Center for Supercomputing Research & Development

Center for Supercomputing Research & Development
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign

CSRD Rpt. No. 597

UILU-ENG-86-8008

DOE/ER/25001--86

DE88 003527

THE STRUCTURED MEMORY ACCESS ARCHITECTURE: AN IMPLEMENTATION AND PERFORMANCE-EVALUATION

Joseph Cyr

August 1986

Center for Supercomputing Research and Development
University of Illinois
305 Talbot - 104 South Wright Street
Urbana, IL 61801-2932
Phone: (217) 333-6223

This work was supported in part by the National Science Foundation under Grant No. US NSF DCR84-10110, the U. S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001, and the IBM Donation, and was submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Electrical Engineering, August, 1986.

[illegible]

THE STRUCTURED MEMORY ACCESS ARCHITECTURE: AN IMPLEMENTATION AND PERFORMANCE EVALUATION

BY

JOSEPH B. CYR

B.A., Aurora University, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1986

Urbana, Illinois

ABSTRACT

The Structured Memory Access (SMA) architecture implementation presented in this thesis is formulated with the intention of alleviating two well-known inefficiencies that exist in current scalar computer architectures: address generation overhead and memory bandwidth utilization. Furthermore, the SMA architecture introduces an additional level of parallelism which is not present in current pipelined supercomputers, namely, overlapped execution of the *access* process and *execute* process on two distinct special-purpose, asynchronously-coupled processors. The Memory Access Processor (MAP) executes the access process which is that portion of the instruction stream that is involved in instruction and operand fetching and storing. The Computation Processor (CP) performs the "useful" computations on the operands fetched by the MAP, i.e., executes those instructions that perform computations and tests on program data. Each processor executes a separate instruction stream to perform its specific task which, together, are functionally equivalent a conventional program.

By using simulation results, the MAP is shown to expedite processor-memory traffic by efficiently computing instruction and operand addresses using special-purpose pipelined function units (i.e., the Address Generation Unit and the Instruction Fetch Unit), and at the same time, reduces the demand on memory bandwidth by requiring less interaction with memory to support the access process. Our simulation results show that, for typical numerical programs, the MAP is capable of achieving *slip*, i.e., running sufficiently ahead of the CP, so that operand fetch requests for data items required by the CP are issued early enough and rapidly enough for the CP rarely to experience any memory access wait time. In this manner the SMA tolerates long memory access time, albeit high bandwidth, paths to memory without sacrificing performance.

Comparison with the Cray-1 in nonvector mode shows that the SMA architecture's features provide improved performance in scalar processing over existing high performance scalar machines. Since the CP is rarely required to wait for operands to arrive from memory, its instruction issue rate is improved and, hence, function unit utilization is increased. The "dual" instruction stream feature, inherent in *decoupled access-execute* architectures, enables each SMA processor's program to be significantly smaller than the conventional single instruction stream program and also frequently allows two instructions to be issued in a single cycle. Speedups, including reductions in memory wait time, often exceed two.

Acknowledgements

I would like express my gratitude to Professor Edward S. Davidson for providing me with the opportunity to come to the University of Illinois and work at the Center for Supercomputing Research and Development. His advice and assistance, both academic and administrative, are greatly appreciated; his insight and guidance, throughout the course of this research, was invaluable.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1.

INTRODUCTION

Much attention is being given to the development of computer architectures that execute vector and/or parallel programs efficiently. The performance of these supercomputers, however, is constrained by the well-known "Amdahl Effect." Code segments which are inherently scalar tend to dominate the performance of many parallel programs on these machines. Hence, it is an increasingly important objective in computer design to develop architectures which exhibit high performance for scalar tasks.

Current scalar architectures (e.g., VAX 11/780) do not take full advantage of the regular memory accessing patterns of most programs. The computation of operand addresses for array references, for example, typically constitutes a large portion of the CPU activity of many programs. Several memory references and arithmetic operations may be necessary simply to determine a single operand address. For data items contained in multidimensional structures, this overhead may constitute a substantial portion of the total execution time. More sophisticated scalar machines (e.g., IBM 360/91, CDC 6600, and many more recent high performance processors) address this problem by dividing program execution into I-unit and E-unit operations, and pipelining the flow through these units. Varying degrees of access and execution overlap are obtained depending on the density of dependencies in the instruction stream. Though the mechanics involved in computing operand addresses is not minimized by this approach, memory access wait time, as seen by the E-unit, is reduced. It is clear from these machines that substantial improvement in scalar processing performance can be achieved if the CPU overhead due to instruction and operand address generation can be minimized so that the fraction of time that the CPU is able to spend on productive computations can be increased. Moreover, if the amount of memory referencing required to support the I-unit were minimized, further performance improvement would result.

Pipelining has been successfully used to exploit parallelism within scalar instruction streams. However, many vector supercomputers are admittedly inefficient when processing scalar tasks; they are effective only when used to process *vectorizable* tasks. In this thesis we examine and develop an architectural technique, *decoupled access-execute,* used in the Structured Memory Access (SMA) and several other architectures to introduce a further specialized level of parallelism. By splitting a conventional scalar instruction stream into two, the machine can execute the resulting streams somewhat independently on two asynchronously interacting processors. Figure 1 represents a high-level model of this type of architecture. In essence, a conventional program is divided into an *access* process and an *execute* process [Hamm77]. Each instruction is split into two distinct subtasks which are executed in parallel; jointly, they perform the original function. In addition, each processor is specialized with hardware features for efficiently performing its assigned tasks. Specifically, the Computation

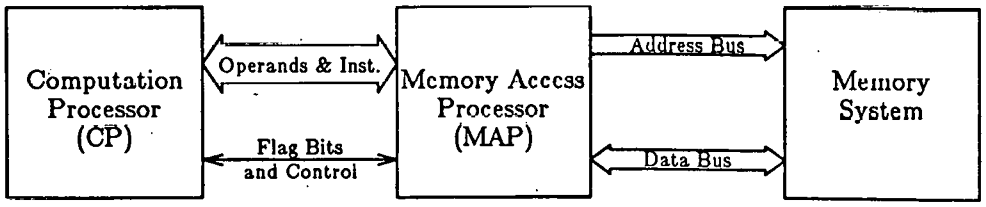

Figure 1. The SMA Organization — A high-level model.

Processor (CP) contains multiple pipelined function units, and the Memory Access Processor (MAP) contains an Address Generation Unit and an Instruction Fetch Unit which are designed to compute operand and instruction addresses efficiently while minimizing the total amount of memory traffic. The processors perform communication and synchronization through hardware queues that enable asynchronous execution of the access and execute processes. The key to the high performance of this architecture is the ability of the access processor to *slip* with respect to the computation processor, and run several instructions ahead, thereby supplying a continuous stream of operands to the computation processor which can then run uninterrupted. Studies have shown that speedups of greater than two are possible for some of the Lawrence Livermore loop benchmarks with such a decoupled access-execute organization [HsPG84], [Sohi83], [Smit84].

