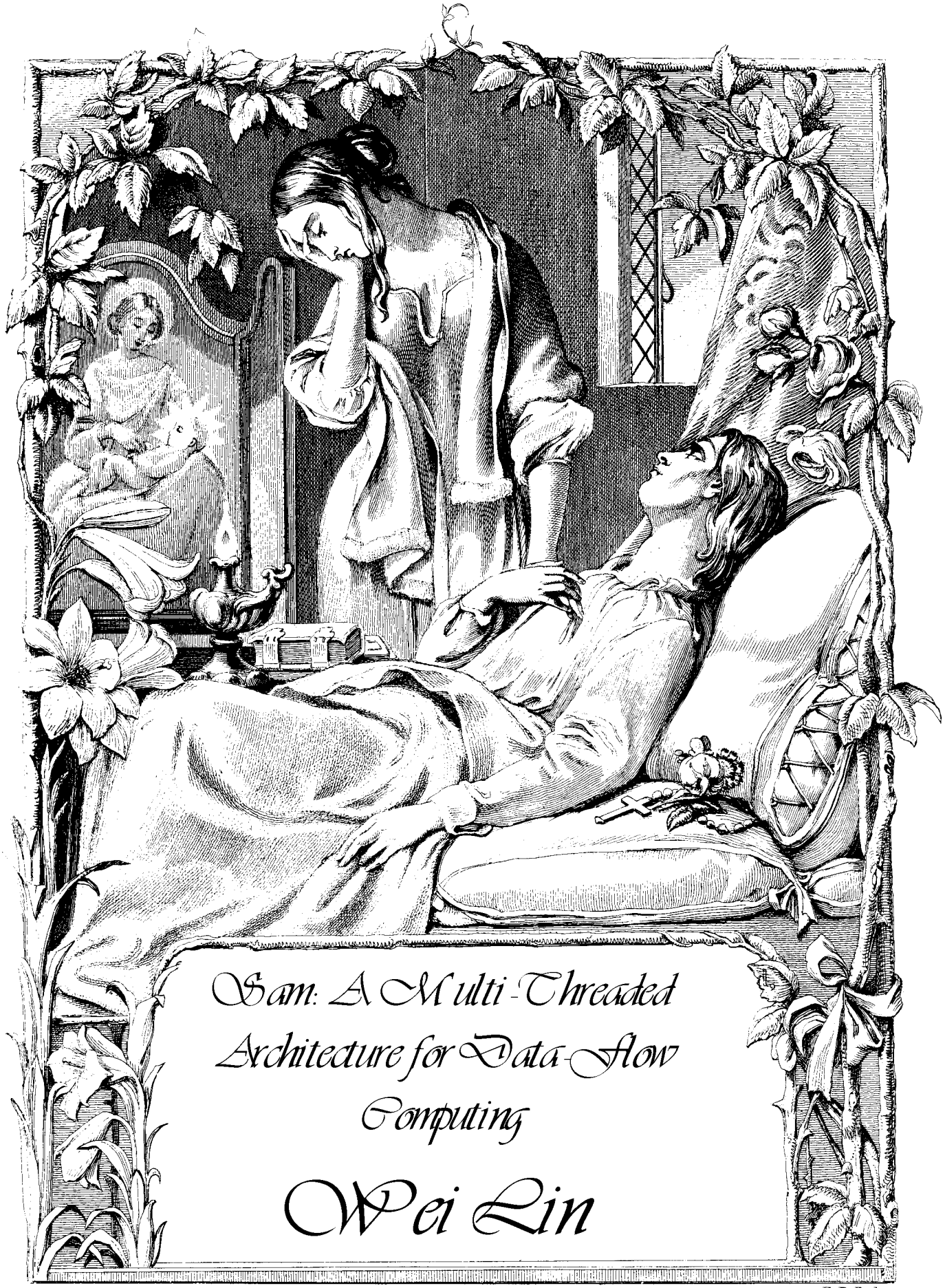


Sam: A Multi-Threaded Architecture for Data-Flow Computing

Wei Lin

D. Maclise, R.A. F. P. Becker.

SAM: A Multithreaded Pipeline Architecture for Dataflow Computing

Wei Lin, Peter M. Maurer

Department of Computer Science and Engineering

University of South Florida

Tampa, FL 33620

* This work was supported in part by the University of South Florida Center for Microelectronics Research.

The SAM Multithreaded Architecture

Wei Lin, Peter M. Maurer
Department of Computer Science and Engineering
University of South Florida
Tampa, Florida 33620
Phone: 813-974-4758

Email: wlin@eggo.csee.usf.edu, maurer@screamer.csee.usf.edu

Abstract

SAM is a multithreaded pipeline architecture which supports multiple instruction threads running on a single shared high speed pipeline. It supports a dataflow-style of programming in which the parallelism in the program is represented as a directed acyclic graph. Because the SAM architecture is basically an extension of the von Neumann architecture, conventional sequential programming is also supported. The SAM architecture resembles a RISC machine with a few additional features. Both experimental data and mathematical analysis show that the SAM architecture can achieve a pipeline efficiency of nearly 100% with minimal overhead.

Keywords: Multithreaded Architecture, Multithreading, Parallel Architecture, Dataflow, Pipeline, RISC

1. Introduction

Over the past several years, there has been considerable effort expended on research into various types of parallel computer architectures, and the algorithms that run on them. One of the most appealing approaches is the dataflow machine[3,7,8], primarily because it provides instruction level parallelism, thus allowing all of the inherent parallelism in an algorithm to be exploited. Furthermore, dataflow machines do not require vectorization as do high-performance pipe-lined processors, they do not force lock-step operation as do SIMD architectures, and do not have the enormous synchronization problems that may occur on a typical MIMD architecture.

Despite the intellectual appeal of dataflow machines, they have not, as yet, enjoyed wide-spread commercial success. When one compares the histories of the dataflow machine and the von Neumann architecture, this is not surprising. The von Neumann architecture has been the subject of intense research and development by thousands of workers for many decades. On the other hand, the dataflow architecture has had a much shorter history, and by comparison, has received the attention of only a relatively small group of experts. This would make no difference if the dataflow machine were an evolutionary development of the von Neumann architecture. But it is not. The dataflow machine represents a revolutionary approach toward computation, and despite the excellent research that has been done so far, will probably require considerably more research and development before it reaches its full potential[4].

The architecture presented in this paper, which is a further development of that presented in reference [1], is designed to be a bridge between the von Neumann architecture and the dataflow machine. This architecture, which is called the SAM architecture for reasons which will be explained below, is an evolutionary development of the von Neumann architecture which has the ability to perform dataflow-like computations. The SAM architecture can support instruction level parallelism within a single process, with very little overhead. Furthermore, the degree of parallelism is controlled by the program, not by the structure of the hardware. However, unlike the dataflow architecture, the SAM architecture requires explicit instructions for controlling the parallelism, and

thus requires more overhead than an ideal dataflow architecture. As will be seen below, however, the number extra of instructions required is quite small.

From the beginning, the SAM architecture was designed to be a general purpose machine, and was intended to be implemented as a single-chip microprocessor. For this reason, the SAM architecture was designed to be implemented using hardware structures that are commonly found in existing von Neumann architectures. Furthermore, considerable thought has been given to such mundane, but necessary, issues such as I/O interrupts, traps, abnormal process termination, and programming structures. Many of these issues present a considerable challenge in an architecture that supports multiple instruction streams, or threads as they are more commonly called, in a single process. Particularly when the creation of a new thread does not involve the participation of the operating system, for example, suppose that it is necessary to forcibly terminate a process containing multiple independent threads. Since the operating system has no knowledge of these threads, finding them and killing them represents a formidable design challenge.

