THE ORGANIZATION AND LANGUAGE DESIGN OF MICROPROCESSORS FOR AN SIMD/MIMD SYSTEM

Howard Jay Siegel
Philip T. Mueller, Jr.
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

June, 1978

ABSTRACT

A dynamically reconfigurable large-scale multimicroprocessor system capable of operating as one or more independent SIMD (single instruction stream - multiple data stream) machines and/or MIMD (multiple instruction stream - multiple data stream) machines is described. User microprogrammable bipolar bit-slices are used as building blocks in the construction of computational and control units. The semi-stack architecture and instruction set of these units are described. Instructions unique to parallel processing are included in the system design. A highly flexible subroutine linkage facility is presented. Methods for analyzing the instruction set are discussed.

Research sponsored in part by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under Grant No. AFOSR 78-3581. The United States Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation hereon.

CH1387-0/78/0000-0311$00.75

I. INTRODUCTION

As a result of the microprocessor revolution, a dynamically reconfigurable large-scale multimicroprocessor system which employs parallelism to perform image processing tasks more rapidly than previously possible is now feasible. A multimicroprocessor system can use parallelism to achieve the real time image processing required for such applications as robot (machine) vision, automatic guidance of air and space craft, and air traffic control. Furthermore, there are many image processing tasks which can be performed on a parallel processing system, but are prohibitively expensive to be performed on a conventional computer system, e.g., remote sensing by satellite [1].

Two types of parallel processing systems are single instruction stream - multiple data stream (SIMD) machines and multiple instruction stream - multiple data stream (MIMD) machines [2,3]. An SIMD machine typically consists of a set of N processors and N memories. A control unit broadcasts an instruction to all of the processors and all active ("turned on") processors execute that instruction at the same time. Thus, a single stream of instructions drives all the processors. Each processor executes the instruction using data taken from a memory to which only it is connected, i.e., each processor uses data from a different memory. This provides a multiple data stream. Examples of such machines are the Illiac IV [4,5] and STARAN [6,7]. An MIMD machine typically consists of N processors and N memories, where each processor may follow an independent instruction stream. As with SIMD architectures, there is a multiple data stream. Examples of such machines are C.mmp [8] and Cm* [9]. Both SIMD and MIMD machines include an interconnection network for interprocessor communications.

The use of parallel processing for image processing has been limited in the past due to cost constraints. Most systems used small numbers of processors (e.g., [4]), processors of limited capabilities (e.g., [10]), or specialized logic modules (e.g., [11]). With the development of the microprocessor and related technologies it is reasonable to consider parallel systems using a large number (e.g., 1024) of complete processors. In the field of image processing both SIMD and MIMD machines would be of great use. This is discussed in [12,13].

Due to the low cost of microprocessors, computer system designers have been considering various multimicrocomputer architectures, such as [9,14-18]. The system described here differs from others in that:

(1) it may be partitioned to operate as many independent SIMD machines following the same or different instruction streams;

(2) parts (or all) of the system may be operating as independent MIMD machines, while the rest of the system is operating as one or more SIMD machines;

(3) the processors used for performing the computations can transfer data simultaneously; and

(4) a variety of problems in image processing and pattern recognition will be used to guide the design choices.

In the design of the system proposed here various image processing tasks have been and will be considered. The philosophy of examining the problem and then designing the machine which can best solve the problem, under certain economic and technological constraints, will be used. It is felt that this will lead to a system that will function efficiently not only for image processing, but for a large class of similar computational problems, such as speech processing, remote sensing using multispectral data, and waveform processing in biomedical engineering.

Section II is a brief overview which describes the major points of the system. Section III discusses various methods of communication between a control unit and its processors in an SIMD machine. Section IV gives some details of a microcomputer architecture which is currently being considered for use as a processor in an SIMD or MIMD environment. Section V describes the instruction stream handler of the control unit for SIMD mode. Section VI discusses some software considerations.

II. SYSTEM OVERVIEW

PASM, a partitionable SIMD/MIMD system, is being developed as a result of the work done in [19-24]. A brief system overview is given here. Certain aspects of the system are discussed in more detail in [12,13]. Figure 1 is a block diagram of the basic system components: the Parallel Computation Unit, the Micro Controllers, the Control Disk, the Memory Management System, the Memory Disk, and the System Control Unit.

The heart of the System is the Parallel Computation Unit (PCU), which contains N processors, N memory modules, and an interconnection network. The PCU processors are microprocessors that perform the actual SIMD and MIMD computations. The PCU memory modules are used by the PCU processors for data storage in SIMD mode and both data and instruction storage in MIMD mode. The interconnection network provides a means of communication among

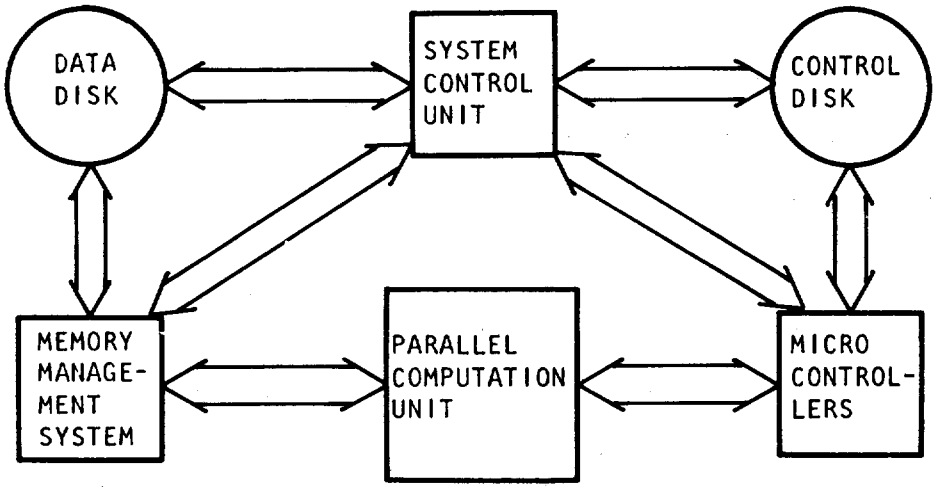


Figure 1: Block diagram overview of PASM.

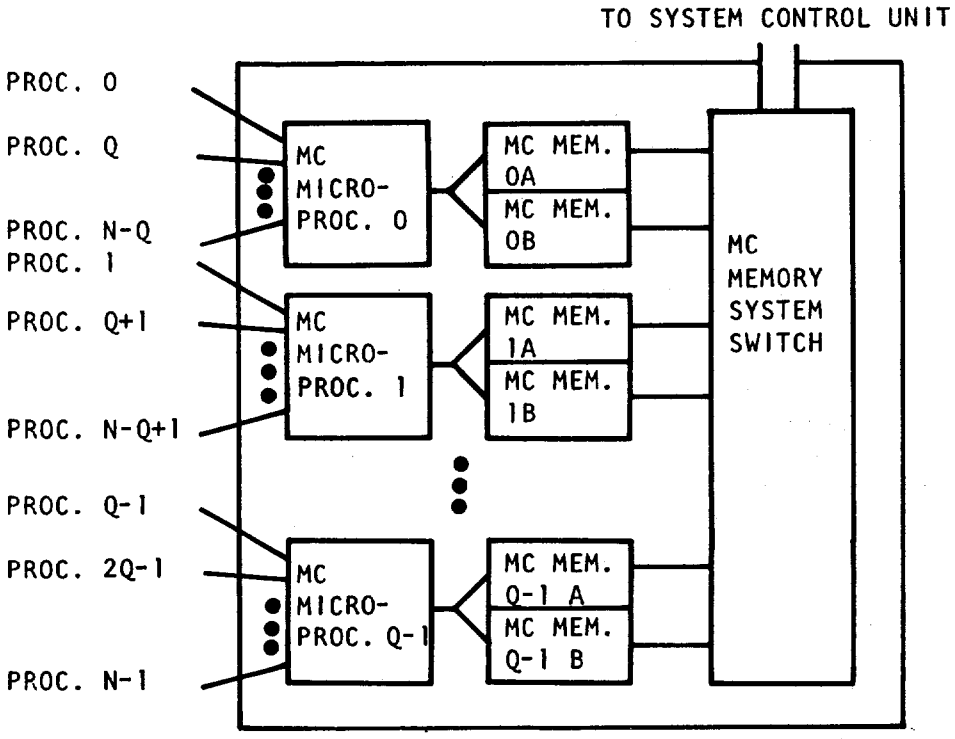


Figure 2: PASM Micro Controllers.

