DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

URBANA, ILLINOIS 61801

(NASA-CR-173998) RESEARCH IN THE DESIGN OF HIGH-PERFORMANCE RECONFIGURABLE SYSTEMS Semiannual Status Report, 1 Apr. - 30 Sep. 1984 (Illinois Univ.) 33 p HC A03/MF A01 CSCL 09B G3/60

N84-34957

Unclas 23192

TITLE

Research in the Design of High-Performance Reconfigurable Systems

Second

Semiannual Status Report

April 1, 1984--September 30, 1984

NASA Grant # NAG 5-377

Project Personnel

Graduate Research Assistant

Scott D. McEwan

Andrew J. Spry

Principal Investigator

[illegible]

D. L. Slotnick

Table of Contents

Research in the Design of High-Performance Reconfigurable Systems

1. Introduction and Summary.

An initial design for the "Bit Processor" (BP) referred to in prior reports as the Processing Element or PE has been completed. It is described in Section 2. Eight BP's, together with their supporting random-access memory, a 64 k x 9 ROM to perform addition, routing logic, and some additional logic, constitute the components of a single "Stage". The Stage was previously referred to, simply, as the "Basic Component". An initial Stage design is given in Section 3. In that section, we demonstrate how Stages may be combined to perform high-speed fixed or floating point arithmetic. Stages can be configured into a range of arithmetic modules that includes bit-serial one or two-dimensional arrays; one or two dimensional arrays of fixed or floating point processors; and specialized uniprocessors, such as long-word arithmetic units.

One to eight BP's represent a likely initial chip level. The Stage would then correspond to a first-level pluggable module. As both this project and VLSI CAD/CAM progress, however, it is expected that the chip level would migrate upward to the Stage and, perhaps, ultimately the box level. Our design effort will do everything possible to facilitate this transition. The BP RAM, consisting of two banks, holds only operands and indices. Programs are at the box (high-level function) and system level.

At the system level initial effort has been concentrated on specifying the tools needed to evaluate design alternatives. It appears that the array simulator generator developed and used extensively on the MPP program can be readily extended to handle our non-homogeneous system design. Section 4 briefly describes the ASG and indicates its intended use. We have not as yet determined what design and simulation tools will be needed at the box level. Section 5 is a detailed statement of our research plans at the BP, Stage, and system levels.

2. Initial Bit Processor (BP) Description.

The initial design of the Bit Processor, hereafter referred to simply as a BP, was guided by the following objectives. First, the BP was to be capable of stand alone operation in the bit serial vertical mode. Second, the BP was to be capable of operation in parallel with other BP's and supporting hardware in horizontal mode. Third, the BP was to have a dual input bus and separate output bus to allow two separate banks of memory and a two address assembly language.

As an initial design stage the MPP's processing element (PE) was chosen as the basis for the BP because of its excellent bit serial processing ability. However, there are some notable differences in the PE's capabilities and the BP's design objectives. First, the PE has its routing logic directly associated with each PE. The BP's routing logic has been migrated to the next level of the design hierarchy to allow for microprogrammable routing logic that can be configured for various horizontal and veritcal mode operations. Second, the PE has a separate register for I/O of bit planes. The means of I/O from the L buffer, described in the prior report, to the BP processor memories has not yet been addressed. As a result, at this stage of the design the BP has no register dedicated to the I/O task. Third, the PE was not designed to operate in a horizontal mode. Although some of the functions needed for horizontal mode operation could be provided by the PE using software emulation, the BP will be designed to provide for coupling into a hardware horizontal mode component called a "Stage". Fourth, the multiple input and separate output busses proposed for the BP will enhance the BP's bit processing capability when compared to the MPP's PE. With these objectives and differences in mind the design of the BP was started. The resulting BP design is given in Figure 1, which will be referred to implicitly in much of