In Chapter 2 we discuss specific organization and implementation issues of the SMA architecture. The Memory Access Processor is treated in detail since it embodies the more novel aspects of the SMA architecture. We do not consider the particular design details of the Computation Processor since the design of powerful arithmetic and logic units is well known. It is sufficient, for our purposes, to assume that the CP has the attributes of some existing high performance machine; hence, a Cray-like scalar CP organization is discussed briefly. In Chapter 3 the results of detailed simulation experiments are discussed, and a performance evaluation, using the Cray-1 as a standard of comparison, is presented. Overall conclusions are presented in Chapter 4.

CHAPTER 2.

SMA SYSTEM ORGANIZATION

The SMA architecture is based on the fact that programs can be split into an *access* process and an *execute* process. Each process is executed on its own processor, each of which contains specialized hardware features designed to obtain high performance by exploiting the intrinsic characteristics of its associated process. A system block diagram of the SMA architecture is shown in Figure 2. The Memory Access Processor (MAP) executes the access process which is that portion of the instruction stream that is involved in instruction and operand fetching and storing. The Computation Processor (CP) performs the "useful" computations on the operands fetched by the MAP, i.e., executes those instructions that perform computations and tests on program data. This architecture is an adaptation of the organization originally proposed by Pleszkun and Davidson [Ples82], [PlDa83], [PSKD86]. The basic concept of SMA originated from the idea that improved performance could be obtained by reducing the overhead of the access process and maximizing the overlap between memory access and computation. These objectives were addressed by investigating methods of reducing the amount of memory referencing required to support the access process and executing the two processes in parallel on distinct processors. Kahhaleh and Sohi each undertook performance modeling experiments, which helped to identify system bottlenecks, and suggested enhancements to the SMA architecture [Kahh83], [SoDa84], [Sohi83]. The SMA architecture shown in Figure 2 also includes some features of the Decoupled Access/Execute architecture (DAE) proposed by Smith [Smit82], [Smit84]. The decoupled access-execute architecture has also been under investigation for VLSI implementation [GHLP85].

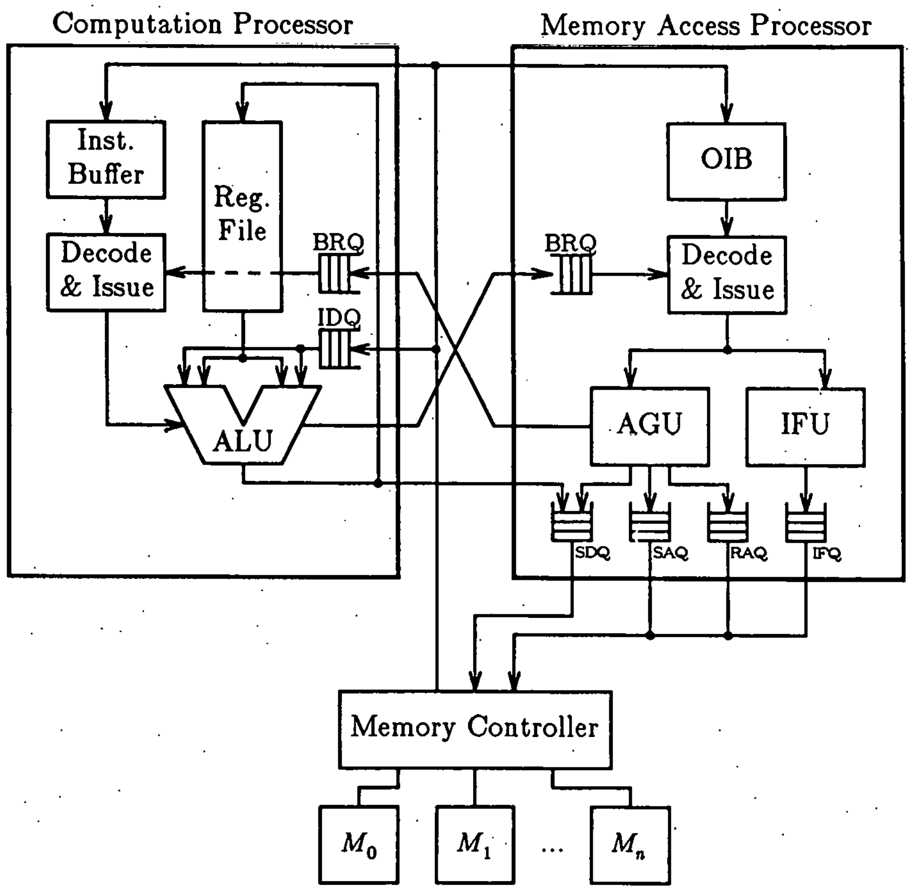

Figure 2. SMA Architecture Functional Block Diagram.

In this chapter we focus on the MAP organization and implementation issues. After a high-level overview of the operation of the SMA architecture, we discuss, in detail, some of the functional requirements and design tradeoffs, and the final recommended MAP organization. The CP organization is modeled closely after the Cray-1 scalar architecture, and therefore, is discussed only briefly. This chapter concludes with a programming example which highlights the salient features of the SMA architecture's operation and software requirements.

2.1. System Overview

Program execution is initiated by the operating system by setting up the MAP Instruction Fetch Unit (IFU) with information necessary to load the OIB with the starting instruction blocks of a program. Appropriate instruction blocks are sent to the MAP and the CP. Under the control of the MAP program, the Address Generation Unit (AGU) computes the addresses of operands that will be used in the CP. An operand fetch is initiated when the AGU places an address in the Read Address Queue (RAQ). The Memory Controller responds to the fetch request by receiving the address from the RAQ, fetching the selected word from memory, and forwarding the word to the CP's Input Data Queue (IDQ). The CP accepts operands from the IDQ, computes new values as dictated by the executing CP program and places output values in the Store Data Queue (SDQ). The AGU also generates memory addresses for the output values computed by the CP, and places them in the Store Address Queue (SAQ). A write to memory occurs when the corresponding SAQ and SDQ entries are both available at the heads of these queues. Memory requests are queued in the RAQ and SAQ in the order in which their addresses are generated by the MAP program. The RAQ has priority over the SAQ, and new read addresses are checked against pending writes in the SAQ. Thus, the correct sequence of memory references is maintained. In this manner, overlap between the access and computation phases of a program is achieved; instruction and operand addresses are computed in the MAP concurrently with the processing of the CP program. A more thorough description of the execution of an SMA program is given in

Section 2.4.