the PCU processors and memory modules. The processors and memory modules are physically numbered (addressed) from 0 to N-1, where $N=2^n$. The interconnection network can be partitioned into independent sub-networks of varying sizes, which are powers of two. The only constraint is that the physical addresses of the P processors and memory modules in a partition have the same $\log_2 N - \log_2 P$ low-order bits. Further details about the PCU can be found in [12,13].

The Micro Controllers are a set of microprocessors which broadcast instructions to the PCU processors in SIMD mode and orchestrate the activities of the PCU processors in MIMD mode. The Control Disk stores the control instructions for the Micro Controllers as well as the programs for the PCU processors in SIMD mode. The Memory Management System controls the loading of the PCU memory modules with data for PCU processors operating in SIMD mode and with data and instructions for PCU processors operating in MIMD mode. The Memory Disk stores these data and instruction files. The System Control Unit is a conventional machine, such as a PDP-11, and is responsible for the overall coordination of the activities of the other components of PASM.

Many computations can be more efficiently executed if the N PCU processors and memories are partitioned into many smaller groups of processors and memories, each group behaving like an SIMD or an MIMD machine. The basic method to provide multiple controllers to broadcast instructions so that the system can be partitioned into independent SIMD machines is shown in Figure 2.

A Micro Controller is a microprocessor which is attached to a memory module. Each memory module consists of a pair of memories so that memory loading and computations can be overlapped. In SIMD mode, each Micro Controller fetches instructions from its memory module, executing the control flow instructions (e.g., branches) and broadcasting the data processing instructions to the PCU processors. There are $Q=2^q$ Micro Controllers, physically addressed (numbered) from 0 to Q-1. Each controls N/Q PCU processors, where possible values for N and Q are 1024 and 16, respectively. The physical addresses of the N/Q processors which are connected to a Controller must all have the same low-order q bits so that the interconnection network can operate in the partitioned environment. The value of these low-order bits is the physical address of the Micro Controller.

An SIMD machine of size MN/Q, where $M = 2^m$ and $0 \leq m \leq q$, is obtained by loading M Controllers with the same instructions. The physical addresses of these M Controllers must have the same low-order $q - m$ bits. This is because the physical addresses of all PCU processors in a partition of size MN/Q must have the same low-order $q - m$ bits in order for the interconnection network to

function properly. The Micro Controller Memory System Switch allows the M memories to be loaded in parallel [13].

III. MICRO CONTROLLER TO PCU PROCESSOR COMMUNICATIONS

Both the Micro Controllers and PCU processors will be constructed from user microprogrammable bipolar bit-slice microprocessors. Using current technology, user microprogrammable bit-slices are faster than single chip microprocessors [25]. The user microprogrammable capability is especially important so that instructions special to parallel processing may be implemented.

Since the PCU processors are microprogrammable, a question is raised. What information should the Micro Controller send to its PCU processors? If the PCU processors are only to be run as part of an SIMD machine, then a large savings on hardware will be realized by having the Micro Controller send control signals to the PCU processors, rather than sending instructions which need to be decoded. The hardware that may be left out by sending control signals is the microprogram sequencer and microprogram store. When this hardware is multiplied by N=1024 (a copy for each PCU processor) a great savings can be realized.

If the decision is made to send control signals, rather than "assembly instructions," another question must be answered. Which is more cost effective: sending encoded control signals, and have the encoded control signals decoded by the PCU processors, or sending full control words?

Encoding the control signals requires fewer board-to-board connections, but requires additional decoding hardware. In addition to having fewer interconnections, the firmware in the Micro Controllers is greatly reduced, i.e., the size of the microstore is greatly reduced. For example, suppose that the PCU processors use 60 control signal lines, but only about 1000 different control words are used (microstore sizes for the PDP11, IBM 360, and IBM 370 are given in [26]). Then it is possible to encode the control words so that each control word requires only ten bits. The microstore word size is reduced from 60 bits to ten bits. Note however, that now at each PCU processor a logic array which has ten inputs and 60 outputs is required for decoding.

Sending the full control word allows more flexibility, i.e., any possible control word may be realized. However, this requires many more board-to-board connections, perhaps three times as many connections for PASM. In the example above, six times as many connections would be required.

In a multiprocessor design described by [27] the control units send encoded control signals to the processing elements. In that system the processing elements are dynamically allocatable, so every control unit (the design specified eight control units) had to be able to send signals to every processing element. In order to keep the bus complexity and cost down encoded control signals were sent. The same decision is made here.

If the PCU processors in PASM are to be able to operate independently as an MIMD system, then each PCU processor must be able to decode raw "assembly instructions." This means that each PCU processor will require its own microprogram sequencer and store, which defeats the purpose of having the Micro Controllers translate the "assembly instructions" into a sequence of encoded control signals.

In the system under consideration, only a limited number of the Q sets of N/Q PCU processors will be able to operate in MIMD mode. This compromise was made on the basis of economic constraints and application requirements. This will allow the system to avoid having N microprogram sequencers and stores, but still operate in a partial MIMD mode. The exact number of PCU processors to be capable of either mode of operation will be determined after an analysis based on a variety of image processing tasks.

A PCU processor in this dual mode must be capable of accepting "assembly instructions" from its associated memory, for MIMD mode, or encoded control signals from the Micro Controller, in SIMD mode. This means that in general, the signals from the Micro Controller must be blocked out while the PCU processor is in the MIMD mode. Therefore each PCU processor will need some type of switch to block the Micro Controller signals. If the Micro Controller sends unencoded control signals a very large switch will be needed at each PCU processor to block the control signals. Thus, the decision to encode control signals simplifies this problem.

In addition to sending information signals, e.g., encoded control signals, some error correcting signals should also be sent. Error correcting signals should correct transmission errors that might occur. Since this is a tightly coupled system, and the transmission distances will be relatively short, not much error correcting capability will be required. Perhaps only single error correction and double error detection is needed.

In section IV the PCU processor architecture is discussed in detail. The instruction format constraints discussed there apply to MIMD mode. Section V expands on how the same instruction format will be handled by the Micro Controller in SIMD mode.

IV. PCU PROCESSOR ARCHITECTURE

A. Introduction

Choosing the PCU processor architecture is a critical step in this design. The PCU processor does all of the major data manipulations, and therefore must be designed to do data manipulations efficiently.

The architecture choosen combines the implied operand ability of the stack machine with the indexing power of the conventional multiregister machine. This combination yields an architecture which is a departure from both stack machine architecture and conventional architectures.

B. Stack Machines

A stack computer is a computer which uses a data structure of the form of a last-in first-out list, i.e., a stack [28-30]. All operations involve the top elements of the stack, e.g., an ADD instruction might add the two top elements of the stack together, leaving the result as the new top element of the stack.