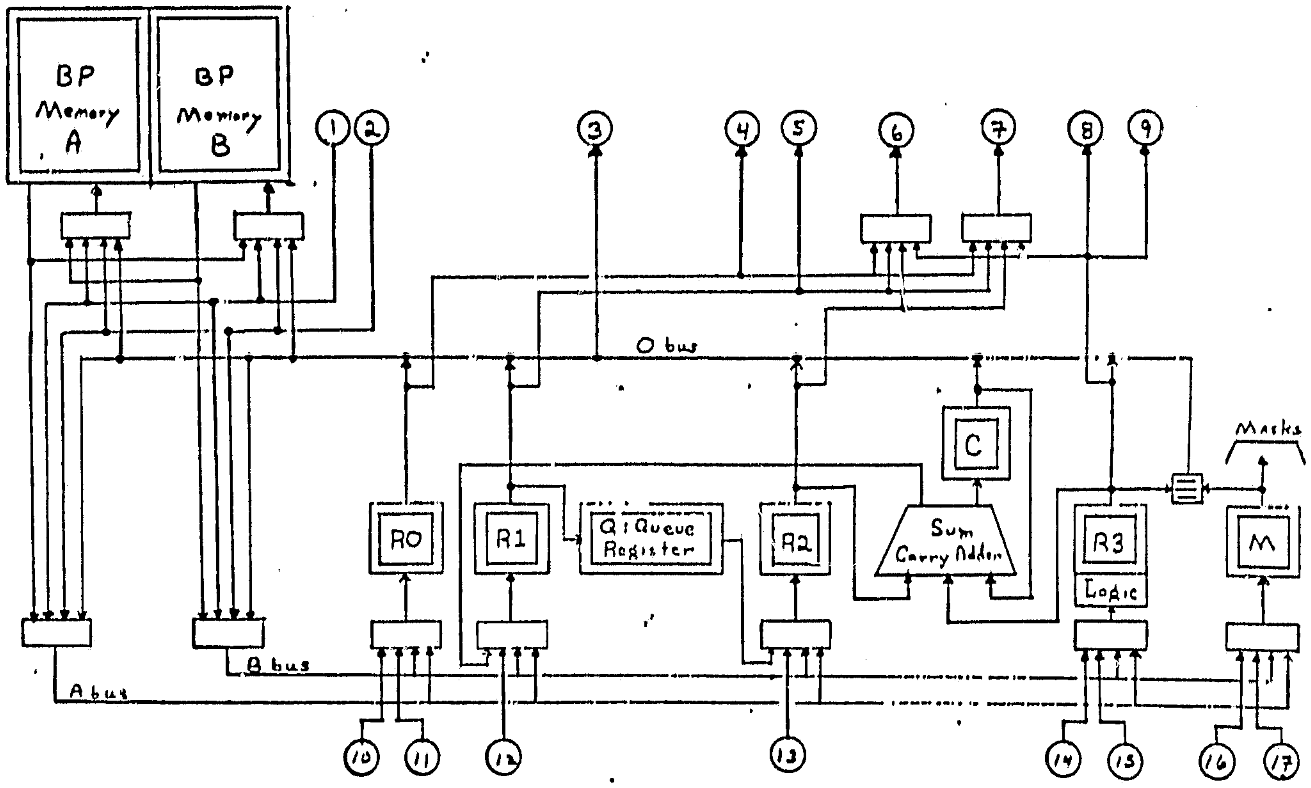


Figure 1: Bit Processor (BP).

the following discussion.

The entire range of vertical mode operations provided by the MPP's PE has been retained in the BP. (Note: to distinguish the BP and Stage registers and busses of the same name all BP registers and busses will be given lower case names.) Individual BPs can be masked out of an operation by loading the **m** (mask) register with local data. Each BP contains a single bit sum carry adder which receives its input from the **r2**, **r3**, and **c** (carry) registers and sends its outputs to the **r1**, and **c** registers. A variable length shift register, referred to as a **q** (queue) register, connects the **r2** and **r1** registers as the head and tail of a 4 to 16 bit length queue. (A major difference between the BP's **q** register and the PE's shift register is that the **q** register will probably be smaller than the PE's shift register because long word operation will be performed more efficiently in horizontal mode.) The **r3** register can be used to form any of the 16 bit-level two variable logic functions. Register **r3** is also used as the source and destination for the routing logic. The equivalence function of the mask register and **r3** (**r3 = m**) is also provided. Thus, there is full a full functional correspondence between the BP's registers and the PE's registers as shown in Table 1.

BP and PE Register Correspondence.	
BP register	PE register
c	**C**
m	**G**
r3	**P**
r2	**A**
r1	**B**
r1	No analog
q	Shift Register
No analog	**S**

Table 1: Correspondence of bit processing registers.

In addition to the above capabilities, which were also provided by the MPP's PE, the following additional capabilities are provided by the BP. The BP has three busses. Two busses (**a** and **b**) are used for independent input from various sources, including two separate memory banks, and the L buffer. The other bus (o) is used for output to various destinations, including the memory banks, the sum-or tree, and the input busses. The registers **r0** - **r3** and M can be loaded from either of the input busses. The registers **r0** - **r3** and c, and the **r3** = **m** function result can be put onto the output bus. The four registers **r0** - **r3** will be referred to as general purpose registers because of their equivalent load and read characteristics. Since the output bus can be linked to either of the input busses the capability is provided for register to register transfers at the same time as input from one memory. Various input and output points to and from the individual general purpose registers (**r0** - **r3**), the mask register (**m**), and the busses are provided to enhance horizontal mode operation. These BP entry and exit points, shown in Figure 1 as numbered circles, are listed in Table 2. Many of these additional capabilities have been provided to enhance the BP for its use in the horizontal mode Stage described in detail in the next section.

Although not a part of the BP itself, the memory structure associated with the BP also provides additional power. As stated in the prior report, two banks of memory were to be provided for the BP. Since on-chip address decoding is furnished in 1 X N configurations for all memory sizes N, there is no reason to preclude vertical mode BP local memory indexing. Potential uses include the skew-storage of matrices, which permits rows or columns of a matrix stored in an array to be accessed without remapping memory. The time spent on the index add would thus generally be more than recovered. Index sets corresponding to desired row or column sequences can be stored at some sequence of fixed locations in BP memory.

In summary, the PB design provides at least equivalent power to the MPP's PE for word lengths shorter than the length of its **q** register. The addition of the general purpose registers (**r0** - **r3**), the multiple bus structure, and the locally indexable multiple memories provide additional flexibility and power for the BP in vertical mode. In addition the multiple bus structure, the multiple locally indexable memories, and the general purpose registers will allow a two address assembly language to be used for programming the BP in both horizontal and vertical mode. Since the BP is designed to be coupled into a horizontal mode the processor is significantly more powerful than the MPP's PE in horizontal mode operation.

ORIGINAL PAGE IS
OF POOR QUALITY

Input and Output Points.	
Figure 1 I/O Number	Bit is To or From
1	From bit i of L buffer
2	From ADD ROM sum (1 bit)
3	To Sum-or tree
4	To next lower bit of 16 bit Stage level product register for horizontal mode multiplication
5	To next lower bit of 16 bit Stage level product register for horizontal mode multiplication
6	To high order address of ADD ROM
7	To low order address of ADD ROM
8	To routing logic
9	To zero detect logic
10	From ADD ROM sum (1 bit)
11	From next higher bit of 16 bit stage level product register for horizontal mode multiplication
12	From next higher bit of 16 bit stage level product register for horizontal mode multiplication
13	From ADD ROM sum (1 bit)
14	From ADD ROM sum (1 bit)
15	From routing logic
16	Stage level mask bit
17	Currently unused

Table 2: Bit Processor Input and Output Points.