Internally the SAM architecture resembles a conventional pipelined SISD processor. The difference lies in the way the next instruction is selected for execution. Instead of maintaining a single program counter that points to the next instruction to be fetched, the SAM architecture maintains a collection of program counters that are serviced in round-robin order. As will be shown below, if a sufficient number of threads are active, this can result in increased pipeline efficiency because the pipeline does not need to be purged due to a conditional branch instruction, or interlocked due to data dependencies between consecutive instructions. The concept of sharing a single pipeline among several independent instruction streams has appeared in the literature many times[12,9,2]. However existing solutions to the problem of managing such a pipeline generally assume that the pipeline sharing is done at the process level, or at the thread level where the operating system has complete knowledge of every existing thread. Furthermore, some architectures allow a particular process to occupy only a single slot in the pipeline which implies that a minimum number of processes must be active for full pipeline utilization to be achieved. In the SAM architecture, when there are insufficient threads to completely fill the pipeline, no holes will be left in the pipeline.

Thus a thread may have several instructions in the pipeline simultaneously. In fact, the SAM architecture is capable of functioning as a conventional pipelined uniprocessor, running a single thread. Pipeline interlocks are provided to permit the SAM architecture to function correctly when there are fewer than the optimum number of threads.

The paper is organized in six sections. Section 2 describes the basic architectural features. Section 3 introduces the simulation model. Section 4 analyses the performance of this architecture. Section 5 shows some program examples that were run on the simulator. Section 6 draws conclusions.

2. Basic Architectural Issues

To the casual observer, the SAM architecture resembles a conventional von Neumann processor with the addition of a few additional instructions added to support instruction level parallelism. The two most important of theses are the **Split** and **Merge** instructions (hence the name SAM for *Split And Merge*. The Split instruction resembles a conventional conditional branch, but actually has two successors instead of one. After the execution of a Split, execution continues at the branch address *and* continues execution with the instruction following the Split. The Merge instruction is the reverses the operation of the Split instruction. The Merge operation has a single operand which is usually initialized to zero. When the Merge instruction is executed and its operand is zero, it sets the operand to 1, and terminates the execution of the thread that executed it. When its operand is non-zero, it is reset to zero and execution of the thread continues with the instruction following the merge. The split and merge instructions allow the degree of parallelism to vary with time, just as does a dataflow architecture. Figure 1 shows how the combination of **split** and **merge** can be used to evaluate the assignment statement "e=(a+b)+(c+d)".

(place figure 1 here)

The first instruction (split L1) generates a new thread L1 which calculates c+d. As soon as the first instruction finishes the execution, two threads are activated. The original thread (instructions

2 and 3) may finish before thread L1 is terminated, and vice versa. When the first merge instruction (*either* 3 or 6) is executed, the corresponding thread will be terminated. The second merge will lead the execution go to the address provided by the instruction following it (if merge at 3, the following jump instruction will go to 7; if merge at 6, the successor 7 is the next instruction address.). In this way, merge plays a role in synchronization between two threads. Figure 1 shows that the split instruction adds three or four instructions of overhead to each thread. If the end of both threads is moved to line 6, the overhead can be reduced to three instructions per thread. Assuming that all instructions execute in one time unit, each thread must be at least four instructions long for there to be any benefit from the parallelism, introduced by a split.

It is relatively obvious how one goes about creating several streams using a collection of split instructions, and as long as every split has its corresponding merge, programming is relatively simple. However, there are cases where it is convenient to think of the Merge instruction as a gateway which allows only every second thread to pass. In some cases it is convenient to have a gateway through which only every nth thread will pass, where n is some arbitrary number. (For convenience we will call this an n-pass gateway.) As long as n is known beforehand, this is relatively easy to achieve. For example, if we wish to construct a gateway through which only every eighth stream will pass the following set of instructions may be used.

```
GATEWAY:  Merge A
          Merge B
          Merge C
```

By adding additional entry points to this sequence it is possible to construct a gateway that permits every nth thread to pass, where 1¡n¡8. For example the following gateway allows every fifth thread to pass. Four of the threads must enter at point G1, while the fifth must enter at G2. By using similar constructions with differing entry points, any n-pass gateway can be constructed.

```
G1: Merge A
    Merge B
```

G2: Merge C

N-pass gateways are quite useful if one wishes to deviate from conventional cobegin-coend parallel structures. Suppose, for example, suppose the most natural form for a computation is that of an acyclic directed graph, where each vertex represents a sequence of instructions to be executed serially. Suppose a vertex contains indegree n¿0 and outdegree m¿0. The sequence of instructions representing the node will be preceded by an n-pass gateway, and followed by m-1 split instructions and a branch. Obviously if n=1 no gateway is required, and if n=1 no split instructions are required. The branch targets of the splits (and the branch) are the entry points of the gateways of the successor vertices. The predecessor vertices will branch or split to the branch targets of the n-pass gate. If a graph has more than one vertex of indegree zero, a null starting vertex is added which serves as the single starting point for execution. All vertices with indegree zero are treated as successors of the starting vertex. Similarly if there is more than one vertex with outdegree zero, a single null vertex is added as the successor of all such vertices. Figure 2 illustrates directed acyclic graph processing in the SAM architecture.

(place figure 2 here)

Up to this point it has been assumed that all instruction threads execute in the same environment, but this restricts the way code can be parallelized, because two separate streams that execute simultaneously must have physically separate instruction sequences. This is especially troublesome if the elements of a large array are to be processed in parallel using essentially the same sequence of instruction. In this case it is necessary to allow code to be shared between threads, but since each stream must perform operations on different memory locations, it is necessary to provide each stream with its own private environment that allows it to select the objects upon which it is to operate. The environment of a thread must contains all the temporary variables used at each step of execution, as well as the parameters that apply to that particular part of the computation. Conventional register-swapping techniques cannot be used, since the streams are executing simultaneously. One method of supporting multiple environments is to have several sets of data

and address registers that are replicated for each instruction thread, as in the HEP processor[9]. However, this approach has two drawbacks. First, in a shared pipeline, each stage may correspond to different instruction thread at different time. The number of required registers is enormous, and a great deal of circuitry is required to pass the register values from one stage to the next. And, as is usually the case, only a small fraction of this data will be accessed during each pass through the pipeline. Second, since not all threads require the same number of registers, and because the size of the register file must meet the maximum need, Many registers will be wasted for small environments.

(place figure 3 here)

SAM presents a different approach to solving this problem. Instead of using registers, SAM uses a concept called the short memory. Each thread has a 256-word block of memory called its short address space. In addition to the program counter, each thread has an environment pointer or EP that contains the main memory address of the short address space. This register is passed from stage to stage instead of passing an entire register file. Most instructions contain at least one short memory address, thus the EP will contain useful information for most instructions. A special instruction is provided for changing the contents of the EP, thus allowing the program to determine whether two threads execute in the same or different environments. Furthermore, a single thread can use several different EP pointers (one at a time) to effectively increase the size of the short memory beyond the 256 byte limit.

One reason for using the short memory is to reduce the size of instructions. An 8-bit address consumes less space than a full 32-bit address, and thus allows instructions to be fetched more quickly. The second reason stems from the fact that the short memory is intended to be mapped to a small fast cache inside the CPU, so keeping the most frequently used data in the short memory will greatly reduce bus traffic. The full address of a short-memory operand is computed by adding the short address to the contents of the EP. It is intended that this memory be fast enough so that all read and write operations can be finished in one clock cycle (corresponding to one stage in the

pipe). Figure 3 illustrates how the short memory cache is implemented in the current simulation model. (Note that the specifications for this cache may change in the future as more knowledge is gained both through simulation and prototyping the existing design.) As Figure 3 shows, the small memory is divided into several segments. There are two steps in addressing short memory locations. The first step is to locate the correct segment, which is accomplished by comparing the high order bits of the full address with the segment tags of the small memory. The second step is to locate the position in the segment using the eight low-order bits of the full address. These two steps can be carried simultaneously. Instructions support two types of addresses: short addresses and long addresses. A short address refers to the small fast cpu memory, while a long address refers to a location in main memory. Each thread can use a portion of or the entire cpu small memory.