The execution paradigm outlined above is similar to that of the pipelined I-unit/E-unit organization of the IBM 360/91 which uses distinct function units for the instruction-handling and execution tasks. The SMA architecture, however, differs from the Model 91 in that each SMA processor executes a distinct instruction stream, which allows the processors to operate much more autonomously. Several performance enhancements are realized by this approach. First, it is known that the scalar performance of most machines is constrained by the maximum instruction decode and issue rate of one per cycle [Flyn72]. The effect of the so-called Flynn bottleneck is diminished in the SMA architecture by supplying two physical instruction streams; one to each processor. With this feature the SMA is able to double its maximum instruction issue rate. Second, since the sources of CP operands are contained only in the CP Register File or the head of the IDQ, and the only destinations are the Register File or the tail of the SDQ, the architecture of the CP is particularly amenable to a RISC implementation [PaSe81], [PaDi80]. The format of CP instructions contain only opcodes and register tags; no addressing modes are required for memory referencing since this is taken care of by the MAP. Benefits of a RISC implementation are a compact instruction set architecture and more economical hardware and firmware implementation due to fewer and simpler instructions. Third, though the usual data dependency and hazard problems exist in the CP instruction stream, the SMA architecture is able to speed up memory accessing by forwarding some previously computed operand results back to the CP quickly by implementing store-fetch forwarding in the MAP. By examining the contents of the SAQ for each read request, some memory referencing can be eliminated. An associative search of the SAQ determines whether the AGU has generated a read address that matches a previously generated store address that is still enqueued. When this condition is detected, the associative search logic signals the Memory Controller to abort the memory read request and forward the corresponding data item from the SDQ back to the CP's IDQ. If an RAQ-SAQ match is found, but the corresponding SDQ entry is empty, the Memory Controller waits for the data item to arrive and, in the meantime, is free to service the instruction fetch queue. Data forwarding can be resolved in the MAP, and no special tagging is required. This technique is similar to the forwarding that occurs with the "multi-access feature" of the IBM 360/91 [Ande67].

As a result of the data buffering that takes place in the hardware queues, the processors are capable of running asynchronously. The MAP can execute several instructions ahead of the CP limited only by 1) one of its queues becoming full, 2) the occurrence of a data dependent branch which requires information from the CP, via the Branch Queue (BRQ), in order to determine the flow of control, or 3) the situation where the access process contains more instructions and, consequently, runs slower than the computation process. Note that the first possibility does not present any performance degradation since it does not cause the CP to wait. A significant advantage of the SMA architecture is that, in the absence of data dependent branches which need to be resolved in the CP, the MAP instruction stream experiences few data dependencies (see Section 3.2). This is a result of the fact that most of the information necessary to support the access process is contained within tables in the AGU. After the AGU tables are initialized no further information is required from the main memory, and few instructions depend on previously issued instructions that require more than one cycle to execute. These phenomena, in the absence of the limitations sited, will enable the MAP to slip ahead and *prefetch* operands for the CP.

Branch control must be coordinated between the two processors such that each performs similar branches within its own instruction stream. Data dependent branches may be resolved in either the CP or the MAP. Conditional branches based on the value of program data contained in a memory location, or in a general purpose register, are resolved in the CP and communicated to the MAP by transmitting a bit to the MAP's BRQ. Conditional branches based on loop indices, or data structure dimensions, are resolved in the MAP and communicated to the CP similarly. For example, a common high-level programming construct is that of a Fortran DO loop that repeats until a loop index reaches its final value. For each iteration, the MAP tests an index register to determine whether the loop has been completed and sends an appropriate bit to the CP's BRQ to indicate the outcome of the test. The CP first executes its instructions corresponding to the DO loop, then executes a *bfq* (Branch From Queue) instruction. The *bfq* instruction causes the control unit either to branch back to the beginning of the loop and reexecute the loop instructions, or to continue with the next sequential instruction, depending on the bit value at the head of the BRQ. Thus, for each iteration of the loop, the CP determines whether to exit the loop based on queued branch test outcome information supplied by the MAP. The opposite case of a branch resolved in the CP is handled similarly; however, in this case the MAP executes the *bfq* instruction at the end of each loop iteration. For performance reasons it is desirable to permit the MAP to maintain slip by producing code wherein the maximum number of conditional branches is resolved solely in the MAP. A branch instruction that is resolved in the CP requires the MAP to stop and wait for the CP to "catch up" and transmit the outcome of the branch test. The MAP must suspend operand fetching during this interval; hence, the steady stream of operands to the CP is interrupted. After the branch is resolved in the CP, the CP will experience a memory wait time, slightly longer than the memory cycle time, waiting for the stream of input operands to resume. During this process, all slip is lost while the CP catches up, but once the branch is resolved, the MAP attempts to restore slip again. In contrast, the outcome of a branch test that is resolved in the MAP is generally determined before the CP reaches the *bfq* instruction. The MAP can proceed with instruction and operand fetching without delay, and the CP can resolve its *bfq* instruction in a single cycle. The MAP effectively performs branch lookahead for the CP and thereby maintains its slip in this case.

The peak performance of the SMA architecture is achieved when the CP is constantly supplied with operands (i.e., the IDQ is never empty). When this is the case, the CP never experiences any delays waiting for the memory to respond to operand fetch requests, the major cause of CP wait in conventional machines. We do not expect the SMA architecture to achieve this ideal operating flow continuously, due to branching and various overhead factors such as setting up the AGU and IFU; however, we do expect fairly long bursts of execution at peak rate. This expectation is justified by simulations that show speedups in excess of two for some programs, due to a combination of effective parallelism (less than two) and reduction of memory access wait time in the CP.

2.2. Memory Access Processor

The Memory Access Processor (MAP) is a special purpose processor designed to reduce the demand on the memory system bandwidth and to expedite instruction and operand fetching by employing efficient hardware mechanisms for generating addresses. The main performance objective in the design of the MAP is to issue operand fetch requests at a rate sufficient to keep the Computation Processor (CP) continually active. This goal implies fetching operands from memory at a rate equal to that which the CP consumes operands from the IDQ. If this objective can be achieved, the CP will run at the rate at which it can perform register transfers, and memory access will appear to be transparent. Ultimately, it would be desirable to *stream* operands to the CP at a rate that allows the CP to perform computations approaching the speed of a vector machine with a single memory port. For this to be possible, it is necessary for the MAP to issue fetch requests at a rate approaching one per cycle, and for the CP to contain pipelined function units and a sufficient number of internal registers.