The stack architecture leads to two instructions which are non-existent in conventional machines. The push and pop instructions are used to put data on the stack (push), or take data off of the stack (pop).

The advantage of a conventional stack machine over a conventional register machine is that one or two operands, the top elements of the stack, are always implied. This means that the instructions for a stack machine require fewer bits than instructions for a register machine. Fewer bits per instruction implies more compact code and faster execution. The execution is faster because fewer memory fetches for instructions are required.

A disadvantage of a stack machine is that to have an infinite stack, or at least a very large stack, one must use main memory. This means data near the top of the stack will be moved back and forth between main memory and the fast registers (assuming there are some) [28]. This can slow down the system.

Another disadvantage is there is only one top of stack, so it is very awkward to have, for example, partial results, loop counters, and array index values in use simultaneously.

Note that to put operands on and to take operands off the stack requires push and pop instructions. However, push and pop instructions are equivalent to the load and store instructions of a conventional machine, except that for the stack machine the push

destination and the pop source operands are implied rather than explicit.

C. The PCU Processor

The PCU processor architecture under consideration is based on the AM2900 bipolar bit-slice microprocessor family [31]. The AM2901 is a user microprogrammable cascadable four bit-slice which contains a 16 word two port register file, an ALU, and a register for temporary results.

To take advantage of the implied operand in a stack machine the 16 word register file will be treated like a stack. External to the AM2901s there will be a four bit binary up/down counter which will serve as a stack pointer into the register file. Note that the stack is for arithmetic operations and partial result storage only, and is very limited in size. By having a stack of limited size, i.e., only as large as the register file, a disadvantage of stack machines given above is avoided. All of the data on the stack resides in fast registers so the system is not slowed down by transfering stack elements to and from main memory. For larger stack usage, such as for parameter passing, only main memory will be used. This is discussed in detail in section VI.

When a push or pop instruction is executed, the implied operand is contained in the word (or words) of the register stack which is pointed to by the stack pointer. If a push operation results in a stack overflow, or a pop operation results in a stack underflow, an interrupt will be generated.

In order to provide index register and loop counter facilities, any word of the register file will be directly addressable for these operations. This removes a disadvantage of stack machines mentioned previously. By having each word of the register file usable as a loop counter or index register, it is easy to have partial results, loop counters, and array index values in use simultaneously. The decision to have any word of the register file directly addressable was made on the basis of the following reasoning.

In order to have enough opcodes for all of the desirable instructions the opcode should be at least eight bits in length. These eight bits do not include any bits to represent register numbers, so some additional bits for register numbers must be included.

How many bits should be added? Adding only one or two bits allows for two or four index registers or loop counters. Clearly this is too few. Adding three bits gives eight index registers or loop counters. While this is sufficient, how are the other five

bits in the byte used? Furthermore, if an instruction uses two registers, then how are the two bits left over used?

By using four bits to specify a register number the basic data size becomes a nibble (four bits). Now the instructions fit into the memory well. An opcode takes two nibbles, a register number takes one nibble, an instruction which uses two registers takes four nibbles (two nibbles for the opcode and two nibbles for the two register numbers), and an address takes four nibbles (a 16 bit absolute address). If each word of the register file is made directly addressable, then the instructions fit into memory with no waste.

Instructions may now end in the middle of an eight bit byte (the smallest piece of data directly addressable). This scheme of having instruction boundaries in the middle of the smallest piece of data directly addressable is implemented in the CDC 6000 Series [32] (the smallest piece of data directly addressable is the word).

One of the problems associated with having instructions start in the middle of a word is: how can one transfer control (i.e., explicitly branch) to an instruction which starts in the middle of a word? The solution used by the CDC 6000 Series is to pad the rest of the word with a null instruction, i.e., a no-op, so that the instruction to which one wants to transfer control starts on a word boundary (this padding is called forcing upper). This allows a jump in the normal manner.

In the CDC machine instructions may start in the middle of a word, but the opcode may not cross word boundaries. The situation here is more complex, since opcodes may cross byte boundaries. If the last four bits of a byte are to be padded with a null instruction, the opcode for the null instruction must be four bits. If the opcode were eight bits, the next instruction would still start in the middle of a byte. This means there are instructions with four bit opcodes and instructions with eight bit opcodes, which greatly complicates the situation.

To simplify the situation the opcode for the null instruction is made to be eight bits, but there are 16 consecutive opcodes which are identical in the most significant four bits which represent the null instruction, e.g., hex F0 through hex FF. However, the null instruction is now handled differently by the microcode. Instead of going to the next eight bits for the next opcode, the microcode appends the next four bits to the least significant four bits of the null instruction opcode to obtain the next opcode.

For example, suppose hex F0 through hex FF are the opcodes for the null instruction. Then the sequence of memory bytes (in hex): F5 AB FC would be interpreted by the microcode as the instruction which has an opcode in hex of 5A. If this particular instruction did not require any addition bits, then the next opcode in hex would be BF.

D. Data Conditional Instruction Hardware

In conventional computers data conditional instructions, such as conditional branches, are not a problem to implement. This is because there is only one data stream so, as in the case of the conditional branch, there can never be a conflict as to where control should be transferred. In an SIMD envirnonment, a conditional branch is not uniquely defined, i.e., it is possible for one processing element to satisfy the condition while another processing element does not.

To allow data conditional computations in PASM each PCU processor has a Conditional Mask Stack (CMS). This bit stack is completely separate from the arithmetic stack discussed previously. The top element of the CMS is the active/inactive status of the PCU processor. If the top element is a "1" then the PCU processor is active, i.e., executes instructions it receives from its Micro Controller. If the top element is a "0" then the PCU processor is inactive. A PCU processor may be disabled by either inhibiting various clock signals or by blocking its control signals. Data conditional masks are more fully described in [3,33].

The IF A THEN B ELSE C statement of conventional structured programming languages is replaced by WHERE A DO B ELSEWHERE C. By using the keyword "WHERE" the programmer is made more aware of the parallelism underlying his program, i.e., in all of the PCU processors where A is true do B.

A statement such as WHERE A DO B is implemented by pushing A onto the CMS, then executing B, and finally popping one item off of the CMS. The instructions required for this appear in the Appendix (WPSH, CMPOP).

A statement such as WHERE A DO B ELSEWHERE C is somewhat more complicated to implement. First, the complement of A is pushed onto the CMS, then A is pushed onto the CMS. B is executed; one item is popped off of the CMS; C is executed; and one more item is popped off of the CMS. The two pushes are executed by the single instruction WEPSH.

Two flip flops, the Condition Flip Flop (CFF) and the Accumulator Flip Flop (AFF), are provided to facilitate the evaluation of conditionals (A in the examples above). Instruc-

tions for various logical operations between the CFF and the AFF appear in the Appendix.

The WPSH, WEPSH, CMPOP, and ICMS are privileged instructions, executed by all PCU processor (both active and inactive). If WHERE statements are nested, an inner WHERE can not incorrectly activate a disabled PCU processor. This is because the CMS is loaded from the CFF, which must remain unchanged (i.e., equal to 0) in a disabled PCU processor until the PCU processor is reactivated by the appropriate number of CMPOPs. To allow nesting in WHERE-ELSEWHERE statements the CMPOP instruction also clears the CFF so that a PCU processor which is deactivated by the CMPOP can not be incorrectly reactivated by an inner conditional statement.