(place figure 4 here)

The allocation of internal memory for a new short address space is done by executing the *SETEP* instruction.Figure 4 gives an example for multiple environments. In this example two threads, *thread0* and *thread1*, share one segment of internal memory. The cpu memory allocated for *thread0* starts from address zero, while that for *thread1* starts from address 8. Both of them can extend to address 255, the end of the internal memory segment. However, SAM also allows a thread to use more than one segment. An example is shown in figure 5, in which *thread0* gets two segment by assigning two different values to the high order bits of the EP. The allocation of cpu memory segment is transparent to the users. As long as there is an empty segment, it can be allocated to any thread that requires it. When a segment is allocated, its tag is set to the high 24-bit of the corresponding EP. The release of cpu memory will be done automatically by hardware. If after a certain amount of time an allocated memory segment has not been touched, its tag will be cleared provided there is no reserved information in this segment. If a thread can not get a segment of cpu memory, it will be suspended.

(place figure 5 here)

Another important issue in a multithreaded architecture is the subroutine call. In a sequential machine, subroutine calls is implemented by a stack. However, in a multithreading environment, a system stack is rather hard to maintain. One solution is to use a linked list rather than a stack. But this may cause stack operations to be quite time consuming. SAM uses another solution. Every thread in SAM may have its own stack or share a stack with other threads, however, in most cases when a thread performs subroutine calls, it will be allocated its own stack. If other threads share this stack, they will not normally perform subroutine calls. To maintain the stack, SAM provides a Stack Pointer (SP) register for each thread. The management of subroutine calls is still being investigated. Although our current solution is workable, *any* solution must take into account that the simple LIFO paradigm of subroutine calls does not apply to a multi-threaded architecture.

The interrupt handling mechanism is another unique characteristic of the SAM architecture. When a hardware interrupt is granted, SAM will generate a new thread for the interrupt routine. The environment pointer for this thread will be set by the interrupt routine itself. Each device that can generate an interrupt must return a unique interrupt vector, which can be used to determine the interrupt routine entry point. Since no thread is actually interrupted, it is not necessary to save the state of any thread when an interrupt is serviced, thus speeding up the servicing of interrupts. Since no stack is involved, the interrupt return will differ from that used on a conventional machine. SAM provides two ways to terminate the servicing of an interrupt. One is to kill the thread by executing a dequeue instruction[14]. The other is to use a merge instruction, which can provide a synchronization between a thread and the interrupt process, and is useful in overlapped I/O operations.

A software interrupt (a system call or a trap) can be issued by any executing thread. Like hardware interrupt, it will generate a new thread. But the environment of this thread will be the same as the one which generated it. Any interrupt routine may have several threads executing simultaneously, thus providing for faster interrupt servicing, and enhanced I/O overlap operations.

Priority and masking are basic mechanisms of an interrupt system. In SAM every thread has a processor state word, which contains the priority of the thread. Since interrupt in SAM actually

does not interrupt any program, as long as the maximum number of threads has not been reached, the interrupt will be granted. When no additional thread can be created, a priority check will be done. The thread with the lowest priority will be suspended and the interrupt routine will begin execution. A hardware masking is also needed for many applications. Since there may be several threads running simultaneously, and since inhibiting interrupts must be done on a global basis, SAM leaves this problem to software. Specifically, any thread that wants to enable or disable interrupts must go through the operating system. Because operating system support for the SAM architecture is still the subject of ongoing research, the specific mechanism for controlling global resources, such as the interrupt mask, has not yet been completely determined. However, the mechanism will probably be based on conventional control structures such as semaphores, which will be supported by a Test-And-Set type of instruction.

3. The Simulation Model

A functional simulator has been built[14] based on the architectural principles described in the previous Section. It has 32-bit address bus and 32-bit data bus. Conventional handshaking signals are used to control the timing between the cpu and memory. In this model, SAM main memory is similar to those used in most contemporary machines. All threads share one main memory and one pipeline in the cpu. The overall consideration of the size of this model is aimed at a microprocessor chip. It must be emphasized that this model is a prototype developed to study the architecture, so some of the features of both the model and the architecture may change as more information is gained through simulations. We plan to design a prototype VLSI implementation based on the model, as soon as we feel the model is stable.

The SAM simulator contains a thread queue which contains all register information for all threads. A thread is physically defined to be an item in thread queue. Every item in this queue has six registers as shown in figure 6. The SID is a thread identification register, while the PID is a process identification register. The PID can be used to identify the program, task, or process in

the operating system that owns the thread. Both the SID and the PID are 16-bit registers. The EP is a is a 32-bit register which contains the main memory address of the short memory. The PSW is another 16-bit register used as a processor state register. The PC is 32-bit program counter. Another 32-bit register, SP, contains the Stack Pointer for the thread. Each queue slot contains one thread. Each queue slot can be in one of five states, as shown in figure 5. *Empty* means the slot is not being used. It can be filled by a new generated thread. *Wait* indicates the thread is waiting for some event and it can not currently be executed. *Ready* means that the thread can issue a new instruction to the pipeline. Running means that an instruction is being fetched from the thread. After the instruction is fetched, the thread will go back to ready state, so if there is only one pipeline, at any time only one item in the thread queue can be in the running state. *Dequeue* indicates that the thread is terminating execution. The thread will be kept in the queue for one rotation of the fetch cycle and then be cleared to the *Empty* state. The fetch cycle processes the queue in a circular fashion. A token is cycles through the thread queue, advancing one slot every clock cycle. Waiting and Empty threads are skipped without wasting a clock cycle. The thread queue is one of the most important parts in this model. Every clock cycle it will select a thread, and pump the six registers to the pipeline. When a thread is generated it will be assigned a slot in the thread queue. When a thread is merged or dequeued, the slot will set to empty. Because the thread queue is limited in size, the number of threads is also limited. We are currently investigating several different alternatives for handling thread-queue overflow. One purpose of the model is to allow us to select the best method for handling this situation.

(place figure 6 here)

(place figure 7 here)

The overall structure of the execution unit is shown in figure 7. A four-stage pipeline is used. The first stage is instruction fetch, which uses the address given by PC. The model contains an instruction cache, which is the source of all instruction fetches. If there is a cache miss, a main memory read for a block of instructions will be issued, and a cache miss signal will be sent to

the successive stages of the pipe. This will produce a "hole" in the pipeline. The second stage is decoding which generates the operand addresses. The third stage is cpu small memory read and write. The cpu memory has two read and two write ports. So two read and two write operations can be executed in one clock cycle. The final stage is an ALU with Barrel Shifter. To support this pipeline, other three modules are included in the simulator. A storage unit takes care of interfacing with memory. Inside the storage unit, there is memory operation queue as shown in figure 8. All the memory requests from all pipeline stages are queued here. The storage unit selects the memory operation from the top of the queue, and issues a memory read or write signal to the bus. The interrupt module handles hardware interrupt requests, interrupt masking, and acknowledgements.

(place figure 8 here)

All the operations in every pipeline stage can be finished in one clock cycle. One execution cycle consists of four clock cycles. The ready threads in the thread queue will be selected using a round robin strategy. When a thread is selected, it will be changed to running state. After one clock cycle it will be set back to ready state, so it can be immediately selected again, if necessary. If an instruction needs to wait for a slow operation, such as a main memory read, the corresponding thread will be set to wait state. In this case, the thread will be skipped during the thread-queue scan.

There is two ways to fill a slot in the thread queue. One is executing a **split** instruction, the other is by interrupt. A **merge** instruction can kill a thread. Besides merge, the simulator provides a **deq** instruction which will terminate a thread and clear the slot containing it. The thread queue structure in SAM produces an efficient way of thread switching. Every clock cycle, it will pump a thread to the pipe without additional time overhead.