3. Initial Stage Description.

3.1. General Description of the Stage.

The Stage, previously referred to as the "Basic Component", can be considered the atomic unit of the horizontal mode of operation. As an autonomous unit it is capable of all (8 bit) fixed point arithmetic and logic operations. The Stage is designed to be coupled into a long horizontal word unit capable of both fixed and floating point arithmetic as well as long word bit wise logic operations. An array of Stages coupled into long word processors will still be capable of routing in the direction perpendicular to the word. This will allow vectors of long word processors to be produced. It will also be possible to set up two-dimensional arrays of long word processors.

The block structure of a single Stage is shown in figure 2. The heart of each Stage is a set of eight BPs. The BP's registers (except the c register) are either literally connected into, or function as, horizontal registers. Each stage has micro-programmable routing logic which will allow transfer of data to neighboring BPs, Stages, and words via the **R3** register. A 64K X 9 ROM is provided to perform 2's complement fixed point addition. A stage level carry propagate and generate are produced from the ROM's 8 bit sum and carry output. A stage level zero detect and equivalence function are provided for the **R3** register. Finally, Stage level masking is provided by distributing a mask signal into each BP's mask register. The Stage configuration shown in Figure 2 is an interim design. A number of questions concerning Stage level local memory indexing, fixed point multiplication and division, and floating point operation will have to be addressed prior to the final design of the Stage.

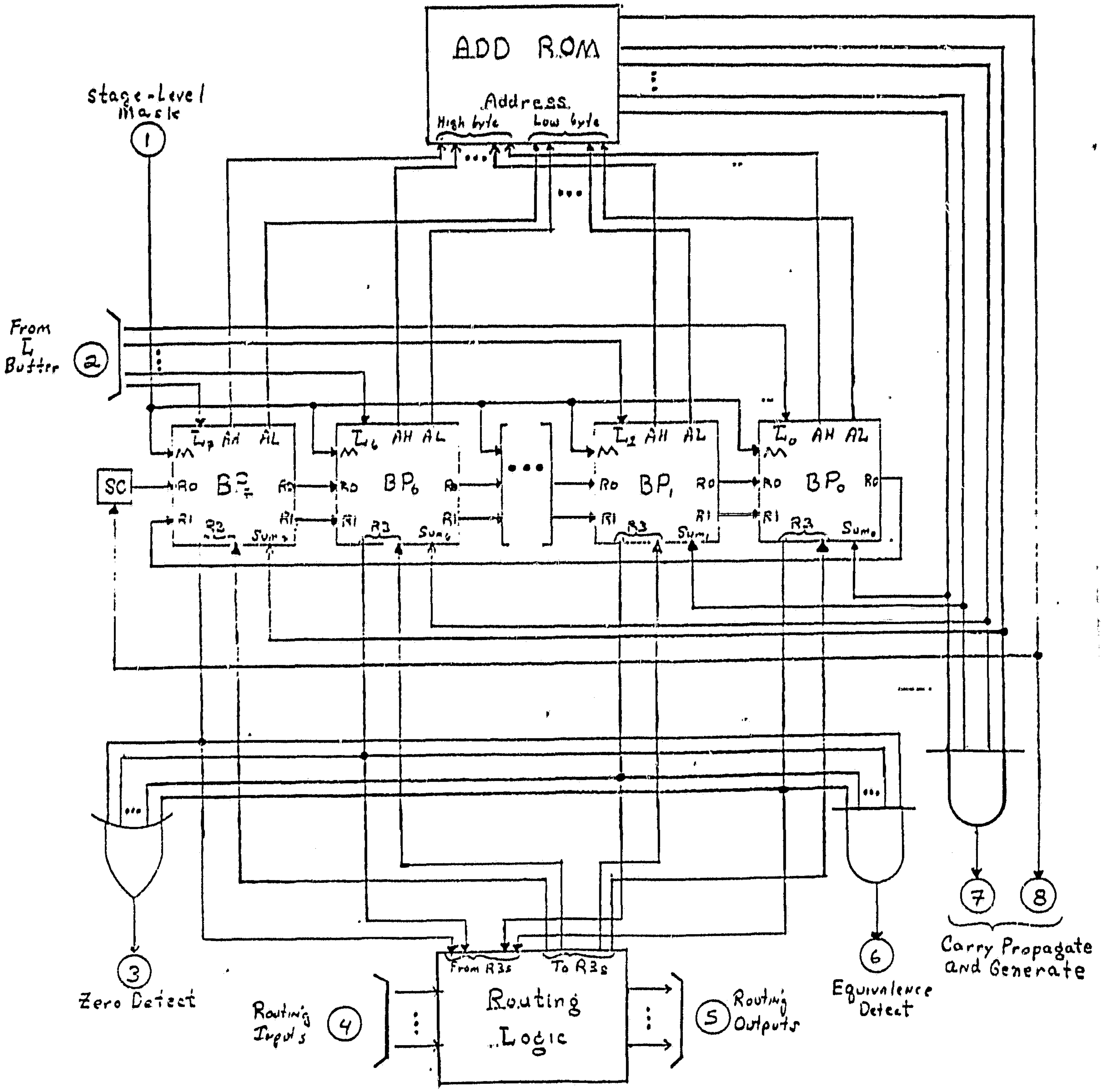


Figure 2: Block Diagram of a Stage.