V. MICRO CONTROLLER ARCHITECTURE

A. Instruction Stream Handler

One aspect of the Micro Controller, its unique Instruction Stream Handler (ISH), is described here. Recall that in SIMD mode the Micro Controller decodes PCU instructions and broadcasts encoded control signals to the PCU processors. For the reasons discussed above, the PCU processor architecture is based around an arithmetic stack. Having each word of the register file directly addressable led to an instruction stream which is best viewed as an array of four bit elements (nibbles). Off-the-shelf memories are usually based on eight bit bytes, which creates a problem. The ISH must shape the eight bit byte stream from the memory into a usable four bit nibble stream. This can be done in a way which allows instruction fetching to overlap instruction execution.

The ISH looks like a first-in first-out data structure (a queue) implemented in hardware, but there are complications. Each element of the queue is a four bit nibble. Since the memory is eight bits wide each memory fetch adds two items to the queue, not one. Items may be removed from the queue two at a time, e.g., an eight bit opcode, or one at a time, e.g., a register number.

The ISH can be described by a "black box" and a counter. The "black box" has an eight bit input port, four control lines and an eight bit output port. The counter keeps track of how many items are in the queue.

The eight bit input port is connected to the output of the memory. The four control lines (SHFT1, SHFT2, LOAD, and RESET) control the activity of the ISH. A control line being "0" implies no action. If SHFT1 is a "1" one nibble is removed from the queue. If SHFT2 is a "1" two nibbles are removed from the queue. If LOAD is a "1" two nibbles are added to the queue. If RESET is

a "1" all of the nibbles in the queue are removed, i.e., the queue is emptied.

The controls for the counter are set up so that when SHFT1 is a "1" the counter decrements by one, when SHFT2 is a "1" the counter decrements by two, when LOAD is a "1" the counter is incremented by two, and when RESET is a "1" the counter is set to zero.

The eight bit output port may be interpreted as an opcode, a register number, or a constant, depending on what the microprogram expects it to be. The most significant four bits of the eight bit output port are the first item in the queue, the least significant four bits are the second item.

The memory fetch of instruction stream data is overlapped with the current instruction execution. The address for the next byte of the instruction stream is applied to the main memory address lines at all times. When a shift operation will result in the number of nibbles falling lower than three, e.g., a SHFT1 signal when the counter shows there are three nibbles left, or a SHFT2 signal occurs when the counter shows there are four nibbles left, the same clock pulse which executes the shift will also execute a load.

The only occasion which may cause problems is when the instruction being executed needs to fetch an operand from main memory. The microprogram will then set the main memory address lines to the address of the desired operand. While the operand fetch is in progress any shift signal to the ISH should be accompanied by a load inhibit signal which will prevent the queue from being loaded by data which is not the next byte in the instruction stream.

VI. SOFTWARE

A. Introduction

One of the most important decisions made when designing a computer is the choice of instruction set. In the case of PASM two decisions are needed, since two instruction sets are needed: one instruction set for PCU processors in MIMD mode and one instruction set for PCU processors and Micro Controllers in SIMD mode.

The instruction set for MIMD mode is basically a conventional uniprocessor instruction set. Certain instructions for parallel processing, such as an interprocessor data transfer, must be added.

The instruction set for SIMD mode is more complex than the one for MIMD mode. It consists of two major types of instructions: Micro Controller instructions and PCU instructions. The Micro Controller instructions are instructions whose execution primarily affects the Micro Controller. On the other hand, the execution of a PCU instruction primarily affects PCU processors.

The Micro Controller instructions are basically control flow instructions. For example, a jump instruction is a Micro Controller instruction. There are also several Micro Controller instructions which are unique to parallel processing, such as "broadcast," "if any," "if all," and masking instructions (to enable and disable PCU processors under program control) [13,22,33].

Many of the instructions included in the PCU instructions come from the MIMD instruction set. However, some instructions are left out and others are changed. The unconditional control instructions of the MIMD instruction set, such as the jump, are considered a part of the Micro Controller instructions, and are not included in the PCU instructions. The conditional control instructions, e.g., BCS - branch if the carry bit is set, are converted to conditional mask instructions. Recall that the PCU instruction set for MIMD mode contains some instructions special to parallel processing which are included in the PCU instructions.

For the most part, as stated previously, the instructions chosen for PCU processors in MIMD mode will resemble conventional instruction sets. One of the major areas which deviates from conventional instruction sets is the area of subroutine linkage. Several special instructions are included to facilitate subroutine linkage. This is discussed in detail in subsection C.

B. Instruction Set

Several factors strongly affect the selection of an instruction set. (1) The instruction set must be complete enough so that all simple operations use only a few instructions. This gives the instruction set the flexibility needed to meet unforseeable demands efficiently. (2) The instruction set should include instructions which execute common sequences of operations, e.g., decrement a register and branch on some condition (this can be used at the bottom of most loops). These types of instructions will decrease the number of instructions needed for a particular algorithm. This also decreases the effective number of bits per operation, e.g., in the example above two instructions are obtained (decrement a register and branch) for the price of one. (3) The number of bits per instructions should be as small as possible. This may lead to an instruction set that has many different instruction formats. Having many different instruction formats may make the instruction decoding hardware more complex.

(4) Finally, in order for the primary high-level language to be coded efficiently, the instructions needed to support it must be included. In other words, while designing the machine level language the high level language features desired for this machine must be considered. The primary high-level language will reflect the fact that this system is being constructed to process images using parallelism.

C. Subroutine Linkage Convention

A subroutine linkage convention (SLC) is a set of rules implemented by both software and hardware to allow communications between the calling routine and called routine. It is important to have a SLC in mind when choosing an instruction set so that the instructions needed to implement the SLC are included.

An SLC is developed for a single processor environment, and then it is modified slightly to fit an SIMD environment.

The first step in constructing an SLC is to list the properties which one would like it to have.

Parameters - It should be possible to send none, one, or more parameters to a subroutine.

Results - It should be possible for a subroutine to return none, one, or more results to the calling routine. The calling routine should be able to specify which, if any, results it wants from a subroutine.

Recursive Calls - Recursive subroutines should be possible.

Embedded Calls - The result(s) from one call may be the parameter(s) for another call, e.g., COS(ARCSIN(X)).

Minimal Overhead - The SLC should require very little overhead, both in time and space.

Allowing recursive calls implies a stack storage for return addresses, local variables, and parameters. This imposes a structure on the SLC, which leaves only the details of the location of the return address, the parameters, the local variables, and results on the stack to be resolved. This stack, which will reside in main memory, will be called the Subroutine Linkage Stack (SLS).

By convention, register number zero will be the SLS pointer, i.e., it will contain the address of the first free space above the top of the SLS. The JSR (jump to subroutine) instruction automatically puts the return address on the SLS by using register

zero as a stack pointer. The RET (return from subroutine) instruction automatically takes the return address from the SLS using register zero as a stack pointer. Note that since register zero points to the first free space above the top of the SLS for the JSR instruction register zero is used as a pointer, then incremented. For the RET instruction register zero is decremented then used as a pointer.

Ideally the calling routine would push the parameters and the return address onto the SLS. Control would then be transfered to the subroutine entry point. Preceding the return to the calling routine the subroutine would place its results onto the SLS in the locations occupied by the parameters and return address. This scheme is shown in Figure 3.

There are several problems associated with the scheme described above. First, suppose the subroutine is sent fewer, or more parameters than it is expecting. How will the subroutine know where to start putting results? Second, if there are more results than parameters the return address must be moved. Third, suppose the parameters are used directly in the calculation of the results. Then the results must be kept in some temporary locations until all of the results are computed. After all of the results are computed, they need to be moved from their temporary locations to their return locations. Finally, suppose the calling routine does not want all of the results. Then the calling routine must rearrange the results it received. Clearly all of these problems have simple solutions, but they contribute to the overhead of the SLC. In order to reduce the overhead, the SLC shown in Figure 4 should be used.