The SAM instruction set, which was designed according to the RISC philosophy[5,6], contains 36 instructions. Apart from moves between main memory and short memory, each instruction is 32 bits long. As long as the instruction is in the cache and all operands are in the short memory, the execution of every instruction will be finished in four clock cycles. Table 1 lists all these instructions.

As mentioned above, the design of a subroutine call/return mechanism in a multithreading architecture is a more complex issue than on a single-threading machine. Furthermore, a conventional call and return contain too many operations to be completed by our pipeline without introducing delays. One goal of the SAM instruction set design is that all instructions must have he same degree of complexity. Thus the SAM call instruction has only two forms, one with an immediate 24-bit address contained in the instruction, and one with an indirect 32-bit address which must be contained in the short memory. Thus the call instruction has only one main memory access (to save return address in the stack) and less than two short memory operations. This allows the call instruction to be completed in one pass through the pipeline.

The conventional return instruction requires a main memory access to finish its operation. Because of the structure of the pipeline, it is impossible to complete this memory access and perform the necessary operations on the stack pointer and PC within the required time limit of four cycles. Therefore, SAM splits the conventional return operation into two instructions: pop and ret. This approach will requires slightly more storage than the conventional approach, but no additional time.

(place table 1 here)

Since SAM's pipe line can be shared by many threads, the conditional branch instruction will not seriously affect the throughput as is the case in conventional pipelined architectures. In fact, as will be proved in the following section, the multithreaded architecture can completely eliminate the conditional branch problem as long as the number of ready threads exceeds a certain threshold. However, if the number of ready threads falls below this threshold, it is necessary to handle conditional branches in the conventional way. In other words, it will sometimes be necessary to purge wrongly prefetched instructions from the pipeline. Write-read conflicts between consecutive instructions present a similar problem. As long as the number of threads exceeds a certain threshold, no write-read conflicts exist (except those present due to programming errors.) However, when the number of threads falls below this threshold, it is necessary for the pipeline to maintain data

coherence within a single thread. The pipeline uses conventional mechanisms for purging wrongly prefetched instructions and for maintaining data coherence. The SID and PID registers are used to identify the threads from which instructions have been fetched.

4. Performance Analysis

For multithreaded pipeline architectures, some researches[11,13] have shown the throughput improvement by multithreading and various performances of different thread scheduling policies. In order to build a SAM machine, it is of fundamental importance to understand how much performance gain can we get, and what kind of thread scheduling strategy should be used. In this section we will analyze the performance characteristics of the model presented in last section. The comparison made here is in terms of clock cycles, and the aim is to increase pipeline efficiency by eliminating holes in the pipeline. To facilitate this analysis, the following definitions are required.

Definition 1: *For a given task, the total number of dispatch cycles for T is defined as follows:*

$$T = t_e + t_o$$

Where t_e is the number of cycles in which a correct instruction is fetched by the pipeline, and t_o is the number of cycles in which no correct instruction is fed into the pipeline.

Definition 2: *The efficiency of the pipeline is defined to be*

$$\eta = \frac{t_e}{T}$$

Definition 3: *The instruction dispatch interval τ is defined to be number of clock cycles that must elapse before the next instruction can be dispatched correctly.*

Definition 4: *Suppose in an instruction thread there are no branching or data coherence problems. Then the entire instruction dispatch interval is due to cache misses. The maximum value of this interval is denoted as τ_c.*

Definition 5: *Suppose in an instruction thread there are no branching problems and no cache misses, then the total instruction dispatch interval is caused by data coherence problems. The maximum value of this interval is denoted as* τ_d.

Definition 6: *Suppose in an instruction thread there are no data coherence problems and no cache misses, then the total instruction dispatch interval is caused by branch problems. The maximum value of this interval is denoted as* τ_b.

Definition 7: *The number of the active threads in the thread queue is denoted by* m, *and the number of the stages in the pipe is denoted by* n. *The probability of a cache miss for the thread* i *at time* t *is* $p_c^i(t)$, *while* $p_d^i(t)$ *is the probability of a data coherence problem occurring in the thread* i *at time* t. *The cache block size is denoted by* l, *while* x *is the number of clock cycles required for a memory operation.*

For the simplicity, we assume that after the program has been running for some and that $p_c^i(\mathrm{t})$ and $p_d^i(\mathrm{t})$ have stabilized, and that in the interval under consideration, the following is true.

$$p_c^1(\mathrm{t}) = p_c^2(\mathrm{t}) = \cdots = p_c^m(\mathrm{t}) = p_c$$
$$p_d^1(\mathrm{t}) = p_d^2(\mathrm{t}) = \cdots = p_d^m(\mathrm{t}) = p_d$$

Due to this assumption, we sometimes ignore the t in the rest of the paper.

Lemma 1: *At any clock cycle, for a given thread* i,

$$\tau_c^i = (m-1)x[p_c(l-p_d)+p_d] + lx/2$$

PROOF: There are two parts of delay which contribute to the τ_c^i. One is the queueing delay τ_q which is the average number of memory requests already in the memory operation queue when the cache miss occurs. The other component is τ_l which is the amount of time required to load the desired instruction into the cache from memory. Since instructions are loaded as a block, and since the desired instruction may be located at any position within the block, τ_l must be computed as an average rather than an absolute value.

$$\tau_q = \sum_{i=1}^{m-1}[p_c^i lx] + \sum_{i=1}^{m-1}[(1-p_c^i)p_d^i x]$$

$$= (m-1)p_c lx + (m-1)p_d(1-p_c)x$$
$$= (m-1)x[p_c(l-p_d)+p_d]$$

$$\tau_l = \overline{l} \cdot x = lx/2$$

So, $\tau_c^i = \tau_q + \tau_l = (m-1)x[p_c(l-p_d)+p_d] + lx/2$

Lemma 2: *At any clock cycle, for a given thread i,*

$$\tau_d^i = (m-1)x[p_c(l-p_d)+p_d] + x$$

PROOF: The proof is similar to lemma 2.

$$\tau_d^i = \tau_q + x = (m-1)x[p_c(l-p_d)+p_d] + x$$

Lemma 3: *At any clock cycle, for a given thread i,*

$$\tau_b^i = n$$

The proof is obvious.

For round robin the worst-case dispatch interval for a thread i, is given by the following formula.

$$\tau^i(t) = max\{\tau_c^i(t), \tau_b^i(t), \tau_d^i(t)\}$$

To evaluate the effectiveness of our round robin scheduling policy, we will compare it to the priority scheduling algorithm, which has been proposed for some shared pipeline schemes.

Definition 8: *The Optimal Scheduling Algorithm will select the thread with the minimum instruction dispatch interval from all the active threads in the thread queue.*

The minimum instruction dispatch interval is given by the following formula.

$$\tau_{optimal}(t) = min\{\tau^1(t), \tau^2(t), ..., \tau^m(t)\}$$

Lemma 4: *If a task is finished in T clock cycles, the total overhead time t_o is given by*

$$t_o = \sum_{t=0}^{T} t_o(t)$$

in which

$$t_o(t) = \begin{cases} 1, & \tau(\mathrm{t}) \geq 1 \\ 0, & \tau(\mathrm{t}) < 1 \end{cases}$$

The proof is immediate from the definition of t_o and $\tau(t)$.