The MAP contains three main function units: The Address Generation Unit (AGU) computes operand addresses, the Instruction Fetch Unit (IFU) computes instruction addresses, and the Operand and Instruction Buffer (OIB) stores the MAP program instruction blocks and immediate operands. Special purpose registers in the AGU are used to hold loop count variables, base addresses of scalar data areas, and data structure parameters. These AGU operands are fetched with special *load* instructions. On arrival at the OIB, these operands bypass the buffer and are stored directly into the AGU registers. Arithmetic hardware in the AGU generates operand addresses from the information stored in its registers, and most addresses are computed without the need for additional information from main memory, once the AGU information is initialized. The IFU executes *prefetch* instructions which cause instruction fetch requests to be issued for instruction blocks that will be needed by the MAP and the CP. The OIB receives the MAP instruction blocks fetched by the IFU. Instruction blocks containing loops are trapped in the OIB which enables the MAP to reexecute loops in a manner similar to the loop mode execution of the IBM 360/91 [Ande67]. The OIB achieves a high hit ratio due to the deterministic prefetching of instructions. Hardware queues buffer the memory requests produced by the AGU and IFU, and smooth the interface between the processors and the memory subsystem.

In the next three sections we discuss the functional implementation of the three main units that make up the MAP and address some of the overriding issues affecting the instruction set design and system software requirements.

2.2.1. Address Generation Unit

The Address Generation Unit (AGU) is an arithmetic function unit used to compute the addresses of scalars, vectors, and multidimensional data structures. Operand addresses generated by the AGU are placed in the RAQ or SAQ depending on whether the address pertains to a read request or a write request, respectively. Our goal in the design of the AGU is to issue one read or write request per memory cycle, thus, making most efficient use of the memory bandwidth and maximizing the rate of operand transfer to the CP. This requirement virtually dictates a pipelined implementation since address generation typically requires that several arithmetic operations be performed for each memory reference. Also required is a fairly large register set that holds vector parameters, indices, and base addresses. It is essential to maintain this information in fast registers in the AGU in order to reduce the amount of memory referencing required to obtain information needed to generate operand addresses [Ples82]. In addition to these special hardware requirements, it is necessary to define several special instructions for controlling the access process. These are discussed after the functional behavior of the AGU hardware is presented.

In general, computation of the address of an arbitrary element in an n-dimensional data structure requires n-1 multiplications and n additions. Since data structures are usually accessed in a nonrandom fashion, we can streamline the address computation process by storing some intermediate information (e.g., base addresses of data structure subdimensions and last address computed). There are many possible hardware configurations for performing the necessary arithmetic. The main tradeoff to be considered is hardware complexity versus software complexity. More powerful hardware capability generally reduces the burden placed on the software, whereas an economical hardware implementation tends to put more demands on the compiler. Furthermore, greater computational flexibility and ease of programming can be obtained at the cost of greater register requirements and a more complex instruction set architecture. In what follows, we examine several possible AGU implementations and the tradeoffs that we considered before reaching our final design decisions.

In the SMA architecture, values in memory are considered to be one of three types: 1) instructions, 2) scalars, or 3) vectors and multidimensional data structures. Instruction address generation is considered in the discussion of the Instruction Fetch Unit in the next subsection. The AGU computes addresses for the latter two.

To generate addresses of scalars efficiently, the AGU contains a small set of Scalar Base Registers (SBR) which can be dynamically loaded by software. Scalar data items are grouped into blocks by a compiler, and the base addresses of scalar data areas are loaded into SBR entries. References to scalars are performed by specifying an SBR and a displacement in conjunction with the *fetch* (or *store*) instruction. In essence, groups of scalars are treated like one-dimensional arrays. Displacements are relatively small integers requiring just a few bits for encoding; thus, any scalar reference can be specified in a single word instruction where the displacement is immediate data, and the base is indicated via an SBR tag. The AGU can compute the effective address of a scalar and issue a fetch request, simply by adding the displacement contained in the instruction to the contents of the specified SBR and placing the result in the RAQ. In this case, the only computation that is required is a single addition. The SBRs are also used to contain argument and stack pointers for subroutine and interrupt processing.

Generation of addresses for the elements of a vector or multidimensional data structure requires knowledge of the base address of the structure, the stride of each dimension, and the values of indices used to select a specific element. The AGU contains three sets of registers to hold this information. The Structure Definition Table (SDT) contains the characteristic parameters for one or more data structures, i.e., the base address and the dimension strides. The Access Pattern Table (APT) contains information that associates index registers with the particular dimensions of data structures defined in the SDT. Each APT entry also contains an offset value which is used to modify index values prior to address computation. This offset feature is frequently useful in numerical applications where index values, used to select particular array elements, are commonly modified by some small integer (e.g., A(i,j+1)). Index Registers (IR) contain the current value, final value, and step size of index values used, for example, in DO loop constructs of Fortran programs.

2.2.1.1. An Initial AGU Implementation

As a preliminary design for the AGU organization, we considered the hardware required for a pipelined implementation of the straightforward multidimensional array address computation algorithm (i.e., n additions and $n-1$ multiplications for each element of an n-dimensional structure). A function unit that implements this algorithm is shown in Figure 3. This AGU implementation contains much flexibility in addressing arbitrary elements of n-dimensional data structures and, at the same time, requires minimal effort from the

