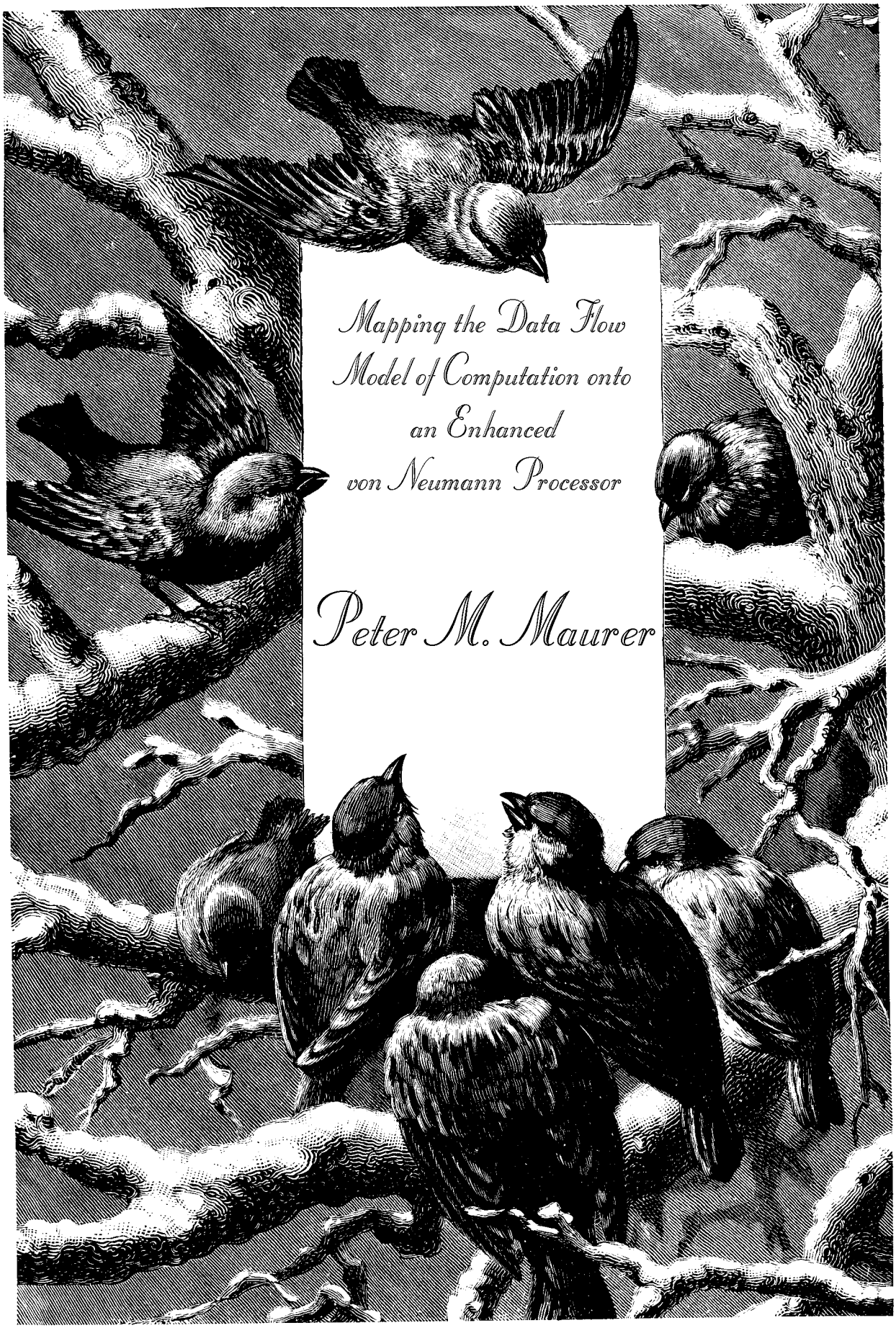


# *Mapping the Data Flow Model of Computation onto an Enhanced von Neumann Processor*

*Peter M. Maurer*

# MAPPING THE DATA FLOW MODEL OF COMPUTATION INTO AN ENHANCED VON NEUMANN PROCESSOR*

Peter M. Maurer

Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

Abstract -- The SAM architecture is an enhanced von Neumann processor that contains inexpensive features for supporting data flow style of parallelism. The architecture gets is name from the basic instructions for supporting parallelism, Split and Merge. It is shown that these instructions can be used to implement the parallel structure of an arbitrary acyclic data flow graph. Features for supporting dynamic parallelism and multiple run-time environments are presented. Implementation issues for supporting instruction execution and the handling of faults and interrupts ar also discussed.

## 1. Introduction.

One of the main focuses of current research in computer architecture is the design of hardware organizations that support the parallel execution of instructions (see [1] for several examples.). Data flow parallel architectures continue to receive a great deal of attention [3] [4]. In a data flow architecture an instruction may execute as soon as its operands become available, permiting a degree of parallelism bounded only by the flow of data between instructions. In spite of their intuitive appeal, data flow machines have been slow to reach the marketplace, and it appears that much work must be done to make data flow machines competitive with other parallel architectures[5].

In spite of the objections raised by [5], data flow is appealing, and it is reasonable to ask whether it can be adapted to a more conventional architecture. The approach taken in this paper is to start with a von Neumann processor, and by adding features, enable it to execute programs in the highly parallel manner characteristic of data flow machines. The objective is to develop inexpensive parallel architectures that can exploit parallelism without sacrificing compatiblity with existing software. Compatiblity with existing software is important because it represents an enormous investment for the computer user, and it is necessary to preserve this investment. The architecture presented in this paper is called the SAM architecture for reasons that will be explained in section 2. The features of this architecture are similar to those found in multi-threading machines [2][6][7], but are somewhat simpler. In spite of this, the features presented here can be used to program some of the more complicated features found in other machines.

Section 2 describes the architecture and the primitive features for supporting parallelism. Section 3 shows how the architecture supports arbitrarily complex static parallelism. Section 4 introduces features that support dynamic parallelism and multiple run-time environments. Section 5 discusses implementation issues, and section 6 draws conclusions.