Lemma 5: *Assume that round robin scheduling is being used, and that thread i is being accessed at time t. Let r(m)=$\tau^i(t)/m$. If*

$$r(m) \leq 1$$

then

$$t_o(t+m)=0.$$

PROOF: By round robin schedule, the next time after t that cpu will select thread i is t+m.

$$\tau^i(\mathrm{t+m}) = \tau^i(\mathrm{t}) \text{ - m.}$$

$$\tau^i(t+m) = \begin{cases} \tau^i(t) - m, & \tau^{\mathrm{i}}(\mathrm{t}) \geq m \\ 0, & \tau^{\mathrm{i}}(\mathrm{t}) < m \end{cases}$$

Since $\tau^i(t) \leq m, \tau^i(t+m) = 0.$

From lemma 4, we have

$$t_o(\mathrm{t+m})=0.$$

The lemma 5 shows as long as $r(m) \leq 1$, there is no time overhead for the thread i. If all threads satisfy the conditions of lemma 5, then at any time t, $t_o(\mathrm{t})=0$, which implies that the pipeline can be kept full.

Theorem 1: *Using round robin scheduling, when $m \geq n$, the conditional branch will not degrade pipeline throughput.*

The proof is obvious.

Theorem 1 proves that the branch problem can be completely eliminated in multithreaded architecture. However the data coherence problem is not so simple.

Theorem 2: *In round robin scheduling, if p_c and p_d are stable, the data coherence effect r(m)=τ_d^i/m is a monotonic function. As m increases, r(m) approaches a constant.*

PROOF: From lemma 2,

$$\tau_d^i = p_c l x(m-1) + p_d x(m-1) + x$$

$$\text{r(m)} = \tau_d^i \ /\text{m} = (m-1)x[p_c(l-p_d)+p_d] \ /\text{m} + \text{x/m}$$

$$= x[p_c(l-p_d)+p_d] + x(1-[p_c(l-p_d)+p_d])/m$$

When $p_c(l-p_d)+p_d > 1$, r(m) is an increasing function.

When $p_c(l-p_d)+p_d < 1$, r(m) is a decreasing function.

When m=1, r(1)=x.

Furthermore,

$$\lim_{m\to\infty} r(m) = x[p_c(l-p_d)+p_d]$$

.

Figure 9 illustrates these results.

(place figure 9 here)

Theorem 2 reveals several interesting points:

1. When the cache miss rate and data coherence rate are independent of the number of threads that are running, and $p_c(l-p_d)+p_d < 1$ the data coherence effect can be decreased to certain extent by increasing the number of threads.

2. When the cache miss rate and data coherence rate are independent of the number of threads that are running, and $p_c(l-p_d)+p_d < 1/x$ the data coherence effect can be completely eliminated by increasing m, because

$$\lim_{m\to\infty} r(m) = x[p_c(l-p_d)+p_d] < 1$$

3. When $p_c(l-p_d)+p_d > 1$ the data coherence problem is worse on a multithreaded architecture than on a single threaded architecture. Furthermore, the more threads running the poorer the performance will be.

4. A careful examination of the formula for r(m) shows p_c plays a more important rule than p_d.

In reality the probabilities p_c and p_d may not be stable or completely independent of the number of running threads. But it is possible to approximate these conditions through careful design. Data coherence (p_d) can be controlled, to a certain extent, by a sophisticated compiler. If the cache hit rate is high, then p_c will tend to stabilize and will tend to be independent of the number of running threads. On the other hand, recall in last section, SAM will suspend a thread if a data coherence problem occurs. This will improve the real performance somewhat, but as shown in Theorem 2, it can not completely solve the data coherence problem. After this thread is set to the waiting state, the remaining threads may still have the same problem if p_c and p_d are high.

Theorem 3: *If p_c and p_d are constant and sufficiently small, then there exists an N, such that when $m \geq N$, the overhead for round robin approaches that of the optimal scheduling algorithm.*

PROOF: Suppose $p_c(l - p_d) + p_d < 1$.

From lemma 1, we have

$$\tau_c^i = (m-1)x[p_c(l-p_d)+p_d] + lx/2, i = 1, 2, ..., m.$$

Let

$$N_1 = \frac{x[p_c(l-p_d)+p_d] + lx/2}{x[p_c(l-p_d)+p_d] - 1}$$

If m $\geq N_1$ then

$$r(m) = \frac{\tau_c^i}{m} < 1, i = 1, 2, ..., m.$$

From lemma 2, $\tau_d^i = (m-1)x[p_c(l-p_d)+p_d] + x$, i=1,2,...,m.

Let

$$N_2 = \frac{x[p_c(l-p_d)+p_d] + x}{x[p_c(l-p_d)+p_d] - 1}$$

If m $\geq N_2$ then

$$r(m) = \frac{\tau_d^i}{m} < 1, i = 1, 2, ..., m.$$

From lemma 3, τ_b^i = n, i=1,2,...,m.

Let N = max$\{N_1, N_2, n\}$

For round robin, $\tau^i = max\{\tau_c^i, \tau_d^i, \tau_b^i\}$

If m $\geq$ N, for all times t then $r(m) = \frac{\tau^i}{m} < 1, i = 1, 2, ..., m$.

From lemma 5, t_o(t)=0, for all threads. So the total overhead $t_o^{RR} = 0$.

For optimal scheduling, $\tau_{optimal} = min\{\tau^1, \tau^2, ..., \tau^m\}$

When $m \geq N$,

$$\tau^i < 1, i = 1, 2, \cdots, m$$

Then

$$\tau_{optimal} < 1, and, t_o^{optimal} = 0$$

Therefore $t_o^{optimal}$(t) = t_o^{RR}

Round robin is simpler to implement than the priority scheduling algorithm and Theorem 3 suggests that it will perform well.

5. Example

Several programs have been run the simulator to test its performance. The programs discussed here perform an 8 by 8 matrix multiplication. We have implemented this algorithm using different numbers of threads, and have counted the total clock cycles, the number of cache miss, and the number of idle clock cycles for each program.

The programs are written in SAM assembly language[14]. The multiplication is implemented by a subroutine using the BOOTH algorithm. All data are 32-bit integers. The SAM source code contains two parts. The first part is the data definitions, while the second part is the instructions. The SAM assembler translates the source code into executable code. The executable file has a jump instruction at the beginning, followed by the data area and then the instructions. The reset sequence of the simulator will cause a single thread to begin execution at address zero. The preliminary jump instruction will cause the processor to enter the executable part of the program. The source code for all of our test programs will be made available on request.

Initially, we wished to minimize the effect of cache misses, to allow us to focus on the other issues. Therefore all of our initial programs were designed to have good locality. Three versions of

the matrix multiply program were run. The first used two threads, the second used four threads, and the third used eight threads. The results are listed in table 2.

The size of the program varies, depending on the number of threads used. Obviously, the fewer threads used, the shorter the program. Shorter programs tend to have a higher cache hit rates. Consequently the implementations with fewer threads have lower cache miss rates. Since our test programs all have good locality, the difference in cache miss rate only affects the initial part of the computation. After this initial portion, the cache miss rate will not be a factor.

(place table 2 here)

Although the number of threads is statically defined for each program, since these threads must be created and destroyed, the actual number of active threads varies for each program. During program in execution, the number of active threads number is always less or equal to the static number. In fact, in our test programs, the number of active threads is stable (at the maximum) most of the time. Table 2 shows that when the number of threads is equal to half of the number of pipeline stages (four), the pipeline efficiency is only about 75%. When the thread number increases to or exceeds four, the pipeline is full most of the time. The experimental results shows when the condition of $m \geq n$, and the cache miss rate is low, the overhead is almost totally eliminated, thus confirming the analysis of Section 4

Further observations of examples 2 and 3 are given in tables 3 (a) and (b), where several snap shots of the execution are listed. These tables show that most of time these two programs can keep the pipeline full. There is a throughput degradation at the beginning and end of the execution, because at the beginning there is a high cache miss rate, and at the end the number of active threads decreases. This is also in accordance with the analysis given in Section 4. The comparison of the first 400 clock cycles in the two programs identifies that four threads gives better performance than eight threads. This coincides with the conclusion in theorem 2. At that period of time the cache miss is high, the condition of

$$p_c(l - p_d) + p_d > 1$$

is satisfied. Thus, the more the threads, the lower the throughput, primarily because of the higher cache miss rate.