3.2. Functional Description of the Stage.

3.2.1. Data Busses. The Stage has three 8 bit data busses. These busses consist of the three single bit BP busses operated in parallel (see Figure 1.) and are referred to as the **A**, **B**, and **O** busses. The **A** and **B** busses can be loaded with a single byte from the L-buffer, from the ADD ROM's sum, from the **O** bus, or from memory (A memory on **A** bus, B memory on **B** bus). The **O** bus can be sent to the **A** or **B** busses, the A or B memory, or to the sum or tree.

3.2.2. General Purpose Registers (R0 - R3). In horizontal mode the 8 bit general purpose registers consist of the BP's 1 bit registers (**r0** - **r3**) operated in parallel. They can all be used as an operand source for stage level addition. For the logic operations, multiplication, and division the registers have specific functions. The **R3** register is used to perform all bit wise logic functions. The **R0** and **R1** register together with a stage carry bit register (SC in Figure 2) are used as a 17 bit shift register for accumulating the product for 8 bit multiplication. The **R1** and **R2** registers function as the tail and head of the 8 bit wide **Q** register used in multi stage multiplication. In addition to these single Stage horizontal registers, the **R3** register together with the routing logic can be used as a multi-Stage long word shift register.

3.2.3. Mask Register. The Stage-level mask register **M** consists of the 8 BP mask registers (**m**) 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. 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.

3.2.4. ADD ROM and Carry Signals. Stage level addition is performed by using two addends stored in the general purpose registers to address a 64K X 9 ROM containing the 2's complement sums of the address bytes. Using this table lookup addition allows high speed Stage level arithmetic. To speed up multi-Stage arithmetic a Stage level carry propagate and generate are produced for use in carry look ahead logic at the board level. The carry propagate is the logical AND of the sum bits from the ROM, while the carry generate is the carry output from the ROM.

3.2.5. Routing Logic. Microprogrammable routing logic is provided at the Stage level. This logic is used in both the vertical and horizontal modes to provide communication paths between the BPs. The routing logic should provide for rich connectivity in two dimensions for the vertical mode of operation. In the horizontal mode of operation the routing logic should provide two levels of function. Long words should be connected by a nearest neighbor connection in the direction perpendicular to the word to provide for a linear array (or vector) of long words. The connections along the long words themselves should be rich enough to allow logical and arithmetic shift operations, fast floating point mantissa alignment operations, sign extension, and special guard bit handling in floating point operations. The complete function of the routing logic will depend on the range of connections needed to provide both veritcal mode BP communications and efficient horizontal mode Stage level communications for fixed and floating point arithmetic. The design of the routing logic will be an area of research in the next period.

3.2.6. Zero and Bit Wise Equivalence Detection. The contents of the **R3** register can be checked for a zero contents at the Stage level. This operation will be useful in controlling floating point alignment shifts and in long word logic operations. In addition a Stage level output is available for checking on the result of a bit wise equivalence operation. These outputs can be cascaded to provide for long word zero and bit wise equivalence detection.

3.3. Long Word Logic Operations.

The Stage's **R3** register is the primary logic engine for long word operation. As stated above a Stage level zero detect is provided for the contents of the **R3** register. These zero detects can be cascaded into a long word zero detect. In addition the Stage level equivalence functions can be cascaded to form a long word equivalence function. Bit wise logic operations can be carried out on the long word contents of the **R3** register and another value. The second operand of the logic operation is any value that can be loaded into the **R3** register from the data busses or routing logic. The 16 bit wise logic functions of two variables are provided by the BP's **R3** load logic. In addition to providing the bit wise logic operations the **R3** register can also be used as a long word shift register. By configuring the routing logic for an end around shift versus, the shifting on of a zero or propagating the sign bit, the long word shift register **R3** is capable of both logic and arithmetic shift operation. These desirable shift abilities will be used as an input in determining the capabilities of the routing logic.

As a first pass in timing for the long word logic operations a single operation machine cycle will be used as a unit of measure. All bit wise logic operations can be performed in one machine cycle. 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 shift operations will depend on the power of the routing logic connections. For a full power of two network (single cycle routes at distances $\pm 2^0, \pm 2^1, \pm 2^2, \pm 2^3, \cdots$) a shift of distance D could be performed in $LOG_2(D)$. With a partial 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 . The desired timing of the shift operations will be used as an input in determining the final horizontal mode routing logic.

3.4. Fixed Point Arithmetic.

3.4.1. Single Stage Fixed Point Arithmetic. Fixed point arithmetic can be logically divided into single and multi-Stage operations. At the single Stage level it is reasonable to talk of single and double precision addition and subtraction as well as single precision multiplication and division. The hardware shown in Figure 2 is capable of the 8 bit single precision fixed point addition and subtraction. Additional hardware will be needed for multiple precision operations and for multiplication and division.

For a first pass estimate of the times required for single precision addition and subtraction we will again use the machine cycle as the unit of measure. Consider the addition of two numbers stored in different memory banks. Since the ADD ROM provides two's complement addition in one machine cycle the operation can be performed in three cycles by the following micro operations.

$\mathbf{R2} \leftarrow \mathrm{MEM[OP1]}; \quad \mathbf{R3} \leftarrow \mathrm{MEM[OP2]}$

$\mathbf{SC,R2} \leftarrow \mathrm{ROM}[\mathbf{R2,R3}]$

$\mathrm{MEM[OP1]} \leftarrow \mathbf{R2}$

Consider the subtraction of two numbers stored in different memory banks. This operation can be performed in four cycles by the following micro operations.

