N O T I C E

THIS DOCUMENT HAS BEEN REPRODUCED FROM
MICROFICHE. ALTHOUGH IT IS RECOGNIZED THAT
CERTAIN PORTIONS ARE ILLEGIBLE, IT IS BEING RELEASED
IN THE INTEREST OF MAKING AVAILABLE AS MUCH
INFORMATION AS POSSIBLE

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

URBANA, ILLINOIS 61808

TITLE

Research in the Design of High-Performance

Reconfigurable Systems

Fourth

Semiannual Status Report

April 1, 1985 -- September 30, 1985
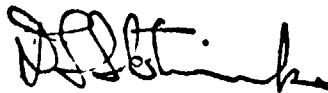
NASA Grant # NAG 5-377

Project Personnel

Graduate Research Assistant

Scott D. McEwan

Gregory J. Smith

Andrew J. Spry

Principal Investigator

D. L. Slotnick

Table of Contents

1. Introduction and Summary.

The initial control and programming philosophies of the RELAPSE are discussed in Section 2. A block diagram showing the relationship of the Arithmetic Units (composed of Stages and Bit Processors), to the Functional Units, and other components of the RELAPSE is used to guide this discussion. The latest version of the Bit Processor design is presented in Section 3. Included in this section is a detailed discussion of the Bit Processor's new scratch pad memory component. The section also clarifies the usage of the Bit Processor's processing registers, and Input/Output functions. The final design phase of the Arithmetic Unit is underway by a study of the Proposed IEEE Floating Point Standard. The decisions on conformation to this standard will be used as inputs into the finalization of the designs of the Bit Processor, Stage, and Arithmetic Units of the RELAPSE.

Section 4 discusses an update to the previous Semi-Annual Status Report. It details the failure of the method by Richards to do multiple input addition. Section 5 of the report deals with two practical multiple input adders and compares them to a two input uniprocessor in both time and cost constraints.

An appendix containing the detailed documentation of the MPP Simulator Programmer's Documentation concludes the report. The more general simulation tools of the ASW simulator from which the MPP Simulator is derived will be used to simulate the Functional Units of the RELAPSE system. The construction of simulations of the RELAPSE will be done using the functions of the ASW described in the documentation.

2. Control Levels of the RELAPSE.

Figure 2.1 shows a block diagram of the functional decomposition of the RELAPSE computer system. The diagram will be used to detail the current thinking and areas of study in the control structure and programming of the RELAPSE system. The figure shows three levels of control: the system level, the functional unit level, and the arithmetic unit level. An earlier stage level control has been determined unnecessary and has been dropped to simplify the overall design.

The system control level is responsible for arbitrating communications and data flow between functional units, scheduling input and output, and scheduling usage of the shared memory resources. Although the system controller is shown as a centralized control in Figure 2.1 it will in practice be a distributed control. The functional unit control level is responsible for communication within the functional units. This includes the sequencing and synchronization of operations on the arithmetic units and on the usage of any internal functional unit buses. The arithmetic unit control level is responsible for controlling and sequencing the micro-operations of the stages, routing logic, and bit processors to perform the basic arithmetic of the RELAPSE system.

The existence of three distinct levels of control implies that a single language implementation would be impractical for the RELAPSE system. Such a single language implementation would have to provide the ability to program the system from the level of the single bit operations of the stages and bit processors up to the high level linear algebra oriented user interface. A three "language type" system under consideration would parallel the control level structure of the RELAPSE. This language structure will allow the individual languages of the RELAPSE to be more easily tailored to specific tasks and provide control level dependent features and optimizations that would be impractical in a single language system.
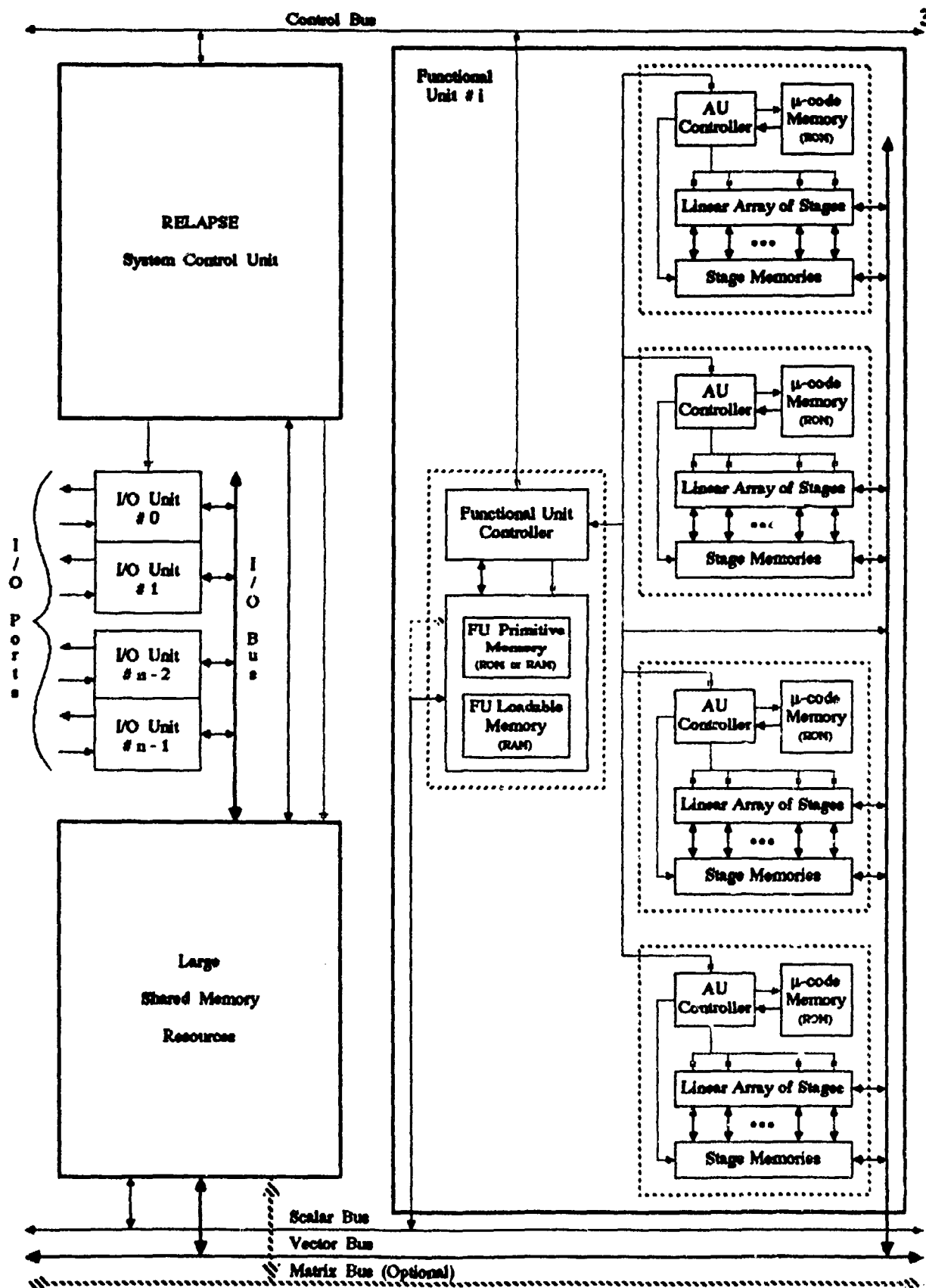
Figure 2.1: Control Structure of the RELAPSE.