(place table 3 here)

(place table 4 here)

There is one type of overhead that we have ignored in the analysis. That is the extra cost of merge instruction. Like branch, merge will cause wrong prefetching. The strategy to deal with this problem is exactly the same as with a branch. As long as the condition of theorem 1 is satisfied, it will not cause any overhead. On the other hand, the merge will probably be executed less frequently than the branch instruction. Furthermore most of the time when a merge is executed there will more than one thread in the pipe. This will tend to reduce the possibility of wrongly prefetching instructions.

6. Conclusion

The SAM architecture represents a combination of both von Neumann and dataflow technology. As in a dataflow architecture, the program and its parallelism can be represented by a directed acyclic graph, and because its control is based on the von Neumann architecture, it can be programmed as if it were a conventional sequential machine. Both our analysis and our experimental data suggest that the ability to share a pipeline between many threads can dramatically increase pipeline efficiency, as long as the cache miss rate is acceptably low. It should be noted that a high cache miss rate can dramatically reduce the efficiency of *any* processor, so SAM is not unique in that respect.

Because of the evolutionary nature of SAM, as opposed to the revolutionary nature of the dataflow architectures, SAM can be implemented using either conventional hardware structures or extensions of conventional hardware structures. SAM can support dataflow-style computation efficiently, although not at the single-instruction level, while its hardware structure is comparable to

that of a conventional RISC machine. The cost of synchronization among the threads is simple and fast. Furthermore since SAM supports conventional von Neumann-style programming, operating system techniques, such as memory management, that have been successfully used on commercial machines can be ported directly to the SAM architecture. Software can be easily ported as long as the appropriate high-level-language compilers are available. Furthermore, since SAM is an extension of the von Neumann architecture, it is conceivable that the multi-threading techniques used in the SAM architecture could be adapted to existing commercial microprocessors without a great deal of effort.

Unlike existing RISC machines, SAM does not rely on a delayed branches or non-interlocked pipelines to improve pipeline efficiency. Because the delayed branch scheme cannot completely avoid using no-op instructions after the branch[10,15], SAM can potentially improve pipeline throughput beyond that of a conventional RISC architecture.

From the implementation point of view, SAM architecture is suitable for implementation in VLSI technology. SAM hardware structure is simple, and can be implemented in one high-frequency chip. On the other hand, some constraint on the SAM architecture is the VLSI chip size and pin count. Because the cache miss rate is a critical factor in the SAM architecture, it is important to have as large a cache as possible. Additional area is required for short memory segments. The pin-count limitation is important, because it limits the size and number of external busses that can be used in the architecture. Even if it were possible to complete one memory operation every cycle, pumping all data and instructions through a single 32-bit bus may limit the throughput of the SAM architecture. Using a 64-bit memory bus as found in some high performance RISC processors may provide a partial solution to this problem.

As it is shown in the previous sections, cache miss rate has an important effect on the performance of a multithreaded machine. It is still not clear what kind of cache miss rate can be reached with a reasonable cost. We believe that our future work with simulators and hardware prototypes will help answer this question, but as yet, no satisfactory data is available. It may be that a multithreaded architecture needs a more complicated caching mechanism than a conventional sequential machine.

The data coherence problem is another important consideration. We are currently investigating the effect that pipeline forwarding would have on the complexity of our memory access mechanisms, but finding the best method to control this problem in the SAM architecture is still an open problem.

Reference

1. P.M. Maurer, Mapping the Dataflow Computation Model into an Enhanced Von Neumann processor, Proceedings of International Conference on Parallel Processing, 1988,
2. R.S. Nikhil, Arvind, Can dataflow Subsume von Neumann Computing?, The 16th ACM Annual International Symposium on Computer Architecture, May 1989, pp.262-272.
3. J.B. Dennis, Data flow supercomputers, IEEE Computer Magazine, Nov. 1980, pp.48-56.
4. D.D. Gajski, D.A. Padua, D.J. Kuck, and R.H. Kuhn, Second opinion on data flow machines and languages, IEEE Computer Magazine, Feb. 1982, pp.489-500.
5. D. Patterson, C.H. Sequin, A VLSI RISC, IEEE Computer, Vol.15, No.9, 1982, pp.8-22.
6. J.L. Hennessy, VLSI Processor Architecture, IEEE Transactions on Computers, Vol. C-33, No.12, 1984, pp.1221-1246.
7. Arvind, D.E. Culler, Dataflow Architectures, Annual Reviews in Computer Science, 1986, Vol 1, Annual Reviews Inc., 1986, pp. 225-253.
8. Arvind, L. Bic and T. Ungerer, Evolution of Data-Flow Computers, Advanced Topics in Data-Flow Computing, Prentice Hall, 1991, pp.3-34.
9. B.J. Smith, Architecture and Applications of the HEP Multiprocessor Computer System, Real Time Signal Processing IV, Proceedings of SPIE, 1981, pp.241-248.
10. T. Gross, Code Optimization of Pipeline Constraints, Technical Report No. 83-255, Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford university, Stanford, CA 94305, 1983.
11. M.K. Farrens and A.R. Pleszkun, Strategies for Achieving Improved Processor Throughput, The 18th ACM International Symposium on Computer Architecture, May 1991, pp. 362-369.
12. W.J. Kaminsky and E.S. Davidson, Developing a Multiple-Instruction-Stream Single-Chip Processor, Computer, Dec. 1979, pp.66-76.
13. H. Hirata, et al., An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads, The 19th ACM International Symposium on Computer Architecture, May 1992, pp. 136-145.
14. A functional simulator for SAM architecture, Technical Report, Computer Science Department of USF, August 1992.

15. R.F. Cmelik, et al, An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks, Proceeding of the 4th Architectural Support for Programming Languages and Operating Systems, April 1991.

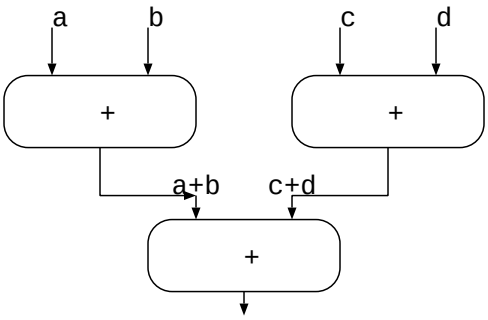
Author's Information

Wei Lin received the B. S. and M. S. degrees in computer science from Shanghai Jiao Tong University, China, in 1982 and 1988, respectively. He is currently working towards a Ph.D degree in the department of computer science and engineering with the University of South Florida. His research interests include parallel architectures, VLSI and artificial intelligence. Mr. Lin is a student member of IEEE Computer Society.

Peter M. Maurer received the B. A. degree in mathematics from Saint Benedict's College, Kansas, in 1969 and the M. S. and Ph.D degrees in computer science from Iowa State University in 1979 and 1982, respectively.

He has worked for AT&T Bell Laboratories and Information Systems where he was involved with VLSI design verification and CAD tool research and development. Currently, he is an Associate Professor of Computer Science and Engineering with the University of South Florida. His interests include computer architecture, VLSI design automation, and VLSI design.

Dr. Maurer is a member of the Association for Computing Machinery, the American Mathematical Society, and the Mathematical Association of America.

