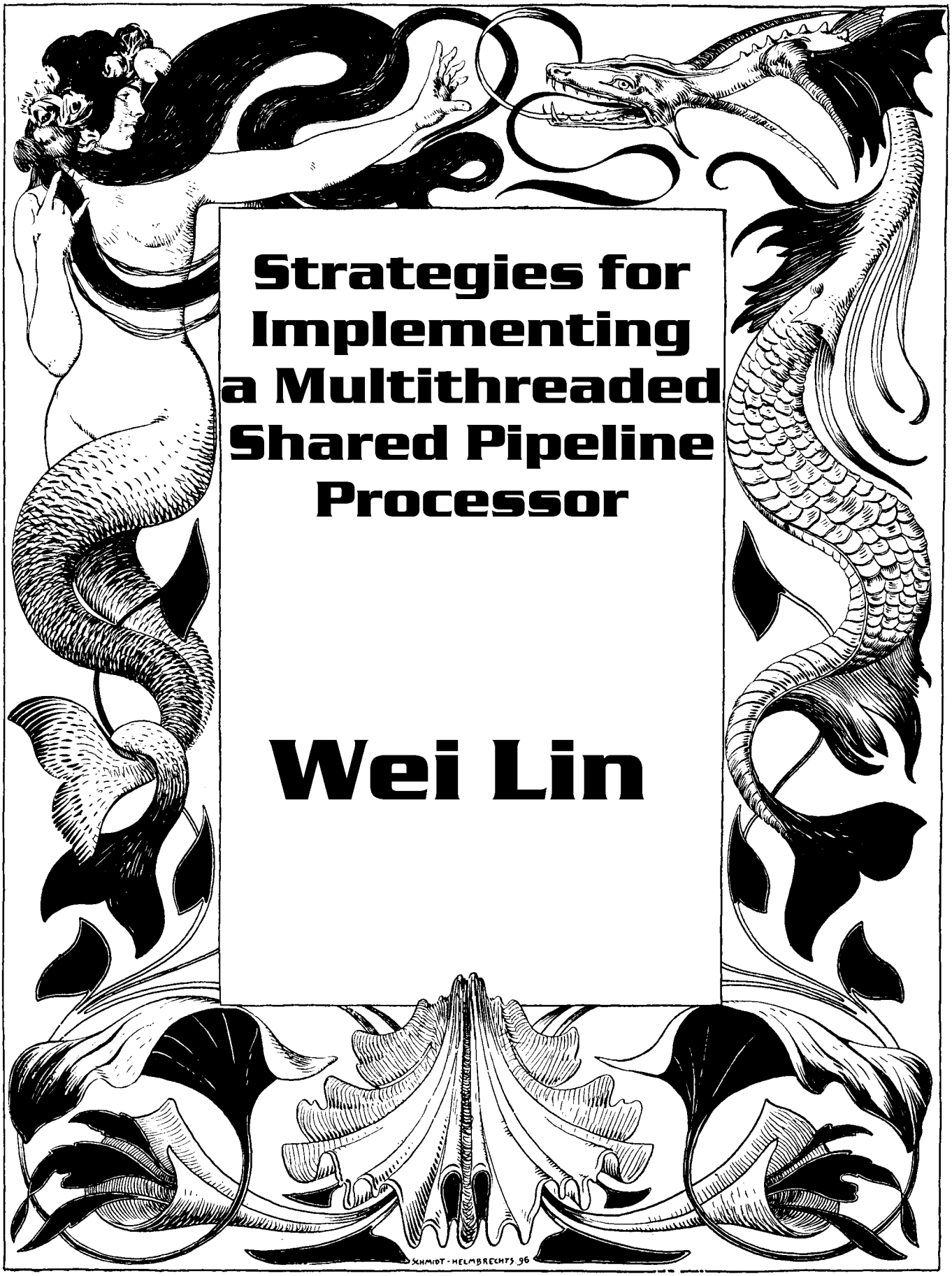


Strategies for Implementing a Multithreaded Shared Pipeline Processor

Wei Lin

Strategies for Implementing a Multithreaded Shared Pipeline Processor

Wei Lin

Peter M. Maurer

University of South Florida

Tampa, FL 33620

Ph: (813)974-4758

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

Abstract

The management of parallelism, the simultaneous management of multiple environments, and the synchronization of cooperating threads are some of the fundamental issues that need to be solved in a multithreaded pipelined architecture. In this paper, we present several different solutions to these problems which are currently implemented in a cycle-accurate software simulator of a multithreaded architecture called the SAM architecture. The simulator is a software prototype of a chip we expect to design and build within the coming year. The architecture contains two special instructions that allow parallelism to vary with time, and a two-level thread queue that allows the degree of parallelism to expand, theoretically, without limit, while still allowing for efficient dispatching of instructions. A concept called the short memory is introduced to support multiple environments. Because we intend to implement the SAM architecture as a real, functioning, general-purpose machine, we have also addressed such mundane issues as interrupts, traps, system calls, and process synchronization.

