DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

URBANA, ILLINOIS 61801

TITLE

Research in the Design of High-Performance Reconfigurable Systems

Third

Semiannual Status Report

October 1, 1984 -- March 31, 1985

NASA Grant # NAG 5-377

Project Personnel

Graduate Research Assistant

Scott D. McEwan

Andrew J. Spry

Principal Investigator

D. L. Slotnick

Table of Contents

Chapter 1

INTRODUCTION.

Computer aided design and computer aided manufacturing have the potential for greatly reducing the cost and lead time in the developement of VLSI components. As this potential becomes a reality the way is paved for the design and fabrication of a wide variety of economically feasible high-level functional units. It has been frequently observed, however, that current computer systems have only a limited capacity to absorb new VLSI component types other than memory, micro processors, and a relatively small number of other parts. The first purpose of the proposed research is to explore a system design which is capable of effectively incorporating a considerable number of VLSI part types and will both increase the speed of computation and reduce the attendant programming effort. A second purpose of the research is to explore design techniques for VLSI parts which when incorporated by such a system will result in speeds and costs which are optimal according to the criterion described in the next section.

It is hoped that the work proposed here will lay the groundwork for future efforts in the extensive simulation and measurements of the system's cost-effectiveness and then, possibly, lead to prototype developement. This proposed research is only the fundamental theoretical and design underpinning for such an effort.

1.1 Computational Time Constraints.

The criterion for judging the hardware design deals with the time constraints placed on the solution to a given problem by different architectures and algorithms. A simple example will be used to introduce the idea. The problem to be considered can be stated as follows: compute the fixed-point sum of k numbers of length l in a time not to exceed some given constant T.

That is, we want to perform the operation:

$$S = \sum_{i=1}^{k} a_i$$

in time $\leq T$.

On a uniprocessor system with a single accumulator the time required to perform this calculation, denoted by $t(\text{ADD } a_i)$, is given by

$$t(\text{ADD } a_i) = 1 \times t(\text{LOAD ACC}) + (k-1) \times t(\text{ADD TO ACC})$$

If $t(\text{ADD } a_i) > T$ on the available uniprocessors an attempt can be made to use a special functional unit, such as an asynchronous adder.

The asynchronous adder (if one were available) would take advantage of the fact that the longest expected carry sequence in the addition of two binary numbers of length l is bounded by $\log_2(l)$. Because the carry propogate time dominates $t(\text{ADD})$ such an adder should be faster than a uniprocesor. With an asynchronous adder the first half adds and carry saves are done in parallel so the expected value of $t(\text{ADD } a_i)$, $E(t(\text{ADD } a_i))$, is estimate by

$$\begin{aligned} E(t(\text{ADD } a_i)) &\leq t(\text{LOAD}) + (k-1)\ t(\text{ADD two Numbers}) \\ &\approx t(\text{LOAD}) + (k-1)\text{Log}_2(l)\, t(\text{ADD } a_i) \end{aligned}$$

If this value still exceeds T, a reasonable next step would be to use a ROM based adder.

In a ROM based adder the summands are used to address a ROM which contains a table of the sums of all the numbers of a the word length l. Such a ROM requires an address space of $2^l \times 2^l$ words where each word contains the sum and carry of the summands that address that location. Therefore, the ROM has a size of $2^{2l} \times (l+1)$. For $l = 8$ this is a 64K by 9 bit ROM. This memory requirement is quite reasonable and yeilds an effective single cycle add. However, because the memory requirement grows exponentially with l it rapidly becomes unrealistic.

For example for $l = 24$ (e.g., the length of a small floating point mantissa) the size of the ROM would exceed 10^{15} bits. Such memories do not exist and if they did would be prohibitively expensive for such a simple operation as addition.

This example illustrates that the problem should be restated in a more realistic manner as follows: compute the fixed-point sum of k numbers of word length l in a time not to exceed some given constant T and at a cost not to exceed some given number of dollars D. To meet this new problem, combination approaches of ROM-assisted sequential logic could be examined. In such a system small ROM's would be used to add sub-words and the results would be combined with more small ROM's and logic circutes to obtain the final result. If these approaches also fail to solve the problem other special purpose functional units will have to be examined.

A method of increasing the computation speed is to use operations that have more than two inputs. One possible system could use k-input adders. A simple serial approach devised by R. K. Richards [1] is illustrated in Figure 1.1. Using this approach yeilds an estimated time of:

$$t(\text{ADD } a_j) = 2(k-1)\,t(\text{HALF ADD}) + t(\text{PROPOGATE CARRY})$$

$$\leq 2(k-1)\,t(\text{HALF ADD}) + \frac{l(l-1)}{2}\,t(\text{HALF ADDS})$$

where the crude estimate is obtained by a worst case assumption (l carries have to be propogated, one from each digit position) for each digit position and summing the arithmetic progression. For $k \gg l$, however, it serves to establish that this approach could result in a faster addition operation. If not, then more costly k' input parallel adders with and without ROM adders for word lengths l' where k' divides k and l' divides l can be investigated. Should none of these combinations obtain the desired cost and performance goals array based functional units can be considered.

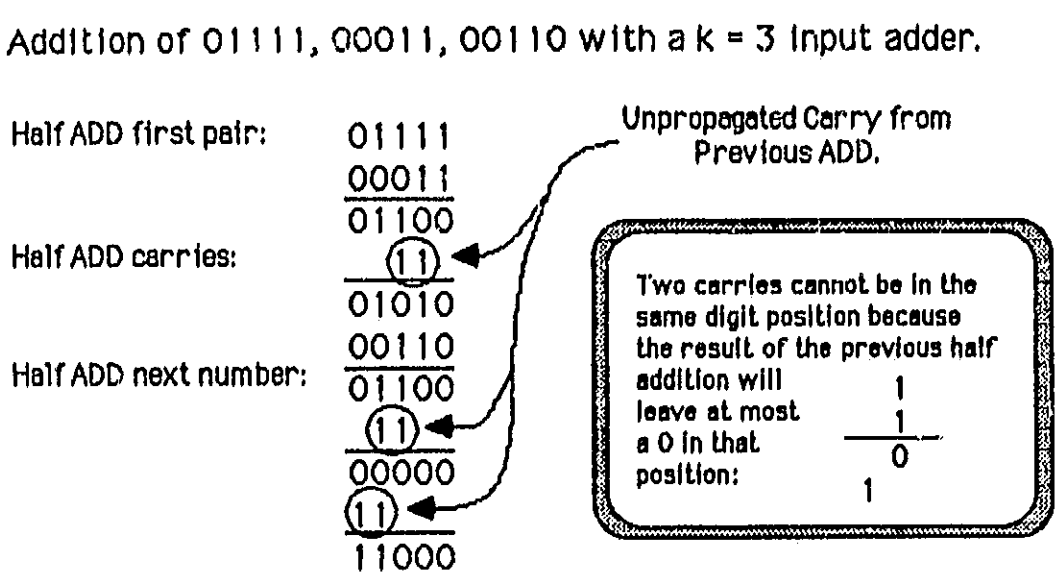


Figure 1.1: ADD Scheme for k Input Adder (R. K. Richards).

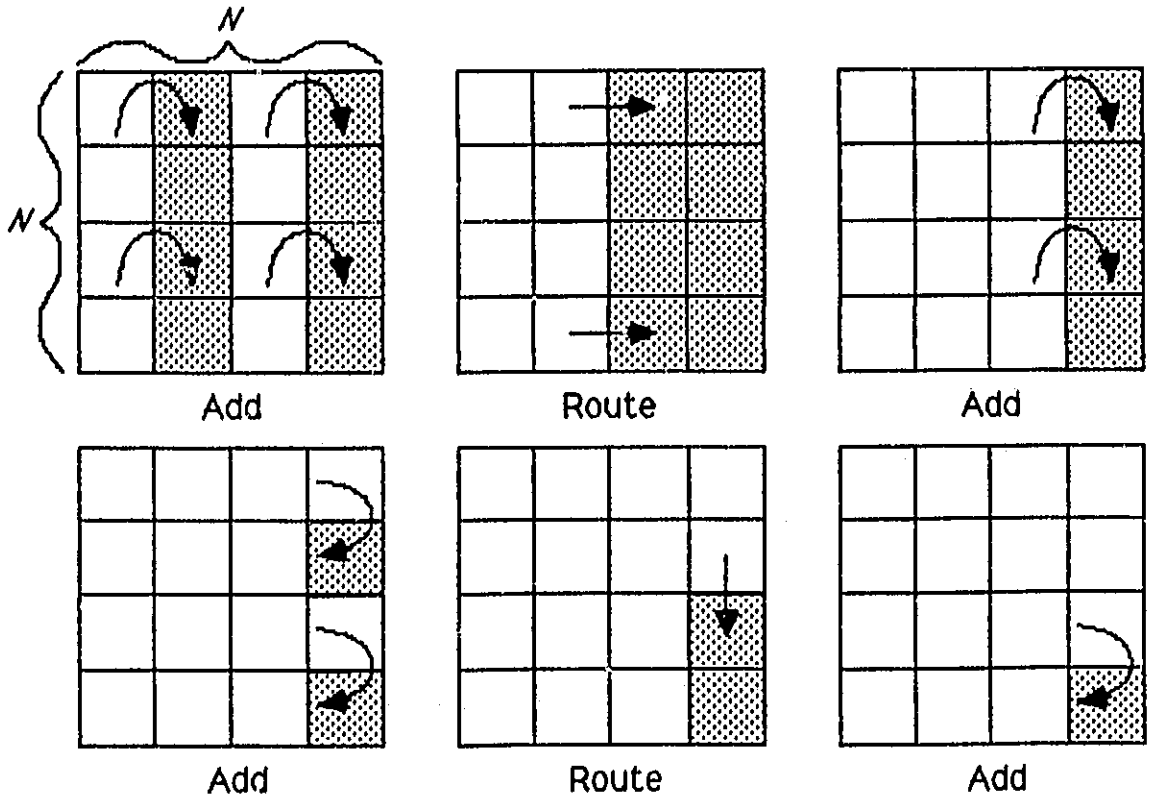


Figure 1.2: Array Add for k=16 and a PE Word Length of *l*.

Figure 1.2 shows a method devised by Cocke and Slotnick [2] of adding n^2 numbers in an $n \times n$ mesh connected array. For simplicity it is assumed that the processing elements of the array have a word length l. The resulting addition time is given by

$$t(\text{ADD } a_i) = 2 \log_2(n)\, t(\text{ADD}) + 2 (\log_2(n) - 1)\, t(\text{ROUTE})$$

where t (ROUTE) is taken to be the time for an average route. The actual route distances increase from length 1 to length $n/2$ during the course of the computation making the average routing distance directly proportional to n. The execution time of these routes will, of course, be a function of the connectivity of the array. If a full $n \times n$ array is to expensive a smaller array can be used with more memory per PE.

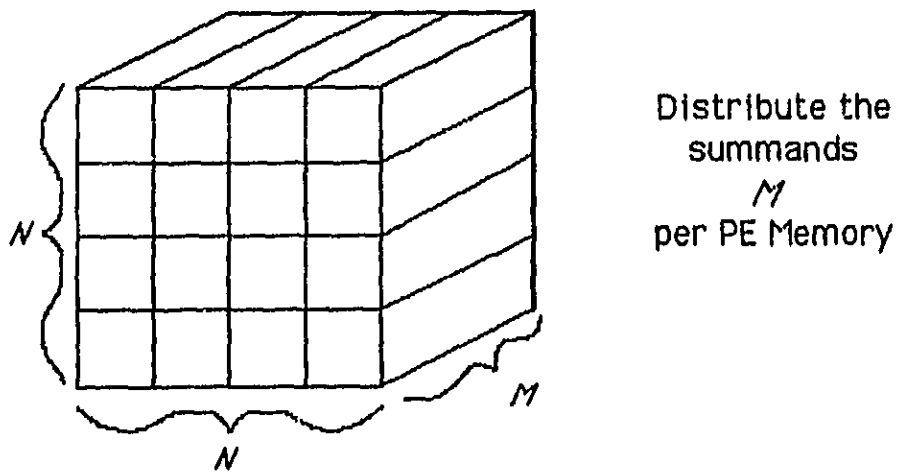


Figure 1.3: Optimal Addition of $k=M \cdot N^2$ Numbers in an $N \times N$ Array.