The arithmetic unit language type is a microcode language that will allow the expression of the full capabilities of both the horizontal mode operation of stages and the vertical mode operation of bit processors. This language provides the basic arithmetic operations of the arithmetic units and the capability to customize this basic arithmetic. Arithmetic unit level programs consist of ROM based micro code. The customizations provided will allow multiple word formats, numerous roundoff techniques, and access to important sub-arithmetic operations such as logical operations, shifting operations, routing operations, and operation on sub word formats such as floating point mantissas and exponents. The set of customizations will allow the programming of arithmetic operations on new and nonstandard word formats to be done from the functional unit level.

The functional unit language type is a high level assembly language that must provide the convenient expression of scalar, vector, and matrix operations as well as the inter-arithmetic unit communication within the functional unit. The language will provide primitives for synchronization and be extensible to different functional unit architectures. As shown in Figure 2.1, the functional unit control memory is divided into two segments: a primitive memory, and a loadable memory. The primitive memory is used to store the machine code for the basic operations of the functional unit such as bus control protocols and basic vector or matrix arithmetic operations. This control memory defines the basic functionality of the unit and will be either stored in ROM or loaded at system boot time. The second segment of the memory is a loadable memory. This segment allows the extension of a basic functional unit to some specific task. In a RELAPSE system there may be a number of functional units that have the same basic underlying architecture and primitives. It is the loadable memory that will hold the algorithms that distinguish these otherwise identical

functional units. This memory can also be used to reconfigure a single functional unit to do different processing tasks for different jobs. This memory will usually be loaded at the start of some processing job.

The system level language type will provide synchronization and control flow primitives that allow the full use of the multiprocessing capabilities of the functional units. It will also allow the development of a mathematically oriented user interface that will contain as its heart a linear algebra oriented programming language for the RELAPSE system. It is this user interface and linear algebra programming language that will provide the Linear Algebra processing system of the RELAPSE.

Programming the RELAPSE system will require the generation of programs for two of the three control levels of the system. For most applications a new system level program will be required in the linear algebra programming language. A number of system wide parameters (such as round off algorithm) will be set via the user interface. Modification of these parameters will be allowed either during processing or between jobs without recompiling the application. For most applications the built in programs of the functional units will be used without modification. If some function required by an algorithm is not provided by a functional unit a new algorithm may be written at the functional unit level.

3. Latest Bit Processor.
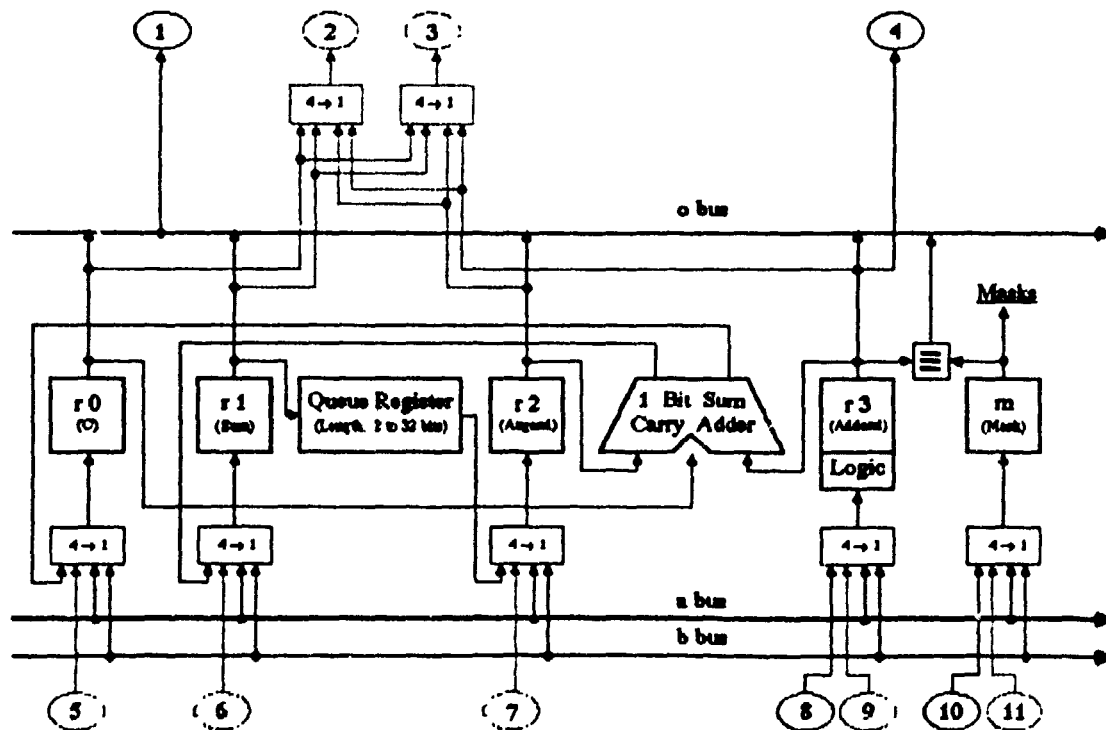
3.1 Bit Processor Layout.

The updated bit processor layout is shown in Figure 3.1. The major changes form the layout presented in the third Semi-Annual Status Report is the reduction of the number of processing registers by 1, the introduction of a scratch pad register memory, and a clarification of register inputs. The scratch pad memory is discussed in detail in the next section. The remainder of the changes are summarized below.

The general purpose registers of the bit processor have been designated as processing registers. It is assumed that the contents of these registers will be destroyed and modified by any horizontal or vertical mode algorithm. It is for this reason that the scratch pad memory was added to the bit processor design as a location for temporary values. The use of the processing register inputs and outputs have been clarified. The specific uses are given in Table 3.1. The input lines to these registers have been divided up as follows: line 1 is for vertical mode arithmetic, line 2 is for horizontal mode arithmetic, line 3 is for input from the a bus, and line 4 is for input from the b bus. Any register (except the mask register) can be placed on the o bus. The remaining output uses of the registers are specific to vertical and horizontal mode arithmetic and are explained in Table 3.1. and Table 3.2.