Keywords: Multithreaded Architecture, Multithreading, Dataflow, Multiprocessing

Strategies for Implementing a Multithreaded Shared Pipeline Processor

1. Introduction

The multithreaded architecture has gained much attention in recent years particularly in conjunction with dataflow research[1-4, 14,15]. In the dataflow model[5], a loop-free computation can be represented as an acyclic graph in which any node can be fired as soon as each of its input arcs contains a token. Such graphs exhibit much inherent parallelism. In fact, it is theoretically possible for all instructions in the program to execute simultaneously, restricted only by the flow of data between instructions. The von Neumann architecture, though well understood, lacks mechanisms to support this kind of fine-grained parallelism. Pure data-flow machines[9], on the other hand, will probably require more research to reach the same evolutionary state as von Neumann machines.

The multithreaded architecture can bridging the gap between pure dataflow and the von Neumann architecture. The multithreaded architecture builds on research that already exists for von Neumann architectures, and provides mechanisms for supporting fine-grained parallelism. They typically provide special instructions to create and destroy instruction streams, or *threads*, thus providing instruction-level parallelism within a single process[1]. All threads are assumed to execute in parallel, although the true order of execution depends on the underlying architecture. Compared with the dataflow machine, a multithreaded machine needs explicit instructions to control the parallelism, thus requiring more overhead than a dataflow machine. But because they are merely enhanced von Neumann machines, multithreaded architectures an avoid many of the difficulties that arise in the design of dataflow machines[6].

A popular scheme for the implementing a multi-threaded architecture is the shared pipeline[12]. The fundamental idea is that the architecture maintains several program counters, each of which represents an independent instruction stream, all of which share the pipeline. It has been shown that a shared pipeline model can yield high throughput with minimal cost[16]. In the USF multithreaded

architecture (called SAM) the active threads are threads contained in a structure called the thread queue. Threads are executed in parallel inside the pipeline, and are serviced in round-robin order. SAM also contains an instruction cache from which all instruction fetches are made. As shown in [16], when a sufficient number of threads are active, the pipeline remains full and that the ability to keep threads active is heavily dependent on a high cache hit rate.

SAM contains a single four stage pipeline. As in most RISC architectures[7,8], an instruction never remains in any pipeline stage for more than one clock cycle. Most instructions are 32-bits long and can be completed in a single pass through the pipeline. The structure of SAM is shown in Figure 1. It has a 32-bit address bus and a 32-bit data bus. The main memory is physically the same as that of most contemporary machines.

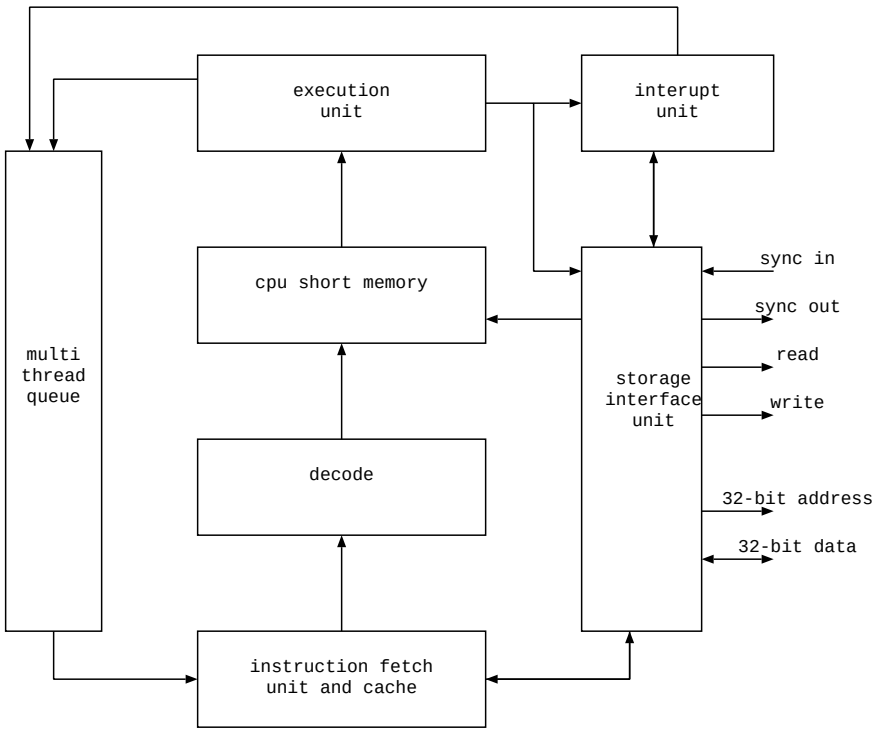


Figure 1. An overall structure of SAM

Two unique features of SAM are the thread queue and the short memory. The thread queue is used to control the multiple threads, while the short memory provides an efficient way to support multiple environments. The storage interface unit is also somewhat different from that found in

most contemporary microprocessors. It contains a storage operation queue, which buffers all the main memory access issued by cpu.

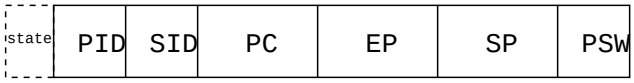
The hardware of SAM is reasonably simple so that the whole structure can be fabricated in one chip. Although this issue is not addressed here SAM also has features that allow several SAM chips to operate in a single system.

2. The Thread Queue

As stated above SAM[1] provides two instructions, Split And Merge, that allow the degree of parallelism to vary with time. The Split instruction resembles a conventional conditional branch, but has two successors instead of one. Split not only continues execution at the branch address, it also continues execution with the instruction following the Split. The Merge instruction reverses the Split operation. Merge has a single operand which works as a flag. When Merge is executed and its operand is zero, it sets the operand to one, and terminates the execution of the thread. When the operand is nonzero, it is reset to zero and execution of the thread continues with the instruction following the merge. These instructions allow the degree of parallelism to vary with time.