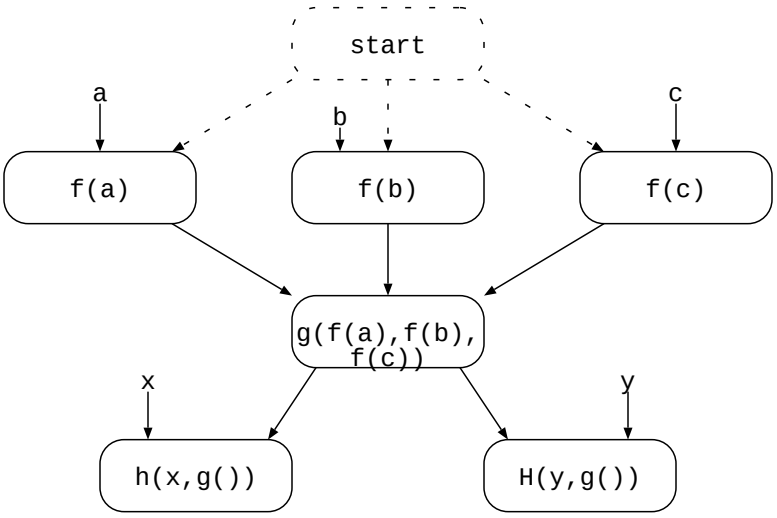


a)Dataflow graph of e=(a+b)+(c+d).

```
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,t2,3
```

b) Evaluating e=(a+b)+(c+d) by split and merge.

Fig.1. Evaluation of an expression.



(1). The dataflow model

```
        SPLIT A
        SPLIT B
          computation
          of f(a)
        JUMP C
A:        computation
          of f(b)
        JUMP C
B:        computation
          of f(c)
C:      MERGE A
        MERGE B
          computation
          of g(f(a),
          f(b),f(c))
        SPLIT D
          computation
          of h(x,g())
        JUMP E
D:        computation
          of h(y,g())
E:      MERGE D
```

(2) The program

Figure 2. The computation of two functions:
h(x,g(f(a),f(b),f(c))), and H(y,g(f(a),f(b),f(c))).

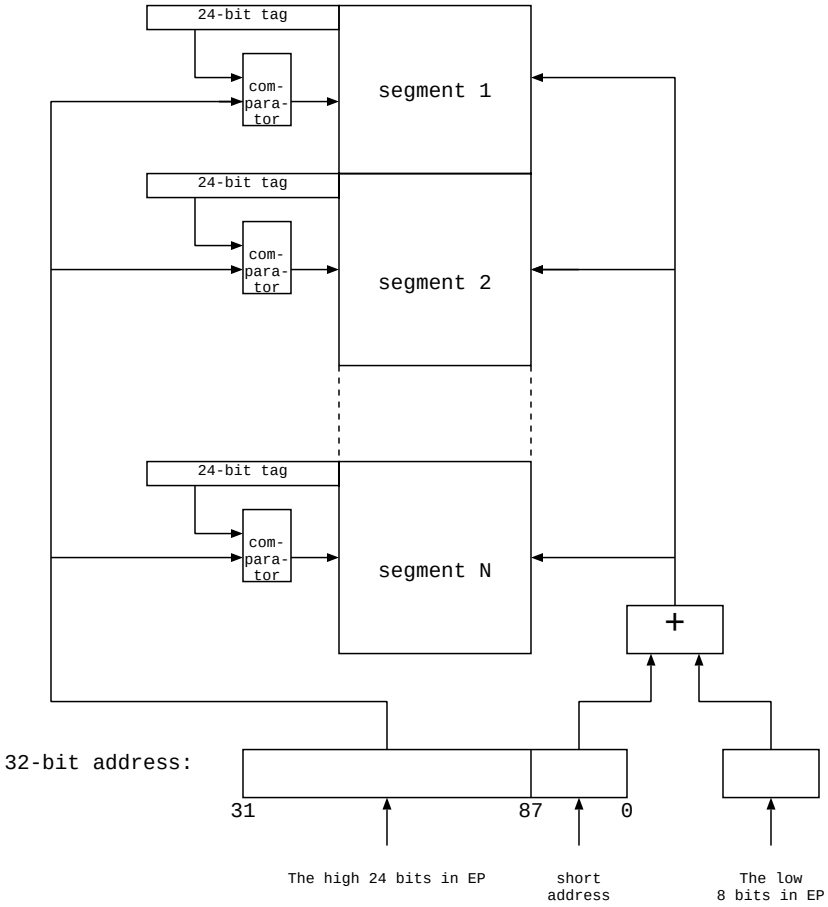
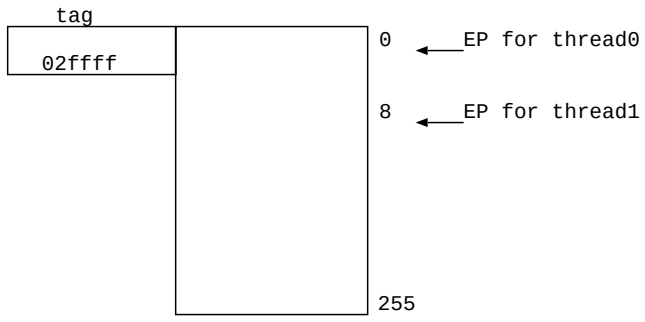


Figure 3. CPU memory organization.

```
THREAD0: MOVIWS 02FFFF00,0;     02FFFF00->0
         SETEP 0;               (0)->EP

         .....

         SPLIT THREAD1;         generate a
                                thread
         ......

         MERGE THREAD1;         merge the
         JUMP A;                the thread

THREAD1: MOVIWS 02FFFF08,0;
         SETEP 0;
         ......
         MERGE THREAD1
A:       ......
```

(a) Example of setting Environment Pointer.

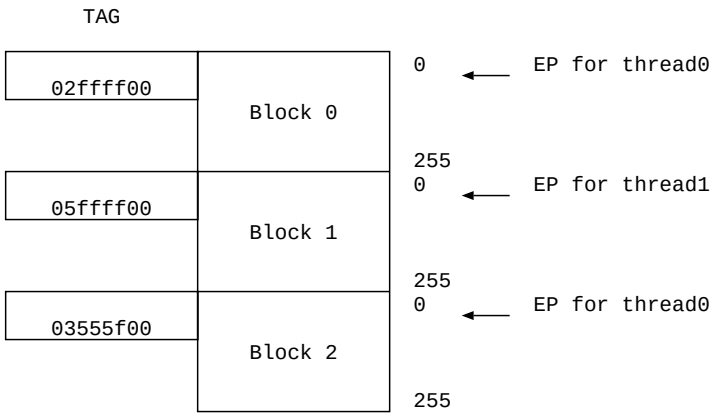


(b) Space allocation in cpu small memory.

Figure 4. Multithread Environment, Example 1.

```
THREAD0: MOVIWS 02FFFF00,0;     02FFFF00->0
         SETEP 0;               (0)->EP

         ......

         MOVIWS 03555F00,0;     allocate
         SETEP 0;               another block
         ......

         SPLIT THREAD1;         generate a
                                thread
         ......

         MERGE THREAD1;         merge the
         JUMP A;                the thread
THREAD1: MOVIWS 05FFFF00,0;
         SETEP 0;
         ......
         MERGE THREAD1
A:       ......
```

(a) Example of setting Environment Pointer.



(b) Space allocation in cpu small memory.

Figure 5. Multithread Environment, Example 2.

An item in the thread queue:

state	code	explaination
empty	000	The empty slot
ready	001	the thread in the slot is ready to dispatch an instruction
running	010	the thread in the slot is fetching an instruction
wait	011	The thead in the slot is waiting for some event and can not dispatch a new instruction
dequeue	100	Wait for n clocks and then turn to empty

Figure 6. The thread queue and states of a thread.