The remaining clarification deal with the input and output from the bit processor itself. The bit processor has four input/output ports. Two of these ports are devoted to the two bank memory. One port is devoted to I/O directly with the processing section of the BP. This port makes the registers of the BP directly accessible from outside the BP. The final I/O port is devoted to I/O with the memory of the BP's. This port provides an I/O path to the memories without using any BP processing registers.
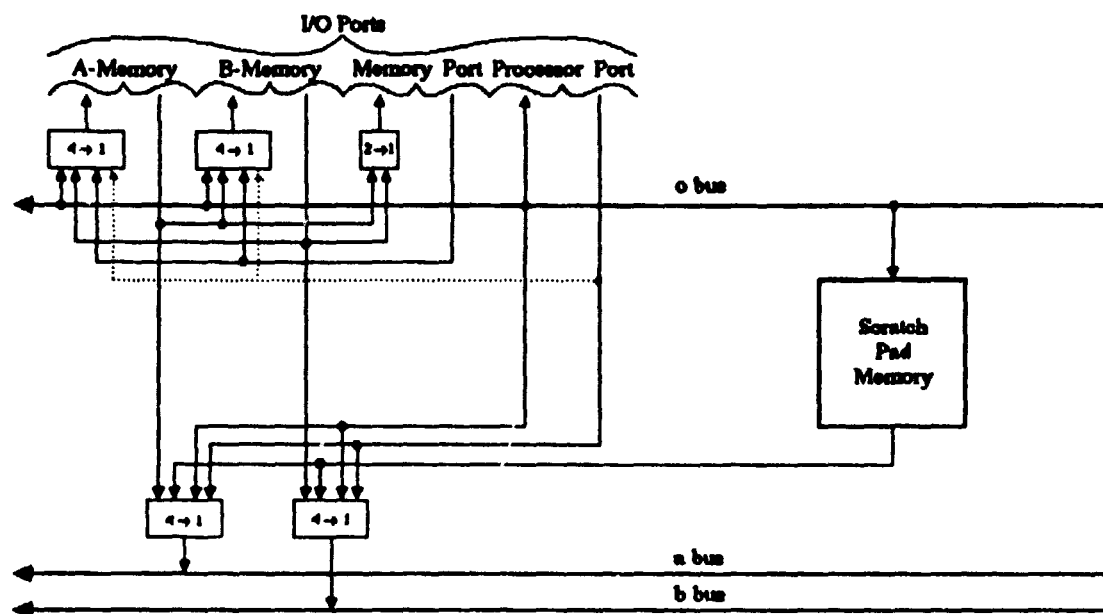
(a) BP Data Connections.

(b) BP Processor Section.

Figure 3.1: Bit Processor Layout.

Data Source and Destination of the BP Processing Registers.		
Reg.	Sources of Input	Destinations of Output
r0	The carry bit from the sum carry adder, one bit of the sum from the add ROM, the a bus, and the b bus.	The o bus, one bit of the add and multiply ROM address, and 1 bit of the sum carry adder input.
r1	The sum bit from the sum carry adder, one bit of the low order byte of the product, the a bus, and the b bus.	The o bus, one bit of the add and multiply ROM address, and the q register input.
r2	The q register, one bit of the high order byte of the product, the a bus, and the b bus.	The o bus, one bit of the add and multiply ROM address, and 1 bit of the sum carry adder input.
r3	The routing logic, the a bus, and the b bus.	The o bus, one bit of the add and multiply ROM address, the equivalence function, the routing logic, and 1 bit of the sum carry adder input.
m	The bit and stage level masks, the a bus, and the b bus.	The equivalence funciton, and the bit processor masks.

Table 3.1: Source and Destination of Processing Register Data.

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

Table 3.2: Input and Output Points of the BP Shown in Figure 3.1.
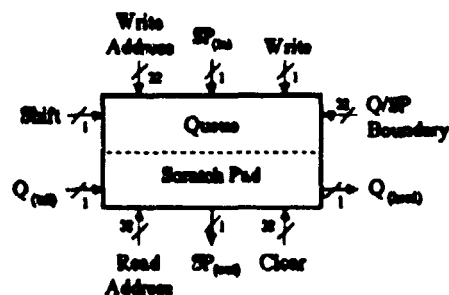
3.2   Queue Register and Scratch Pad Memory.

The Queue and Scratch Pad Memory (Q/SP), shown in Figure 3.2, is a component of the Bit Processor. It is a combination of a fixed length queue register and a scratch pad memory. The proposed total size of the queue and scratch pad portions of the unit is 32 bits. The boundary between the two portions is software reconfigurable by setting the length of the queue portion. The queue's length is decoded from a 5 bit control value supplied by the control unit.

The queue portion of the unit receives its inputs from the BP's processing register r1 and sends its output to the BP's processing register r2 (Figure 3.2(c)). On each shift cycle of the queue the input is placed in the queue's tail bit and the queue's head bit is available as the output. The bit output can either be loaded into the r2 register or neglected (i.e.when the queue is being filled for the first time). Each internal bit of the queue receives the value of the preceding bit in the queue and makes its contents available for the succeeding bit in the queue during each shift cycle (Figure 3.2(a)). The tail of the queue is considered bit 0 and the head is considered bit l-1 where l is the length of the queue. Thus it requires N shift cycles of the queue to move a bit through a queue of length N.

The scratch pad portion of the unit receives its input from the BP's output bus (o-bus) and its output can be placed on either of the BP's input buses (a-bus or b-bus). Each bit of the scratch pad memory is individually addressable. The read and write addresses of the scratch pad memory are decoded from 5 bit values supplied by the external control unit. A write control signal is set on any cycle that will perform a write operation while the read operation can be done on any cycle. Read and write operations of the scratch pad memory can be performed simultaneously. If the same cell of the scratch pad is read from and written into on the same cycle the value read will be the previous contents of the cell and the new contents of the cell will be the
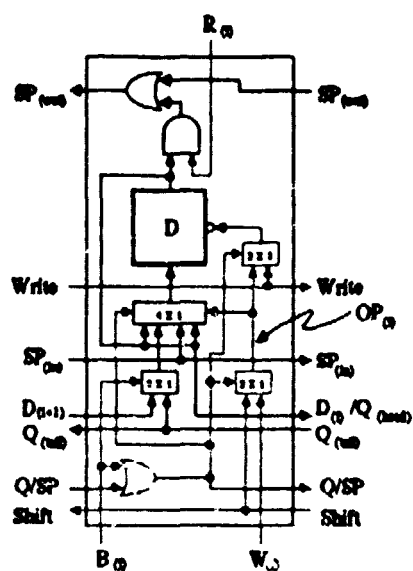
(a) Queue & Scratch Pad Cell.

(b) Queue & Scratch Pad Unit Connections.
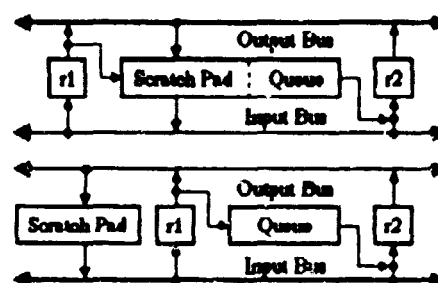
(c) Data Connections to the BP Buses.

(d) Queue & Scratch Pad Layout.