## 2. The Basic Architectural Features.

The SAM architecture supports arithmetic and logical instructions as well as conditional and unconditional jumps. Initially it is assumed that conditional jumps perform both a comparison and a conditional jump, and that all instructions are memory to memory. It will be possible to relax these restrictions later. Two addressing modes are supported, the full-address mode which provides direct addressing using full-width addresses, and the short-address mode which requires fewer address bits than

---

* This research was supported by the university of South Florida Center for Microelectronics Design and Test.

the full-address mode and is used to access the low-address portion of memory. Short addresses may be either direct or indirect. The portion of memory addressable in the short-address mode is called the short-address space, and will be described more fully in section 4. In some implementations, portions of the short-address space may be mapped to registers or a high-speed cache. There are no programmer-addressable registers.

The SAM architecture is a MIMD machine that allows the degree of parallelism to vary with time. Two types of parallelism are supported, static parallelism where the degree of parallelism is determined at compile time, and dynamic parallelism where the degree of parallelism depends in part on the data being processed. There is no upper limit on the degree of parallelism. The features for supporting parallelism are motivated by the differences between the execution histories of sequential machines and those of data flow machines. On a sequential machine each instruction has exactly one predecessor and exactly on successor, while on a data-flow machine, each instruction has several predecessors and successors. In order to support parallelism whose degree varies with time, it is necessary to have instructions that have more than one prececessor and successor. In the SAM architecture the "split" instruction has one predecessor and two successors, while the "merge" instruction has two predecessors and one successor. These instructions form the core around which the rest of the architecture is designed, hence the name "SAM" for "Split And Merge." The split instruction has the format of an unconditional jump, one successor is the branch target, while the other is the following instruction. The split instruction creates two independent instruction streams. The merge instruction has one operand that is normally initialized to zero. When its operand is zero, the merge instruction sets it to 1 and terminates the execution of the current instruction stream. Otherwise the it sets its operand to zero and contiues execution with the next instruction. The merge instruction operates atomically on its operand. Figure 1 shows how the combination of split and merge can be used to evaluate the statement "e=(a+b)+(c+d)." with the sub-expressions evaluated in parallel.

```
1        split   L1
2        add     a,b,t1
3        merge   x
4        jump    L2
5    L1: add     c,d,t2
6        merge   x
7    L2: add     t1,t2,e
```