Figure 1.3 illustrates the optimum way to store $k = m(n^2)$ numbers in an $n \times n$ array for addition, where each processing element (PE) is assumed to have at least m words of storage. Consider solving a problem of size $k = 4n^2$ on such a system. This problem can be solved on an $n \times n$ array with 4 words of memory per PE in

$$t(\text{ADD } a_i) = t(\text{LOAD}) + (3 + 2\log_2(n))\, t(\text{ADD}) + 2(\log_2(n) - 1)\, t(\text{ROUTE})^*$$

or on a $2n \times 2n$ array with 1 word of memory per PE in

$$t(\text{ADD a}_i) = 2(1+ \text{Log}_2(n))\, t(\text{ADD}) + 2\,(\text{Log}_2(n))\, t(\text{ROUTE})^*$$

where $t(\text{ROUTE})^*$ is the mean routing time. The $n \times n$ array uses one more add, saves one route, and requires 4 times the memory of the $2n \times 2n$ array. The $2n \times 2n$ array saves one add, uses one additional long route, and requires 4 times the number of processors as the $n \times n$ array. If the reasonable assumptions (for large n and a nearest neighbor connection) that the long route will require more time than the addition and that memory is less expensive than processors are made, the $n \times n$ array will have the better cost-performance ratio. This analysis shows that increasing the PE memory size, or increasing the speed of the PE as discussed in the case of the $n \times n$ array, would be more cost effective than increasing the array size. The resulting array size will depend on all of the time constraints of the individual algorithm involved and of course on the value of D.

One final parallel approach that will permit a time solution for any $T > 4\,\text{Log}_2(n)$ cycles can be implemented by giving each PE a word length (l) ROM adder, and cross-bar-connections. However, for quite reasonable choices of T, k, and l this will exceed any reasonable D.

For this problem, a reletively complete cost performance (T/D) trade off study is possible with paper and pencil. For a floating point inner-product calculator which is the heart of a fairly popular convolver box such an analysis it is at best difficult. For a mesh calculator, which solves in a restricted T/D subspace by direct or iterative methods only the Laplace Equation for severely circumscribed classes of boundary values and desired result accuracies, the problem is not paper and pencil solvable in any sense.

In summary the characteristics of the general problem are as follows. There is some computation C to be performed in a time $\leq T$. A machine (M) is desired that can solve the problem in time $t(M, C) \leq T$. In addition the cost of the machine $d(M)$ must not exceed a

maximum D. In the optimum sense this problem is stated as follows. Find a machine M for which

$$t(M, C) \leq T \quad \text{and}$$
$$d(M) = \text{Min}$$

Obviously this problem is solvable so the existence of a solution is trivial. The solution is, however, not unique. The optimization problem is an intractable, nonlinear, multi-dimensional problem, so a more realistic statement of the problem is find any machine M for which

$$t(M, C) \leq T \quad \text{and}$$
$$d(M) \leq D$$

where D may be a function of the processing time $D=D(T)$. No existence theorem can be stated for this problem because of its cost condition. Even this problem is too general for any practical solution. The next section further restricts this optimization problem by choosing a problem and a design space that shows particular promise of meeting the optimization goals.

1.2 Implementing a Parallel Machine with VLSI Components.

It is obviously an insurmountable task to consider all algorithms on all classes of machines in terms of the cost performance ratio optimization as developed above. To make the problem more tractable a reasonable choice of problem domain and machine architecture must be specified. It is the intent of this research to use VLSI technology as the basis in designing components of a new class of machines. This machine would have an overall architecture suited to solutions of a particular problem domain. To make such a machine cost effective a rich problem domain, such as linear algebra, must be chosen. As will be shown in the next chapter the linear algebra problem domain is useful in a large number of physical and mathmatical applications. This problem domain also has the benifit of a large body of algorithms for solution

of its basic operations which can be used to guide the system level design.

To meet the two design criterion of making extensive use of VLSI components and having the architecture reflect the problem domain a two pronged design strategy is necessary. First, a reconfigurable high level modular design reflecting the problem domain (or a reasonable subset of the domain) must be created. This design will consist of a number of functional units, controllers, processors, communication switches, and memories operated in parallel. The system level design must provide for extension to, or a change in, the subset of the problem domain that is implemented. The design must also include the ability to incorporate new functional units and new technologies at the functional unit level without extensively disturbing the system level design. To manipulate the design task at this level will require the establishment of a consistent set of accessable design rules based on a consistent family of interconnection techniques. After the functional units of the system have been determined the best means of implementing them using current and anticipated VLSI technology will be determined. This proposal presents the design of a number of VLSI components that can be linked into an illustrative (Inner Product) functional unit that is consistent with the overall design of the reconfigurable linear algebra processing system.

Chapter 2

THE RECONFIGURABLE LINEAR ALGEBRA PROCESSING SYSTEM. (RELAPSE)

[2]re·lapse \ri-'laps\ SINK, SUBSIDE < ~ into deep thought >

2.1 The Linear Algebra Problem Domain.

As stated in the introduction a reasonable problem domain must be chosen before a coherent high level system design can be undertaken and before the cost performance ratio optimization can be addressed. Two creteria were used to determine which application areas to investigate for the problem domain. First, the set of application areas would have to be large enough to adequately explore the system's application scope. Second, the application areas would have to benefit from the higher computer performance likely to be provided by the proposed system. The application areas described below show considerable promise of yeilding to the design approach described above.

The first application area included in the set is image processing. This area includes geometric distortion determination and correction, FFT, image histogramming, statistical clustering of the ISODATA type, and some rudimentary semantic image classification techniques such as template matching. Each of these individual calculations and several subsets of them are candidates for execution by functional units. Study of this area will likely provide a starting basis for the study of radar and other signal processing applications.

Another related application area is the VLSI layout problem. In particular it deals with images composed of a limited number of constituent types. The main problems here are the related ones of placement and routing. Two approaches can be used: heuristic techniques which attempt to reduce combinatorial complexity by sacrificing optimality and rigorous mathematical programming approaches (both linear and quadratic) which are computationally overwhelming.

Both approaches will be investigated as they offer distinct and interresting design opportunities; the former for functional units possibly useful in a variety of AI-type applications and the latter in a large class of optimization problems discussed below.

The linear programming application area is of interest because, in addition to its intrinsic importance, it offers the opportunity to study large, sparse matrix handling including inversions. This application area is perhaps the single most valued application currently performed on medium and large scale machines. An appropriate long word functional unit can, it is expected, be of considerable value. This area also stresses the relation between the functional units and the systems shared (secondary and tertiary) memory resources.

The numerical weather prediction application area is also of interest. The solution of partial differential equations, tipified by numerical weather prediction, depends on handling large sparse banded matrices, that is matrices where the non-zero elements are highly structured into (diagonal) bands whose location is determined by the choice of the differencing scheme. Both iterative and direct methods will be explored from the viewpoint of the subject of this proposal. As with the linear programming application area, both computation and the storage interection in the system are stressed by this application.

The application area of input--output analysis also shows promise of benefiting from the functional design approach. This technique, initiated by Wasily Leontieff, has been applied to a large and growing number of other areas in addition to economic analysis. At its heart is the inversion of a large, dense matrix. For parametric studies, many matrix inversions are usually required. This area will focus attention on the most basic numerical problem; the inversion of high order dense matrices. Attention will be paid to estimating conditioning, involving eigenvalue calculation, and attendant sensitivity analysis. This is, perhaps, the area richest in algorithmic history and should provide an instance where different functional unit approaches can be systematically contrasted.

This group of applications constitutes a reasonable first set of application areas for system design. It is expected that others will be added to the list or substituted as work progresses. This set is obviously too ambitious for the limited scope of a doctoral research program. It is for this reason that the linear algebra domain has been selected as the first problem domain for a system design. The linear algebra problem domain is a subset of many of the more important application areas in the initial set. As will be seen in the next section, a linear algebra based machine would also be capable of processing any of the application areas that contain the linear algebra problem domain as a subset. This is possible because of the inclusion of a powerful uniprocessor as a functional unit in the overall system design. This uniprocessor is capable of performing the calculations of a particular application that do not have a dedicated functional unit in the system.

2.2 Organization of the RELAPSE System.

Current systems may incorporate only a few reasonably high-level specialized functional units such as convolver boxes, FFT calculators, or pipelined high speed floating point units. This may be viewed as a point of departure for the proposed system level design. The question that needs to be asked is what additional high-level functions can be implemented in a flexible framework designed to facilitate cooperation between them and how can that framework be specified in a compliant manner. The functional units of the system should be those whose direct implementation in VLSI will increase the computational effectiveness of the overall system and make its programming easier. The mathmatical description of the problem domain should also serve as a guide in the choice of the functional units of the system.

As stated in the introduction the overall organization of the system should meet the following criterion. The framework should reflect the organization of the problem domain which in this case is the domain of linear algebra. The framework should allow for easy extention to

additional functional units that perform various computational tasks in the problem domain. The framework should also support a high level (possibly multi-programmed) programming environment for the problem domain.

Figure 2.1 illustrates the overall system configuration. The data paths are shown by heavy lines and the control paths are shown by light lines. The figure shows the major components of the system. A main control unit with the capability of a medium size general purpose computer manages the system through the three sub-controllers shown. At the top of the figure special purpose functional units are shown. These units communicate data through a high order switch that connects each functional unit to many (or all) of the others via a full cross-bar. Since each of the functional units implements a high level mathematical function it is reasonable to assume that the relative proportion of data movement to processing is not large. Because of this the switch network does not need to have a very high bandwidth.

Below the functional units are a group of shared memory resources. These communicate with both the input output units at the bottom of the figure and with the functional units. They buffer results between processing by the functional units and provide input/output buffers. The switching network connecting the functional units to the memories is, for the same reason as given above, one of high-order connectivity but not necesarily wide bandwidth. However, a number of special high bandwidth connections may be provided for such items as bulk image data from an input/output peripheral unit.

At the bottom of the figure are a group of peripheral devices that provide the input and output functions of the system. These peripheral devices may include special devices that handle bulk image data and other relatively low-precision (fixed point) sensor data. These devices are connected to the memory units via a high-order high bandwidth switching network. The connections needed for data from some of the peripherals (such as the bulk image data) may require some of the connections between the memory units and the functional units to also be high bandwidth.

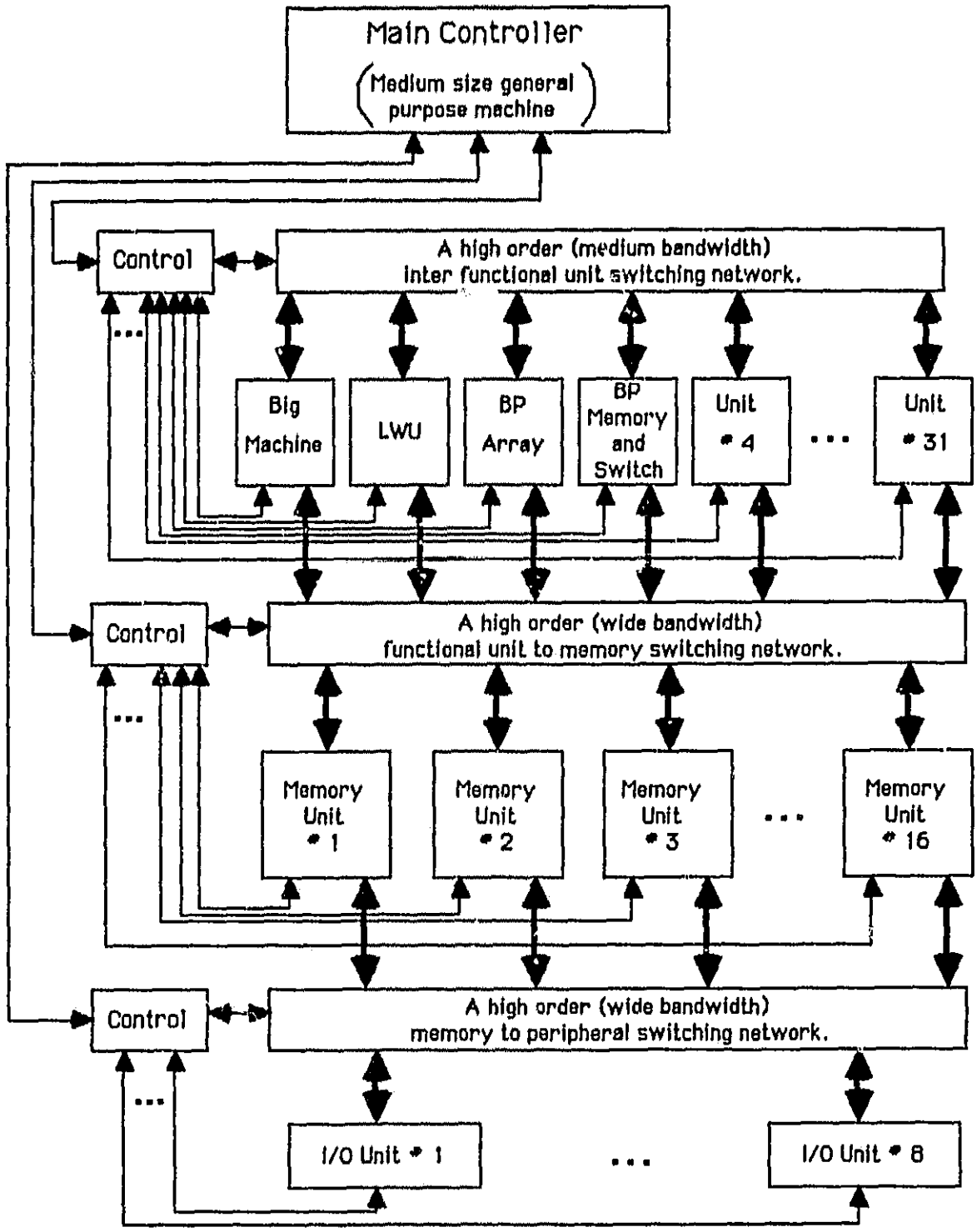


Figure 2.1: Overall Organization of the RELAPSE System.