Figure 3.2: The Queue and Scratch Pad Memory Unit.

value written in that cycle. The scratch pad memory thus acts like a small dual ported random access memory.
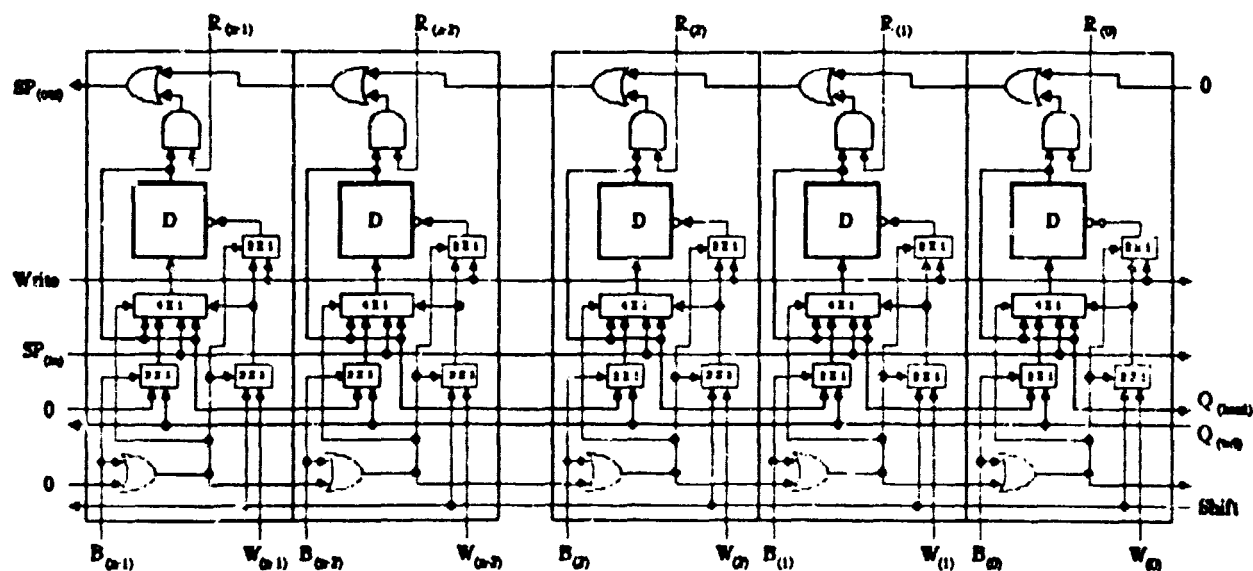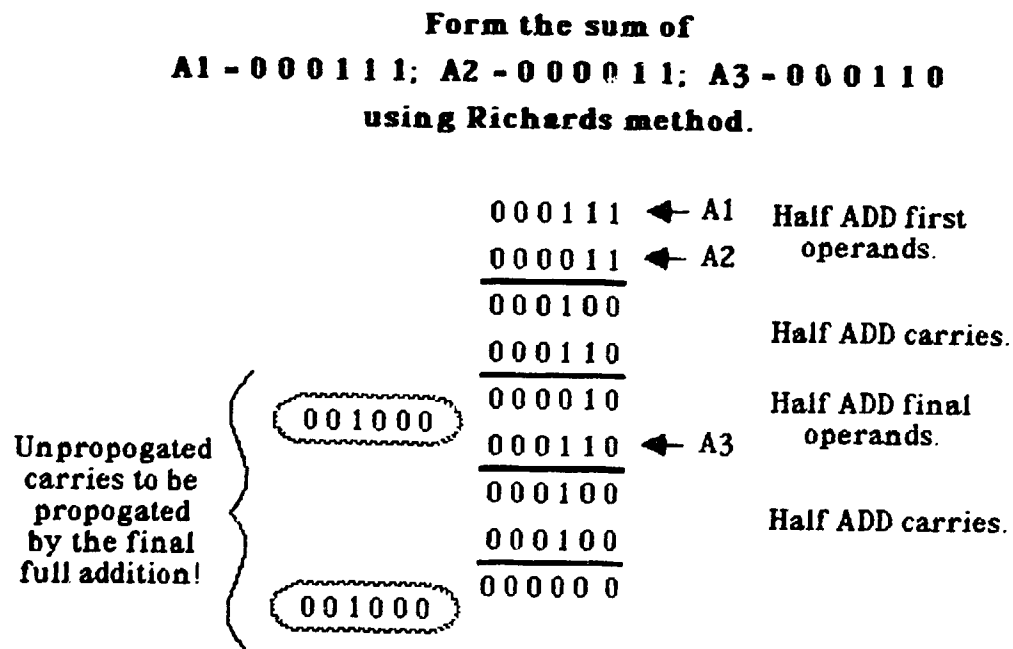
The software configurable boundary between the queue and scratch pad memory portions of the Q/SP unit provides hardware protection against writting into the queue via a scratch pad memory write. In the event of a scratch pad write operation to the portion of the Q/SP unit that is currently the queue the write operation will simply fail and the contents of that bit of the queue will remain unchanged (provided no queue operation also occurred on that cycle that would change the value). Figure 3.2(a) shows that no other hardware boundary checking done in the Q/SP itself. It is up to the controller to generate an interrupt signal a scratch pad write operation is in the legal address range. There is currently no hardware protection against reading from the queue portion of the Q/SP unit via a scratch pad read. This is not a design feature however and may not be retained in the final design of the Q/SP unit.

In summary the Q/SP unit of the Bit Processor (Figure 3.2(b)) is a 32 bit register that can function as a 1-32 bit "shift register" queue, as a 1-32 bit scratch pad memory, or as a 1-32 bit combination of both. The register has two data inputs and two data outputs (black arrows in the Figure), one pair for the queue and one pair for the scratch pad. Control lines (grey arrows in the Figure) are provided for designating the read address, the write address, setting the queue's length, performing a shift operation, performing a write operation, and asynchronously clearing the entire unit. To reduce the overall number of control lines to the Q/SP the 32 bit control signals (the read and write addresses, and the queue boundary) will be decoded from 5 bit input control signals.

4. Richards' Method Update.

The addition method by Richards (Page 3 and Figure 1.1 in the thesis proposal and SAR #3) does not work in general. Briefly, the method was proposed to allow the addition of more than two numbers with a single adder circuit using half adds, carry saves, and a final full addition to propagate any remaining carries. The method relied on the fact that the carries resulting from a half addition can be added to the "half sum" to generate an unpropagated carry. The unpropagated carries are then propagated during a final full addition step. The method does guarentee that two carries cannot occur in the same digit position on subsequent steps (half adds) but it does not guarentee that multiple carries into the same digit position will not occur on alternate steps. An example where the method fails is shown in Figure 4.1.