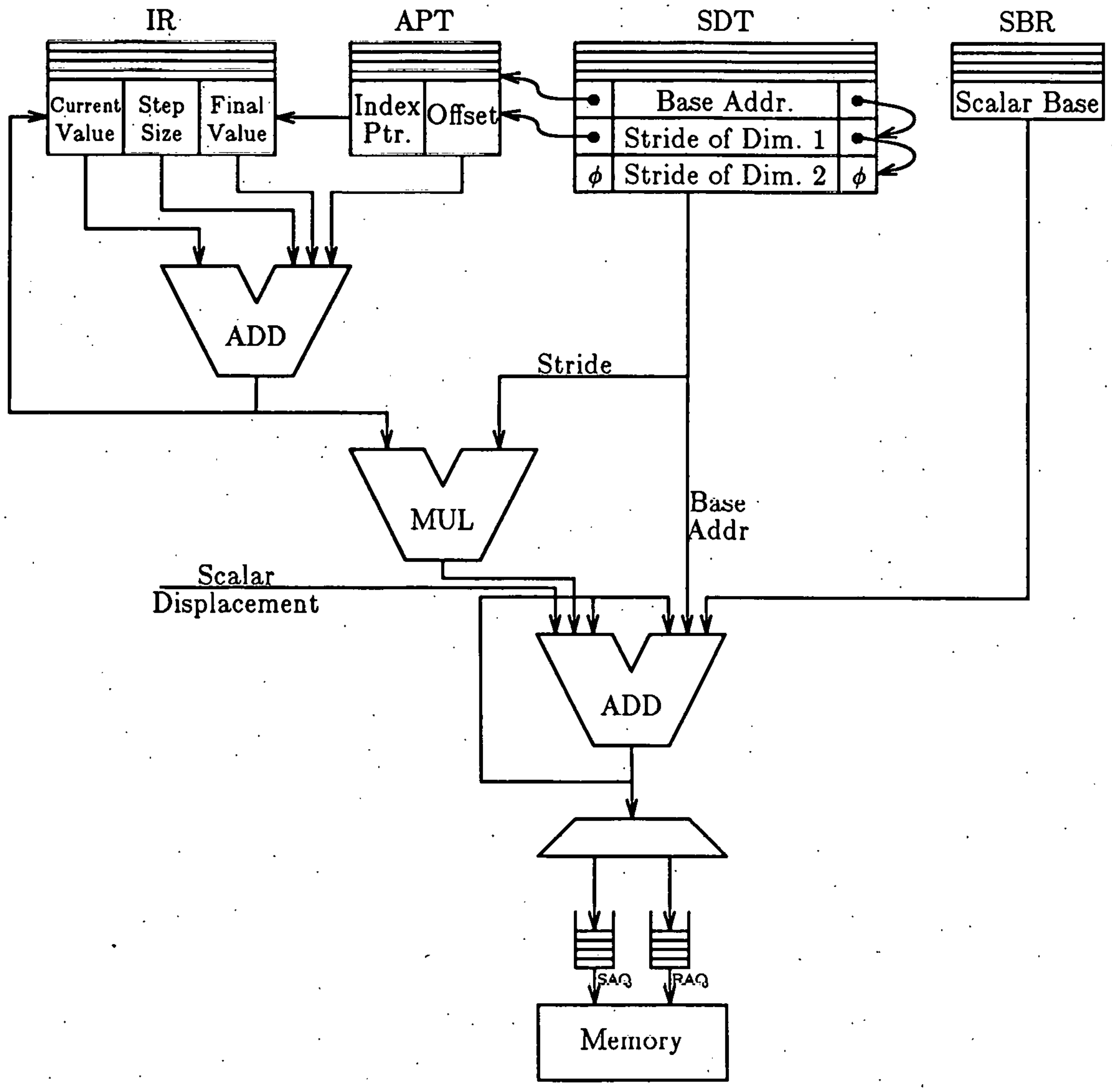

Figure 3. Address Generation Unit – An Initial Implementation.

compiler to generate code. Note that no effort was made to take advantage of the generally nonrandom nature of memory referencing present in most programs. As a consequence, the resulting hardware and control sequencing are relatively elaborate.

To fetch a data structure element requires specification of an SDT entry that contains the base address of the structure, in conjunction with the *fetch* instruction. To compute the effective address of an element of an n-dimensional data structure, the index values for each dimension (modified by APT offsets, if necessary) are multiplied by the corresponding strides and successively added to the base address of the data structure. That is, $EA = B_1 + \sum_{i=1}^{n} S_i * (I_i + O_i)$, where B_1 is the absolute base address of the data structure, and I_i, O_i, and S_i are the index, offset, and stride, respectively, of the i^{th} dimension. The base addresses and strides are stored in the SDT, offset values are stored in the APT Offset field, and the index values are stored in the IR Current Value field (see Figure 3). Each SDT register contains two pointers: one points to the appropriate APT register, which is used to enable the correct Offset and IR Current Value, the other points to the successor SDT entry, which indicates the registers to be used in the next phase of the computation. The control unit cycles through SDT entries until a null SDT pointer is encountered, and the summation is accomplished by the feedback loop at the second adder. Every data structure address is essentially computed "from scratch." No intermediate addressing information is retained, and no assumptions are made about the next address computation. Note that multiple data structure definitions in the SDT with similar traversal characteristics may share common APT and IR registers.

As a simple example of how address generation takes place in this unit, consider the computation of the address for an array element A(2,5), where A is a 32×32 matrix. This data structure is two-dimensional; therefore, three SDT entries are used: one contains the base address of the array A, and the other two contain the strides of the two dimensions. During the first cycle, the index register containing the Current Value 2 is gated into the first adder along with the corresponding APT Offset, which is 0 in this case. The IR and APT registers containing these values are enabled by the APT pointer in the SDT entry which contains the Base Address of A (refer to Figure 3). The base address of A is sent to the second adder. During the second cycle, the SDT entry containing the stride of the first dimension, which has the value 32, is gated to the multiplier along with the result of the first adder, which is simply 2. At the same time, this SDT entry enables the index register containing the Current Value 5, via its corresponding APT register, which, again, contains an Offset of 0. On the third cycle, the SDT entry containing the stride of the second dimension, which is just 1, is sent to the multiplier along with the result of the first adder which is 5, and the result of the previous multiply, 64, is forwarded to the second adder to be added to the base address of the structure. On the final cycle, the result of the second multiply, 5, is forwarded to the second adder to be summed with the result of the previous addition at that adder, which was the base address of A plus 64. The result of this final addition is the base address of A plus 69, which is the address of A(2,5). This result is then placed in the RAQ. (Note that this algorithm assumes that the array subscripting ranges from 0 to 31.)

Although this may not seem like a particularly expedient approach, there are some redeeming advantages. Because of the independence of each address computation, completely random references can be generated, i.e., index values can be modified arbitrarily between memory references with no additional overhead. This random access support is a very useful feature for some applications, and one that current vector processors do not have. The requirements placed on the compiler are simply to compute the stride of each data structure dimension from the high-level language declarations, determine the correct pointers for each SDT entry, and generate instructions to load these values into the SDT prior to use. Similarly, the APT and IR entries are taken directly from the source statements of the high-level programming language. Disadvantages of this implementation are that a good deal of multiplexing is required at the adder inputs, and the algorithm to control the sequencing of the computation is fairly complex. Furthermore, inclusion of multiplication hardware is undesirable in terms of cost and speed, and the summation process that takes place in the feedback loop requires several cycles (one for each dimension) causing the algorithm to be slower than we require.

2.2.1.2. An AGU Implementation with Intermediate Addresses

To streamline the address computation algorithm, we can do two things: eliminate the multiple cycle summation, and eliminate the need to perform multiplications. The former can be achieved by retaining information from previous computations. For example, accessing a two-dimensional array can be treated like a vector if an intermediate address, representing the base of the second dimension, is retained for use in subsequent address calculations. With respect to a conventional innermost loop, all accesses to an array lie within a single dimension, and are some offset distance (derived from the inner loop index, J) from this intermediate base address. The base address of the second dimension (B_2) is a function of the absolute base address of the data structure (B_1), the stride of the first dimension (S_1), the index governing the first dimension (I), and the offset of the index value (O_1), i.e., $B_2 = B_1 + S_1 * (I + O_1)$. Each time I is incremented in the outer loop, B_2 must be recomputed. The effective address of an array element can now be computed in a single pass through the pipeline; $EA = B_2 + S_2 * (J + O_2)$. Generalization of this technique to higher-dimensional data structures is obvious.