Suppose the subroutine SUB is called by a statement that looks like:

```
(X,Y)=SUB(P1,P2,...,Pn)[3,2];
```

and the exit for SUB looks like:

```
RETURN(R1,R2,R3,R4,R5);
```

The first statement means call subroutine SUB with parameters P1,P2,...,Pn. The [3,2] at the end of the statement is matched with the variables on the left hand side of the subroutine call statement. This means that when SUB is finished its third result should be put in location X and its second result should be put in location Y. The second statement is an exit from a subroutine which sends R1,R2,R3,R4,R5 as results back to the calling routine.

If the above statements are used then the SLS would look like Figure 5. The "2" and "3" below the return address specify which

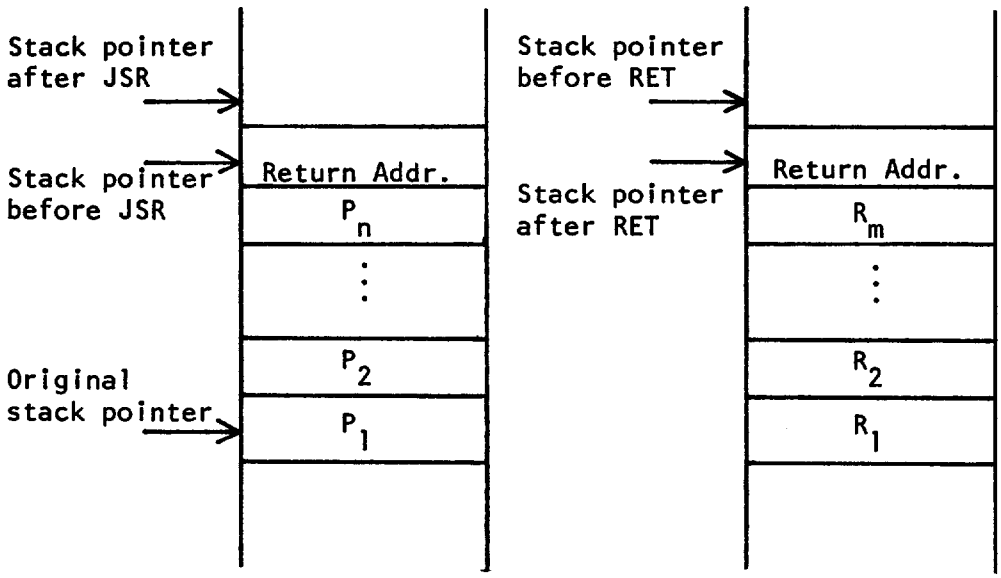


Figure 3: The SLS before and after subroutine call (ideal).

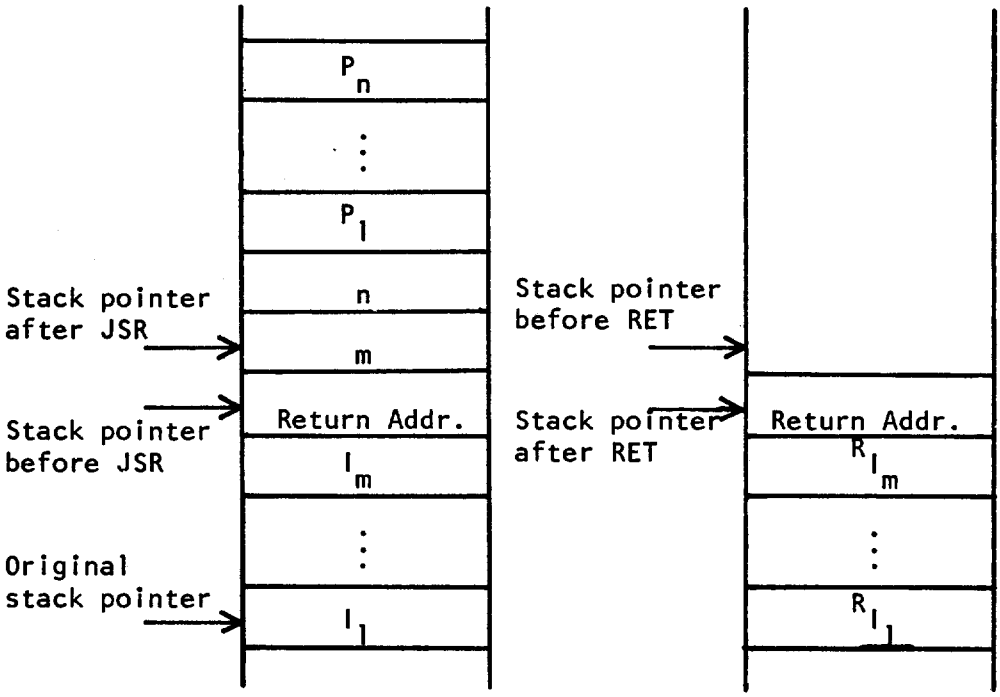


Figure 4: The SLS before and after subroutine call. I_1, ..., I_m are indicer of results desired.

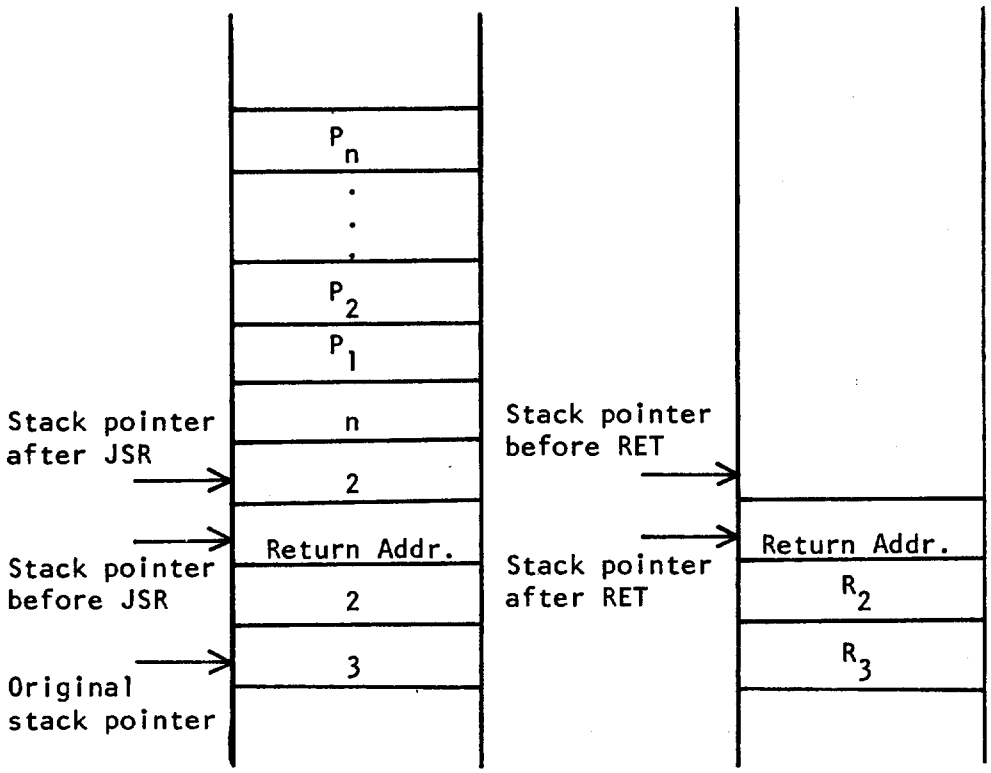


Figure 5: The SLS before and after subroutine call for example given in text.

results are to be returned. The "2" above the return address tells the subroutine how many results are wanted. The "n" above that tells how many parameters were sent.