Since there can be many sequential threads of execution, each with its own Program Counter (PC) a storage area is required to hold all unique information pertaining to a thread. In SAM the thread queue is used for this purpose. The thread queue is a set of frames, each of which defines a thread. The frame structure is shown in figure 2(a). To support a thread, four registers are needed. The PC points to the next instruction to be fetched for this thread. The SP is a stack pointer as used in conventional machine. (Note that except under exceptional conditions, stacks cannot be shared between threads.) The EP is an environment pointer, which will be explained in the next section. The PSW contains status information for the thread, such as privileged state and condition codes. Two 16-bit registers (PID and SID) are used for thread and owner-process identification.

For every frame, there are eight states as shown in figure 2(b). *Empty* means the slot is not being used. *Wait* indicates the thread is waiting for some event and it cannot currently be executed. *Ready* indicates that a new instruction can be issued to the pipeline. *Running* means that an instruction is being fetched. After the instruction is fetched, the thread returns to the ready state. At any given time only one item in the thread queue can be in the running state. *Dequeue* indicates that the thread is terminating execution. It will remain in this state for one rotation of the fetch cycle and then be cleared to the empty state. *Sleep* indicates that the thread is being dumped to main memory. As soon as the contents of the frame are sent to the storage queue, the slot can becomes *empty*. *Suspend* is similar to *wait*, but is used to reduce busy waiting during process synchronization. *Awaken* state means that a dumped thread is being restored.



(a) The frame structure.

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 thread in the slot is waiting for some event and can not dispatch a new instruction.
dequeue	100	The thread is being terminated.
suspend	101	The thread is suspended
sleep	110	The thread is being dumped
awaken	111	A thread is loading into the slot from main memory

(b) The states of a frame.

Figure 2. The frame in the thread queue.

The thread queue is divided into two parts. One part is in the cpu and the other is in the main memory. *Active* threads must be in the cpu part of the queue. Swapping threads between the cpu thread queue and the main memory is discussed in section 6. The fetch cycle processes the cpu part of the queue in a circular fashion. Assuming there is at least one active thread, an instruction

is fed into the queue each clock cycle. Wait, Empty, Suspend and Sleep slots are skipped without wasting clock cycles. When a thread is generated it will be assigned to a slot in the cpu thread queue. When a thread is merged or dequeued, its slot will be set to empty.

To support the programming of the above architecture, four special instructions have been designed. They are *split, merge, deq, deqpid. Split* has the format of *split x* where *x* is an address. When this instruction is executed, it performs the following operations:

If the thread queue has an empty slot, it will:

(1) generate a stream identification number to SID in an empty frame;

(2) copy the current PID, PSW, and EP to the new frame;

(3) put x in the PC of the new frame;

(4) set the state of the new frame to ready.

Otherwise, the cpu will dump the thread of the lowest priority to memory. After a slot has been released, the cpu will execute steps (1) to (4) above.

Merge has the same format as *split*, but is treated differently. Furthermore, because the merge instruction is far more likely to cause pipeline hazards than other instructions, we are currently investigating alternatives to the operation of this instruction. The following technique could be used to support the merge operation under certain restricted circumstances, but additional work is needed in this area.

A merge table is used in the cpu to support the merge operation, which is shown in figure 3. The merge table allows the *the test and the set or reset* to be finished in one clock cycle. The instruction *merge x* will perform the following operations:

(1) If x is in the merge table, x is erased and thread continues;

(2) If x is not in the merge table, it is put into the table and the thread is terminated.

This modification of the original proposed merge operation[1] has two advantages:

1) Easy to program. The user can use the same x as they used in the *split*.

2) Pipeline hazards are avoided.