Form the sum of
A1 - 0 0 0 1 1 1; A2 - 0 0 0 0 1 1; A3 - 0 0 0 1 1 0
using Richards method.

```
                            0 0 0 1 1 1  ← A1    Half ADD first
                            0 0 0 0 1 1  ← A2      operands.
                            ‾‾‾‾‾‾‾‾‾‾‾
                            0 0 0 1 0 0
                                                 Half ADD carries.
                            0 0 0 1 1 0
           ⎧  ┌─────────┐   ‾‾‾‾‾‾‾‾‾‾‾
           ⎪  │ 0 0 1 0 0 0 │ 0 0 0 0 1 0        Half ADD final
Unpropogated⎨ └─────────┘                          operands.
carries to be            0 0 0 1 1 0  ← A3
propogated               ‾‾‾‾‾‾‾‾‾‾‾
by the final             0 0 0 1 0 0
full addition!⎪                                  Half ADD carries.
           ⎪             0 0 0 1 0 0
           ⎪  ┌─────────┐  ‾‾‾‾‾‾‾‾‾‾‾
           ⎩  │ 0 0 1 0 0 0 │ 0 0 0 0 0 0
              └─────────┘
```

Figure 4.1: Failure of Richards Method with Three Inputs.

In the example two unpropagated carries are generated in the 2^3 digit position. This implies that two carry propagate additions will be needed to propagate these carries and the method fails. In general on each alternating step (half add of carries) a new carry can be generated for each digit position so at most N-1 unpropagated carries can accumulate at any digit position when there are N input numbers. Therefore, the method will still need N-1 carry propagate additions in general to propagate the carries across the final "half sum."

5. Multiple Input Adders.

A number of different multiple input adders can be built using binary trees of two input adders. The individual adders in these trees can be bit serial or bit parallel and they can use any addition speedup techniques such as carry look ahead addition or ROM assisted addition. In order to distinguish the speedup provided by multiple input addition from the speedup provided by bit parallel versus bit serial addition, it is necessary to study how multiple input adders built from both bit serial and bit parallel adders compare in the solution of a suitable problem. One such suitable problem can be stated as follows: compute the fixed point sum of k numbers of word length l in a time less than some constant T and at a cost less than some constant D.

A k input adder can be constructed from a set of bit serial full adders with separate 1 bit carry registers. If the basic word format of the machine is an l bit parallel word, a parallel to serial conversion register (of length l) will be needed to convert each input operand, and a serial to parallel conversion register (also of length l) will be needed to convert the output. At each level in the addition tree a single 1 bit register for each adder will also be needed to store the bits of the "partial sum" for the next level. Since an overflow can result from any of the intermediate sums, the carry out of each of the bit serial adders must be considered when detecting overflow of the k input sum. An example of a such an adder tree for $k = 4$ inputs is shown in Figure 5.1.

A k input adder can also be constructed from a set of bit parallel adders (of word length l) where each adder has a set of input registers for its input operands. Thus at each level of the tree there are twice as many l bit registers as there are parallel adders. Since an overflow can result from any of the intermediate sums, the carry out of all the parallel adders must be considered when detecting overflow of the k input sum. Figure 5.2 shows a $k = 4$ input adder constructed from parallel adders. In order to achieve the fastest addition speed each parallel adder should use some
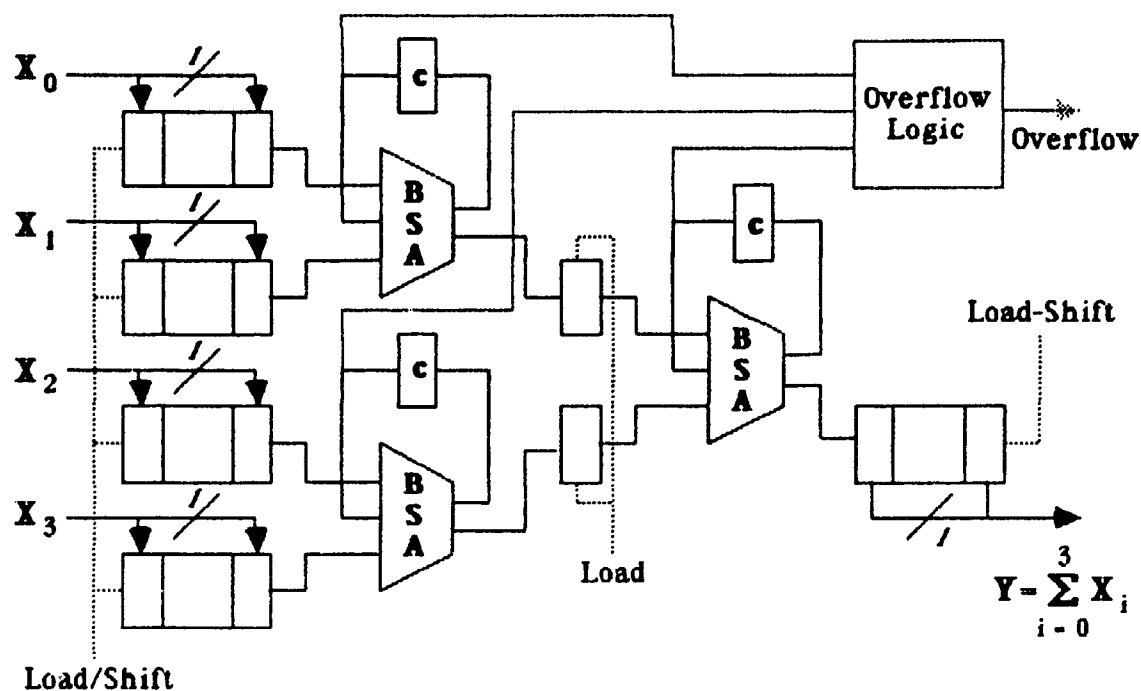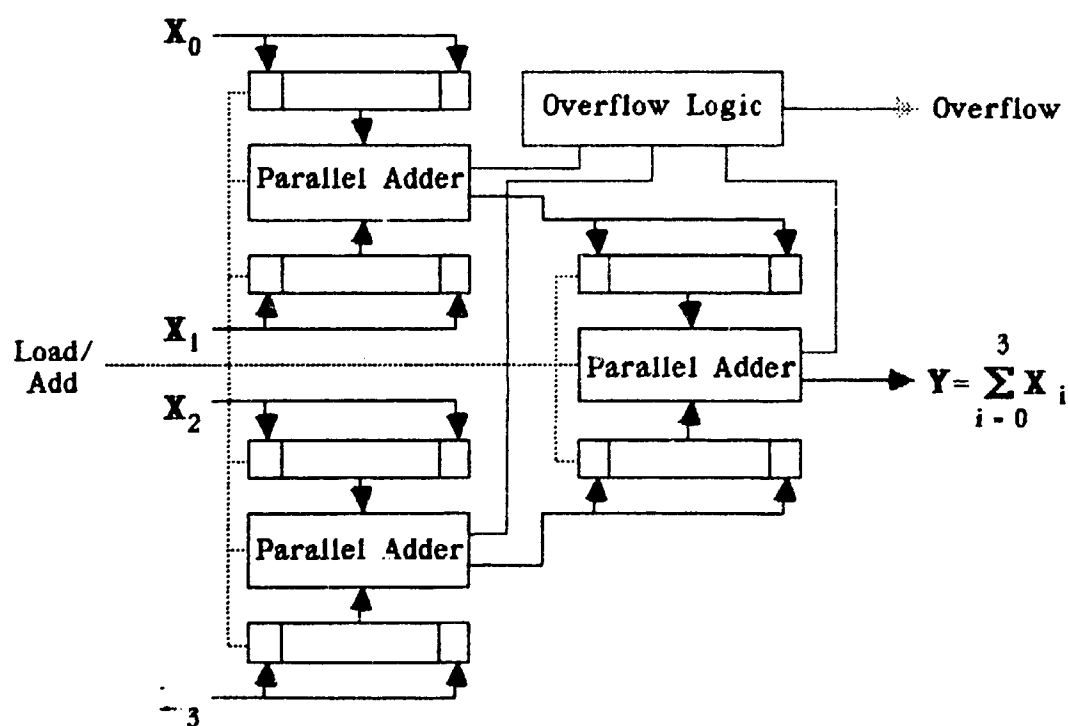
Figure 5.1: Four Input Bit Serial Adder Tree.

$$Y = \sum_{i-0}^{3} X_i$$

Figure 5.2: Four Input Parallel Adder Tree.

$$Y = \sum_{i-0}^{3} X_i$$

technique to reduce carry propagation delays such as carry look ahead addition.