An impression of the scope of the system design can be gained from noting that a large conventional uniprocessor and a parallel array of processors are shown on the level of functional units. The large conventional uniprocessor is the "default" functional unit which handles those parts of calculations that no specific functional unit exists for. The absence of a special functional unit may result from the lack of a sufficiently frequent need, a low place in the design priority, or from the system being populated to capacity.

The parallel array processor is similarly reguarded as a functional unit. In the figure the array processor is shown as a set of three functional units (the LWU, BP Array, and BP Memory and Switch). This unit can be used to clarify the design philosophy of the system. The BP Array is an array of bit processing elements. The BP Memory and Switch provides the inter-processor routing connections for BP Array, processor to processor-memory connections, and the processor memory. The Long Word Unit (LWU) functional unit is an up-to-now unimplemented functional unit. Its purpose is to handle long words composed of a single status bit (mode or mask bit) from each BP. Since the number of BP's may be large (e.g., a 128 x 128 array in the MPP) these words will be long. The type of processing to be done on these words varies with the context. For BP array control they would be used mainly to test for zero. It is also sometimes necessary to know the position of each one, the number of ones in a row (or column) of an array, or some other more complex function of the mode words. The Long Word Unit could provide these functions. The Long Word Unit will also be useful when each BP has local address modification. (The BP design presented in the next chapter provides this capability.) In this context the local index sets become sequences of long words and effective address calculations may be viewed as long index word calculations influenced by the values of the long mode words.

Were this the only class of applications for a Long Word Unit it could be incorporated in either the BP Array or BP Memory and Switch units. Preliminary analysis indicates that an appropriately designed Long Word Unit may also be beneficial in the processing of large sparse matrix calculations. For this reason the LWU is a separate unit that may be accessed by other functional units independently of the busy state of the remainder of the array processor.

Figure 2.1 suggests a rigidly centralized control philosophy with the traditional roles played by function requests, completion signals, and queueing structures. Actually no explicit control structure is intended by the diagram. A significant amount of data flow control is expected to be used to mediate data transfer between the functional units.

A number of fundamental issues in the overall design of the system will be addressed within the scope of the proposed research. A determination will be made of which subset of linear algebra functional units should be implemented to provide a consistent functional base for estimation of system performance. A more precise characterization of these functional units, the memory units, and the peripheral units will be made. The control sturcture of the system will be further specified. This will include both the data communication protocols and the functional unit control formats. The populations of the different system components and the richness of their interconnections will also be determined.

At this point a few thoughts can be expressed in reguard to programming the system. One of the desired goals of the design is to reduce the application programming effort. It is likely that the overall programming effort will be reduced by this system design approach. There is nothing mystical about this claim. The reason for the programming simplification is that a large proportion of the programming disappears into the design of the VLSI functional units. When programming to use these units only the appropriate input and output parameters (scalars, vectors, and matrices) need to be passed. The system level operating system, which can possibly be a multi-programming operating system, should provide the high level functionality needed for this style of programming.

Programming the system at the application level will be done in a high level functional language for the problem domain. To acheive this goal the results of several centuries of mathematics in identifying a problem's cleanly separable computational elements will be relied upon. It is this mathmatical base that will be a primary input into determining the functional units to be implemented. With this approach it is beleived that evolutionary change of the functional units should cause no reprogramming difficulty if the changes only reflect the manner in which a functional unit performs its function rather than the function itself.

Chapter 3

THE BIT PROCESSOR, THE STAGE, AND ARITHMETIC UNITS.

The functional units of the RELAPSE consist of a hierarchy of simpler components. At the highest level of the hierarchy are the arithmetic units (AU's) which provide the bitwise logic operations and multiple precision arithmetic of the functional units. The AU's are in turn composed of 8-bit processors, called stages, which provide a high speed single precision arithmetic for the arithmetic units. Each stage is in turn composed of a set of eight 1-bit processors (BP's), the hardware needed for high speed single precision arithmetic, and the hardware that allows the steges to be coupled into atithmetic units. The bit processors are the smallest computational unit of the hardware. They are single bit processors that can be operated in a bit serial mode or in cooperation with other BP's as part of the stage and artihmetic units in a bit parallel mode.

The hierarchical design of the arithmetic units has a number of advantages over a monolithic design. At the lowest level the design consists of a small number of simple components amenable to VLSI implementation. The small number of distinct components decreases the complexity of the design. This in turn reduces the probability for design errors and reduces the design cost. In addition the computational power of the stage and bit processor, which is far from negligible, can be utilized in units that are not composed directly of arithmetic units such as arrays of bit processors and special long word processors.

3.1 The Bit Processor.

The design of the bit processor represents a compromise between efficiency in low precision (4 to 7 bit words) fixed point operation, and higher precision (8 bits or longer words) fixed and floating point operation. The efficient use of memory and processing time in

the 4 to 7 bit word lengths of many signal and image processing problems point toward a bit serial, variable word length, mode of operation. Problems that require rich connectivity also point toward the bit serial mode of operation since many low cost (single bit bus) connections can be provided. The higher precision fixed and floating point word lengths needed for sparse and dense matrix inversions and aspects of image processing problems such as FFT and convolutions point toward a bit parallel mode of operation. The bit parallel mode of operation will also be more efficient for problems exhibiting a lower order of parallelism.

The BP has the following general characteristics. It has two modes of operation, a bit serial and a bit parallel mode, refered to as the vertical mode and the horizontal mode. In the horizontal mode eight BP's are used in conjunction with additional hardware to create a high speed 8-bit processor. The BP's have a dual memory, two input buses, and one output bus. The BP's are operated synchronously from a central control unit. The control units are programmable in a two address assembly language that produces encoded micro instructions. The BP's routing logic is also programmable to allow for rich connectivity in the vertical mode and to provide data communication paths for the horizontal mode.

A large number of bit serial processors have been developed for array machines including the Solomon, the DAP, and the MPP. The MPP's processing element was chosen as the point of departure for the BP because of its excellent bit serial processing capabilities. There are, however, few remaining overall similarities between the BP and the MPP's PE. The MPP's PE is not designed to be coupled into bit parallel procesors, is only a one address processor, and has only a nearest neighbor connection for its routing logic.

3.1.1 Functional Description of the Bit Processor.

Figure 3.1 gives a block diagram of the design of the BP. The a and b buses are used for input to BP registers. The o bus is used for output from BP registers. Each input bus can be

loaded from one memory module, the output bus, or the L buffer. The connection of the output bus to the input buses allows register to register transfers in one cycle. The two seperate input buses also allow the input of two operands from memory in one cycle provided they are stored in separate memory modules.

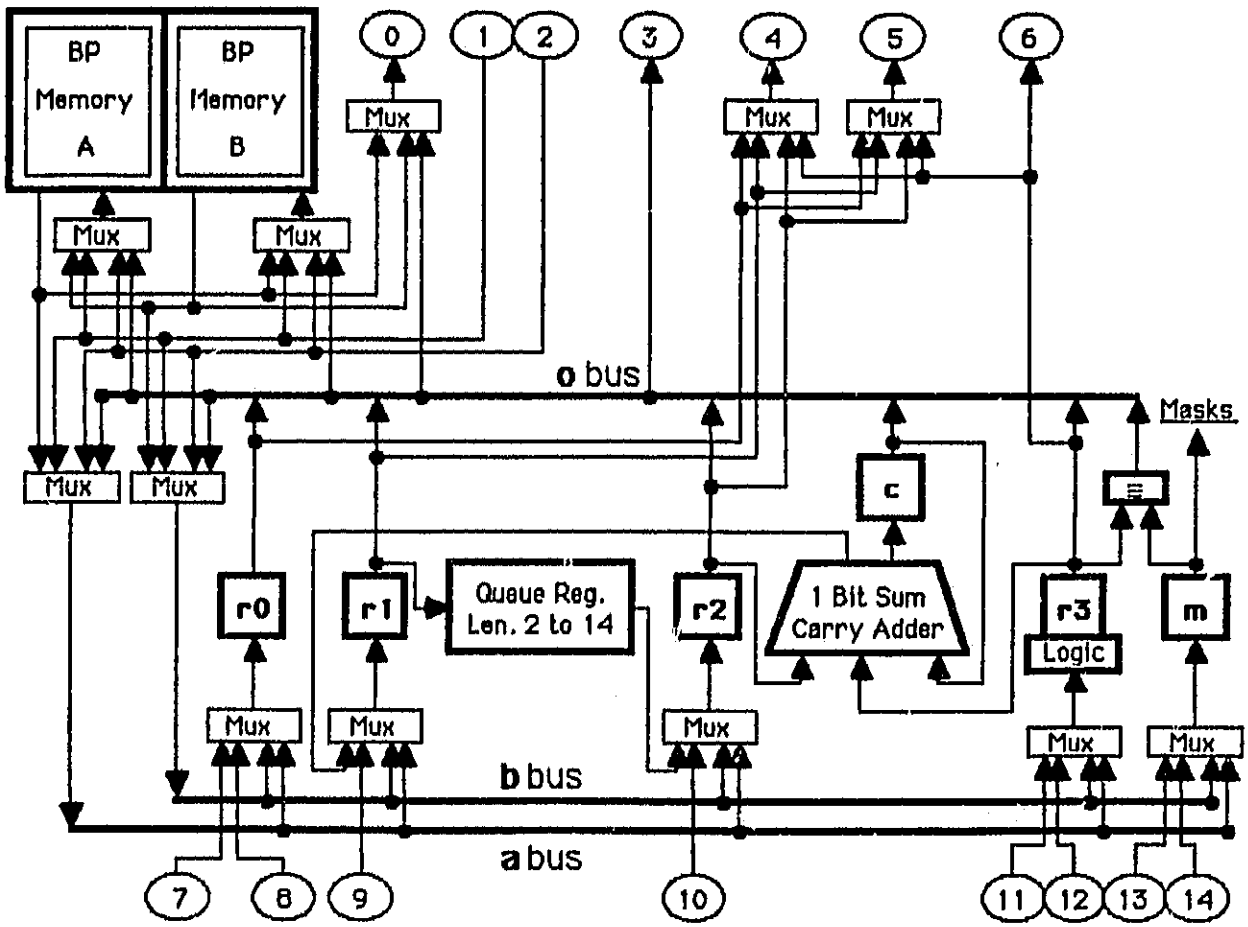


Figure 3.1: The Bit Processor with Its Associated Memory.

The two memory modules of the BP are composed of standard commercial memory chips with on chip address decoding. Data is input to and output from the BP to the L buffer by stealing a BP processing cycle. Data input from the L buffer can be stored either in the memory modules or directly in a BP register. Data output to the L buffer can originate from either a memory

module or from the **o** bus. The input and output connections from the BP to the L buffer are shown in Figure 3.1 by the circles labeled 1 and 0.

The L (buffer not shown in the figure) is used in the vertical mode to reformat the data from a bit parallel format of the host machine to the bit serial format of the BP. The L buffer also provides a speed matching buffer in both modes of operation of the BP. The implementation details of the L buffer are one topic of the proposed research.

Source and Destination of Data for the BP Registers.