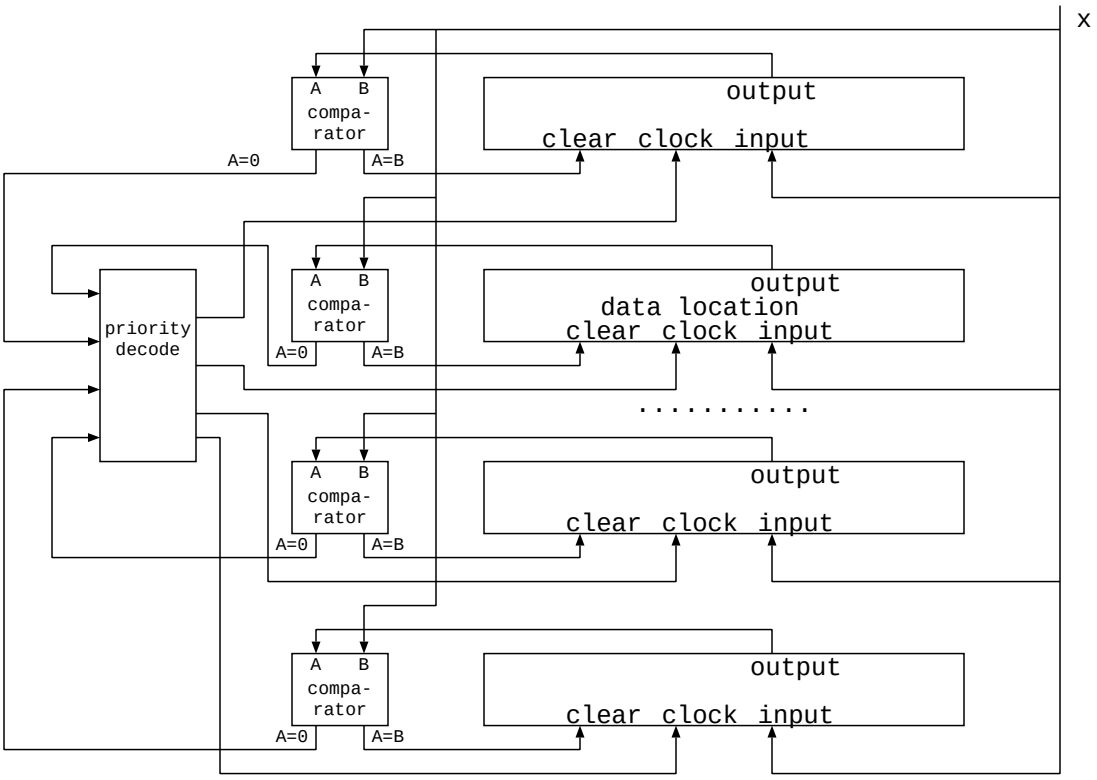


Figure 3. Merge table.

Unfortunately, this technique can have problems if the merge instruction is used in unconventional ways, because it assumes that the merge operand is initialized to a non-zero value. This approach will fail if it is necessary to initialize a merge operand to zero.

The *dequeue* instruction provides another approach to terminate a thread. Two forms of dequeue instruction are used. *deqpid N* will terminate the threads whose PID is equal to N. *deq* will terminate the current thread. These two instructions are used by operating system and interrupt subroutines. Unfortunately, the cpu has no knowledge of those threads that have been dumped to memory. Therefore, the dequeue instruction cannot delete these threads. For that reason, a dequeue table is provided which contains the PID of each dequeued thread. When a *deqpid N* instruction is executed, N is put into the dequeue table. If there are no memory threads, the dequeue table is purged. When a frame is called back from memory, its PID is compared with dequeue table. If there is a match, this thread will be discarded. If the dequeue table overflows, it will cause an interrupt, the interrupt subroutine can use this pid number to clear the memory part of the queue.

3. Short Memory

Another fundamental issue for a shared pipeline architecture is supporting multiple environments. An execution environment is a set of temporary memory locations used to distinguish one computation from another, even if the same code is being executed by two different threads. In conventional machine, the cpu registers are used for this purpose, requiring register saves and restores for process switches and subroutine calls. However this approach can not be used in SAM, because the environment may change with every clock cycle. Another approach is to use cache instead of register file. Since the translation speed and the complexity of a pure data cache affects system performance and cost, it was considered as impractical environment management[17]. Thus, SAM uses a concept called *short memory* to solve the problem.

Instead of using registers, SAM has a cache-like small memory inside the cpu, which is organized as illustrated in Figure 4. Short memory consists of several 256-byte blocks. Each block has a 24-bit tag. Each thread a 256-byte block of memory called its short address space. Whose address is contained in the environment pointer or EP. 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 its short memory beyond the 256-byte limit.

As in most architectures, there are two reasons for using a short memory (or a conventional register file). The first is to reduce the size of instructions thus reducing the time required to fetch an instruction. In the SAM architecture, a short address is 8-bits long which requires less fetch time than a full 32-bit address. The second to reduce instruction execution time. Typically data can be fetched from a register file much quicker than from main memory, and the use of several internal busses allows two or more operands to be fetched simultaneously. SAM can fetch two operands from short memory in one clock cycle. (Short memory is dual ported.) The short memory is in