The bit serial adder tree shown in Figure 5.1 can be thought of as a bit serial arithmetic pipeline containing $\lceil \log_2(k) \rceil$ steps. After the setup time of $\lceil \log_2(k) \rceil$ addition cycles the first bit of the k input sum is loaded into the output register. An additional $l - 1$ addition cycles are then required to determine the remainder of the k input sum. Therefore, the total time required to compute the k input sum using the bit serial adder tree (denoted $T_k(B)$) is

$$T_k(B) = l \, (\text{ADD A}_j) = (l - 1 + \lceil \log_2(k) \rceil) \times l \, (\text{FULL ADD})$$

The bit parallel adder tree shown in Figure 5.2 can also be thought of as an addition pipeline that has a setup time of $\lceil \log_2(k) \rceil$ steps. In the parallel adder tree pipeline, however, the final result is available after the setup time so the pipelining characteristic is only valuable if there are multiple k input additions to be performed. Using carry look ahead addition the time required by the bit parallel adder tree to solve the k input addition problem (denoted $T_k(L)$) is given by

$$T_k(L) = l \, (\text{ADD A}_j) = \lceil \log_2(k) \rceil \times (\, 2\lceil \log_2(k) \rceil + l \, (\text{FULL ADD}) \,)$$

The speedup of the bit parallel adder tree over the bit serial adder $S_k(L)$ is found by dividing the time required to solve the problem on the bit serial adder by the time required to solve the problem on the bit parallel adder. The speedup for the carry look ahead parallel adder tree is

$$S_k(L) = \frac{(l - 1 + \lceil \log_2(k) \rceil) \times l \, (\text{FULL ADD})}{\lceil \log_2(k) \rceil \times (\, 2\lceil \log_2(k) \rceil + l \, (\text{FULL ADD}) \,)}$$

Analyzing this equation shows that the bit serial adder tree is faster only for very short word lengths ($l < 16$) when there are many inputs to be added ($k > 128$). In all other cases the bit parallel adder tree will be faster than the bit serial adder tree. For example for a 16 input addition of 64 bit operands the parallel adder tree has a

speedup of 6.7 over the bit serial adder tree. The advantage of the bit parallel adder tree over the bit serial adder tree is directly proportional to the word length l and inversely proportional to the number of inputs k.

In order to determine the speedup of the multiple input adders over two input adders the time required to solve the k input addition problem on a two input adder must be known. Two different two input adders will be used as the base line for the speedup analysis: a carry propagate adder and a carry look ahead adder. The time required to do a single addition using a carry propagate adder is $l \times t$ (FULL ADD) because, in the worst case, the carry has to be propagated across the entire l bit word. The time required to do a single addition using a carry look ahead adder is $2 \lceil \log_4(l) \rceil + t$ (FULL ADD) assuming logic with a fan in of 4 is used. The 2 input adders solve the problem in an iterative manner so a total of $k - 1$ additions are needed to form the k input sum. Therefore, the time needed to solve the addition problem using the carry propagate adder (denoted $T_1(P)$) and the carry look ahead adder (denoted $T_1(L)$) are given by

$$T_1(P) = t\,(\text{ADD A}_j) = (k - 1) \times l \times t\ (\text{FULL ADD})$$

and

$$T_1(L) = t\,(\text{ADD A}_j) = (k - 1) \times (2 \lceil \log_4(l) \rceil + t\ (\text{FULL ADD}))$$

The 2 input carry look ahead adder will be used in both comparisons. The 2 input carry propagate adder will be used in the speedup comparison of the bit serial adder tree because that adder tree suffers from the carry propagation delay. In that adder tree the carry propagation is done in parallel by all the adders in the tree but still requires $O(l)$ cycles to propagate.

The speedup of the bit serial adder tree over the carry propagate adder (denoted $S_1(B)$), determined by dividing $T_1(P)$ by $T_k(B)$, is

$$S_1(B) = \frac{(k - 1) \times l}{(l - 1 + \lceil \log_2(k) \rceil)}$$

From this it is easy to see that for all word lengths l and two or more inputs ($k \geq 2$) this adder is faster than a single carry propagate adder. The speedup increases rapidly as both l and k incerase because the addition time of the bit serial adder tree depends on the sum of the word length and number of inputs rather than their product. The same general results are obtained by analyzing the speedup of the bit serial adder tree over a 2 input carry look ahead adder. The speedup function does not increase as rapidly as it does in the carry propagate adder case because the 2 input carry look ahead adder time depends on the $\log_4$ of the word length instead of linearly on it.

The speedup of the bit parallel adder tree over the carry look ahead adder (denoted $S_1(L)$), determined by dividing $T_1(L)$ by $T_k(L)$, is

$$S_1(L) = \frac{(k-1)}{\lceil \log_2(k) \rceil}$$

As with the bit serial adder tree the bit parallel adder tree is faster than a 2 input adder. When comparing the carry look ahead adders the speedup in only a function of the number of inputs. If the speedup comparison is done with a 2 input carry propagate adder the speedup will be greater than that shown by a multiplicative factor of the word length.

The final part of the problem statement deals with a cost analysis of the different adders. A detailed cost analysis would have to involve a determination of the hardware necessary to deliver all k operands to the k input adders, as well as a cost analysis of the adders themselves, and any conversion registers needed in the bit serial adder tree case. To get an order of magnitude estimate of the costs involved only the adders, their necessary associated registers, and data connections will be considered. As a basis for this comparison the cost of a single 1 bit adder, its associated data lines, a carry register, and two 1 bit input registers will be denoted

D(BIT ADDER). The 2 input carry propagate adder is composed of l of these bit adders except that only one carry register is needed. Thus the cost of the 2 input carry propagate adder $D_1(P)$ is $O(l \times D(\text{BIT ADDER}))$. The 2 input carry look ahead adder is more difficult to analyze because of the carry look ahead circuitry. If the number of gate delays is used as a comparison factor the carry look ahead circuitry for each bit position will be roughly as complex as the adder for reasonable word lengths and logic fan ins. The bit cost of a carry look ahead adder will therefore be about 1.5 times the bit cost of the carry propagate adder so $D_1(L) = O(l \times D(\text{BIT ADDER}))$. The bit serial adder tree contains $k - 1$ bit serial adders each with an associated 1 bit carry register, two 1 bit operand registers, and 1 bit data connections. Thus the cost of the k input bit serial adder tree $D_k(B)$ is $O(k \times D(\text{BIT ADDER}))$. The k input carry look ahead adder tree contains $k - 1$ carry look ahead adders which have a cost of $D_1(L)$ so the cost of the carry look ahead adder tree $D_k(L)$ is $O(k \times l \times D(\text{BIT ADDER}))$. Comparing these estimates it is easy to see that the k input bit serial adder tree costs $O(k / l)$ more than either of the 2 input adders and that the k input bit parallel adder tree costs $O(k)$ more than the 2 input adders. It can also be seen that the k input bit parallel adder is $O(l)$ times as expensive as the k input bit serial adder.