Register	Sources of Input	Destinations of Output
r0	The **a** bus, the **b** bus, and one bit of the sum from the add ROM.	The **o** bus, and one bit of the add and multiply ROM address.
r1	The **a** bus, the **b** bus, the sum bit from the sum carry adder, and one bit of the low order byte of a product.	The **o** bus, the queue register input, and one bit of the add and multiply ROM address.
r2	The **a** bus, the **b** bus, one bit of the high order byte of a product, and the output of the queue register **q**	The **o** bus, the sum carry adder, and one bit of the add and multiply ROM address.
r3	The **a** bus, the **b** bus, one bit of the the sum from the add ROM, and the input from the routing logic.	The **o** bus, the sum carry adder, the routing logic, the zero detect logic, the equivalence function, and one bit of the add and multiply ROM address.
m	The **a** bus, the **b** bus, and the stage level mask control.	The bit processor mask lines, and the equivalence function.
c	The carry bit from the 1 bit sum carry adder.	The **o** bus, and the sum carry adder.

Table 3.1: The Inputs and Outputs of the BP Registers.

The inputs and outputs of the BP registers are given in Table 3.1. The BP has four general purpose registers, r0 through r3, which form the primary processing registers of the BP and in turn the stage. All the general purpose registers can be loaded from the input buses and written to the output bus. The r0 register, which has no special function in the vertical mode, can be used as a storage location for data. The remaining general purpose registers have special functions in the vertical mode.

The r2 and r1 registers form the head and tail of the BP's queue register (q). The q register is a shift register of variable length that serves as a partial result queue for bit serial arithmetic (e.g., as a partial product register for multiplication). The length of the q register can be set to 2, 6, 10, and 14 bits. By choosing the next length larger than the size of the word being procesed bit serial algorithms can be customized to execute efficiently on the BP. For word lengths larger than the q register the horizontal mode of operation is more efficient than vertical mode because partial results have to be stored in memory.

The r3 register is the logic engine of the BP. The logic hardware associated with it can perform the 16 bit-level logic functions of two variables. The contents of the register and the bit being loaded are used as the inputs to the logic hardware. The r3 register is also the source and destination register for the routing logic. In one operation the contents of r3 can be loaded from and written to another BP using the routing logic. The routing logic provides a nearest neighbor connection in two dimensions and an abbreviated power of 2 connection in one dimension. The details of the routing logic will be discussed later. The bit level logic and routing functions of the BP's r3 register are used by both the stage and arithmetic units.

The r1, r2, r3, and c registers are used in conjunction with the q register and a 1-bit sum carry adder to provide vertical mode arithmeitc. The sum carry adder takes as its inputs the values stored in the r2, r3, and c registers and produces a sum and carry output. The sum bit is loaded into the r1 register where it can be stored in the q register if desired. The carry bit is loaded in the c register where it can be cycled back for the next bit of the sum.

The **m** register of the BP is used to hold a mask bit. This bit is used to control the execution of a masked instruction according to the value of some local data. Only BP's that contain a 1 in the **m** register will participate in masked operations. The **m** register can be loaded from local data via the input buses or from a stage level input in the horizontal mode. In vertical mode the **m** register can be used to perform exception handling. For example the **m** register can be cleared by an algorithm to indicate an overflow. Once the **m** register is cleared its BP will no longer participate in the masked instructions of the algorithm. The contents of the **m** register can be loaded onto the output bus only through the **r3 ≡ m** function. This function can be used to determine if the **m** register was set or cleared to determine if exceptions occured during a bit serial algorithm. The use of the stage level mask in multiple precision horizontal mode arithmetic will be described later.

The input and output connections of the BP shown in Figure 3.1 by the labeled ovals are listed in Table 3.2. Connections 1 and 0 provide the 1 bit input and output paths between the **L** buffer and the BP. Connection 3 provides access to any value on the **a** bus. This connection can be used for a zero detect by taking the logical OR of a number of BP's either at the stage level or in a tree arrangment for a matrix of BP's. This connection can also be used to obtain the value of the the **r3 ≡ m** function. Connections 12 and 6 provide the input and output paths from BP to the routing logic. The remaining connections are extensions to the BP for use in the horizontal mode and will be described later.

As stated before all BP's are operated synchronously under the command of a micro programmed control unit. The control unit structure will depend on the orgainzation of the component the BP's are used within. For example, the BP's organized into the stages will have a different control unit than a set of BP's organized into an array processor. All operations done by the BP above the level of addition and 1-bit logic must be programmed. The horizontal and vertical modes of operation will have separate assembly languages to distinguish the functions available in the different modes. For example, the operation of multiplying two numbers would

require a call to a control unit which would execute a micro code subroutine to read two operands from memory, add them one bit at a time using the carry sum adder, and form the partial products in the q register. The subroutine for this operation would be written in the vertical mode assembly language because it uses the carry sum adder which is unavailable in the horizontal mode assembly language. The operation of the BP's in horizontal mode will be described in conjunction with the stage below.

BP Input and Output Points.	
Input/Output Number.	Bit is To or From.
0	To bit *i* of the L Buffer.
1	From bit *i* of the L Buffer.
2	One bit of the Sum from the add ROM (horizontal mode).
3	To sum-or tree, and zero detect logic.
4	One bit of the high order byte of the add or multiply ROM address (horizontal mode).
5	One bit of the low order byte of the add or multiply ROM address (horizontal mode).
6	To the routing logic.
7,11	One bit of the Sum from the add ROM (horizontal mode).
9	One bit of the low order byte of the product from the multiply ROM (horizontal mode).
10	One bit of the high order byte of the product from the multiply ROM (horizontal mode).
12	From the routing logic.
13,14	Stage and arithmetic unit level mask inputs.
8	Currently unused.

Table 3.2: Input/Output Points of the Bit Processor.

3.2 The Stage.

The stage is the atomic unit of the horizontal mode of operation. In the horizontal mode all arithmetic is based on the 8-bit single precision arithmetic of the stage. Each stage has hardware that provides high speed 8 bit addition and multiplication. Each stage also contains additional hardware that allows it to operate with other stages to form long word arithmetic units. When grouped into long word units each stage can be considered a one digit procesor where the digits have a base of 2^8.

Figure 3.2 shows the block structure of a stage. At the heart of each stage is a set of eight BP's. A 64K X 9 bit add ROM and a 64K X 16 bit multiply ROM are used to perform the high speed 2's complement single precision arithmetic of the stage. The stage also contains the micro programmable routing logic used to transfer data to and from the r3 reigisters of its internal BP's and the r3 registers of the neighboring stages. Because the BP was designed to be coupled into the stage as well a bit serial arrays the stage uses much of the BP's hardware directly. In addition to the components shown in the figure each stage contains additional hardware that allows it to be coupled into the multi stage arithmetic units.

3.2.1 Functional Description of the Stage.

The stage has three 8-bit data buses, referred to as the A, B, and O buses, which are composed of the 1-bit BP buses operated in parallel (see Figure 3.1). The A and B buses can be loaded with a single byte from the L buffer, from the add ROM's sum byte, from the O bus, or from memory. The A memory can be read on the A bus, and the B memory can be read on the B bus. The O bus can be sent to the A bus, the B bus, the A memory, or the B memory. In addition the O bus can be used as an input to zero detect logic at the stage and word level.

The four 8-bit general purpose registers of the stage (R0 - R3) are composed of the BP's 1-bit registers (r0 - r3) operated in parallel. Any general purpose register can be used

as the source of operands for the single precision arithmetic of the stage and any register can be used as the destination of the sum of a single precision add. The other stage level operations such as multiplication can be performed only on subsets of the general purpose registers. The R1 and R2 registers can be used as the destination for the 16 bit product of the single precision

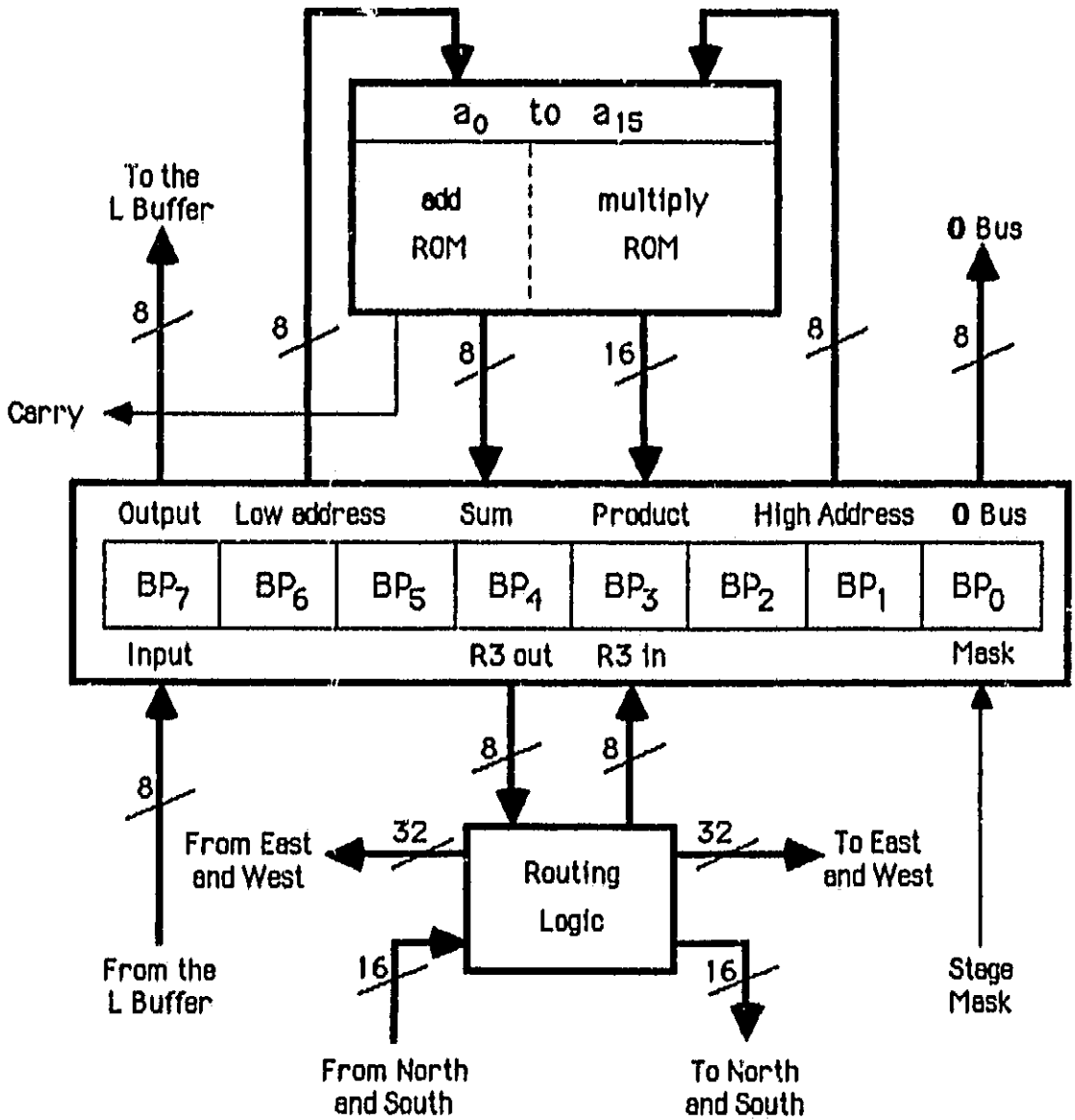


Figure 3.2: Block Diagram of the Stage.

multiply. The R1 and R2 registers also function as the tail and head of the 8-bit wide Q register. The Q register, which is composed of the q registers of the BP's operated in parallel, can be configured into lengths of 2, 6, 10, and 14 words. The R3 register is used to perform all bit wise logic functions using the load logic of the BP's r3 registers operated in parallel. The R3 register is also connected to the micro programmable routing logic.

The stage-level mask register M consists of the BP's m registers operated in parallel. For a stage level mask to occur a single mask bit input to the stage is distributed to the m registers of each BP. The stage level mask bits are connected across the steges to form a word level mask register. This word level mask register can be shifted one stage in each cycle. This allows sections of long words, or entire words, to be masked out of operations. This capability is useful in exception processing and floating point arithmetic, in multiplication and broadcasting.