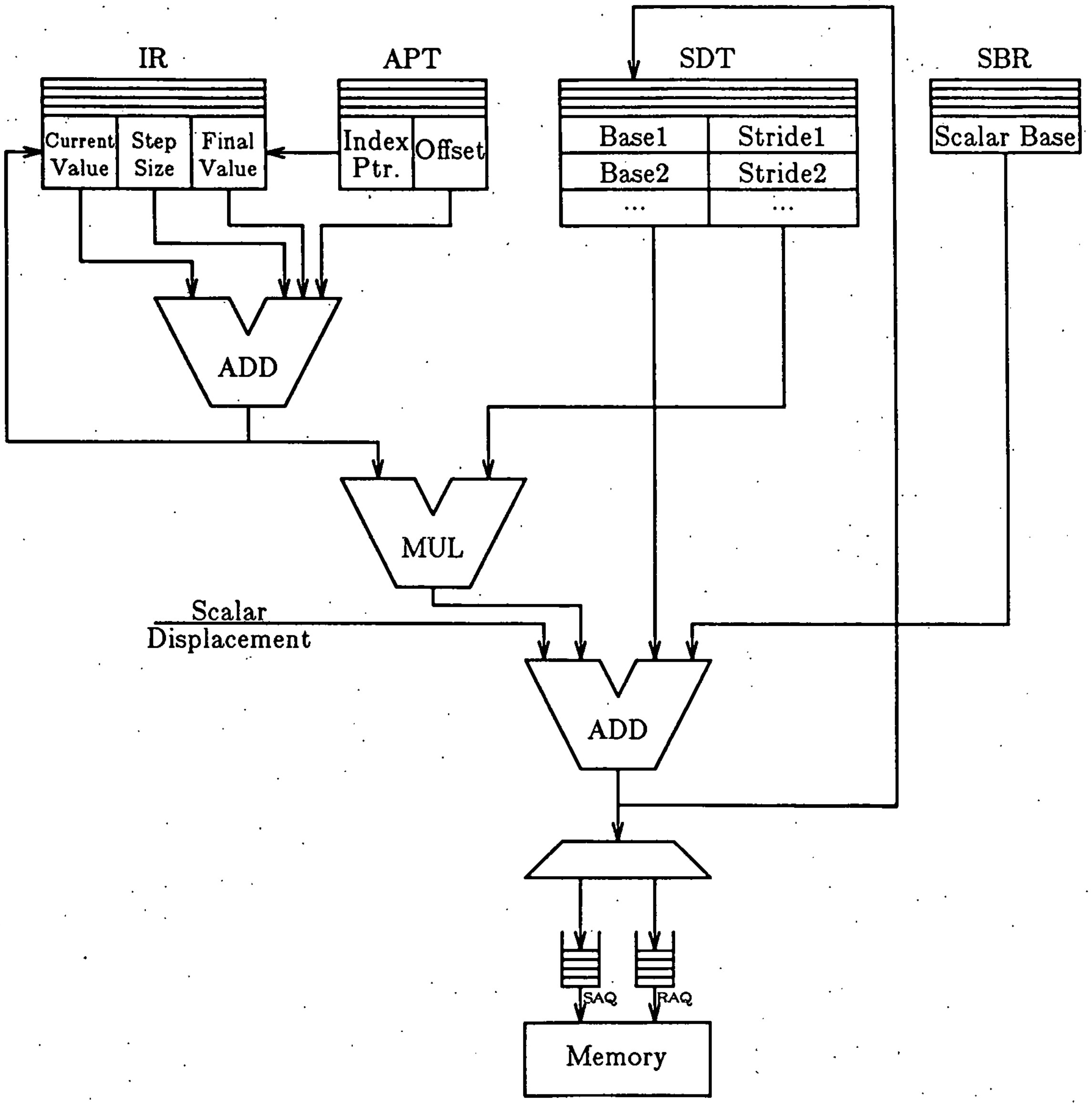

Figure 4. Address Generation Unit with Intermediate Base Addresses.

Figure 4 shows modifications to the AGU organization to support the address computation algorithm just described. In this organization, the SDT associates a base address and a stride for every dimension of each data structure. Each address computation involves only the highest dimension SDT entry; therefore, it is no longer necessary for the SDT registers to maintain links to subsequent entries. Furthermore, the binding of APT and SDT entries can be established at compile time as a part of the normal register allocation activity, so explicit pointers to APT registers are also unnecessary; however, instructions that cause addresses to be computed must now specify an APT register in addition to an SDT register. Finally, the output of the second adder is now available as an input to the SDT in order to update subdimension base addresses. The computation of an operand address is initiated by the *fetch* (or *store*) instruction with the SDT and APT registers corresponding to the highest dimension specified. The compiler is expected to insert instructions to recompute subdimension base addresses as required.

2.2.1.3. An AGU Implementation with No Multiplier

The necessity of performing multiplications can be eliminated in several ways. If the Base Address field of an SDT entry for the highest dimension is used to store the address of the last data item accessed, rather than the base address of that dimension, then subsequent addresses in the inner loop can be computed by adding the stride of the highest dimension to this value. Such an AGU is shown in Figure 5. Each new address that is generated is also stored back into the SDT entry representing the highest dimension base address. Here again, whenever an index governing a lower dimension is modified, the base addresses of all higher dimensions must be recomputed. This algorithm is very efficient in terms of hardware and control for very regular accessing patterns; however, it lacks flexibility in accessing when less regular patterns are required. One particular drawback is that the address calculation is *independent* of the index values. A program cannot modify index values without immediately updating the appropriate SDT registers, regardless of whether an operand address is to