Figure 1. Parallel Evaluation of e=(a+b)+(c+d).

Most of the instructions Figure 1 are self explanatory. The labels "a," "b," "c," "d," and "e" are the variables named in the expression, while the labels "t1" and "t2" are temporary variables. The label "x" is a temporary variable that is initialized to zero. The split instruction on line 1 causes the add instructions on lines 2 and 5 to be executed in parallel. The first two operands of these instructions are added and the result is placed in the third operand. In this example, a separate merge instruction is placed at the end of each instruction stream. An equivalent way to program this example would be to omit the merge instruction on line 3 and move the label "L2" from line 7 to line 6.

Figure 1 shows that the split instruction adds three or four instructions of overhead to each stream (the two merge instructions cannot execute in parallel). If the end of both streams is moved to line 6, the overhead can be reduced to three instructions per stream. Assuming that all instructions execute in one time unit, each stream must be at least four instructions long for there to be any benefit from the parallelism. introduced by a split. However, if the split and merge instructions execute quickly as compared to the other instructions, the length of the stream could be reduced without negating the beneficial effects of parallelizing the code.

At this point it is assumed that all instruction streams execute in the same environment, which restricts the way code can be parallelized. Methods for removing these restrictions will be discussed in section 4.

3. Translating Data Flow Code to Split/Merge Streams.

The translations presented in this section are based on the intermediate form of data flow code presented in [8]. A program is represented as a combinatorial expression of the form (C op0 ... opn), where C is an Abdali combinator[9], and op0 through opn are the operands of the combinator. An operand may be a constant or another expression. The combinator may be of the form $B^n_m$ , $I^n_m$ , or $K_m$ . If it is of the form $B^n_m$ then op0 will be the name of an instruction and op1 through opn will supply the operands of the instruction. If the combinator is of the form $I^n_m$ , op0 through opn will not be present, and if it is of the form $K_m$ , op0 will be a constant and op1 through opn will not be present. For any expression, all subscripts will have the same value . A subscript of m signifies that m inputs are need to evaluate the expression. Combinators of the form $B^n_m$ are used to evaluate n-input functions, those of the form $K_m$ are used to introduce constants into an expression, while those of the form $I^n_m$ are used to select the nth input from a list of m inputs. For example, the expression x+y+1 can be translated into the expression ($B^2_2$ + ($B^2_2$ + ($I^1_2$ ) ($I^2_2$ ) ($K_2$ 1)). As was pointed out in [8] these expressions are simply linearized forms of data flow graphs.