Appendix: MPP Simulator Programmer's Documentation.

1 *Introduction.*

The Architecture Simulation Workbench simulator of the MPP consists of an MPP emulator, which provides full functional emulation of the Main Control Unit, PE Control Unit and the Array Unit, along with a program debugger and a set of routines that control the execution of the simulator modules and provide communication between modules. This document describes the structure, modules and individual files making up the simulator.

Throughout this document the unix directory path conventions are used, where "dir/file" means the file named "file" in directory "dir". The directories used by the MPP simulator are:

mpp, which contains the main routines for the MPP emulator,

libsim, which contains subroutines used to simulate ARU functions,

debug, which contains the source for the debugger, and

libasw, which contains the routines used to simulate parallel operation of the simulator components, provide I/O and inter-module communication. These routines act like a virtual operating system for the simulation.

2 *Structure.*

The simulator is broken up into several modules, with a module being a distinct functional entity which operates in parallel with the other modules of the system. There is a module corresponding to each of the MCU, PCU, IOCU (currently a null program, since the code to use the IOCU is in place but an emulator has not yet been written) and the simulator debugger. Since the C programming language does not provide the capability of running multiple processes, a set of routines is provided to simulate parallel execution in the directory "libasw". Each module acts like a separate program, with calls to the routines in libasw for communication, I/O and synchronization. This section will describe the libasw routines and how the modules of the simulator interact with them.

2.1 *The Simulator Operating System – libasw.*

The directory libasw contains the routines which perform operating system functions for the simulator – creation of modules, communication between modules and I/O with the Vax file system and the user's terminal. Also included in this directory are routines for performing common simulator operations involving simulated MPP memories and register sets. These routines provide a common interface for all memory or register operations, as well as allowing the ASW debugger to transparently transfer data to and from the simulated memories.

The file "libasw/multi.c" contains the heart of the ASW operating system – the routines for simulating concurrent execution of the modules. Each module

has a corresponding module descriptor, a C structure defined in h/multi.h, which contains the module's name, its run time stack, the address to set the Vax stack pointer when the module is run, and information about the module's state. The routine "sp. exec" in libasw/multi.c defines a new module. Sp_exec creates a new module descriptor, allocates stack space, initializes the module's stack, and then places the descriptor in the system run queue. The bottom of the stack is set up so that if and when the main routine of the module finishes, the subroutine return will cause control to jump to the routine sp_die. This routine disposes of the module descriptor and returns control to the scheduler. The address of the main routine for the module is placed on the stack above sp_die, and the initial value of the module stack pointer is set to point to this address, so that the first time the module is scheduled the main routine is called.

The routine "scheduler" (in libasw/multi.c) schedules the execution of modules. Two queues are maintained for modules ready to be run and those waiting for some event. The scheduler first checks to see if there are any modules in the run queue. If there are, the first one is removed from the queue and restarted. If not, the wait queue is traversed to find any modules that are ready to be transferred to the run queue. A module is ready to run when its state matches the system state variable and it is not waiting for elapsed time. If there is no runnable module, the system clock is updated, all modules have their wait time counters updated and the wait queue is again traversed.

When a module is scheduled to run, a pointer to the module descriptor is put in the global variable "u" and the routine "sp_swap" is called. sp_swap alters the Vax stack frame so that the subroutine return address is substituted with the

address saved when the module was suspended. The Vax subroutine return then continues execution at this point.

The routines "sp_sleep" and "wait_for" are used to suspend execution of a module. Each of these routines updates the module descriptor to reflect the event that is to wake up the module (either elapsed time or a change in the system state), places the module on the wait queue and swaps in the scheduler. sp_sleep causes the module to sleep for a number of clock ticks. wait_for causes the module to sleep until a specified bit in the system state is set. A module can signal an event (set a bit in the state) via the routine "sp_ch". The bit can then be cleared by a module calling the routine 'sp_seen", to indicate that the event has been seen.

Each module of the simulator consists of a large loop in which an instruction is interpreted, and then the module calls "sp_sleep" to sleep for the number of clock ticks corresponding to the time used by the instruction. Modules are automatically suspended when they try to perform some operation that cannot be done at that time: for instance, when the PCU tries to take a command of the call queue, if the queue is empty, the queue routine suspends the PCU until a signal is received indicating something was put on the queue. (Since the signal only says that something was put on a queue, without specifying which queue, the routine actually has a loop to check if there is something on the proper queue, resuspend if not, and calls "sp_seen" when it does get something.)

The file "libasw/imc.c" contains routines for communication between modules. These routines are used to define and access intermodule resources. The definition of a resource comes from the "item[NITEM]" table. Currently it

delimits the classes "section of memory", "function", and bit plane of data. "I_creat" makes a resource in one module available to other modules. These resources can then be read or written to using the routines "I_read" and "I_write". A resource can be loaded from or written to a file via the routine "I_file". I_file uses the routines in "libasw/load.c" for loading and unloading resources.

The file "libasw/memory.c" contains routines for creating and using simulated computer memories or register sets. These routines allow all memory accesses to be done through a common interface, which checks for illegal addresses and performs the trace and breakpoint functions for the ASW debugger. The PCU, MCU and ARU memories and registers are all created and accessed with these routines. The routine "M_create" creates a memory space by placing a descriptor for the memory in a table. This descriptor contains the name of the memory (for example, MCU memory is called "mspace"; this name is for printing out messages, such as when a breakpoint is activated), the type (scalar or array), access permission, a flag for turning on the trace function, word length, the memory's size, an array of breakpoints which can be set by the debugger, and a pointer to chunk of VAX memory to be used as the simulated memory. M_create also calls "I_creat" to make the memory available to the debugger. Once a memory has been created, it can be read or written to by the routines "M_read" and "M_write". "M_address" returns a pointer to some address in a memory – this is used mainly by the array routines because it is more efficient to use this pointer to perform an operation directly on an ARU plane rather than reading it into a buffer, performing the operation, then writting the result back.