$\mathbf{R2} \leftarrow \mathrm{MEM[OP1]}; \quad \mathbf{R3} \leftarrow \overline{MEM[OP2]}; \quad \mathbf{SC} \leftarrow 1$

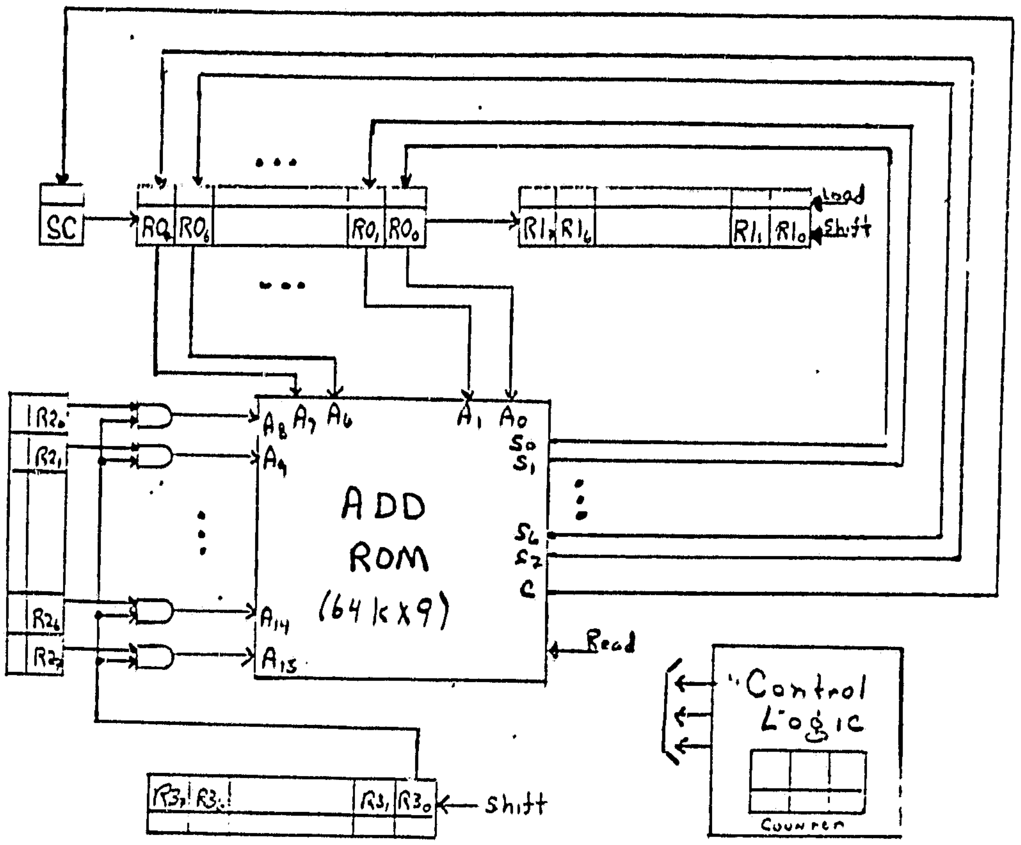
$\mathbf{SC,R3} \leftarrow \mathrm{ROM}[\mathbf{R3,SC}]$

$\mathbf{SC,R2} \leftarrow \mathrm{ROM}[\mathbf{R2,R3}]$

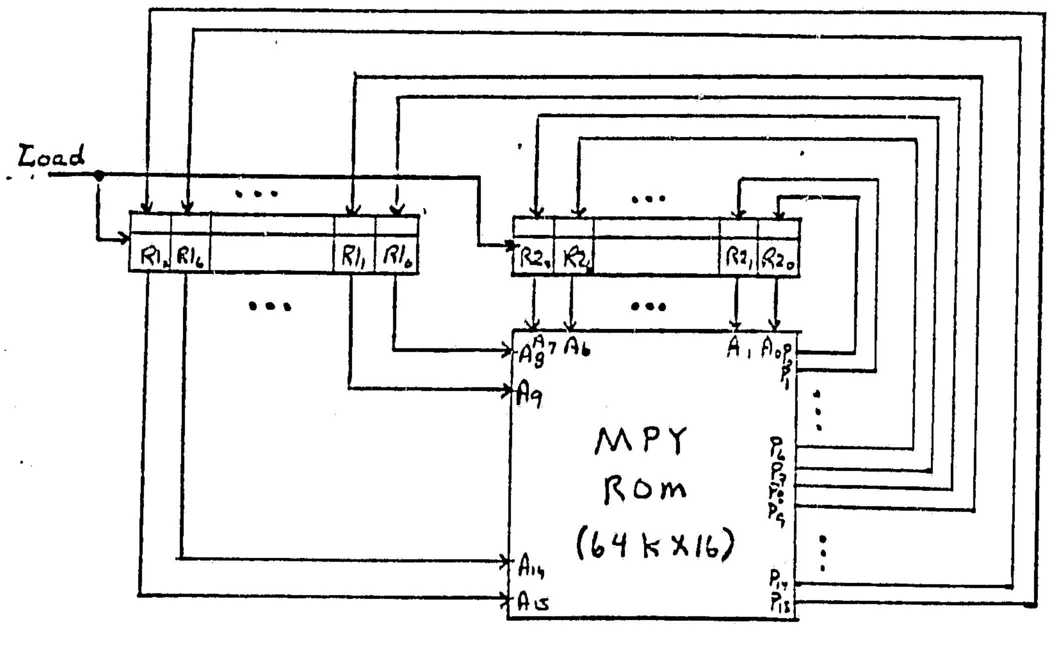
$\mathrm{MEM[OP1]} \leftarrow \mathbf{R2}$

Multiple precision operations will require additional cycles to propagate the carries across the bytes of the words, but in general subtraction will require one additional addition (of the same precision) to generate the 2's complement. Overflow detection hardware will be required at the Stage level. This hardware should work for all precisions desired at the Stage level.

Two possible schemes for a hardware Stage-level single precision multiplication are given in Figure 3. Figure 3 (a) shows how the Stage would be configured for an



(a) Iterative Multiplication.