Micro programmable routing logic is provided at the stage level. This logic is used in both the vertical and horizontal modes to provide communication paths between BP's. In the vertical mode the routing logic provides nearest neighbor connections in two dimensions. This functionality allows the creation of two dimensional mesh connected arrays of bit processors. In the horizontal mode of operation the routing logic provides two levels of function. The nearest neighbor connection will be provided in two dimensions and a nearest stage connection will be provided in one dimension. The nearest neighbor capability can be used in the horizontal mode for one bit shifts in either direction along arithmetic units and for long word shifts perpendicular to the arithmetic units. This capability is simply the result of applying the nearest neighbor connectivity of the BP's in a parallel manner. More importantly a second routing capability is provided for operand shifts in stage increments. This capability can be used to normalize floating point mantissas more rapidly then single bit shifts. To provide multiple precision arithmetic based on the single precison arithmetic (base 2^8) of the stage one cycle shifts of 8 BP's is desirable. The trade offs between a simple nearest stage connection (where BP's are connected at a distance of $\pm 2^3$), and an abbreviated power of two network (where the

BP's are connected at distances $\pm 2^1$, $\pm 2^2$, and $\pm 2^3$) will be investigated. The major advantage of the power of two network is that shifts of a distance D (e.g., in floating point normalization) can be done in $O(\log_2(D))$ time instead of $O(D)$ time. The connections along the arithmetic units will also allow logical and arithmetic shift operations, sign extension, and special guard bit handling in floating point operations. Thus, the complete function of the routing logic depends on the range of connections needed to provide both veritcal mode BP communications and efficient horizontal mode stage and arithmetic level communications. The best method of providing the communication along the stage and arithmetic units will be one topic of the proposed research.

3.2.2 Single Precision Logic and Arithmetic on the Stage.

As stated above the stage provides single precision arithmetic for the arithmetic units. This arithmetic can be considered base 2^8 arithmetic where each stage contains one digit. The descriptions of the single precision arithmetic operations will be given in terms of the micro operations of the stage's components. The timing estimates will be based on a ROM memory cycle time of 50ns. Although there are other cycle times in the stage, the basic cycle time for operations based on ROM lookups is one memory cycle time.

The simplest single precision oerations are the bit wise logic operations. All bit wise logic operations con be performed in one machine cycle using the load logic of the R3 register. With a two address assembly language any logic operation of two variables can be specified as a single statement of the form:

LOGICOP OP1,OP2

Such a logic operation can be performed in at most 3 cycles. This maximum time arises if the first operand is in any other location than the R3 register. In this case the following micro

operations would be used to produce the desired result:

```
R3          ← MEM[OP1]
R3          ← [R3] LOGICOP MEM[OP2]
MEM[OP1]    ← R3
```

As an optimization, statements where the first operand is the **R3** register should be assembled into a one machine cycle operation.

The time required for logical and arithmetic shift operations will depend on the power of the routing logic. For a full power of two network (single cycle routes at distances $\pm 2^0$, $\pm 2^1$, $\pm 2^2$, ...) a shift of distance D could be performed in $O(\text{Log}_2(D))$. With an abbreviated power of two network, the times for a shift of D will of course be greater but will still be an improvement over a distance one shift time of D. High speed shift operations are by far more important at the arithmetic unit level than at the single stage level. In all cases special hardware will be added at the ends of the words to provide for the cycling of bits in logic shifts and for the introduction of the correct bits in arithmetic shifts. This additional hardware will be discussed later in relation to the arithmetic unit. The desired timing of the shift operations will be used as an input in determining the final horizontal mode routing logic.

Single precision addition is performed by a table lookup in a 64K X 9 bit add ROM that contains the 2's complement sum and carry of the operands used as ROM addresses. Any general purpose register can be used as the source for the add ROM addresses. The operands of the addition are read out of the BP's by the connections labeled 4 and 5 in Figure 3.1. The sum from the ROM can be placed on the **O** bus (connection 2 in Figure 3.1), or loaded directy into the **R0**, or **R3** registers (connections 7 and 11 in Figure 3.1). The carry bit from the ROM is available for output to the next stage and is stored in a stage carry register **SC**.

Consider the addition of two numbers stored in different memory banks. Since the add ROM provides 2's complement addition in one machine cycle the operation can be performed in three cycles by the following micro operations.

```
R2          ← MEM[OP1];  R3 ← MEM[OP2]
SC, R2      ← ADD[R2, R3]
MEM[OP1]    ← R2
```

The SC register can be read to detect overflow. If the operands are both in registers then the addition will require only two cycles. Thus, single precision additions can require from 1 to 3 ROM cycles depending on the location of the operands. With current ROM speeds this corresponds to 50 to 150ns.

Next consider the subtraction of two numbers in 2's complement format that are stored in different memory banks. This operation can be performed in four cycles by the following micro operations.

```
R2          ← MEM[OP1];  R3 ← MEM[OP2];  SC ← 1
SC, R3      ← ADD[R3,SC]
SC, R2      ← ADD[R2,R3]
MEM[OP1]    ← R2
```

Here again the actual speed will depend on the location of the operands. For two operands already in registers the subtraction requires only two ROM cycles. Thus, single precision 2's complement subtraction will require between 100 and 200 ns.

Single precision multiplication is performed by table lookup in a 64K X 16 bit multiply ROM that contains the 2's complement product of of the operands used as ROM addresses. Any general purpose register can be used as the source of the operands for multiplication. Figure 3.3 shows how multiplication would be done if the operands are read from the R1 and r2 registers. This operation takes one ROM cycle to obtain the 16 bit product of the operands. The

product is always put onto the R1 , and R2 registers with the low order byte of the product being stored in the R1 register. The maximum time required for multiplication of two values from memory would be 3 ROM cycles. This includes the time needed to obtain both operands from memory module and the time needed to store the result back to memory.

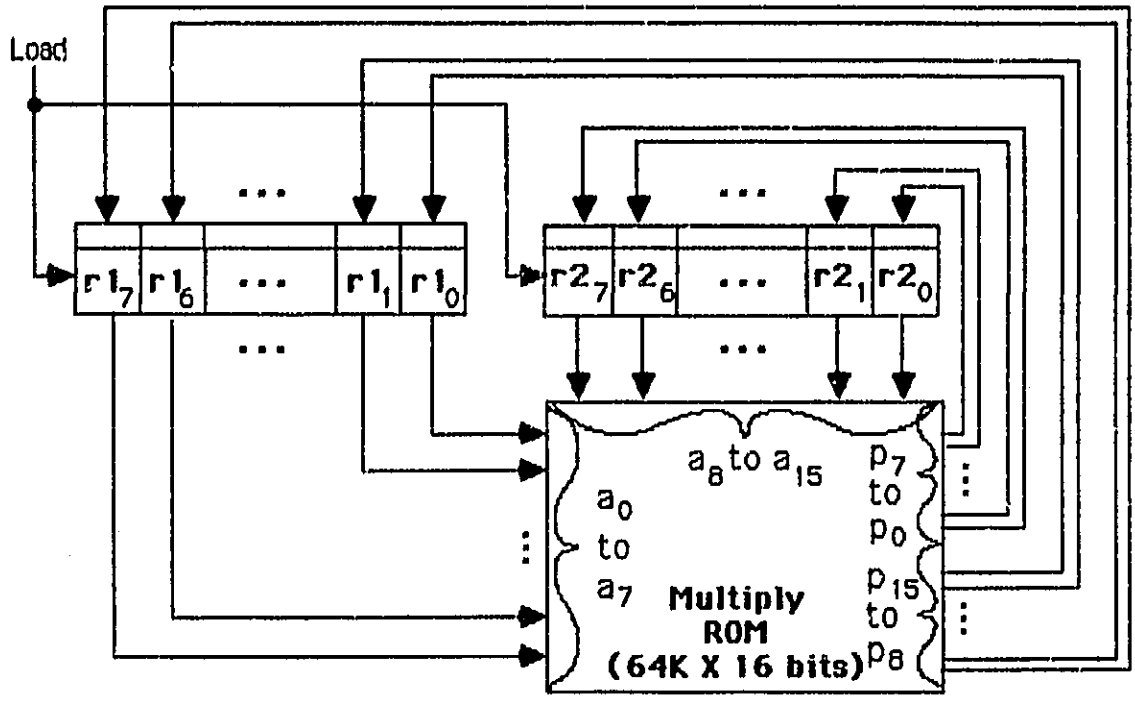


Figure 3.3: Single Precision Multiply.

A number of areas of the design of the stage are still to be determined. The operation of single precision division for the stage is not yet specified. The possibility and usefulness of simultaneous multiplication and addition will be investigated. The amount of autonomous control that each stage will have is also an open question. Under consideration is how much of the hardware neeeded for the floating point and multiple precision arithmetic should be built into the stage as opposed to being put into the arithmetic units and the control units. The stages by their very nature will have to have their operation controlled by a micro programmable control

unit. The details of the assembly language for the stage are yet to be worked out. Finally, the usefulness of sub-stages will be investigated. A sub stage would be a processor that is capable of only one of the basic operations such as multiplication or addition. Such a sub-stage would not necessarily need the memory modules of the stage and would have a smaller set of registers. The high speed multiple precision arithmetic described below uses a set of sub-stages in conjunction with stages to acheive its processing speed by pipelining the multiplication and addition operations.

3.3 Arithmetic Units.

For the purpose of the RELAPSE machine, arithmetic units are defined as any collection of BP's, stages, and sub-steges which perform the multiple precision arithmetic of the functional units. This definition is intentionally general enough to allow many different configurations of processors within the functional units. It will be seen below that although a simple linear array of stages can perform a respectable multi precision multiplication operation a special aritmetic unit can be constructed of stages and additional components to obtain even faster multiplication speeds. These high speed long word multipliers can be used profitability in functional units where the overall execution time is dominated by the multiplication step (such as an inner product functional unit).

3.3.1 Multiple Precision Data Formats.

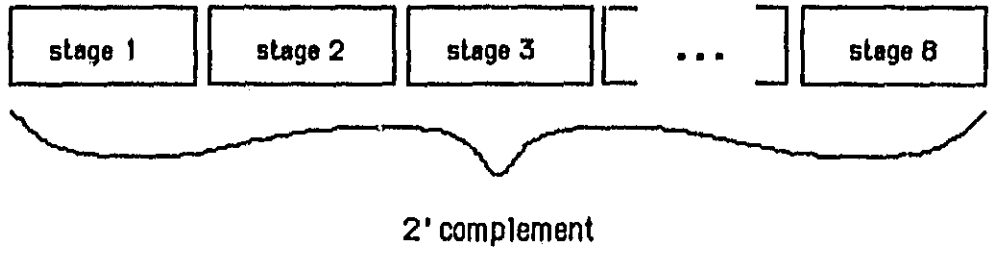
The multiple precision data formats of the RELAPSE machine are greatly influenced by the design of the stage. The stage provides a 2's complement single precision arithmetic that can be considered as either binary arithmetic or base 2^8 arithmetic. The fact that all single precision arithmetic of the stage is performed by ROM table lookup, and the fact that the ROM's contain the 2's complement sum and product of the operands, imply that all arithmetic on the

RELAPSE machine is done in 2's complement. This has no great effect on the fixed point cardinal and integer number formats since 2's complement is a common choice for these data types. It does have an interesting effect on the floating point formats, however, since even the exponent of a floating point number must be in 2's complement. The design of the stage has two other effects on the data formats. Because stages are designed to be coupled into arithmetic units each stage contains the hardware necessary to be the boundary of a data word. This implies that words in the data formats described below can be of any length up to the size of the arithmetic unit. For example, a 128 bit floating point processor built from stages can be reconfigured into two 64 bit processors operated in parallel simply by designating one of the middle stages as a data word boundary. The other effect on the data format is that all multiple precision formats are a multiple of stages in length. Since the stage is an 8-bit processor (the length of one byte on most systems) this effect is minimal.

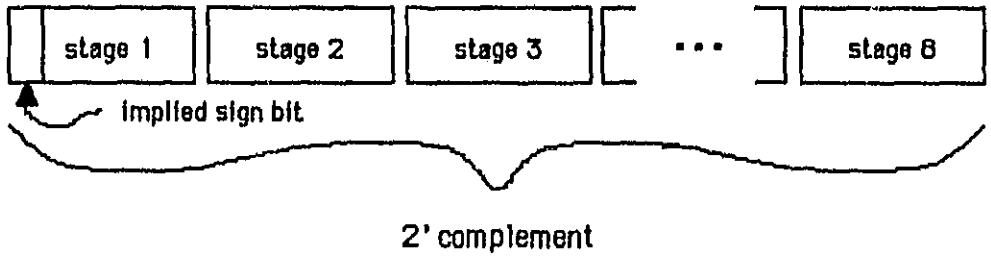
The format of a 64 bit cardinal number is shown in Figure 3.4(a). A cardinal number is formed by connecting a set of stages in a linear array and operating them in parallel. The length of a cardinal number must be a multiple of the length of a stage. Therefore cardinals can be used to represent numbers in the range from 0 to $2^{(8N)}-1$ where N is the number of stages in the arithmetic unit. The arithmetic performed on cardinals is modulo $2^{(8N)}$ arithmetic with optional overflow detection provided by the carry out of the highest order stage.