The file "libasw/queue.c" contains routines for setting up and using queues. The routines are general–purpose; a queue can be created with any number of elements and the queue elements can be anything desired. The routines are used in the simulator for the PE call–queue. The file "h/queue.h" has the declaration for the queue structure. This structure has fields for the queue element size in bytes, the number of elements, the count of elements currently in the queue, the head element of the queue, and a pointer to the block of memory containing the queue. The routine queuesize creates a new queue. The routine "enq" places an element on a queue; "deq" removes the top element. "topq" returns the top element without removing it from the queue. "dumpq" empties the queue. When a module calls "enq" to place an element on a queue which is already full, it is automatically suspended, and the next time an element is removed from the queue, a signal is sent to wake up the module and the operation is completed. Similarly, when a module attempts a "deq" on an empty queue, it is suspended until an element is placed on the queue.

2.2 *Module Structure.*

The simulator currently has 4 modules: the MCU, PCU, IOCU(null program) and the ASW debugger.

Each module (except the debugger) is of the form:

```
main_routine () {

    <variable declarations>


    I_creat ("<module name>", I_MEM, u, u, sizeof *u);
```

This makes the module descriptor available to the debugger. By manipulating this descriptor, the module can be stopped or started by the debugger, and the module status can be read.

$$<space> = M\_create ("<space>", <access>, <word length>, <size>);$$

<space> is the name of some memory space or register set. <access> is some combination of M_READ, M_WRITE, M_DATA, M_INST, M_SPACE or M_ARRAY, ORed together. Any combination is possible, except that M_SPACE and M_ARRAY are mutually exclusive. This parameter specifies whether the memory is an array or not, and what kinds of access are permitted on it. <word length> is the length of a word of the memory. For array memories it is in bit-planes, for others, in bits. Note that for MCU memory this is 8, because MCU addresses are byte addresses. The fact that the MCU works on 2 bytes at a time simply means that all memory accesses are in multiples of 2 bytes. <size> is the size of the memory, in words (i.e., bytes for MCU memory, bit planes for ARU memory). The size of a memory can be changed by the debugger at any time. Some memories are created with 0 words and then dynamically expanded at start up or when loaded with data.

Memories declared with M_create can be accessed with the memory management routines in libasw/memory.c, most notably M_read and M_write. It is not necessary for each module to use M_create for every memory that it uses, but using the memory management routines

provides several advantages. Bounds checking is done automatically, and the trace and breakpoint facilities of the debugger can be used for every memory (even registers) using the routines.

```
STATUS = SP_WAIT;
```

Wait for start command from debugger.

```
for (;; sp_sleep (<speed>)) {
```

Or similar loop, with "sp_sleep" of appropriate number of clock ticks at end. The rest of the program is enclosed in this loop.

```
while (ERROR) {
        wait_for (S_CONTROL);
}
```

As long as the module status is "error", wait for signal.

```
if (STATUS & SP_SINGLE) {
        STATUS |= SP_WAIT;
}
```

If single stepping, wait for control signal.

```
while (STATUS & SP_WAIT) {
        wait_for (S_CONTROL);
}
```

If status is "wait", wait for signal.

```
M_read (<space>, <pc>, <nwords>, &inst, M_INST);

<pc> += <nwords>;
```

Read next instruction, increment program counter.

The rest of the program consists of interpreting the instruction.

3 *Debugger structure.*

The debugger is the overall controller for the simulation. All I/O to the terminal and the file system is handled by the debugger. The debugger controls and monitors the operation of the MPP emulator by manipulating emulator and system variables. Variables which have had been made inter–module resources by "I_creat" or memories created by "M_create" can be accessed by the "open" and "assign" debugger commands. Certain emulator variables are placed in the debugger symbol table at start up. These variables can be seen in "mpp/mppsyms.c".

The debugger main routine is "command" in "debug/command.c". This routine initializes some variables, uses the C library routines "setjmp" and "signal" to arrange for control–C interrupts to cause the debugger to restart (this works fine on unix but only about half the time on VMS), and then calls "yyparse", the debugger command parser. Input to the debugger is initially taken from the file "mpp.ini", which opens channels to emulator resources and defines several debugger variables and procedures. Input is then taken from the terminal.

The debugger uses the routine "read_term" in "libasw/termio.c" to perform non–blocking reads on the terminal. If there is no input available when attempting to read from the terminal, "sp_sleep" is called.

The parser is in "debug/y.tab.c". This file was created by the parser generator program "yacc" from the file "debug/c.y", a BNF–like grammar for the debugger command language.

The parser action is to form a parse tree of the statement, if currently inside a block or procedure definition, add the parse tree to the structure, otherwise execute it and print the result. The routine "eval" in "debug/eval.c" evaluates parse trees. It recursively evaluates each sub–expression in the tree until the entire tree has been evaluated, and passes the result to the calling program (which is either itself, one of the routines in "eval.c" which evaluates particular functions or expression types, or the parser).

The routines in "libasw/imc.c" are used to transfer data between the debugger and the other modules of the simulator.

4 *Simulator execution.*

The file "debug/master.c" contains the main routine for the simulator. This routine does the following:

Calls the C library routine "signal" to set up signal handling. The "trap" and "pipe" (unix signals) signals are ignored; the "terminate" (control–C) signal causes a call to "quit", which halts the simulator.

Interprets the command line options, which, in unix fashion, are of the form "–<option>". The options are "–i<dir>", which tells the the simulator to use the directory "<dir>" rather than the default directory for the debugger initialization file, "–p", which causes writes to the terminal to be held until after a carriage return when something is being typed by the user, and "–w", which is the opposite of "–p" (and the default) – writes to the terminal are sent immediately.

Calls "queuesize" to create the PCU call queue.

Calls "sp_exec" to start each of the simulator modules running.

Calls "lexfile" to open the debugger initialization file.

Calls "multi_task" to take over and start running the simulator modules. At this point, each of the modules described above (MCU, PCU and debugger) begin to operate. The MCU and PCU are in a wait state, waiting for input. The user can now use debugger commands to load a program into memory and start the MCU running. The PCU will wake up upon a call from the MCU.

Each of the modules runs in a non-terminating loop, interpreting its instructions, or simply idling and waiting for input. An abort command to the debugger halts the simulation.

5 *Adding a module to the simulator.*

To add a module to the simulator (for example, a staging memory emulator), one must first write a program for the new module. This can be done pretty much independently of the rest of the simulator, with only a few subroutine calls to connect it to the rest of the system. The call

L_creat ("<module name>", I_MEM, u, u, sizeof *u);

should be put at the top of the main routine. If any memories are to be created using the memory routines in memory.c, a call such as

<space> = M_create ("<space>", <access>, <word length>, <size>);

should be included for every such memory. This can be followed by

STATUS = SP_WAIT;

to indicate that the module is to wait for a signal from the debugger to begin execution. The following sequence of code should be placed before each iteration:

```
while (ERROR) {

        wait_for (S_CONTROL);

}
if (STATUS & SP_SINGLE) {

        STATUS |= SP_WAIT;

}
while (STATUS & SP_WAIT) {
```

```
        wait_for (S_CONTROL);

    }
```

Finally, the program should execute "sp_sleep" after every iteration, to sleep for the simulated amount of time used by the module.