To make the SLC described above easier to implement some instructions must be added to the basic instruction set. A few are: "MOV (Rn++),#con," which first moves the constant following the instruction to the location pointed to by Rn, then increments Rn; and "MOV (Rn++),addr," which first uses the address following the instruction to fetch an operand which is moved to the location pointed to by Rn, then increments Rn (Rn means register number n). To implement subroutine linkage n would be zero. Other related instructions are in the Appendix.

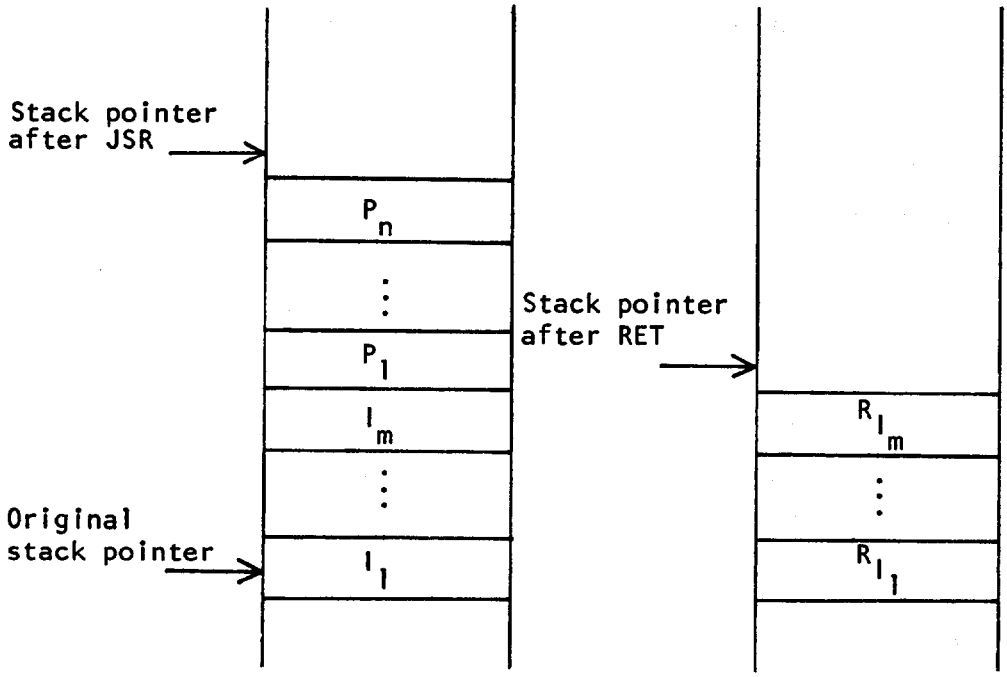
For SIMD mode the SLC should be modified slightly. Each PCU processor will have an SLS as will the Micro Controller. Each PCU processor will push the indices of the the results desired, then push the parameters onto its SLS. The Micro Controller will push all of the control information: the return address, the number of results desired, and the number of parameters sent onto its SLS. This is shown in Figure 6.

D. Instruction Level Simulation

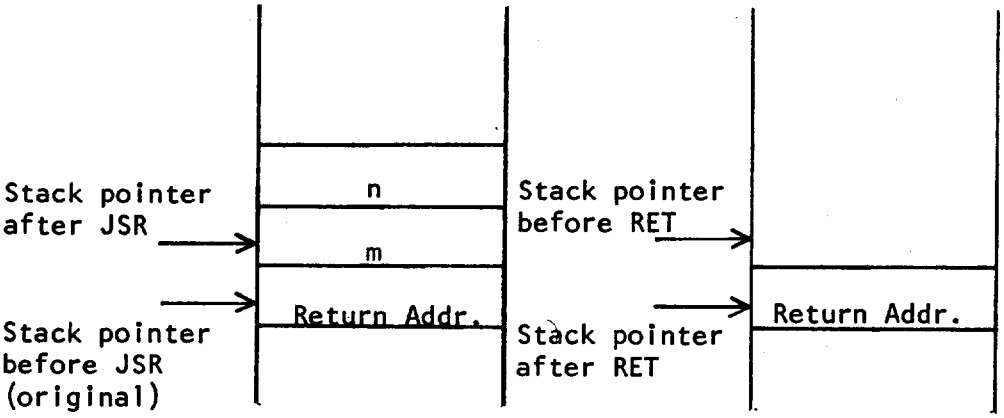
There are two purposes for simulating a computer during the design phase. First, one would like to test for design defects, both in architecture and implementation. Second, a simulator allows work to begin on the operating system and applications programs so that deficiencies in the instruction set may be detected.

In order to test for instruction set deficiencies the simulator need only simulate the architecture of the computer, as opposed to the implementation of the computer. Simulating the architecture means that the simulator need only interpret the instruction and execute it using any operations available without taking into account the exact hardware required to do the task, i.e., all of the hardware is treated as a "black box" which when given specified inputs gives specified ouputs. If the simulation were done on the implementation level then the hardware in the "black box" would be included, i.e., the simulation would be done at the gate level.

The simulator should be able to measure dynamic opcode frequencies. The dynamic statistics which should be collected are: simple opcode frequency, average instruction size, average number of memory reads and writes, execution distance between successful jumps, and relative frequency of successful branches versus unsuccessful branches.



A PCU processor SLS.



A Micro Controller SLS

Figure 6: A PCU processor and Micro Controller SLS before and after subroutine call.

An auxilary program to collect static opcode frequencies is also required. The auxilary program should take as its input either the source file or the object file. The static statistics desirable are: opcode frequency and opcode pair frequency [34].

As pointed out by [35], when optimizing an instruction set one should take in to account the high level language which will be run on the architecture under consideration. This means that when statistics are collected they should be collected on compiled programs rather than assembled programs.

The high level instruction set for an SIMD machine should include many vector and matrix operations. This fact is illustrated by [36] in their attempt to design a language for the CDC STAR-100. To perform a matrix multiplication two (nested) loops are required. Matrix multiplication should be a single instruction in any high level language. To insure the efficiency of a high level language the instruction set must reflect the language desired.

The instruction set listed in the Appendix will be used as a starting point for a high level simulator. The simulation results will provide the information necessary to modify the instruction set to obtain a more efficient set. A variety of image processing and pattern recognition problems will be programmed for these simulations and evaluations.

VII. CONCLUSIONS

A partitionable SIMD/MIMD system, PASM, consisting of a System Control Unit, a Memory Management System, Micro Controllers, and a Parallel Computation Unit was described.

The Micro Controllers and PCU processors are constructed from high speed user microprogrammable bit-slices. The flexibility of the bit-slices allows the Micro Controller to send encoded control signals to its PCU processors and allows the PCU processors to operate in either SIMD mode or MIMD mode.

The PCU processor architecture is based on conventional stack machines, but the two major disadvantages of stack machines are avoided. The instruction stream is nibble oriented in order to be more efficient.

The nibble oriented instruction stream leads to a memory interface problem which is solved by the ISH. The ISH not only solves the interface problem, but it also allows instruction fetching to overlap instruction execution.

Some goals of the instruction set were presented along with a flexible subroutine linkage scheme. The instruction set will be tested using a high level simulator which will gather the statistics mentioned. This will allow the instruction set listed in the Appendix to be improved by adding instructions in areas where it is weak and by removing instructions that are unnecessary.