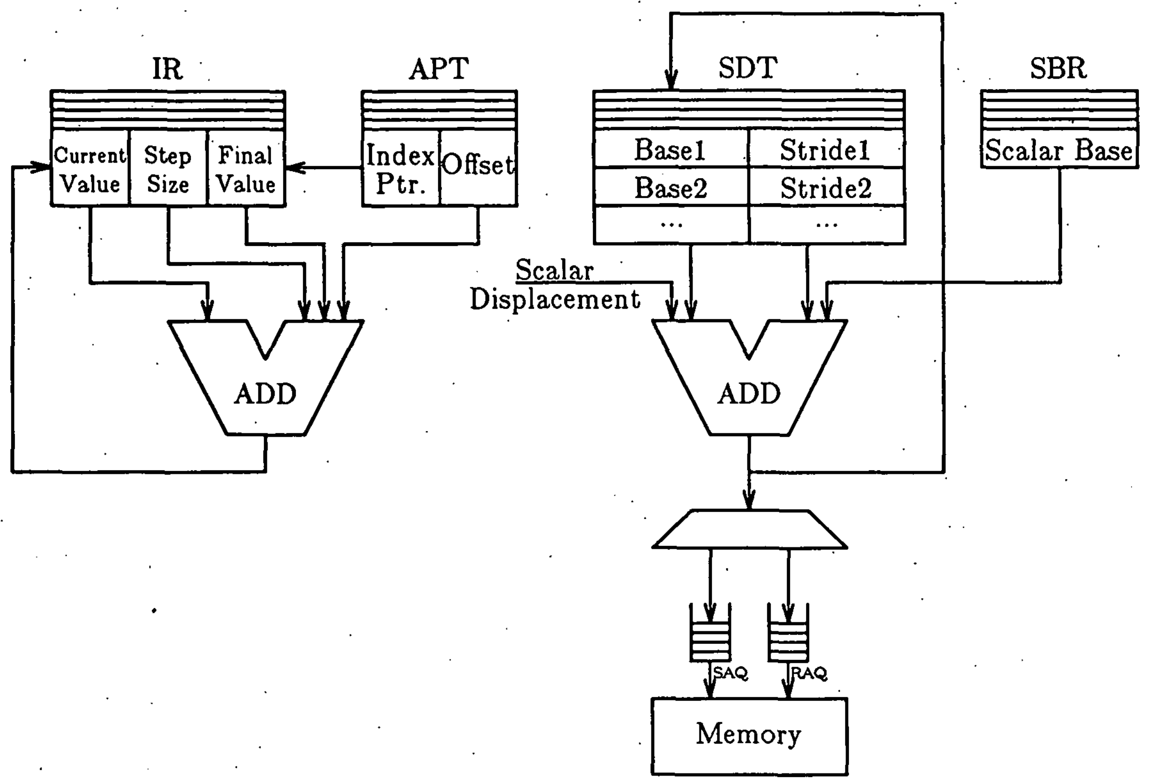

Figure 5. Address Generation Unit with No Multiplier.

be generated. For example, consider a DO loop where the instructions of the loop are contained within a conditional IF statement. For this common situation, operand fetches are required only for iterations when the IF condition is true. In this implementation, however, the SDT values must always reflect the current state of the corresponding index registers and, therefore, must be modified in every loop iteration since they cannot be computed, whereas in the previous AGU design, the computation of subdimension base addresses can be performed just on iterations where actual operand fetches are required. The use of index offsets is also complicated. A larger computational overhead can be incurred using this approach. For some programs the access process may contain enough unnecessary address computation that the MAP becomes the system performance bottleneck. Furthermore, this technique tends to reduce the addressing capability of the machine to that of a vector processor, which can efficiently access only a sequence of data items that are separated by a constant stride.

2.2.1.4. The Final AGU Implementation

A second technique for eliminating multiplications is to normalize the index register values and APT offsets to integral multiples of the stride of the dimension for which the index value is used. For example, in the two-dimensional case, the Current Value, I_1, varies from 0 to $(N_1 - 1)S_1$ in increments of S_1, where N_1 is the upper bound (Final Value) of the outer loop, and I_2 varies from 0 to $(N_2 - 1)S_2$ in increments of S_2, where N_2 is the upper bound of the inner loop. The Step Size of each IR is set equal to the dimension stride, and the APT Offsets corresponding to I_n are also normalized to multiples of S_n. Now the computation of B_2 is reduced to $B_2 = I_1 + O_1 + B_1$, and the effective address of an array element is $EA = I_2 + O_2 + B_2$. Multiplication of the sum of the current index value and the offset by the dimension stride, as is performed in the first two AGU implementations, is no longer necessary since this is implicitly done each time an index value is incremented. Similarly, it is no longer necessary to store the stride of each dimension in the SDT since these values are now contained in the IR Step Size field. This feature reduces the size requirement of the SDT by approximately 50 percent. Furthermore, the loop index values can be modified arbitrarily between address computations.

These normalizations can easily be accomplished by the compiler; however, they tend to limit the amount of IR and APT sharing that can occur between data structures, and consequently more index register space is required. For example, if two arrays are accessed in a nested loop, one in row-major order, the other in column-major order, a particular index variable will index the first dimension in one array, and the second dimension in the second array. Since the two dimensions have different strides and final values, the same IR cannot be used to compute addresses for both arrays. This problem can be solved by maintaining separate index registers for each array, with each being incremented by its corresponding stride. This "dual" index can be set up and maintained by the compiler, and need not be reflected in the high-level programming language. Either of these index registers may be selected for the loop exit test. This situation is actually quite common in conventional computers, and is handled by using more than one index register (if available) and several increment instructions per loop iteration.

A block diagram of this final AGU design is shown in Figure 6. The AGU contains the usual four register sets and two adders in cascade. This implementation does not fully realize the design objective of generating one operand address per cycle due to the pipeline being used to perform functions other than address generation, but, as shown in Chapter 3, the overall performance is generally good. In addition to computing operand addresses, the AGU pipeline also computes the addresses of operands required to initialize the AGU tables, computes the addresses of data structure subdimensions, increments (decrements) the index registers, and tests the IR Current Values against the Final Values in order to set branch signals. The cascaded architecture is very useful in implementing a pipelined increment and branch instruction used for program control. After the first adder performs the increment operation, the result is simultaneously stored back in the IR Current Value field and routed to the