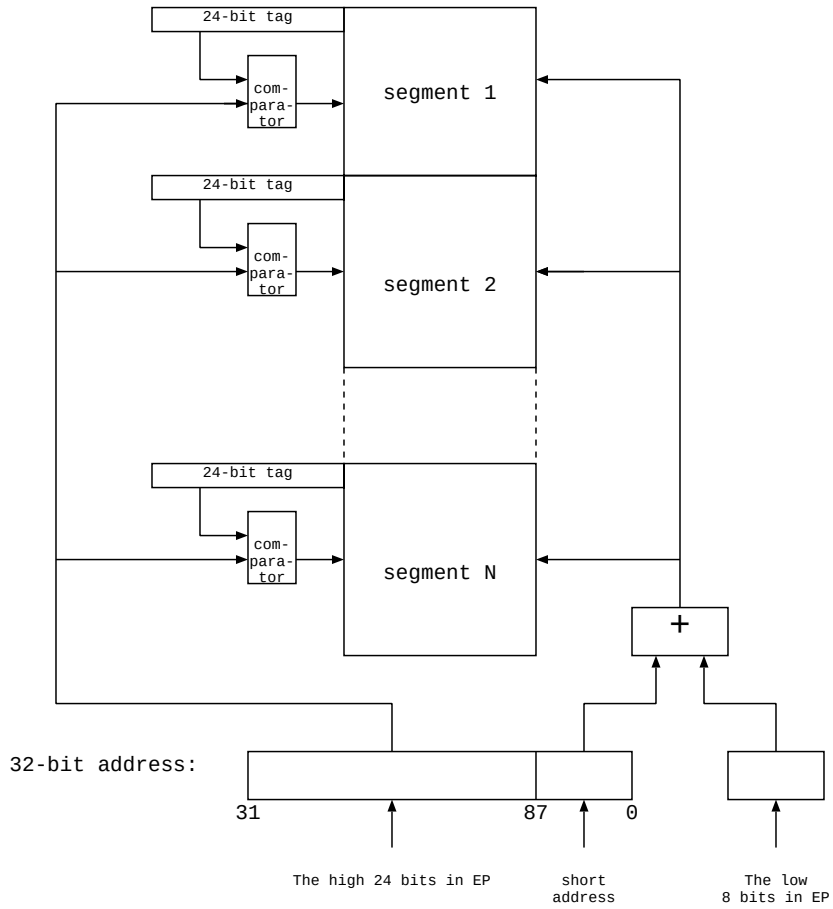


Figure 4. CPU short memory organization.

on-chip which allows operands to be fetched without using a bus protocol. The full address of a short-memory operand is computed by *adding* the short address to the contents of the EP. As shown in figure 4, there are two steps in addressing short memory locations, locating the correct segment by comparing the high order bits of the full address with the segment tags of the short memory (a fully associative match),and locating the position in the segment using the eight low-order bits of the full address. These two steps are carried out simultaneously. Instructions support short and long addresses. Short addresses refer to the short memory, while long addresses refer main memory. Each thread can use a portion of or the entire cpu short memory. It is also possible to partition short-memory blocks among threads that do not require 256 bytes of short memory.

The SAM short-memory can work in two modes. In register mode, short memory works somewhat like a register file. A thread can allocate a block of short memory by setting assigning a new

value to the EP. As long as there is a free block in the short memory, it will be assigned to a thread by setting the its tag to the new value of the EP. Any read-reference to a short memory address that precedes the first write operation to that address will produce garbage. The full 32-bit address generated during short-memory access *does not* correspond to a main memory address.

Compared with the conventional register file approach[3,10], this strategy has the following advantages:

1) There is no need to swap the register files when switching threads.

2) The size of the short memory space used for a thread is flexible, while in the conventional approach the size of the register file is fixed and every thread have the same number of registers whether it needs them or not.

If the system is working in register mode, dumping a thread to memory requires the saving of the short memory block associated with the thread.

In transparent mode, two additional features are added. The first is an auto-memory-load operation will be issued if an instruction reads a short memory location which has never been written. The memory address for the load operation is obtained by adding the EP and the short address. All subsequent reads from that location will obtain the current short-memory contents without accessing main memory. The second operation is a write-through to the corresponding main memory location for all short memory writes.

The primary difference between transparent short memory mode and an ordinary cache is that the contents of the cache is completely under user control. Furthermore, short memory has no line structure, so only those words that are actually accessed are loaded into short memory. The user is free to allocate this area to any suitable purpose. However short memory transparent mode differs from an ordinary cache in one important respect. Only short-memory references go through the cache, therefore if a thread accesses the same memory location using both short and long addresses, unpredictable results can occur. If a thread uses a long address to write to a short memory location, the value written may or may not be visible when accessed through the short memory. An example of transparent mode is shown in Figure 5.

```
EXAMPLE: A short SAM program that adds the contents in memory location
         1000 and 2000, and store the result in 1002.

     1) Under the register file mode:

        SETEP 0;            set EP to 0.
        MOVIWS 1000, 0;     move 1000 to location 0 in a short memory
                            block.
        MOVIWS 2000, 1;     move 2000 to location 1 in the block.
        MOVIWS 1002, 2;     move 1002 to location 2 in the block.
        MOVLIS 0,0,3;       move the content in memory location 1000
                            to 3 in the short memory block. ((0))->3
        MOVLIS 1,0,4;       move the content in memory location 2000
                            to 4 in the short memory block. ((1))->4
        ADD 3,4,5;          add the contents in 3 and 4 and store in 5
        MOVSIL 5,0,2;       move the content in 5 to the memory
                            location addressed by the content of 2.


     2) Under the cache mode:

        SETEP 1000;         set EP to 1000.
        MOVIWS 2000,3;      move 2000 to location 3 in the block.
        MOVLIS 3,0,1;       move the content in memory location 2000
                            to 1 in the short memory block, with a write
                            through, i.e., (2000)->1001.
        ADD 0,1,2;          ((EP)+0)+((EP)+1)->(EP)+2.
```

Figure 5. Example for different short memory modes

As shown in the Figure 5, transparent mode leads to simpler programming. Another advantage of transparent mode is that it eliminates saving the short memory contents if the thread is dumped to memory. On the other hand, write-through may cause a great deal of extra memory traffic, even if the thread is eventually dumped, since every write, including those for temporary variables must be recorded in the memory. This may have a detrimental effect on system performance if transparent mode is used indiscriminately [16]. One method to reduce memory traffic is to limit the write-through effect to certain instructions.

Although the current scheme for implementing the short memory is workable, there are many more questions to be answered. Short memory is currently the subject of a great deal of intense research. Describing the number of alternatives that we have either already explored or are intending to explore would require a lengthy paper in itself.