(b) ROM Lookup Multiplication.

Figure 3: Possible Multiplication Hardware Configurations.

iterative multiply. Such an operation would require 1 cycle to load the operands into the registers, 16 cycles to ADD and shift the product into the **R0**, **R1** product register, and 2 cycles to store the result in memory. This gives a total of 19 cycles for Stage level multiplication. Figure 3 (b) shows how the stage would be configured for a ROM multiplication table lookup. Such an operation will require 1 cycle to load the operands into the registers, 1 cycle to perform the ROM lookup, and 2 cycles to store the result in memory. This gives a total of 4 cycles for Stage level multiplication (only one cycle longer than the time required for single precision addition). The various cost, and space tradeoffs of these and various other multiplication schemes will be considered before a final choice is made for the multiplication hardware.

3.4.2. Multi-Stage Fixed Point Arithmetic. Multi-Stage addition and subtraction are a natural extension of the Stage level single precision operations. The Stage produces a carry generate and propagate output. A simple ripple carry scheme could be used to extend the addition operation but this would require one carry propagate step per stage. For a 64 bit numbers this would result in a 10 cycle add time. Using a carry look ahead at the board level (it is assumed that 8 stages will fit on a board) only one cycle will be needed to correct for the carry. For the 64 bit number this scheme will result in a 4 cycle add time. For longer words the carries can be rippled from board to board with the increase in addition times of only one cycle per board. Thus the time required for long word addition is given by: $R+2$ where R is the number of 64 bit boards.

Multi-Stage multiplication will require the generation of single Stage partial products. These partial products will have to be stored and shifted around while the summation is being performed. The temporary storage of the partial products and partial product sums can be done by using the **Q** register. The shift operations will require Stage length shifts of the **R3** register. Various algorithms for this operation will be compared for efficiency.

3.5. (Normalized) Floating Point Arithmetic.

3.5.1. Floating Point Formats. The choice of a 2's complement ADD ROM has some interesting implications for the choice of the floating point representation of numbers. First, the exponent must be in 2's complement instead of the more common excess notation. Second, the sign bits (implicit because of the choice of 2's complement) must be the high order bit of the mantissa and exponent. Third, there can be no stage level floating point operations because the ROM is set up as a fixed point 2's complement sum. Fourth, the choice of ROM lookup addition and the design of the stage require that the exponent and mantissa be multiples of the stage length (currently 8). A proposed floating point format is given in Figure 4.

3.5.2. Floating Point Arithmetic. The microprogrammable routing logic will be used extensively to provide fast floating point operations. As an example consider the addition of two floating point numbers stored in different memory banks. The configurations of the Stage level masks and routing logic for this operation for 32 bit floating point arithmetic are given in Figure 5. The first step in floating point addition is to compare exponents. This operation is done with the word configured as shown in Figure 5 (a). The exponents are loaded from memory, a subtraction is performed, and the difference δ is put in **R3**. This sub-operation will require 5 machine cycles. The second step is to align the binary points of the mantissas. This operation is done with the configuration of Figure 5 (b). The bits shifted off the exponent are used to mask

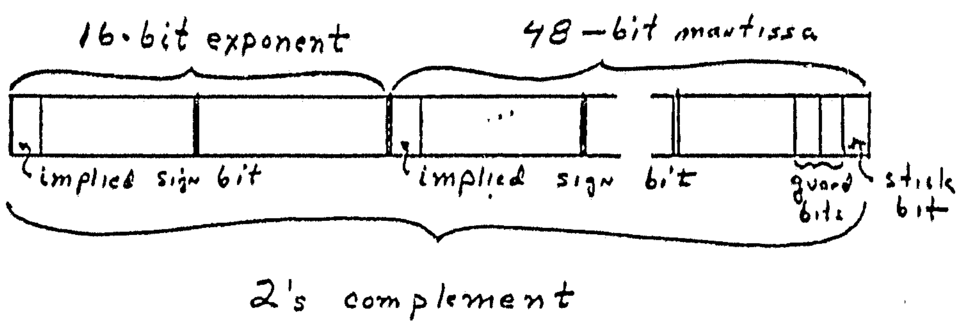
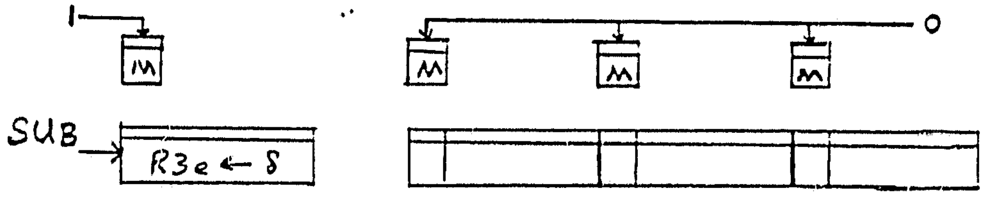
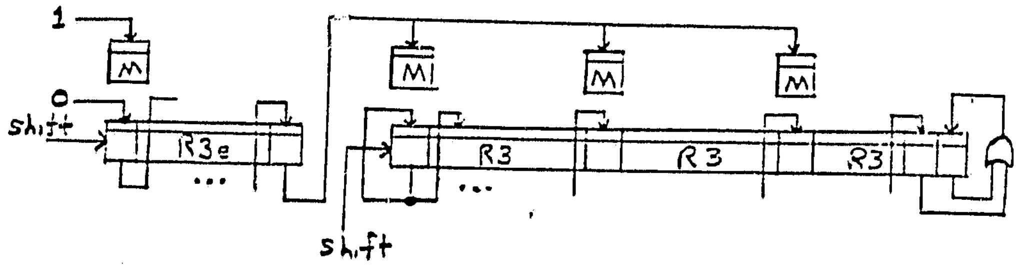