The format of a 64 bit integer number is shown in Figure 3.4(b). The length of the integer number must be a multiple of the length of the stage. An integer can be used to represent numbers that range from $-2^{(8N)}$ to $2^{(8N)}-1$. The arithmetic performed on integers is 2's complement with overflow detection provided by the carry out of the highest order stage.

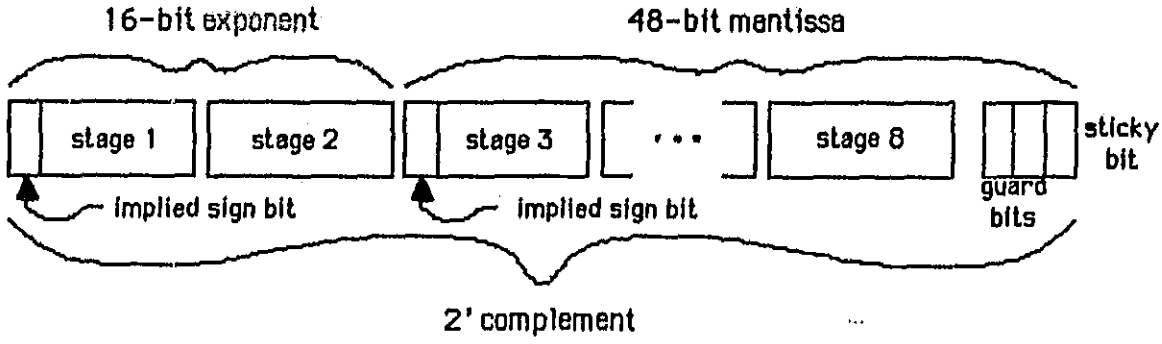
An example of a 64 bit floating point format is given in Figure 3.4 (c). As previously mentioned the entire floating point number must be stored and manipulated in 2's complement. If F is the number of stages in the mantissa and E is the number of stages in the exponent, then for binary floating point numbers (radix base 2) with fractional mantissas the values that



(a) 64 Bit Cardinals.



(b) 64 Bit Integers.



(c) 64 Bit Floating Point.

Figure 3.4: Multiple Precision Word Formats.

can be represented in this format are:

$$2^{(8F-2^{(8E-1)})} \quad \text{to} \quad (1-2^{(8F-1)})2^{(2^{(8E-1)}-1)}$$

and

$$-(1-2^{(8F-1)})2^{(2^{(8E-1)}-1)} \text{ to } -2^{(8F-2^{(8E-1)})}$$

and

$$\pm 0.$$

For the 64 bit floating point format of Figure 3.4(c) where $F = 6$, and $E = 2$ these correspond to:

$$2^{-32720} \text{ to } (1-2^{47})2^{(2^{47}-1)} \text{ and } -(1-2^{47})2^{(2^{47}-1)} \text{ to } -2^{-32720} \text{ and } \pm 0.$$

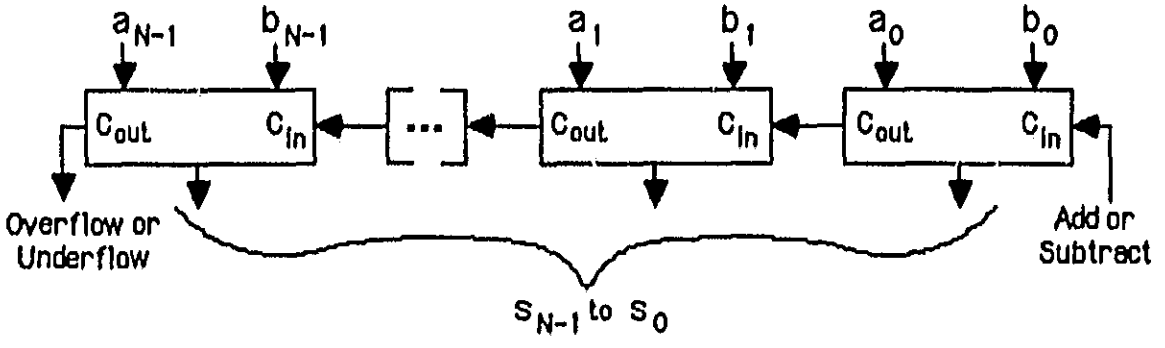
Each stage has the hardware needed to function as the exponent mantissa boundary. Because of this a floating point number with greater precision can be created simply by adding more stages to the mantissa. The only restriction on the size of the mantissa and exponent is that each has to be a multiple of a stage length. The impact, on the stage hardware, associated with its use in handling exponents is described later in relation to floating point addition.

A block floating point format can also be supplied by stage base arithmetic units if the functional unit's controller has block exponent hardware. In a block floating point the mantissas of the values are stored in the arithmetic units and processed there while the exponent is stored and manipulated in the control unit. The exponent hardware can be composed of stages if desired. Such a format would provide faster floating point processing for problems that have a limited dynamic range of real (non-integral) values. The arithmetic of the block floating point would be faster than regular floating point because there would be only an infrequent need to perform global normalizations and except for the global normalization all mantissa arithmetic is essentially fixed point (integer) arithmetic.

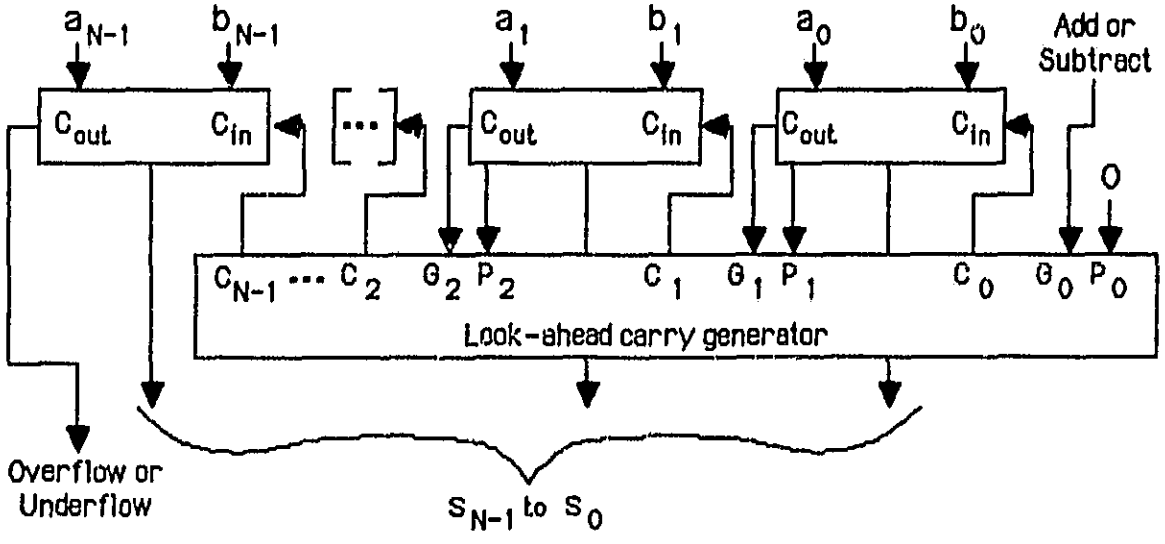
3.3.2 Multiple Precision Arithmetic.

The multiple precision arithmetic of the functional units of the RELAPSE use the data formats described above. These data formats allow words of different length to be created by modularly adding stages to the hardware of the arithmetic units. In this section fixed point addition, floating point addition, and fixed point multiplication are described. Attention will be paid to the construction of the arithmetic units that perform these calculations. In particular the additional hardware of the stage, not described above, needed for multiple precision operations will be discussed. As with the arithmetic of the stage a cycle time of one ROM memory cycle will be used as the unit of measurement for the algorithm times. It should be noted that this is likely to provide a pessimistic estimate because shift times in some algorithms will be faster than the ROM memory accesses. For simplicity, however, and because the final routing logic has not been specified this single cycle time will be used. In addition, all the timing estimates will be given for register to register operations. This is the minimum time in which the described operations can performed. Unless otherwise stated the maximum time required for an algorithm will be 2 cycles longer than the minimum. This time differential results from the delay of reading the two operands from memory and writing the result back to memory.

The first operation to consider is fixed point addition. Two possible configurations for an N stage ($8N$ bit) adder are shown in Figure 3.5. In each configuration the stages provide all the hardware needed to perform the addition in 8 bit slices. In the first scheme the carries are propagated across the stages as a ripple carry. Because of this there will be an N stage delay in obtaining the sum of A and B. The stage, as shown in Figure 3.2, can be coupled into this adder scheme without any additional hardware. In the second scheme a slightly more complicated stage is required. Each stage must make available a carry propagate (P_i) and carry generate (G_i) signal for use in the carry look-ahead circuit. Since each stage can be considered a separate digit an $N-1$ input carry look-ahead circuit will be sufficient for an N



(a) N Stage Ripple Carry Adder.



(b) N Stage Carry Look-ahead Adder.

Figure 3.5: Multiple Precision Adder Configurations.

stage adder. The carry propagate signal can be produced by taking the logical AND of the sum outputs of the add ROM. The carry generate signal is simply the carry output of the stage as before.

The ripple carry adder can add two N digit fixed point numbers in N cycles. One cycle is needed to take the sum of the initial values and $N-1$ cycles are needed to propagate the carry. For a 64 bit fixed point number eight cycles are required giving total time of 400ns. The carry

look-ahead adder's execution time depends on the size of the carry look-ahead circuit. If eight stages and their associated logic are placed on a single board an 8 input carry look-ahead circuit is a reasonable choice. With an 8 input carry look-ahead (note: only 7 inputs of the circuit are actually used) the addition of two 64 bit fixed point numbers will require only 2 cycles. The first cycle is used to create the carry generate and propagate signals from the inputs and the second cycle is used to correct the sums generated on the first cycle for the carries. For word lengths of 128 bits the carry can simply be rippled from one 64 bit group to the next to provide a 3 cycle 128 bit add time. If each 64 bit group also provided a carry generate and propagate signal then one additional level of carry look-ahead can provide a 3 cycle add time for word lengths of up to 512 bits.

Fixed point subtraction can be performed by both of the adder schemes shown in Figure 3.5. To perform a subtraction the 2's complement of the subtrahend must be determined. Using the ripple carry adder of Figure 3.5(a) subtraction will take $2N+1$ cycles. The first cycle is used to load the R3 register with the 1's complement (logical NOT) of the subtrahend. The carry in of the first stage is set to one and an addition is performed to generate the 2's complement of the subtrahend in N cycles. An addition (requiring N more cycles) is then performed to obtain the final result. For a 64 bit fixed point subtraction this requires 17 cycles. The carry look-ahead adder can improve on this performance even more than it could improve on the performance of the addition operation. For the 64 bit addition only 7 input pairs of an 8 input carry look-ahead circuit are used. If the carry in to the subtraction operation is connected to the first carry generate as shown in the Figure 3.5(b) a subtraction can be done in only one more cycle than addition. The carry in to the operation (labeled Add or Subtract) is a 0 if the operation is addition and a 1 if the operation is a subtraction. A subtraction is performed by loading the R3 register with the 1's complement of the subtrahend and then adding. The carry in results in the 2's complement operation being completed as the addition is performed. With this hardware a 64 bit subtraction requires only 3 cycles, which is a significant saving

over the 17 cycles of the ripple carry adder.

The multiple precision fixed point multiplication algorithm for an arithmetic unit composed of a linear array of N stages demonstrates the usefulness of the stage's Q register. To provide a high speed multiplication the fastest possible addition operation is required so it is assumed that the carry look-ahead adder approach is implemented. In addition to carry look-ahead it will be necessary to have a "shiftable" mask register at the word level. This shiftable mask should provide a one stage shift of the stage level mask in a single cycle. The shiftable mask is used to reformat the multiplier from a byte parallel format to a byte serial format where each stage of the word contains the enitre multiplier in its Q register.

The multiplication algorithm works as follows. The multiplier and multiplicand are read from memory and loaded into the R3 and R0 registers. Next the multiplier is broadcast and reformatted. The product is then determined by computing the partial products and accumulating them with fast additions. The Q register of the stage is used to store the product as it is accumulated. After the multiplication step is completed the low order N bytes of the product, located in the Q register of stage 0, are distributed across the stages of the word and the result is stored.

The broadcast operation is done by N circular routes right of the R3 register. Each route has a distance of one stage, and on each route the contents of the R3 register is stored in the Q register. A reformatting step is needed after the broadcast because the Q register of each stage i contains the multiplier in a format that is "rotated" by a distance i (e.g., Q_2 contains $b_1\ b_0\ b_3\ b_2$ instead of $b_3\ b_2\ b_1\ b_0$). The reformatting is done in a total of N masked pop and push operations on the Q register. The mask used is initially all 1's. On each step of the reformatting the mask is shifted one stage left and the leftmost stage level mask receives a 0. This results in each stage i containing byte b_0 of the multiplier in its R2 register ready for the first partial product.