Because the language presented in [8] does not contain conditionals loops or assignments, no provision was made for handling them. In addition because the language is applicative, no provision was made for handling sets of independent statements that communicate by side effects. To make the results of this section as general as possible, it is necessary to introduce the functions "if," "while," "assign," and "set" to handle conditionals, loops, assignments, and sets of independent statements. In addition, the new combinator $Q^n_m$ is introduced to distinguish between sets of statements that are independent, and those that have data dependencies. The combinator $Q^n_m$ is mathematically equivalent to $B^n_m$ , but when code is generated for the expression ($B^n_m$ x0 x1 ... xn) each of the expressions will be evaluated in parallel. When code is generated for the expression ($Q^n_m$ x0 x1 ... xn), the expressions x1 through xn will be evaluated serially. The $Q^m_n$ combinator can also be used near the "leaves" of an expression if it is necessary to place a lower bound on the length of independently executed instruction streams. To illustrate the use of the $Q^n_m$ combinator, consider the usual form of the combinatorial expression for e=(a+b)+(c+d) which is ($B^2_5$ assign($I^1_5$ )($B^2_5$ +($B^2_5$ +($I^2_5$ )($I^3_5$ ))($B^2_5$ +($I^4_5$ )($I^5_5$ )))). When code is generated for this expression, the code-generation algorithm will create two independent instruction streams to evaluate the sub-expressions (a+b) and (c+d) in parallel. The two streams will be merged to complete the final addition. The following slight modification in the combinatorial expression will cause the three additions to be executed serially, ($B^2_5$ assign($I^1_5$ )($Q^2_5$ +($B^2_5$ +($I^2_5$ )($I^3_5$ ))($B^2_5$ +($I^4_5$ )($I^5_5$ )))).

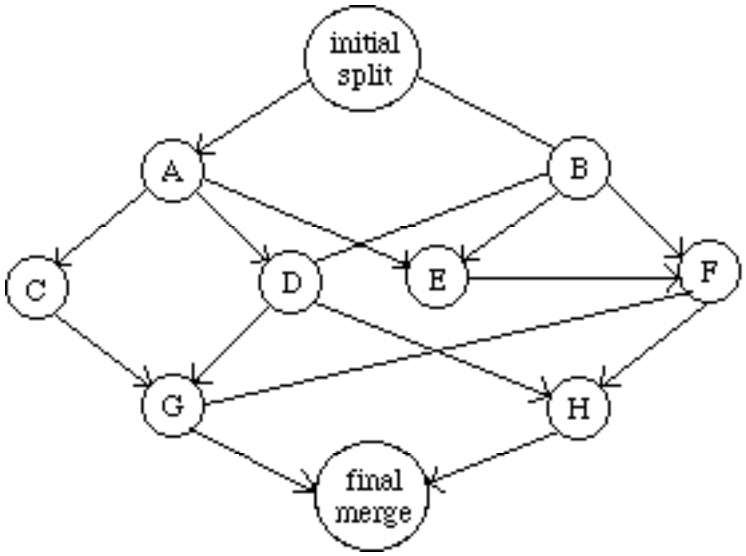
Code can be generated for combinator expressions in a straightforward manner. A separate instruction stream is created to evaluate each operand of a
B-type combinator, while the operands of a Q-type combinator are evaluated serially. The operands of "if" and "while" functions are always executed serially, although parallelism within the evaluation of the

operands is not precluded. When code is generated for the body of a loop, it begins and ends as a single instruction stream, which prevents the iterations of a loop from getting out of sync.

If expressions of the form ($Q_m^n$ set x1 ... xn) are translated using the most straightforward algorithm, parallelism will be lost. For example, consider the following two-statement sequence.

```
a=b+c
e=(d+f)*a
```