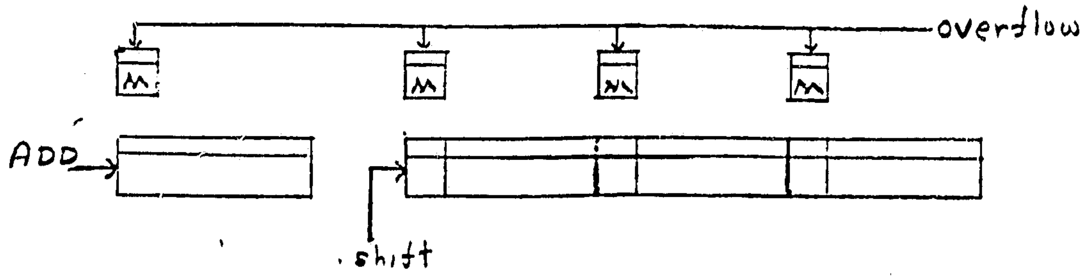
Figure 4: 64 Bit Floating Point Format.



(a) Exponent Comparison Step.



(b) Mantissa Alignment Step.



(c) Renormalization Step.

Figure 5: Configurations for 32 bit Floating Point Addition.

the shift of the mantissa. The Stage-level zero detects can be used to stop the shifting operation. In any case the shift can be stopped after the 5th bit of the δ (see Figure 5 (a)) has been shifted because the significant digits will have been shifted off the mantissa. In is worthy to note that the shift preserves the sign of the mantissa and provides for a "sticky bit" as a guard digit. The third step is the mantissa addition itself. The result of this operation will be placed in **R3**. Register to register addition of the 24 bit mantissas will require 3 cycles. The fourth step is the renormalization of the mantissa. This operation is performed using the configuration given in Figure 5 (c). Notice that the overflow bit is used to mask the words that do not require renormalization out of the operation. This step will require 2 machine cycles for the increment of the exponent. Finally, the result will be stored in memory. This will require one machine cycle. Thus, 32 bit floating point addition can be performed in 16 machine cycles where the maximum alignment shift was required for one of the operands.

Floating point subtraction and multiplication are similarly extensions of the fixed point operations on segments of the word. The stage level masking and microprogrammable routing logic are very important in providing fast floating point operations.

4. System Level - The Extended (Array) Simulator Generator.

The Array Simulator Generator (ASG) is a set of modern software tools built originally for constructing simulators for bit-serial array computers such as the MPP. A simulator created using ASG contains a complete functional emulation of the target machine, as well as a built in debugger/user interface. Although ASG was initially designed to simulate array machines, virtually any machine can be simulated by writing emulators for each functional unit and using ASG routines to connect them and interface with the ASG debugger. A library of routines implementing common array operations is provided to simplify creating emulators for array units (see the Appendix).

ASG can be used to create a simulator for a proposed machine, in such a way that variants of the simulated machine's architecture can be examined. Many hardware parameters (such as memory depth, array dimensions, or the clock speed of any functional unit) can be altered dynamically while the simulator is running, allowing quick comparisons to be made between similar architectures.

An ASG simulator can be used as a powerful software development system. Code which is developed on the simulator can be easily transported to the actual machine. A significant amount of software was developed on an ASG MPP simulator and subsequently ported to the MPP. The only changes necessary were due to the fact that the simulated MPP had a 16 by 16 array rather than the 128 by 128 array on the MPP. The simulator gives an accurate estimate of a program's running time, as well as a count of the number of instructions executed by each unit. If only a timing estimate is wanted for a long runni g program, the simulator can be set up so that subroutines with known run times can be trapped and their run time added to the system clock without bothering to execute the routines.

4.1. Elements of the ASG System.

The ASG debugger is the overall controller for the simulation. Through the debugger, the user has control over all components of the simulated machine. All memory and registers can be viewed or changed; programs can be run, halted or single-stepped; breakpoints can be set or cleared; individual emulated hardware units can be started or halted; all or part of any unit's memory can be loaded from or saved to a file (with corner-turning for the array unit). Certain hardware parameters can be changed dynamically. The system clock can be read to give estimates of program running time.

The debugger is independent of the machine emulator and requires changes or additional code to work with a new emulator. Information about the emulated machine can be given to the debugger by one of two methods: the emulator can place information in the debugger symbol table during start up using an ASG subroutine provided for this purpose, or the information can be extracted from the symbol table of the compiled ASG simulator and placed in a file which is automatically read by the debugger during start-up. The latter method is more flexible, but less portable, as it requires a separate program to extract the symbol table information for each system run on the simulator.

The multi-processing subroutine library contains routines for simulating parallel execution of the functional units of the simulated machine. The library includes routines for creating new processes to be run in parallel, for suspending a process for a period of simulated time, while waiting for some event to occur, and for signaling that an event has occured (e.g., something has been placed on a queue, data is available from an input device, a unit has completed execution of a subroutine). These routines, along with a built-in scheduler, act like a virtual operating system for the simulator.

The inter-process communication library handles communication between units of the simulated machine and between the emulator and the debugger. There are routines for creating a descriptor for a resource held by a process, opening a channel to a resource in another process, and reading and writing to resources.

The queue management library contains routines to create a queue, place an item on a queue, pop an item off a queue, look at the top item of a queue, and clear a queue. Queue elements can be any data type. A process is automatically suspended when it attempts a queue operation that can't be done (trying to take an element off an empty queue or place one on a full queue) and reactivates the process when the operation becomes possible.