The multiplication step is performed by alternating the generation of partial products (using a single precision multiplication and a multiple precision addition), and accumulting these partial products into the product (held on the Q register). The final distribution step is needed because the low order N bytes of the $2N$ byte product will end up stored on the Q register of stage 0. At the end of the algorithm the stages contain the multiplicand in register R0, the high order bytes of the product in register R1, and the low order bytes of the product in the R3 register. The multiplier (originaly in R2) is destroyed during the multiplication.

The broadcast, reformatting, and redistribution steps of the algorithm each require N cycles. The multiplication step includes two multiple precision additions, one single precision multiplication, and a number of shift and queue operations on each iteration. A number of the operations in each iteration of the multiplication step can be performed in parallel so each iteration requires only 7 cycles. Thus, the total multiplication step requires $7N$ cycles. This gives a total multiplication time of $10N$ cycles. This estimate is quite pessimistic for an arithmetic unit where stage length shifts can be performed in one step. In this case the cycle time for the routing and register transfer operations (which account for $5N$ cycles) is being overestimated. After the design of the stage and routing logic is finalized more exact estimates of the multiplication time will be possible.

Figure 3.6 shows a possible design for a high speed multiplication arithmetic unit. The unit is constructed from N full stages (the boxes labeled *) linked into a linear array. These stages compute the partial products using the single precision multiplication of the stage. The multiplicand is stored in a register across the stages 1 byte to a stage. The multiplier is stored in a byte wide shift register that supplies each byte of the multiplier as it is needed for the generation of the partial products. The addition of the single precision partial products to produce a multiple precision partial product, and the accumulation of the product is performed by a set of sub-stages. These sub-stages are connected to the multiplier stages via their buses. The output bus of a multiplier stage is connected to the adder sub-stage directly below it in the

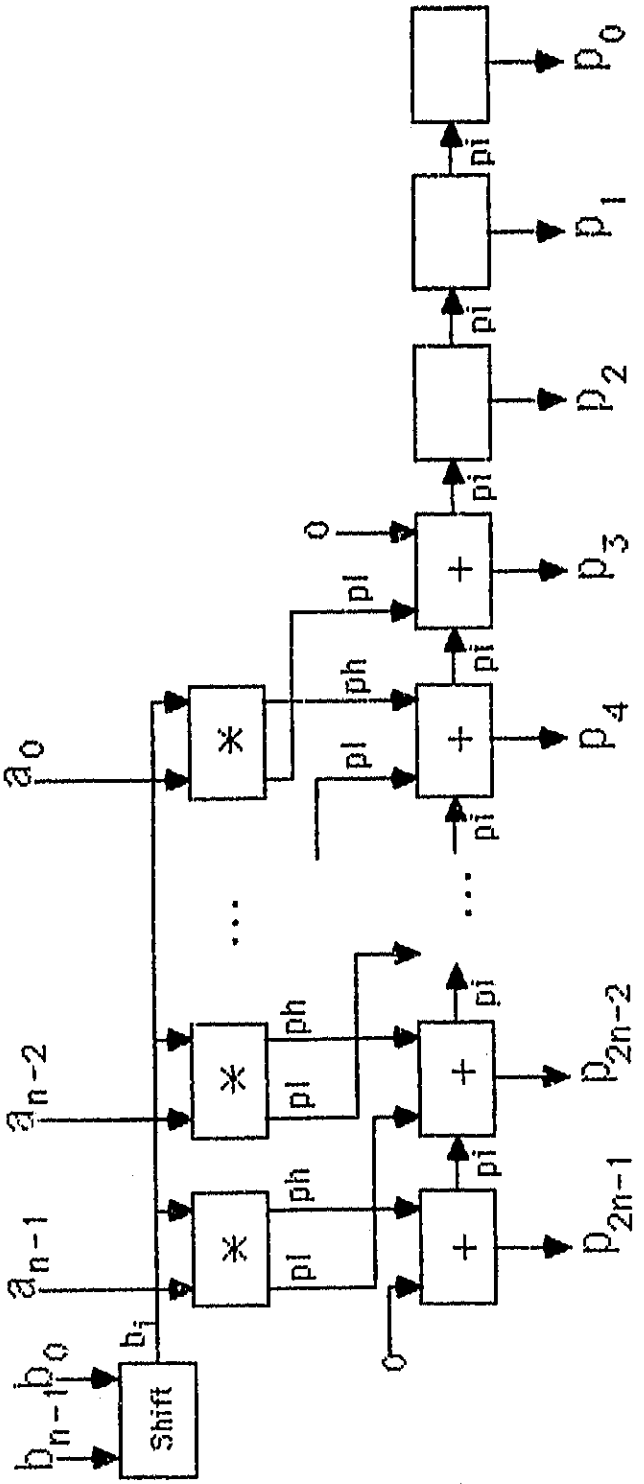


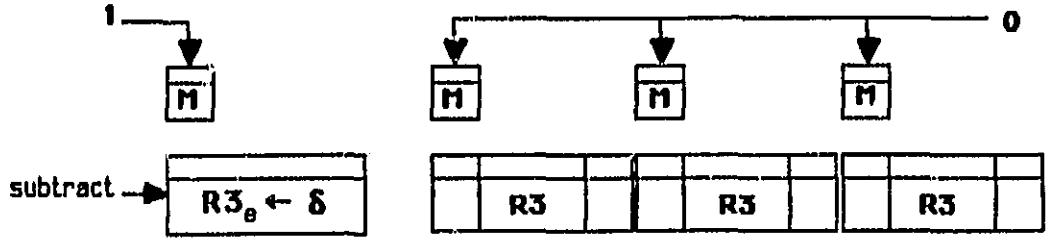
Figure 3.6: High Speed Multiple Precision Multiplication Unit.

figure. As the unit is designed none of the stages would have any individual memory (other than the add and multiply ROM's). The multiplicand (A) would be input directly onto the A bus of the multplier stage and the multiplier bytes (B_i) would be broadcast directly to the B bus of the multiplier stage.

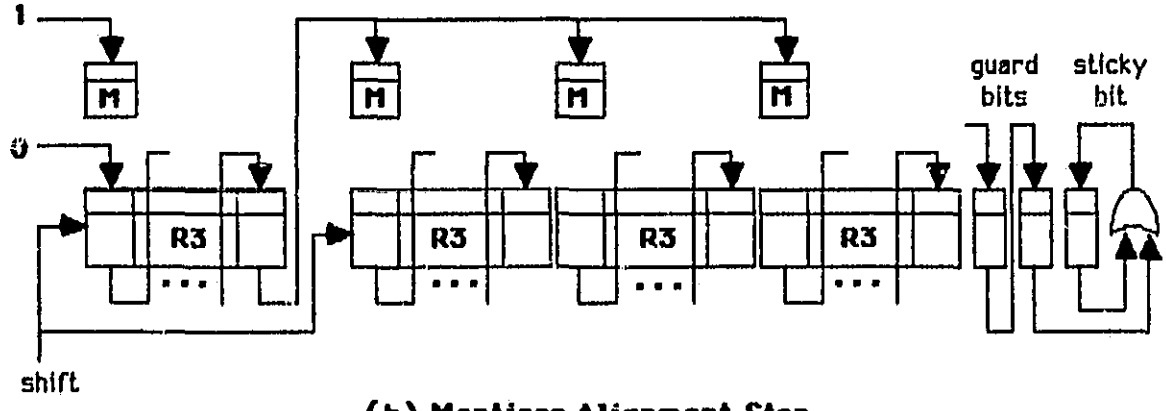
The connection between the multiplier and adder stages is a single byte connection between the O bus of the multipler stage and one of the buses of the adder stage. The positional shift of the low order byte of the partial product (p) in the figure) is performed by transmitting the low order byte to the R3 register of the adder stage and then doing a 1 byte shift to the right while the high order byte is loaded to the adder stage. The remianing special box in the figure is the byte shifter. This shifter could be formed from a set of shift registers similar to the R3 register of the stage. The output of the shifter is used as the input to the broadcast lines.

The multiplier shown in Figure 3.6 can overlap the accumulation of the partial products with the generation of the next partial product. It also has no need for the broadcast, reformatting, and dequeueing steps of the previous multiplier design. The limiting factors in this design are the single byte connection between the multiplier and adder units, and the speed of the multiple precision addition. The algorithm for multiplication on this design performs the addition of the partial product in parallel with the combination of the broadcast and multiplication operations. The output of the high and low bytes of the partial product to the adders and the addition to produce a partial product from them are done sequentially with the first parallel step. Each iteration of the multiplication step requires 7 cycles so the total multiplication speed for the design is $7N$ where $2N$ of the cycles are register transfer and shift operations.

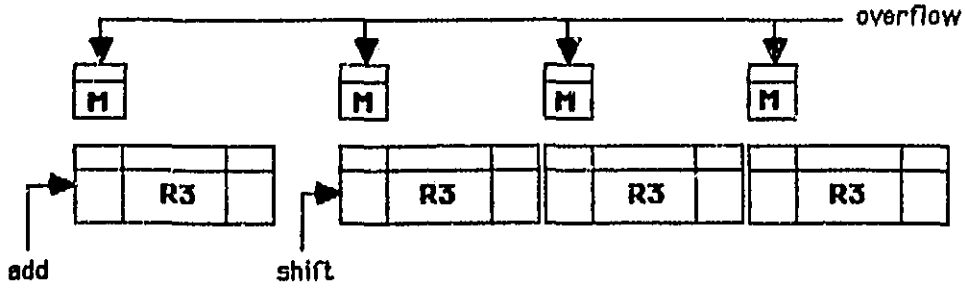
Floating point multiplication is a simple extension of the fixed point operations of multiplicaton and addition applied to the mantissa and exponent of the floating point number. The operation of floating point addition, however, requires additional hardware at the stage level.



(a) Exponent Comparison Step.



(b) Mantissa Alignment Step.



(c) Renormalization Step.

Figure 3.7: Configurations for 32 Bit Floating Point Addition.

Figure 3.7 shows the control configurations for an arithmetic unit which is composed of a linear array of stages during a floating point addition operation. The general algorithm for addition of normalized floating point numbers is to compare the exponents, align the mantissas, add the mantissas, and renormalize the result.

Figure 3.7(a) shows the control configuration of the arithmetic unit during the exponent comparison step. The exponents of the operands (from any register but R3) are subtracted and the difference δ is placed in the **R3** register. The mantissa of the larger number is then placed in the **R3** register for alignment. The control configuration for this operation would have the inverse of the mask of the subtraction step.

Figure 3.7(b) shows how the mantissa alignment is performed. The set bits at position i in the δ from the exponent comparison correspond to an alignment shift of a distance 2^i. With a full power of two network the the alignment shift can be accomplished with the configuration shown in the figure. The bits shifted off the exponent are used to mask the shift of the mantissa. The exponent is shifted right a distance of 1 bit on each cycle and the mantissa is shifted right a distance of 2^i on each cycle. The stage level zero detects can be used to stop the shift operation. In any event the alignment operation can be stopped after the 6th bit of the δ (for a 64 bit word) has been shifted off the exponent because all significant bits will be shifted off of the mantissa at this point. If a full power of two network is not available the control unit will have to determin how many stage length and 1 bit shifts are to be done. It is worthy to note that the shift preserves the sign of the mantissa and that the floating point format provides a "sticky bit" and guard bits.

The mantissa addition step is identical to the addition of fixed point numbers, except that the exponent stages are masked out of the operation. The result of the addition is placed back in the **R3** register so it can be shifted in the renormalization step. At most a single bit renormalization shift will be needed. The words that require this step are those that produced a

carry out during the mantissa addition. Therefore, the overflow can be used to povide a mask for this operation as shown in the figure.

The speed of the operation depends on the power of the shift network used in the alignment step. A full power of two network is probably too expensive for word lengths of greater than 8 bits. If such a network existed, however, the shift would take only $\lceil \log_2(\delta) \rceil$ time. For mantissas between 32 and 64 bits the maximum shift would require only 6 cycles with this network. With an abbreviated power of two network that has a maximum shift of 8 bits the time required for the mantissa allignment would be $\lceil \log_2(\delta) \rceil$ for $\delta \leq 8$ and $4 + \lfloor (\delta/8) - 1 \rfloor$ for $\delta > 8$. For most aligments the abbreviated power of two network will be sufficient to do the entire shift in $O(\log_2(\delta))$ time. Using an estimate of 3 cycles for the alignment the total time for the floating point add is 12 cycles. The initial exponent comparison requires 3 cycles (for all reasonable exponent lengths), two cycles are required to move the mantissas into position for the alignment, three cycles are needed to align the mantissas, and two more cycles are needed for both the mantissa addition and the renormalization.