The subexpressions "b+c" and "d+f" can be executed in parallel, but if the statements are serialized due to the data dependency, this parallelism will be lost. A more sophisticated method of translating these functions is needed. The procedure is easier to visualize if it is assumed that the set of statements has been described as a data flow graph. Each node in the graph represents one statement in the set. (Complex expressions have been broken into separate statements.) All arcs that do not begin and end on a node are omitted, along with all duplicate arcs. The result is a directed acyclic graph with one or more source nodes and one or more sink nodes. Assume that there are j source nodes and k sink nodes. Since the source nodes use only those data items that are assumed to be present before the execution of the set begins, they can all be executed in parallel. The code for the set begins with a j-label msplit instruction, which is the single predecessor of each source node. Similarly, the code for the set ends with a k-label mmerge instruction, which is the single successor of every sink node. Msplit and mmerge are standardized sequences of instructions that create and merge an arbitrary number of streams. An n-label msplit acts as an n-way branch, while an n-label mmerge acts as an n-label branch-target. Their construction is straightforward. The code for a node with m predecessors and n successors begins with an m-label mmerge instruction and ends with an n-label msplit instruction. In each case the labels on the msplit instruction match the labels on the mmerge instruction of the successor nodes. The nodes of the data flow graph can represent arbitrarily complex expressions and are not restricted to individual instructions. A node may have a high degree of internal parallelism as long as it has a single entry and a single exit. This method of translating set functions allows arbitrary acyclic data flow graphs to be implemented using the split and merge instructions. An example of this procedure is illustrated in Figure 2.




```
msplit      LA1,LB1
mmerge      LA1
-----A-------
```

```
msplit      LC1,LD1,LE1
mmerge      LB1
-----B-------
msplit      LD2,LE2,LF2
mmerge      LC1
-----C-------
msplit      LG1
...
```

Figure 2. Data Flow Parallelism with Split and Merge

4. Creating Multiple Environments.

Although a high degree of parallelism can be realized with the split instruction in a single environment, multiple environments are needed to support dynamic parallelism and shared subroutines. One method of supporting multiple environments would be to have severa; data and address registers that are replicated for each instruction stream. Such a mechanism is used in some multiprocessors, but since not all independent instruction streams require separate environments, it is desirable to separate the function of creating an instruction stream from the function of creating a new environment. Recall that the SAM architecture provides a short-addressing mode that is used to access the short-address space. Associated with each instruction stream is a register called the prefix register that contains the location of the short-address space. The prefix register is assumed to contain the high-order address bits of the short-address space, with the low order bits being supplied by the instruction. The instructions "readp" and "writep" are provided to read and write the prefix register. Instruction streams that require separate environments may use these instructions to create a new short-address spaces. The current value of the prefix register is replicated on a split which causes the two independent streams to execute in the same environment. Since the two streams will generally begin execution at two different points in memory, distinct environments can be created for each stream. The merge instruction does not affect the contents of the prefix register.

Although the prefix register can be used to solve the problem of dynamic parallelism and the problem of calling the same subroutine in two different instruction streams, the stack-based addressing scheme for passing arguments and saving return addresses cannot be used in a multi-threading environment without elaborate support mechanisms or rigid controls on how it is used. In a multi-thread environment it is not possible to predict when memory for one set of arguments will be deallocated with respect to the memory for another sets. In the SAM architecture, allocation of space is made the responsibility of either the support software or the compiler, because the most efficient method for doing so depends on the problem being solved,. Efficient implementation of the basic features of the architecture should allow many different allocation schemes to be programmed efficiently.

To illustrate how the prefix register can be used to achieve dynamic parallelism, consider the code illustrated in Figures 3 and 4. It is assumed that each instruction stream requires an environment of size $2^x$ , and that $2^i$ instruction streams are to be created. In addition to a number of temporary variables, each environment contains its starting address, its size, the number of the current instruction stream, the total number of instruction streams, and a pointer to the parent environment. There is also a word initialized to zero, which will be used as a merge target. Figure 3 illustrates the serial creation of instruction streams, while Figure 4 illustrates the logarithmic creation of instruction streams. The logarithmic initiation of instruction streams operates by creating a single environment of size $2^{x+i}$ and repeatedly splitting it in half until environments of the proper size have been created. In the process the proper number of instruction streams will be initiated.

```
allocate 2^(x+i) bytes;
for j=1 to 2^i
    init env j and make it current;
```

```
        split to shared_code;
    endfor
    for j=1 to 2^i
        exec merge in env j;
    endfor

    shared_code:
        ...
    for j=current_stream to 2^i
        execute merge in environment j;
    endfor;
```

Figure 3. Serial Stream Initiation.