The I/O library handles all I/O between the simulator and the host computer's file system. Any part of the simulated computer's memory can be read from or stored to a file. Corner-turning can be performed when loading or saving any array unit memory.

The array operations library contains routines for a variety of bit- serial array operations. These include: array to array memory plane moves, all 16 binary logic operations between two planes of array memory or registers, planewise one bit addition, array shifts, logical or of all bits in a plane, sum of bits in a plane. All operations can be optionally masked with a plane of mask bits.

4.2. Building a Simulator.

In order to build a simulator using ASG, it is necessary to write an emulator for each functional unit of the machine. The existing array operations library can be used to help write an array unit emulator. The start up for each emulator should create descriptors for any resources needed in inter-process communication, and should initialize the debugger's symbol table with addresses of emulator resources. Finally, a main routine must be written to start up the different units as parallel processes and begin multi-processing (a routine exists that can be used as a template for this.)

As boxes are designed it is our intention that emulators will be written and added to the system. It is our hope, that the simulation activity will closely track the design effort.

5. Current Objectives.

5.1. Bit Processor Level.

Logic design refinements will center around the BP register structure. The possible roles of the BP Q register in horizontal mode (primarily, as a partial product queue) will be examined with the objective of finalizing both its length and connectivity. The whole question of reasonable vertical mode uses, bracketing storage and computational efficiency, degrees of problem parallelism expected to be encountered, and required input and output precision, will be addressed.

Routing studies will continue. Although we are now strongly inclined toward power-of-two connectivity, in both vertical and horizontal mode, we will subject this choice to an analysis of its consequences for vector and long-word calculation. A first pass will be made at BP hardware layout, timing, and cost.

5.2. Stage Level.

Hardware for forming a 16 bit product from two 8 bit operands at the Stage level will be added. No hardware is shown at the age level as yet since the means for providing this function is still under study. Some of the alternatives being investigated are an iterative multiply using all four general purpose registers, or a ROM lookup. Tradeoffs between these and other methods will be investigated during the next period.

An instance of the close coupling between design considerations for the BP and the Stage is provided by the relation between BP memory size and the size of the add ROM in the Stage. As we indicated in Section 1, it is unquestionably useful and technologically feasable to allow BP memory indexing on the local level. This causes no difficulty, independent of BP memory size, in vertical mode. The only memory size effect is the time required for the bit-serial index add. In horizontal mode, however, an index at the Stage (or multi-stage) level will require double (or higher) precision utilization of the add ROM. The numerous ways to handle this difficulty, will be examined.

Hardware for the detection and handling of overflow will be designed.

Floating point operation specification and design will continue including developing the cost and performance consequences of the two's complement representation selected.

Stage-level connectivity will be finalized with the objective of providing efficient matrix, vector, and word (standard to long) operation.

The incorporation of Stage-level carry-generate and carry-propogate will be studied from both the utility (primarily for carry look-ahead) and cost viewpoints.

Two "exotic" possibilities will be briefly examined: the implementation of asynchronous Stage long word operation, and the use of n^2 Stages for n byte multiplication/division. Finally, a first pass will be made at Stage layout, timing, and cost.

5.3. System Level.

A principal objective will be to achieve a working extension of the (Array) Simulator Generator. This will permit us to generate and evaluate the first set of box-level designs. An interesting, distantly related question is the desirability of an efficient MPP emulator mode.

It is important to note that we intend to maintain a clear division between box designs utilized the Stage "component" and the incorporation of boxes at the system level. Thus, as the advance of VLSI continues we can substitue more advanced stages into boxes or entirely specialized box-level components, without doing total violence at the system hardware and software level.

Finally, we will select and start the design of a set of dense, banded, and random-sparse matrix/vector boxes together with the specification of their comprehending system environment.

6. Appendix: ASG Subroutine Libraries.

Multi-processing

sp_exec(prog)

Define routine as a sub-process and set up for multi-processing.

multi_task()

Start up multi-processing;

sp_ch(n)

Set bit "n" in global state.

sp_seen(n)

Clear bit "n" in global state.

wait_for(n)

Suspend current sub-process until bit "n" in global state is set, then clear bit "n" and resume execution.

sp_sleep(n)

Suspend current sub-process for "n" clock ticks.

sp_swap(user)

Replace current sub-process with "user". Usually only called by multi_task().

Inter-process Communication

I_creat(name,

Create a descriptor for referencing a resource in a sub-process.

I_open(name)

Open a channel to a resource in another sub-process.

I_write(channel,

Write to a previously opened channel.

I_read(channel,

Read from a channel.

Array Operations

bit_copy(dest,

Move "nplanes" planes of array from "src" to "dest".

bit_logic(dest,

Perform any of the 16 binary logic operations between the planes "src" and "dest", either masked or unmasked.

bit_net(dest,

Array shift "nplanes" planes "distance" pe's, using "type" to determine topology.

bit_or

Find logical or of all bits in a plane.

bit_sum(array)

Find sum of all bits in a plane.

fulladd(sum,

Full add of 2 planes with carry, masked or unmasked.

halfadd(sum,

Half add (1 plane + carry), masked or unmasked.

shiftreg

Shift register operations. Shift register length is given by "code".

Queue Management

queuesize(q,

Set length of queue to "n".

enq(block,

Place block on queue "q".

deq(block,

Remove next item from queue.

topq(q)

Top element of queue.

dumpq(q)

Clear queue.

Miscellaneous Operations

address(itemno,

Return pointer to "count" bytes at offset "offset" in item "itemno".

copy Copy "length" bytes.

clearSet "len" bytes to zero.

bit_size(nrow,

    Change array dimensions to "nrow" by "ncol".

bit_in

    Read in and corner-turn data from a file into array memory.

bit_out(file,

    Corner-turn and write out array planes to a file.