Table 3.3 summarizes the execution times of the operations discussed in this chapter. From the table it can be seen that the addition and subtraction times of the stage based arithmetic units are very good. The values given in the table for the floating point addition are average times based on an assumption of a three cycle mantisse alignment. It should also be noted that the values in the table are for register to register operations. If the operands are to be read from memory and the results stored in memory an additional 2 cycles are required for each operation.

As mentioned earlier in this chapter a number of areas in the design of the bit processor, stage, and arithmetic units are topics of the proposed research. The implementation of the L buffer will be determined. An optimization pass will be done on the hardware of the BP and stage presented here inlcuding such areas as number of registers, queue register length, and control structures. The best form of affordable routing logic for processing multiple precision data will

be determined. Division and square root operations will be specified for both the single precision stage and the multiple precison formats of the RELAPSE system. Finally the usefulness of a concurrent multiplication and addition in the stage and a shiftable word level mask will be investigated.

Summary of Multiple Precision Execution Times.			
Operation	Hardware	Word Length (bits/stages)	Excecution time (cycles)
Fixed Point Addition	Ripple Carry Adder	64/8 128/16	8 16
	Carry Look-ahead Adder	64/8 128/16	2 3
Fixed Point Subtraction	Ripple Carry Adder	64/8 128/16	17 33
	Carry Look-ahead Adder	64/8 128/16	3 4
Fixed Point Multiplication	Stage Array	64/8 128/16	80 160
	Stage and Sub-stage Array with Braodcast	64/8 128/16	56 112
Floating Point* Addition	Stage Array	64/8 128/16	12 13
* The execution time listed is for a 3 cycle alignment shift.			

Table 3.3: Summary of Multiple Precision Execution Times.

CHAPTER 4

FUNCTIONAL UNITS COMPOSED OF BP'S AND STAGES

As stated in Chapter 2 the choice of the linear algebra problem domain was made for three reasons. First, problems from this domain are encountered in many physical and mathmatical applications. Second, the solutions to problems in this domain can be decomposed into computation tasks that are related to each other in a functional manner. Third, there is an extensive body of algorithmic design to draw upon in determining the functional components of the RELAPSE system. It is the intention of the proposed research to select a consistent set of functional units for system evaluation. The initial set of functional units will contain a subset of functional units that provide the same function through different algorithms and a subset of functions that will allow the system to choose the best algorithm for the problem at hand.

4.1 The Inner Product Functional Unit.

The inner product unit was chosen as an initial design study in building functional units from the VLSI components introduced in the last chapter. The unit is a valuable sub-assembly of many other functional units two of which are discussed below. The problem to be solved is stated formally as follows. Compute $\mathbf{Y} = \mathbf{A} \cdot \mathbf{B}$ where $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{Y}$ are vectors of dimension N. It can be shown that the solution to this problem requires at least $O(\log_2(N))$ time with computational units that have two inputs.

Figure 4.1 shows a functional unit that acheives this optimal performance. The unit is constructed of a linear array of multiplier units and a binary tree of adder units. The multiplier and adder units can be any of the designs described in Chapter 3. The multiplication of the pairs of vector elements is performed in parallel requiring one arithmetic unit cycle. The

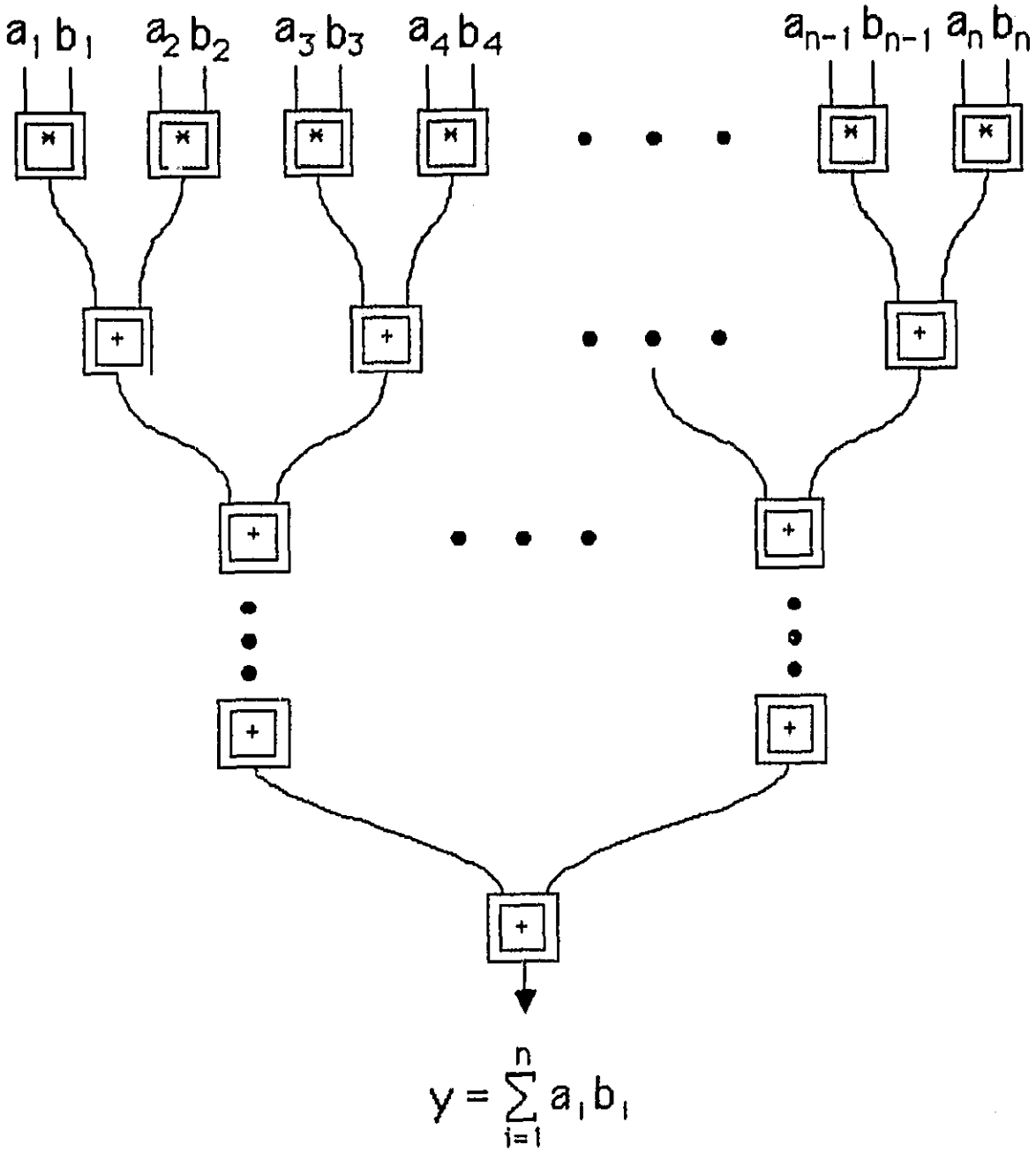


Figure 4.1: Inner Product Functional Unit.

products are then summed using the adder tree. The adder tree has a height of $\lceil \log_2(N) \rceil$ and requires $\lceil \log_2(N) \rceil$ steps to form the sum. Therefore the inner product of two vectors of length N can be calculated in $\lceil \log_2(N) \rceil + 1$ multiplication cycles. The functional unit requires N multiplier arithmetic units and N adder arithmetic units. It is important to note that the funcitonal units could also be constructed out of sub-steges to reduce the hardware costs. The adder units do not require a multiply ROM and the mutiplier units do not require the add ROM. Also the various arithmetic units in the figure do not require any RAM memory. This leaves the input and output buses availble for use in connecting up the adder tree. The initial input is loaded directly onto the input buses of the mutliplication units.

4.2 Matrix Vector and Matrix Matrix Multiply Units.

The inner product tree unit can be used to form a pipeline of inner product calculations where a new inner product problem can be started on each multiplication step. This capability can be used directly to create a Matrix Vector Multiplier functional unit. The N inner product calculations required for the multipication of an $N \times N$ matrix and an N-vector are simply run through the inner product calculator in a pipelined manner. The first result will be available in $\lceil \log_2(N) \rceil + 1$ multiply times and the remaining $N-1$ results will follow one per multiplication cycle. Thus the total time to perform a matrix vector multiply with a pipelined inner product calculator is $N + \lceil \log_2(N) \rceil$ multiplication cycles. The total amount of hardware is obviously the same $2N$ arithmetic units as before. If N such inner product calculators are available the multiplication of two $N \times N$ matrices can also be done in the same amount of time.

It is interesting to contrast these results with the results acheived using the systolic array design approach. The systolic array designs for the matrix vector and matrix matrix multiply calculations are based on an inner product cell [4]. Each inner product cell performs a multiplication and addition each time the array is cycled. Thus, for a comparable word format

the inner product cells are of comparable complexity to the arithmetic units of the inner product tree. The total number of cells needed in the systolic array is dependant on the bandwidth of the matrix they are processing. The inner product tree design above is primarily for random martices in that it contains no optimizations to operate on a banded matrix. In order to provide a valid comparison the matrices to be processed will be assumed to be random. These matrices, therefore, have the maximum bandwidth of $2N$. With this asumption the systolic arrays and the inner product based design bith have the same bandwidth to the outside world.

The systolic matrix vector calculator is linear array of $2N$ inner product cells. It can can form the product of an $N \times N$ matrix and N-vector in $4N$ cycles. Because only half of the cells are active on each cycle the array can be used in a pipelined manner to perform two multiplications in the same $4N$ cycles. The inner product tree used as a matrix vector calculator also contains $2N$ arithmetic units. It can calculate the matrix vector product in $N + \lceil \log_2(N) \rceil$ cycles. This is asymtotically better than the systolic array's performance, even when the systolic array is operated as a pipelined unit.

The systolic matrix matrix calculator is a hexagonally connected array of $4N^2$ inner product cells. It can compute the product of two $N \times N$ matrices in $5N$ cycles. Like the matrix vector calculator, it can be pipelined to calculate 3 matrix matrix products in the same $5N$ cycles. The inner product based matrix matrix multiplier uses N inner product trees for a total of $2N^2$ arithmetic units. It can compute the product of two $N \times N$ matrices in $N + \lceil \log_2(N) \rceil$ cycles. This result is better than the pipelined systolic array. Perhaps the biggest advantage is that it requires only roughly half the hardware.

To make a fully valid comparison between the two types of processors a number of other factors would have to be considered. The complexity of the two basic calculating units would have to be compared. The usefulness of the units in other porblems would also be important. The relative execution speeds would also have to be compared. Without weighing these factors in the comparison the simple comparison of number of execution cycles is somewhat suspect.

4.3 Remaining Work.

The topics of research in the lower levels of the design have already been described in their respective sections. In addition to those topics a consistent set of functional units will be chosen from the following list of linear algebra functions. For each functional unit the performance will be estimated and a communication protocol will be established.

- Elimination step with both partial pivoting and full pivoting.
- Iteration step using the Gauss Seidel, Jacobi, and SOR algorithms.
- L_2 and ∞ norm computation.
- Eigenvalues of matrices using the power method and inverse iteration.
- Deflation step unit.
- Units for storing and inverting tri-diagonal matrices.
- Units for storing and inverting random sparse matrices.

A RELAPSE machine will be designed that contains the functional units selected. The system level communication protocols and control sequencing of the various independent functional units in the machine will be specified. The initial system design will then be evaluated against the background of general purpose and systolic array systems using the ASW simulator. It is hoped that the results will provide insight into the restricted optimization problem posed in Section 1.1.

REFERENCES

[1] D. L. Slotnick, "Time-Constrained Computation," Department of Computer Science, University of Illinois at Urbana-Champaign, Report No. UIUCDCS-R-82-1090, May 1982.

[2] R. K. Richards, "Arithmetic Operations in Digital Computers," van Nostrad, 1955.

[3] D. L. Slotnick and John Cocke, "The Use of Parallelism in Numerical Calculations," IBM Research Memorandum No. RC-55, 1958.

[4] H. T. Kung and Charles E. Leiserson, "Systolic Arrays For (VLSI)," Department of Computer Science, Carnegie-Mellon University, Pittsburg, Pa. 15213, C 1978.