Creating a separate environment for each stream causes the overhead for each stream to be greatly increased. When multiple environments are being used, it may be more convenient to treat the independent streams as individual processes that communicate through a producer/consumer structure as proposed by several others [2]. The simplest way to model the producer/consumer relationship is to follow the producer by the instruction "split x" and precede the consumer by the instruction "x: merge k" where k is a variable that has been initialized to zero. Overruns can be prevented by preceding the consumer by the instruction "y: merge j" where j allocate $2^{x+i}$ bytes;

```
init as one size 2^(x+i) env & make current;
while (env_size > 2^x
    env_size = env_size / 2;
    init new env in second half of current env &
        make current;
    split to x;
    restore parent env
x:
endwhile;
... shared code ...
while (total_streams > 1)
    if (curr_strm_num is even)
        make env at env_adr+env_size current;
    endif;
    exec merge; restore parent env;
    divide curr_strm_num and total_streams by 2;
endwhile;
restore parent env;
```

Figure 4. Logarithmic Stream Initiation.

is a variable that is initialized to one, and following the consumer with the instruction "split y." This scheme will work only if each data item has a single producer and a single consumer. To support multiple producers and consumers, it is necessary to introduce the hardware equivalent of semaphore P and V operations. The P operation is modeled by the "seq" instruction,which has a single operand. If the operand is non-zero when the "seq" instruction will set it to zero and instruction execution continues with the next instruction. If the operand is zero, both it and the program counter for the current stream remain unchanged. The seq instruction operates atomically on its operand,and executes repeatedly until its operand is set to zero. The detrimental effect of the busy wait can be minimized by spinning the seq instruction off into a separate instruction stream. The seq instruction can be used for multiple-producer multiple-consumer problems and other types of synchronization.

5. Implementation Issues

The SAM architecture will be implemented as a shared pipeline similar to that found in the HEP multiprocessor[2]. The number and function of the pipeline stages is not fixed by the architecture, but for definiteness consider the pipeline pictured in Figure 5. This is a typical pipeline augmented with two additional stages to fetch and write stream descriptors. Each stream descriptor contains the current PC for the stream as well as the current prefix register value. The descriptor may contain other items as explained below.

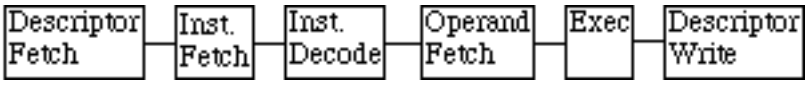



Figure 5. An Augmented Pipeline.

The descriptor fetch stage of the pipeline obtains descriptors from many different sources. In particular, each stage of the pipeline can serve as a source of descriptors, which allows the pipeline to be fully utilized even when only a small number of descriptors exist. When descriptors are fetched from the earlier stages of the pipeline, potential pipeline hazards must be provided for either in the hardware, or by some software scheduling technique.

The descriptor write stage of the pipeline provides for internal buffering for descriptors. When the internal buffer of the descriptor write stage is full and a split instruction creates a new descriptor, several actions can be taken. The descriptor write stage can provide storage management for a circular buffer in some form of backing store, or it can cause a fault to occur when the number of streams reaches a high-water mark. The support software has the choice of suspending the execution of the stream until the number of streams fell below a low-water mark, or of passing the descriptor to a second processor.

To support the simultaneous execution of several processes, each of which may have several instruction streams the SAM architecture provides features for handling interrupts and faults. An interrupt is handled by initiating a new instruction stream in response to an external event. The "return from interrupt" is accomplished by executing a merge instruction with a zero argument. An interrupt vector consists of a pointer to the executable code for handling the interrupt, and a pointer to the short-address space for the interrupt handler.