4. Interrupt and Critical Section Handling

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 will be used to determine the interrupt routine entry point. Since no thread is actually interrupted, it is not necessary to save the state of a thread, thus speeding up the servicing of interrupts. SAM provides two ways to terminate the servicing of an interrupt. One is to kill the thread by executing a dequeue instruction. The other is to use a merge instruction, which can provide a synchronization between a thread and a process waiting for I/O. The merge technique is also useful in synchronizing 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. An 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. As long as the maximum number of threads in the cpu part of the thread queue has not been reached, any interrupt will be granted. When no additional thread can be created in the cpu queue, a priority check is done, and the thread with the lowest priority will be suspended and dumped to memory. The interrupt thread will be inserted into the empty slot in the active state.

The modification of the thread queue has the highest priority in the SAM architecture. During these operations no interrupt will be serviced, and no split instructions will be completed. On the other hand, it may be necessary to have a few slots to be reserved especially for interrupt threads for some critical operations. We are currently investigating the details of this topic.

Another important issue for a multithreaded machine is critical section handling. The basic requirement for the protection of a critical section is an indivisible execution of *test and set*. This is a bit more difficult in the SAM architecture, since in every clock cycle several different threads may be executing in different stages in the pipe. However, the short memory provides several different opportunities for solving this problem.

The current version of SAM adopts the semaphore strategy to synchronize the threads. The P operation is implemented by an *opp m* instruction. The operand m is a short memory address in either mode. *opp m* will decrease m by one, and test the result. If the result is larger than zero, the result is written back to m and the thread proceeds. Otherwise, the state of the thread will be turn into *Suspend*. SAM has a four stage pipeline, and has a simple pipeline interlock mechanism. The first two stages will service an *opp m* instruction as a branch, hence the thread will be blocked for the next four consecutive clock cycles. In other words, until the P operation is finished, no further execution is allowed in this thread. Two consecutive *opp m* execution will cause interlock on m. If short memory is in transparent mode, the cache miss in m will cause the current thread as well as successive threads executing on m to be placed in the *Waiting* state.

The V operation is carried out by an *opv m* instruction. It will increase m by one and write it back. If the result is larger than zero, all the suspended threads will be reset to *Ready* state. Figure 6 illustrates this operation. Using these two instructions, user can protect the critical section inside the memory. One advantage of this approach is that it avoid some busy waiting. This critical section handling mechanism works in a single chip mode. Again, this is a subject of ongoing research. Since the semaphore operations are expected to be used primarily by operating system processes, requiring them to operate in the same environment does not seem overly restrictive.

5. Thread Swapping

The *split* and *merge* mechanism allows for an unlimited degree of parallelism. However, the thread queue size in the cpu can not be unlimited. Therefore the queue is broken into two parts,

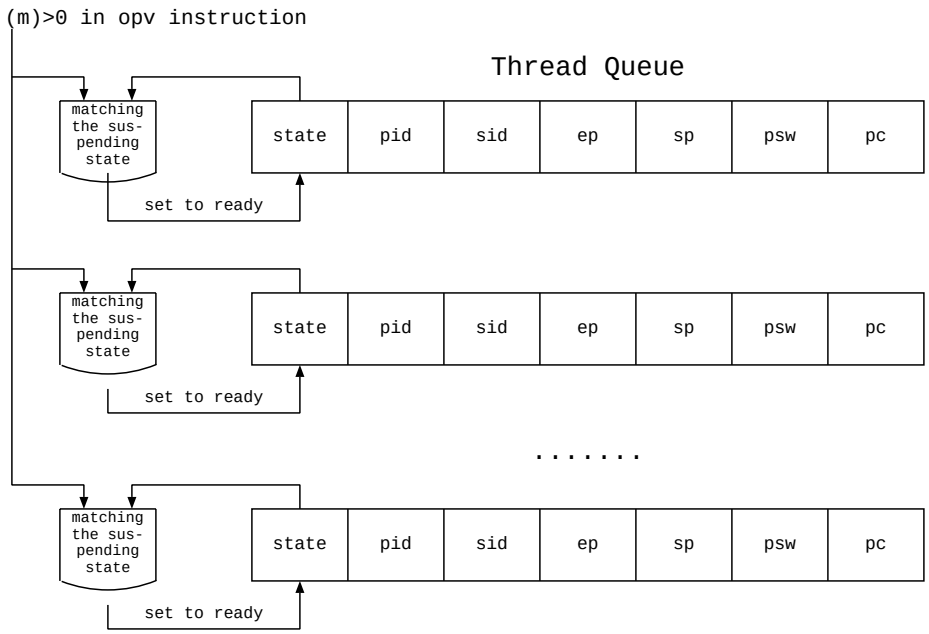


Figure 6. V operation

one inside the cpu and one in memory. This requires that there be a swapping protocol between the two parts of the queue. Normally we refer to this process as dumping and restoring a thread.

There are two steps required to dump or restore a thread. The first step is to save or load the thread queue frame of the thread. The second step is to save or load the short memory space. The time required for these operations is roughly comparable to the time required for a conventional process switch, and in some cases can be completed much more quickly.

The hardware used for this purpose in SAM includes three special registers. The register QP points the end of the thread queue in the memory. The QC serves as a counter. Whenever a thread is dumped, QC is incremented by one; whenever a thread is restored from the memory, QC is decremented by one. Another register DP is used to point to where the storage location of the short memory contents of the thread. Although DP is normally identical to the EP of the thread, both QP and DP can both be set by an operating system instruction. All the decrement and increment operations on the registers are can be performed by the hardware in one clock cycle. In addition, two fields are added in PSW. They are *offset* and *count* as shown in figure 7. The *offset* field shows the starting address in short memory space that needs to be saved during

dumping, while *count* indicates the number of words needed to be saved. These two fields are user programmable.