All of the problems involved in designing the software and hardware of PASM are strongly interrelated. It is the consideration of these interrelations and the coordinated design of the system hardware and software which will aid in producing a machine that is both a versatile and efficient parallel computational tool for image processing and related fields.

REFERENCES

[1] S. Ruben, et. al., "Application of a parallel processing computer in LACIE," 1976 Int'l. Conf. on Parallel Processing (Aug. 1976), pp. 24-32.

[2] M. J. Flynn, "Very high-speed computing systems," Proc. of the IEEE, Vol. 54 (Dec. 1966), pp. 1901 -1909.

[3] H. S. Stone, "Parallel Computers," in Introduction to Computer Architecture, H. S. Stone, editor, Science Research Associates, Chicago, Illinois (1975), pp. 327-355.

[4] G. Barnes, et. al, "The Illiac IV computer," IEEE Trans. Comput., Vol. C-17 (Aug. 1968), pp. 746-757.

[5] W. J. Bouknight, et. al, "The Illiac IV system," Proc. of the IEEE, Vol. 60 (Apr. 1972), pp. 369-388.

[6] K. E. Batcher, "The multidimensional access memory in STARAN," IEEE Trans. Comput., Vol. C-26 (Feb. 1977), pp. 174-177.

[7] K. E. Batcher, "The flip network in STARAN," 1976 Int'l. Conf. on Parallel Processing (Aug. 1976), pp. 65-71.

[8] W. A. Wulf and C. G. Bell, "C.mmp - a multi-mini-processor," FJCC (Dec. 1972), pp. 765-777.

[9] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm*: a modular, multimicroprocessor," in A Collection of Papers on Cm*: A Multi-Microprocessor Computer System, Carnegie-Mellon University, Dept. of Computer Science, Technical Report (Feb. 1977).

[10] J. M. Vocar and R. O. Faiss, "Warp processing using STARAN," 1977 Conf. on Picture Data Description and Management (Apr. 1977), pp. 68-75.

[11] B. Kruse, "A parallel picture processing machine," IEEE Trans. Comput., Vol. 22 (Dec. 1973), pp. 1075-1087.

[12] H. J. Siegel, "Preliminary design of a versatile parallel image processing system," Third Biennial Conf. on Computing in Indiana (Apr. 1978), pp. 11-25.

[13] H. J. Siegel, P. T. Mueller, and H. E. Smalley, "Control of a partitionable multimicroprocessor system," 1978 Int'l.

Conf. on Parallel Processing (Aug. 1978).

[14] G. J. Lipovski, "On a varistructured array of microprocessors," IEEE Trans. on Comput., Vol. C-26 (Feb. 1977), pp. 125-138.

[15] G. J. Nutt, "Microprocessor implementation of a parallel processor," Fourth Annual Symp. on Computer Architecture (Mar. 1977), pp. 147-152.

[16] M. C. Pease, "The indirect binary n-cube microprocessor array," IEEE Trans. Comput., Vol. C-26 (May 1977), pp. 458-473.

[17] C. V. Ramamoorthy, T. Krishnarao, and P. Jahanian, "Hardware software issues in multimicroprocessor computer architectures," First Annual Rocky Mountian Symp. on Microcomputers (Aug. 1977), pp. 73-99.

[18] H. Sullivan, T. R. Bashkow, and K. Klappholz, "A large scale homogeneous, fully distributed parallel machine," Fourth Annual Symp. on Computer Architecture (Mar. 1977), pp. 105-124.

[19] H. J. Siegel, "Analysis techniques for SIMD machine interconnection networks and the effects of processor address masks," IEEE Trans. Comput., Vol. C-26 (Feb. 1977), pp. 153-161.

[20] H. J. Siegel, "Single instruction stream - multiple data stream machine interconnection network design," 1976 Int'l. Conf. on Parallel Processing (Aug. 1976), pp. 273-282.

[21] H. J. Siegel, "The universality of various types of SIMD machine interconnection networks," Fourth Annual Symp. on Computer Architecture (Mar. 1977), pp. 70-79.

[22] H. J. Siegel, "Controlling the active/inactive status of SIMD machine processors," 1977 Int'l. Conf. on Parallel Processing (Aug. 1977), pg. 183.

[23] J. F. Bogdanowicz and H. J. Siegel, "A partitionable multi-microprogrammable-microprocessor system for image processing," 1978 IEEE Computer Society Workshop on Pattern Recognition and Artificial Intelligence (Apr. 1978), pp. 141-144.

[24] H. J. Siegel and S. D. Smith, "Study of multistage SIMD interconnection networks," Fifth Annual Symp. on Computer Architecture (Apr. 1978), pp. 223-229.

[25] E. Lowe, "A 16-bit microcomputer for missle guidance and control applications," 1977 Joint Automatic Control Conf. (Jun. 1977), pp. 17-21.

[26] G. A. Blaauw, Digital System Implementation, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1976), pp. 207-208.

[27] R. D. Arnold, and G. J. Nutt, The Architecture of a Multi Associative Processor, University of Colorado, Department of Computer Science, Technical Report #CU-CS-070-75 (Oct. 1976).

[28] D. M. Bullman, "Stack computers: an introduction," Computer, Vol. 10 (May 1977), pp. 18-28.

[29] R. P. Blake, "Exploring a stack architecture," Computer, Vol. 10 (May 1977), pp. 30-39.

[30] W. M. McKeeman, "Stack computers," in Introduction to Computer Architecture, H. S. Stone, editor, Science Research Associates, Chicago, Illinois (1975), pp. 281-312.

[31] Advanced Micro Devices, The AM2900 Family Data Book, Advanced Micro Devices (1976).

[32] R. Grishman, Assembly Language Programming for the Control Data 6000 Series and the Cyber 70 Series, Algorithmic Press, New York, New York (1974).

[33] H. J. Siegel, Masking Schemes for Determining the Active/Inactive Status of Single Instruction Stream - Multiple Data Stream Machine Processors, Purdue University, School of Electrical Engineering, Technical Report TR-EE 77-25 (May 1977).

[34] B. Peuto and L. Shustek, "Current issues in the architecture of microprograms," Computer, Vol. 10 (Feb. 1977), pp. 20-25.

[35] D. Allison, "A design philosophy for microcomputer architectures," Computer, Vol. 10 (Feb. 1977), pp. 35-41.

[36] V. Basili and J. Knight, "A language design for vector machines," SIGPLAN Notices, Vol. 10 (Mar. 1975), pp. 39-43.

APPENDIX

This following is a list of "assembly instructions" which are desirable in an SIMD environment. Due to the large number of instructions only a subset of the instructions may be implemented.

The following conventions are used:
Rn represents register number n;
i represents one, two, four, or eight;
j can be replaced by: SI (single precision integer), DI (double precision integer), SF (single precision floating point), or DF (double precision floating point).
k can be replaced by S (single precision integer) or D (double precision integer).
m can be replaced by: B (byte), SI(single precision integer), DI(double precision integer), SF(single precision floating point), or DF (double precision floating point).
Instructions are PCU instructions unless otherwise indicated (see section VI).