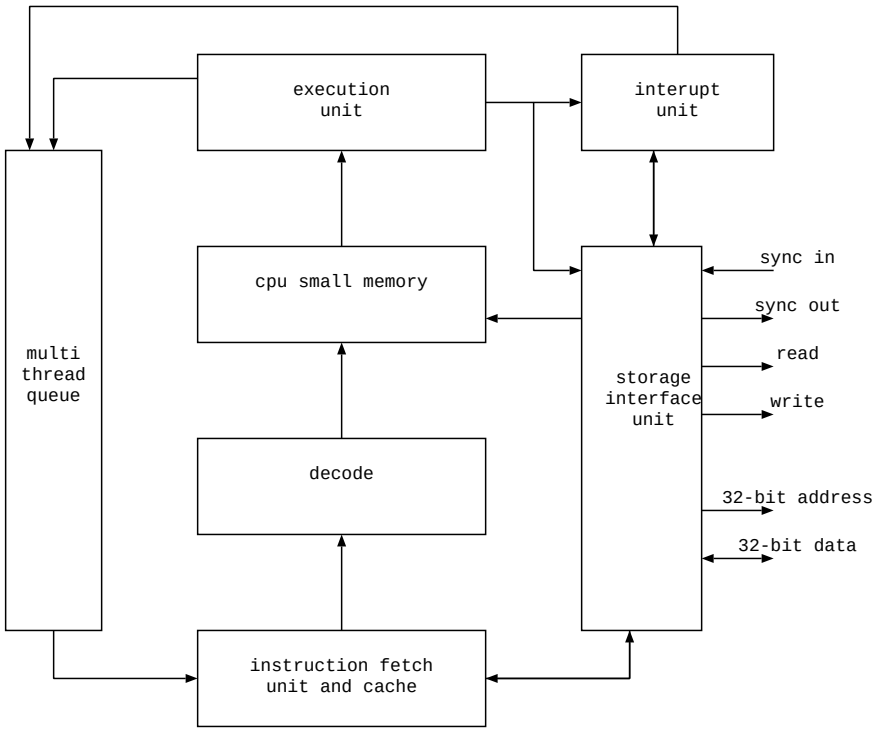
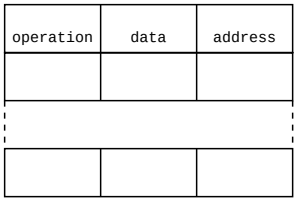


Figure 7. An overall structure of SAM



operation field indicates what kind of memory operation;

data field contains the memory write data. For read operation, this field is no use.

address field stores the main memory address.

Figure 8. Memory operation queue.

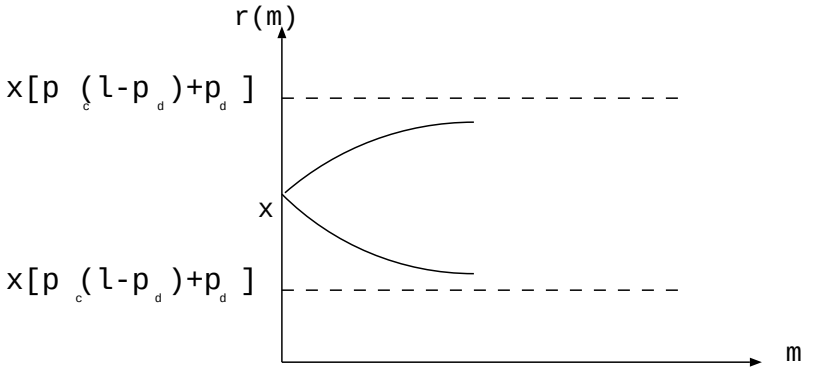
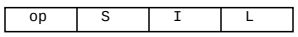
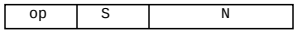
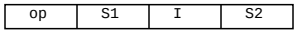
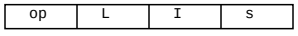
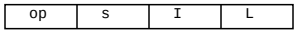
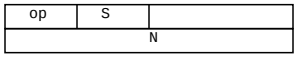
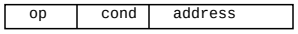
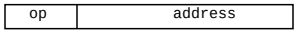
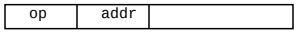
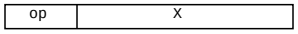
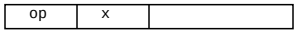
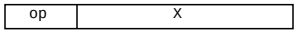
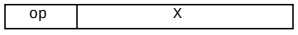
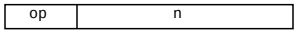
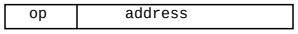
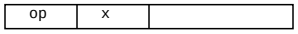
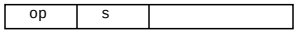
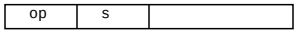
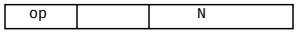
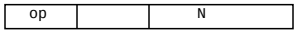
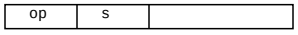
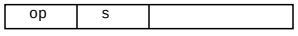
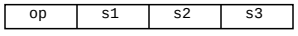


Figure 9. Data coherence effect curve.

Table 1. SAM instruction set.

Mnemonics	operation	format
M0VSLI S,I,L	(S)->(L+I)	
MOVIWS N,S	N -> s	
MOVSIS S1,I,S2	(s1+I) -> s2	
MOVLS L,I,S	(L+I) -> s	
MOVSIL S,I,L	(s+I) -> (L)	
MOVLWS N,S	N -> s	
BC ADDRESS BNC ADDRESS BNE ADDRESS BZ ADDRESS BNZ ADDRESS BP ADDRESS	if carry is on, branch if carry is off, branch if negative, branch if zero is on, branch if zero is off, branch if positive, branch	 as above
BR ADDRESS	address -> pc	
JMP ADDRESS	address -> pc	
SPLITS X	(split instruction)generate a thread, and x -> pc	
SPLITL X	(split instruction)generate a thread, and (x) -> pc	
MERGE X	merge a thread X	
DEQ	terminate the current thread	
INT n	generate a thread n	
CALLS ADDRESS	(pc)->(sp), address->pc	
CALLL ADDRESS	(pc)->(sp), (x)->pc	
POP S	(sp) -> s	
RET S	(s) -> pc	
SETPSW N	N -> psw	
SETPID N	N -> pid	
SETEP S	(s) -> ep	
SETSP S	(s) -> sp	

ADD S1,S2,S3
ADC S1,S2,S3
SUB S1,S2,S3
SBC S1,S2,S3
AND S1,S2,S3
OR S1,S2,S3
XOR S1,S2,S3
SHL S1,S2,S3
SHR S1,S2,S3
RSL S1,S2,S3
RSR S1,S2,S3
	(s1)+(s2)->s3
	(s1)+(s2)+c->s3
	(s1)-(s2) -> s3
	(s1)-(s2)-c -> s3
	(s1)^(s2) -> s3
	(s1)|(s2) -> s3
	(s1)xor(s2) -> s3
	(s1) shift left s2 bits -> s3
	(s1) shift right s2 bits
	(s1) shift left s2 bit with c
	(s1) shift right with carry
		 as above

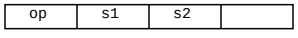
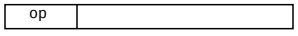
CMP S1,S2	(s1)-(s2)	
HLT	halt	

Table 2. The execution result of an 8 by 8 matrix multiplication.

Example	The length of the program in words	Number of threads used	cache miss	idle clock t	total clocks T	pipeline effiency
1	124	2	124	40538	169544	0.7609
2	152	4	152	218	129139	0.9983
3	200	8	200	228	129141	0.9982

Table 3.(a) The snap shots of the execution of example 3 with eight static threads.

period	cache miss	idle clock t	subtotal clocks T	pipeline effiency
0-200	126	128	200	0.36
201-400	55	55	200	0.725
1000-10000	2	2	9000	0.9998
100000-120000	0	0	20000	1.00
128741-129141	1	15	400	0.9625

Table 3. (b) The snap shots of the execution of example 2 with four static threads.

period	cache miss	idle clock t	subtotal clocks T	pipeline effiency
0-200	108	109	200	0.455
201-400	29	30	200	0.85
1000-10000	1	1	9000	0.9999
100000-120000	0	0	20000	1.00
128739-129136	0	10	400	0.975