Since several program faults of the same type may occur simultaneously, it is necessary to have some method of serializing the first portion of the fault handler, which allows the descriptor of the offending instruction stream to be copied into the environment of the fault handler without destroying data that is still needed to process a previous fault of the same type. This problem is solved by adding a global register for masking faults, and a status register to the descriptor of each stream. When a fault occurs and the corresponding fault-type is masked, the descriptor of the offending stream will have a "suspended" bit set in its status register. A descriptor with the suspended bit set propagates through the pipeline without change. When the descriptor reaches the stage where the fault originally occurred, the stage will schedule the fault-handler. if the fault type is now unmasked. A more expensive solution would be to allow the descriptor write stage of the pipeline to queue descriptors waiting for the fault to become unmasked.

At times the support software may need to terminate a process in response to a program fault or other event. Because each process can have many instruction streams active simultaneously, some mechanism is needed to identify and terminate all instruction streams belonging to the terminated process. To solve this problem a process-id, which can be used to identify and terminate instruction streams, is added to the descriptor of each stream. The process-id is copied into the new descriptor when a split instruction is executed but it may be changed by the support software. One way to accomplish this is to combine the

assignment of process-ids with memory management. For example, the process-id could be a pointer to the segment or page table for the process. The memory management hardware could be used to force the termination of instruction streams. Another method is to pass the process-id of a failed process to the descriptor write stage of the pipeline and allow this stage to purge all descriptors with matching process-ids.

The addition of status bits to the stream descriptor permits the implementation of privileged instructions, allows more conventional compare and conditional jump instructions to be used, and allows for local masking of faults in the instruction streams. There are a number of implementation issues that remain to be solved, but these should be readily addressed as work on the SAM architecture progresses.

## 6. Conclusion.

The SAM architecture is the first step in developing an inexpensive method for supporting data flow style parallelism in a von Neumann architecture. The features presented here are intended to be inexpensive to implement, and easy to use by a compiler. Although the split and merge instructions are simple, it has been shown that they can be used to implement arbitrarily complex static parallelism in a single environment. Using the prefix register to create multiple environments, it is possible to implement arbitrarily complex dynamic parallelism at a cost somewhat higher than that for static parallelism. Implementation issues have been discussed that allow for multiple processes as well as interrupt and fault handling. It is hoped that the SAM architecture will soon lead to the development of one or more small inexpensive multiprocessors.

## REFERENCES

1. R. H. Kuhn, D. A. Padua (eds.) "Tutorial on Parallel Processing," IEEE Computer Society Press, Silver Spring Md, 1981.

2. B. J. Smith "Architecture and Applications of the HEP Multiprocessor Computer System," Real Time Signal Processing IV, Proceedings of SPIE, 1981, pp. 241-248.

3. Arvind, D. E. Culler, "Dataflow Architectures," Annual Reviews in Computer Science, 1986,.Vol 1, Annual Reviews Inc., 1986, pp. 225-253.

4. J. B. Dennis, "Data Flow Supercomputers," Computer Vol. 13, No. 11, Nov. 1980, pp. 48-56.

5. D. D. Gajski, D. A. Padua, D. J. Kuck, R. H. Kuhn, "A Second Opinion on Data-Flow Machines and Languages," Computer, Vol. 15, No. 2, Feb. 1982, pp. 58-69.

6. L. M. Pedersen, "Design for MISP: A Multiple Instruction Stream Shared Pipeline Processor," Technical Report CSG-37, Coordinated Science Laboratory, Computer Systems Group, University of Illinois at Urbana-Champaign, 1984.

7. P. C. Trealeven, R. P. Hopkins, P. W. Rautenbach, "Combining Data Flow and Control Flow Computing," The Computer Journal, Vol. 25, No. 2, 1982, pp. 207-217.

8. P. M. Maurer, A, E. Oldehoeft, "The Use of Combinators in Translating a Purely Functional Language into Low-Level Data-Flow Graphs," Computer Languages, Vol. 8, No. 1, 1983, pp. 27-45.

9. S. K. Abdali, "An Abstraction Algorithm for Combinatory Logic," The Journal of Symbolic Logic, Vol. 41, 1976, pp. 222-224.