Arithmetic Instructions (Operates on top of stack)

ADDj	add two type j numbers.
SUBj	subtract one type j number from another
ADDCj	add the carry and two type j numbers
SUBCj	subtract the carry and one type j number from another
NEGj	take two's complement of a type j number
INCj	increment a type j number
DECj	decrement a type j number
MULj	multiply two type j numbers together
DIVj	divide one type j number by another

Boolean Instructions (Operates on top of stack)

ANDk	logical and two type k numbers together
ORk	logical or two type k numbers together
XORk	logical exclusive-or two type k numbers together
NOTk	take one's complement of a type k number

Shift Instructions (Operates on top of stack)

ASLk		arithmetic shift left a type k number one bit
ASRk		arithmetic shift right a type k number one bit
ASLk	n	arithmetic shift left a type k number n bits
ASRk	n	arithmetic shift right a type k number n bits
ROLk		rotate a type k number left one bit
RORk		rotate a type k number right one bit
ROLk	n	rotate a type k number left n bits
RORk	n	rotate a type k number right n bits
ROLCk		rotate a type k number left one bit through carry
RORCk		rotate a type k number right one bit through carry
ROLCk	n	rotate a type k number left n bits through carry
RORCk	n	rotate a type k number right n bits through carry

Stack Instructions

PSHi	#c	push i byte constant following instruction onto register stack
PSHi	addr	push an i byte item onto the register stack using the next word as an address for the item
PSHi	(Rn)	push an i byte item onto the register stack using Rn as an address for the item
PSHi	(Rn+c)	push an i byte item onto the register stack using Rn+c as an address for the item
PSHi	(Rn++)	push an i byte item onto the register stack using Rn as address for item, then add i to Rn
PSHi	(Rn--)	push a i byte item onto the register stack using Rn as address for item, then subtract i from Rn
PSHi	(Rn+Rm)	push an i byte item onto the register stack using Rn+Rm as an address for the item
POPi	addr	pop an i byte item off of the register stack using the next word as the destination address
POPi	(Rn)	pop an i byte item off of the register stack using the contents of Rn as the destination address
POPi	(Rn+c)	pop i byte item off register stack using sum of next byte and contents of Rn as destination
POPi	(++Rn)	increment Rn by i then pop i byte item off register stack using contents of Rn as destination
POPi	(--Rn)	decrement Rn by i then pop i byte item off register stack using contents of Rn as destination
POPi	(Rn+Rm)	pop i byte item off register stack using sum of contents of Rn and Rm as destination
SWB		swap bytes of top word of register stack
SWW	n,m	swap the top n words with the m words below
DUP	n	duplicate top n words of the register stack
DEL	n	delete the top n words of the register stack

```
PSHSW           push Status Word
POPSW           pop Status Word

PSHZi           pushi bytes of zeros onto register stack
PSH     Rn      push Rn onto the stack

CLR     Rn      set Rn to zero
POP     Rn      pop top of stack into Rn

LDSP    #c      load Register Stack Pointer with 1 nibble constant
                    following the instruction
STSP    addr    store Register Stack Pointer

POPTC           load Transfer Control Register from register stack
TRANS           interprocessor transfer of data
```

Compare Instructions

```
CMPm            compare the two type m numbers on register stack
CMPm    #c      compare a type m number on the register stack with
                    the constant following the instruction
CMPm    addr    compare a type m number on the register stack with
                    an item using the next word as an address
```

Register Instructions

```
LD      Rn,#c   load Rn with 2 byte constant following instruction
LD      Rn,addr load Rn with a 2 byte item using the next word as
                    an address for the item
ST      Rn,addr store Rn using next word as an address

INC     Rn      increment Rn
DEC     Rn      decrement Rn

ADD     Rn,#c   add 2 byte constant following instruction to Rn
ADD     Rn,addr add a 2 byte item to Rn using the next word as an
                    address for the item
SUB     Rn,#c   subtract the 2 byte constant following the in-
                    struction to Rn
SUB     Rn,addr subtract a 2 byte item from Rn using the next word
                    as an address for the item

CMP     Rn      compare Rn with the top of the stack
CMP     Rn,#c   compare Rn with the 2 byte constant following the
                    instruction
CMP     Rn,addr compare Rn with a 2 byte item using the next word
                    as an address for the item
```

CMP	Rn,Rm	compare Rn with Rm

Memory Instructions

MOVi	(Rn++),#c	move the i byte constant following the instruction to memory using Rn as an address, then increment Rn by i
MOVi	(Rn++),addr	move an i byte item, using the next word as an address for the item, to memory using Rn as an address, then increment Rn by i
MOVi	(Rn++),(Rm+c)	move an i byte item, using register m added to the 1 byte constant (c) following the instruction as the address for the item, to memory using Rn as an address, then increment Rn by i
MOVi	addr,(--Rn)	decrement Rn by i, then move an i byte item, using the next word as an address for the item, to memory using Rn as an address
MOVi	(Rm+c),(--Rn)	decrement Rn by i, the move a i byte item, using register m added to the 1 byte constant (c) following the instruction as the address for the item, to memory using Rn as an address

Mask Instructions

SCCC	set Condition Flip Flop (CFF) if carry clear
SCCS	set CFF if carry set
SCVC	set CFF if overflow clear
SCVS	set CFF if overflow set
SCGT	set CFF if greater than zero
SCGE	set CFF if greater than or equal to zero
SCEQ	set CFF if equal to zero
SCNE	set CFF if not equal to zero
SCLT	set CFF if less than zero
SCLE	set CFF if less than or equal to zero
LDA	load Accumulator Flip Flop (AFF) with CFF
LDC	load CFF with AFF
NOTC	complement contents of CFF
ANDA	load AFF with logical and of AFF and CFF
ORA	load AFF with logical or of AFF and CFF
WPSH	"where" push of CFF onto the Conditional Mask Stack (CMS)
WEPSH	"where elsewhere" push of CFF onto the CMS
CMPOP	pop off top element of CMS and clear CFF

ICMS		initialize CMS

PE Address Mask Instrucions

(see section VI, subsection A, and [13,22])

PMSK	#c	decode following five nibble "postive" PE address mask and load it into Mask Vector Register (MVR)
NMSK	#c	same as above except "negative" mask
SMSK	Rn	store MVR in Rn through Rn+3
LMSK	Rn	load MVR with Rn through Rn+3
ANDM	Rn	logical and MVR and Rn through Rn+3, result in Rn through Rn+3
ORM	Rn	same as above for logical or
NOTM	Rn	complement Rn through Rn+3

Control Instructions (Micro Controller Instructions)*

NOP		null (or no) operation
HLT		halt
DBNZ	Rn,addr	decrement Rn, branch if not zero
IBNZ	Rn,addr	increment Rn, branch if not zero
BRA	addr	branch always
JMP	addr	jump to addr
JMP	Rn	jump using Rn
BRS	addr	branch to subroutine
JSR	addr	jump to subroutine
JSR	Rn	jump to subroutine using Rn
RET		return from subroutine
IFANY	addr	if any PCU CFF is set branch to addr
IFALL	addr	if all PCU CFFs are set branch to addr

* NOTE: Rn here indicates register number n in the Micro Controller. Branch means addr is stored as an offset relative to the program counter. The offset may be positive or negative, but must be representable as an eight bit two's complement number.

Micro Controller Register Instructions

CCLR	Rn	clear Rn
CLD	Rn,#c	load Rn with the 2 byte constant following the instruction
CLD	Rn,addr	load Rn with a 2 byte item using the next word as an address
CST	Rn,addr	store Rn using next word as an address
MOV	Rn,Rm	load Rn with contents of Rm
CINC	Rn	increment Rn
CDEC	Rn	decrement Rn
CADD	Rn,Rm	add Rn to Rm and put result in Rn
CSUB	Rn,Rm	subtract Rm from Rn and put result in Rn
CMUL	Rn,Rm	multiply Rn by Rm and put result in Rn through Rn+1
CDIV	Rn,Rm	divide Rn through Rn+1 by Rm put result in Rn and remainder in Rn+1