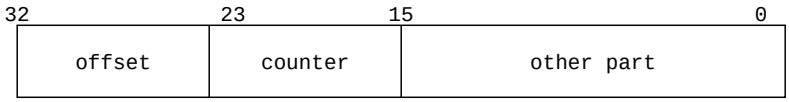


Figure 7. Additional fields in PSW.

There are several possible approaches to implement dumping and restoring in a multithreaded environment. One approach is by hardware. As soon as a dump is begins, the cpu could insert a block of operations to save the thread-queue frame and the short memory. However, SAM does not take this approach. The reason is that inserting a block of consecutive memory operations into the memory queue could seriously degrade pipeline performance. A second approach is to use an interrupt subroutine to do all the saving and loading. This approach will use less hardware but more time than the hardware approach. Furthermore, since the software must increment counters and do comparisons as well as perform moves, this approach mayh also degrade pipeline performance. The third approach, which is adopted by SAM, combines the hardware and software approach, and yields a better performance. (Note that even in the worst case, the effect on performance will be no worse than that which would be experienced during a process switch on a conventional single-threaded processor.)

In the SAM cpu thread queue, one slot, which is normally in the inactive state,, is reserved purely for dumping. When a *split* instruction or an interrupt causes an overflow in the thread queue in cpu, reserved thread is activated, and executes the program shown in figure 8. The first five instructions save the frame registers, and the rest store the short memory into main memory. The address and size of the short memory can be found in PSW. Since the information required to execute these operations are all inside the cpu, they can be generated automatically by the hardware. To do this, a 3-bit counter and a state bit are needed for each slot. When a thread-dump is activated, during the first five cycles to save the frame registers. Then, if the thread is operating in register mode, it will proceed to save the contents of the short memory. ach write to main memory is treated as an instruction. The counter field in the PSW is decremented for each write until the size becomes

zero, and the DP register is incremented. The final instruction is *deq*, which will reinitialize the slot and inactivate it.

```
STO ID;
STO EP;
STO SP;
STO PC;
STO PSW;

STO S;
......
STO S;

DEQ;
```

(a) The program for dumping a thread.

```
LOD PSW;
LOD PC;
LOD SP;
LOD EP;
LOD ID;

LOD S;
.......
LOD S;

normal execution
```

(b) The program for restoring a thread.

Figure 8. Dumping and restoring a thread.

Loading a thread is somewhat different from dumping. When there is an empty slot, the hardware will set the empty slot to the *awaken* state. Slots in the *Awaken* state will be scheduled as if they were in the *ready* state. While in the *Awaken* state, the thread will first execute the program shown in figure 8(b). It will load the frame and, if in register mode, the short memory contents. Then it will change the *awaken* state to the *ready* state. If the thread is operating in transparency mode short memory locations will be flagged as unreferenced, but not loaded. Although it may seem that transparency mode has a big advantage over register mode, at least in terms of dumping, further analysis will reveals that there are drawbacks to this scheme.

We define the *instruction dispatch interval* to be the minimum number of clock cycles before the next instruction can be dispatched. As presented in [16], the instruction dispatch delay caused by a cache miss is

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

Where m is number of active threads, and n is the number of the pipe stages. 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 pipeline hazard occurring at time t. The cache block size is l, while x is the number of clock cycles required for a memory operation.

The delay caused by a pipeline hazard is

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

The delay caused by a branch is

$$\tau_d^i = \text{n}$$

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

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

The following theorem shows how these parameters are affected by thread dumping and restoring.

Theorem 1: *Let $v_c^i(t), v_d^i(t), v_b^i(t)$ be the dispatch delays caused by cache misses, pipeline hazard and branch for a thread i, assuming that thread dumping or restoring is taking place. Then*

$$v_c^i = (m-2)x[p_c(l-p_d)+p_d]+x+lx/2.$$

$$v_d^i = (m-2)x[p_c(l-p_d)+p_d]+2x$$

$$\tau_d^i = n.$$

Definition *For a given thread i, the dispatch delay factor is defined as $r^i(m) = \lfloor \tau^i/m \rfloor$.*

During thread dumping or restoring, it is denoted as $R^i(m) = \lfloor v^i/m \rfloor$.

In round-robin scheduling, $r^i(m)$ identifies when the next instruction can be dispatched. If $r^i(m) = 0$, then an instruction can be dispatched during the next clock cycle.

Theorem 2 *Based on the above thread swapping protocol, if $p_c(l-p_d)+p_d < 1$, $r^i(m)-R^i(m) = 0$, when m goes to big.*

Theorem 2 shows that the register mode thread swapping may not affect the pipeline performance. So the total cost for swapping is 10+2k clock cycles, where k is the size of short memory.

The proofs of the above theorems, which are omitted due to lack of space, are straight forward based on the formulas in [16].

In transparency mode short memory, the swapping operation is much simpler. However, write through will greatly increase p_d, which will affect the important condition: $p_c(l-p_d)+p_d < 1/$. In that case, the overall performance could be greatly degraded[16]. One way to reduce write-through

traffic is to restrict the write-through property to a few instructions. More work is needed in this area.

Another problem with thread swapping is the execution of *split* instructions. During the swap operation, *split* must be blocked. Since the number of active threads is limited, and the swapping operation is short, a split buffer of limited size could be used to buffer these operations. An alternative solution is to inhibit the *split* execution during swapping. This will affect the pipeline performance, in a way that has not been included in the above theorem. However, since the *split* is not executed frequently, its effect can probably be safely ignored. Another problem that can occur when using a merge table is merge-table overflow. In this case SAM will dump the thread causing the overflow to memory, and reexecute the merge once the thread is restored. Our research on the merge table is in its preliminary stages, so there are still several questions to be answered about its management.

6. Conclusions

A functional simulator based on above ideas has been built. The experiments on the simulator with a limited number of threads[16] exhibit encouraging results. They suggest that with a good cache hit rate and appropriate software design, the pipeline can be kept full most of the time.

Compared with dataflow machine, SAM architecture less expensive both in terms of hardware cost and the research and development required to create a fully functional hardware model. SAM can efficiently support interrupt and I/O operations, and is a general purpose machine. SAM can support dataflow-like parallelism, while its hardware structure is almost as simple as that of a conventional RISC machine. Although instruction level parallelism is not as simple as that of a data-flow machine, the only overhead is the split and merge instructions themselves. In addition, the purely sequential program is supported by the SAM architecture, and if desired, it could be run as if it were an ordinary sequential processor. Thus it is possible to maintain software compatibility with existing sequential machines. Although we have provided an entirely new instruction set for

the SAM architecture, it is theoretically possible to alter the instruction set of an existing machine to include many of the features provided by the SAM architecture. This implies that instruction-level compatibility may be possible between existing microprocessors and multi-threading machines such as the SAM architecture.

Compared with RISC machines, multi-threading is a better solution to the branch problem than the delayed branch. When using delayed branches, it is impossible to avoid using at least some nop instructions after the branch instructions[13]. The overall pipeline throughput of a multithreaded machine should be able to exceed that of a RISC machine.

Cache-miss-driven scheduling[3], and priority scheduling[11] are used in some multithreaded machines. Such schemes are used to keep the same thread active as long as possible, thus reducing the number of register-file swaps required. However, because it is not always necessary to dump the short memory to main memory, short memory can often out-perform a register-file system regardless of which scheduling technique is used. Furthermore, round-robin scheduling supported by short memory is much easier to implement than priority scheduling, and can achieve optimal performance[16], provided that sufficiently many threads are active.

Another important possibility is this that the SAM architecture may be able to reduce a large amount of operating system overhead. We believe that the thread queue structure and short memory concept provide better support than the conventional sequential machine for multitasking and multithreading.

Our current experiments are focused on a single pipeline architecture, which can be fabricated in a single VLSI chip. However, we are also investigating features for supporting multiprocessor systems using multiple SAM processors. We believe that the two-level thread queue and the short memory provide efficient support for multiple pipelines in several chips. Future research will investigate the concept of multiple SAM-based multiprocessors.

References

1. P.M. Maurer, Mapping the Dataflow Computation Model into an Enhanced Von Neumann processor, Proceedings of International Conference on Parallel Processing, 1988,

2. H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads, The 19th ACM International Symposium on Computer Architecture, May 1992, pp.136-145.

3. M. Sato, Y. Kodama, S Sakai, Y. Yamaguchi, and Y. Koumura, Thread-based Programming for the EM-4 Hybrid Dataflow Machine, The 19th ACM International Symposium on Computer Architecture, May 1992, pp.146-155.

4. R.S. Nikhil, Arvind, Can dataflow Subsume von Neumann Computing?, The 16th ACM Annual International Symposium on Computer Architecture, May 1989, pp.262-272.

5. J.B. Dennis, Data flow supercomputers, IEEE Computer Magazine, Nov. 1980, pp.48-56.

6. 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.

7. D. Patterson, C.H. Sequin, A VLSI RISC, IEEE Computer, Vol.15, No.9, 1982, pp.8-22.

8. J.L. Hennessy, VLSI Processor Architecture, IEEE Transactions on Computers, Vol. C-33, No.12, 1984, pp.1221-1246.

9. Arvind, L. Bic and T. Ungerer, Evolution of Data-Flow Computers, Advanced Topics in Data-Flow Computing, Prentice Hall, 1991, pp.3-34.

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

11. M.K. Farrens and A.R. Pleszkun, Strategies for Achieving Improved Processor Throughput, The 18th ACM International Symposium on Computer Architecture, May 1991, pp362-369.

12. W.J. Kaminsky and E.S. Davidson, Developing a Multiple-Instruction-Stream Single-Chip Processor, Computer, Dec. 1979, pp.66-76.

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

14. R.S. Nikhil, G.M. Papadopoulos, and Arvind, *T: A Multithreaded Massively Parallel Architecture, The 19th ACM International Symposium on Computer Architecture, May 1992, pp.156-167.

15. G.M. Papadopoulos and R. Traub, Multithreading: A Revisionist View of Dataflow Architecture, The 18th ACM International Symposium on Computer Architecture, May 1991, pp.342-351.

16. Wei Lin, Peter M. Maurer, SAM: A Multithreaded Pipeline Architecture for Dataflow Computing, Technical Report, Computer Science Department of USF, August 1992.

17. D.R. Ditzel, H.R. McLellan, "Register Allocation for Free: the C Machine Stack Cache," Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, pp. 48-54.