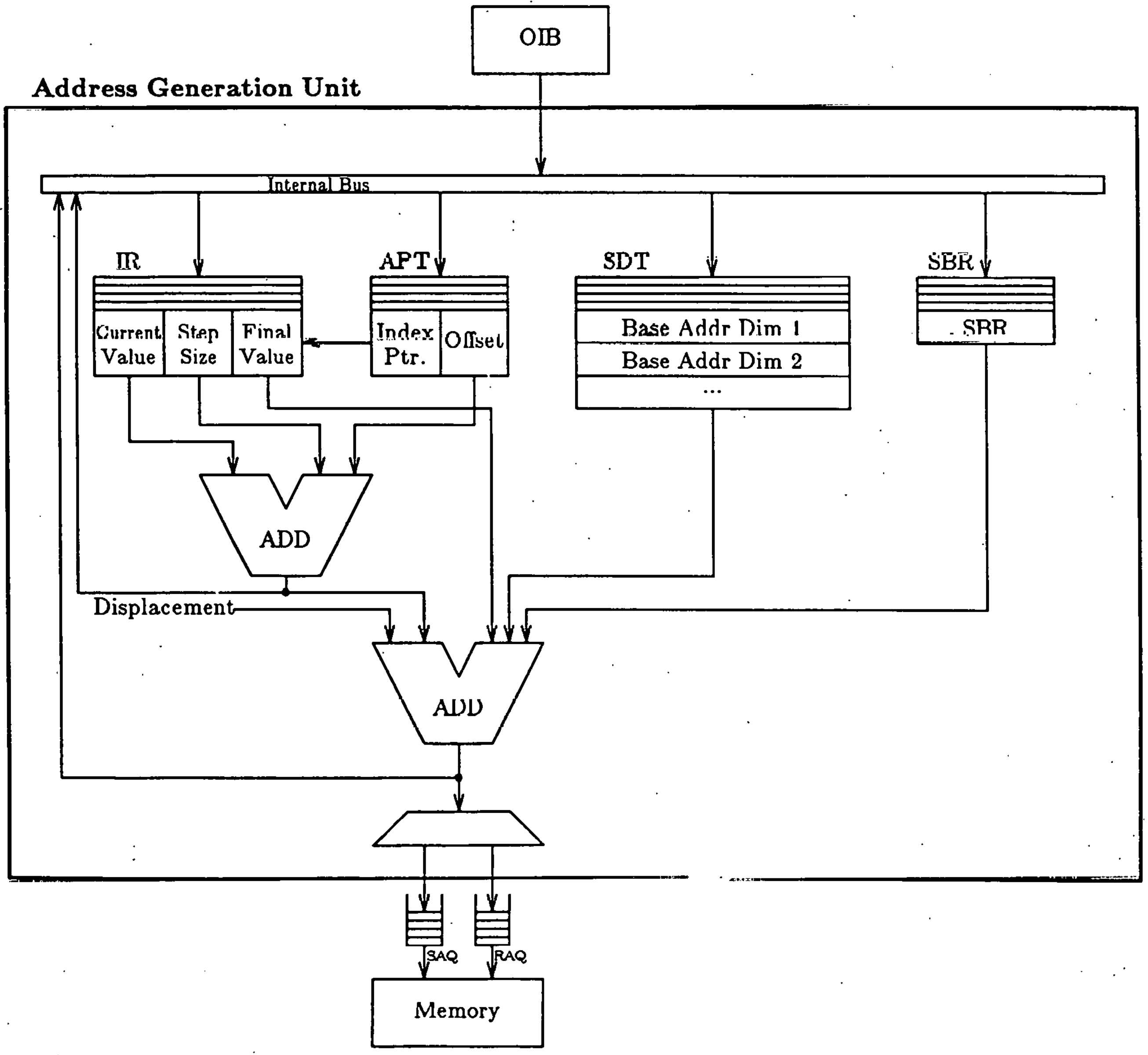

Figure 6. Address Generation Unit – The Final Design.

second adder along with the Final Value where the branch test is performed. The multifunctionality of the pipeline limits the address generation rate to be somewhat less than one per clock, but it also significantly reduces the complexity of the hardware.

To compute addresses of elements of an n-dimensional data structure, n SDT entries are used. One entry contains the absolute base address of the data structure, and the remaining $n-1$ entries contain the addresses of subdimensions within the data structure. Each subdimension base address is associated with a particular index register and is modified by multiples of the stride of the corresponding dimension as the program progresses. That is, each time an index register for a given dimension is incremented, the higher-dimensional base addresses are recomputed by additional instructions. The hierarchical relationships of the SDT registers corresponding to each dimension of a data structure are determined at compile time as registers are allocated. Overhead is reduced by assigning the most frequently changed index to the highest dimension. The compiler also normalizes the Final Values, Step Sizes, and Offsets to integral multiples of the strides of data structure dimensions for which they are used. All of these registers are loaded under software control.

2.2.1.5. Address Generation

As a simple example of address generation in the MAP, consider the fetching of the elements of every third row of an $N \times N$ matrix in row major order. The high-level MAP software to accomplish this task might be the following:

```
FOR I = 1 to N BY 3 DO
  FOR J = 0 to N-1 DO
    FETCH A(I,J+1)
  END
END
```

Note that the structure of the inner loop (i.e., using $J+1$ for $0 \leq J \leq N-1$) is unnecessarily complicated; however, it is useful for illustrative purposes.

The A matrix is two dimensional; therefore, two SDT entries are initialized. The first SDT register is loaded with the base address of the first dimension (B_1), which is the absolute base address of the data structure. The second SDT entry contains the base address of the second dimension, B_2, which is defined to be $B_1 + I + O_1$, where O_1 is 0, and I is initially 0, and ranges from 0 to $(N\text{-}1)S_1$ in increments of S_1 as the program runs. The base address of the second dimension, then, varies as a function of I (B_1 and O_1 are constants); each time index I is altered, B_2 must be recomputed. The SDT entry corresponding to B_1 (say, sdt1) is associated with an APT entry (apt1), which contains an Offset equal to 0, and a pointer to the index register containing I. Initially, this index register has a Current Value of 0, a Final Value of $(N - 1)S_1$, and a Step Size of $3{*}S_1$. The second SDT entry (sdt2) points to an APT entry (apt2), with Offset, O_2, equal to $1{*}S_2$, and a pointer to the index register containing J. The J index register has a Current Value of 0, a Final Value of $(N - 1)S_2$, and a Step Size of S_2. The normalized Offsets, Final Values, and Step Sizes of indices are determined at compile time from the loop bounds and data structure declarations.

In this example we have assumed that $S_1 = N$, and $S_2 = 1$; however, this need not be the case. In general, each data structure element can be of any length, and the loop bounds are not necessarily required to coincide with the data structure dimensions. Arbitrary subarrays can be accessed, with any ordering of the dimensions, by adjusting the initial and final values of index registers and assigning them to the dimensions of the structure appropriately.

The execution of a *fetch* instruction for array element A(I,J+1) is initiated by an assembly language instruction such as:

```
fetch   sdt2, apt2
```

The specified APT register (apt2) enables the index register corresponding to J. The Current

Value of J and the Offset contained in apt2 are gated into the first adder. The result is gated to the second adder along with B_2, the base address of the second dimension, contained in sdt2. The result of the second adder, the address of A(I,J+1), is demultiplexed to the RAQ which completes the address generation process for one data element.

To implement the complete code segment, we also need to initialize and increment the appropriate index registers, recompute B_2 as I changes, and conditionally branch to the beginning of each loop. The following code is representative of the corresponding MAP program.

```
1.              setup  x1, (sbr1)
2.   out_loop:  setup  x2, 3(sbr1)
3.              comp   sdt2, sdt1, apt1
4.   in_loop:   fetch  sdt2, apt2
5.              inc    x2, in_loop
6.              inc    x1, out_loop
```

The two *setup* instructions load the index registers x1, and x2, with initial information from memory corresponding to the I and J indices, respectively. The location of this information in memory is determined by adding a displacement to a previously loaded SBR. The *comp* instruction computes B_2 (stored in sdt2) from B_1 (sdt1) and I (pointed to by apt1). The inner loop is comprised of instructions 4 and 5. After initiating the operand *fetch*, the index register corresponding to J (x2) is incremented by its Step Size and compared to its Final Value. As long as the Final Value is not reached, a branch to the label *in_loop* is taken; otherwise, the program proceeds sequentially. The second *inc* instruction is reached when the inner loop exits. It increments I and closes the outer loop. Note that the index register containing J is reinitialized and B_2 is recomputed whenever the outer loop is executed. Additional instructions (not shown) are required to initialize the SDT and APT registers. These instructions are functionally similar to the index register *setup* instruction, and must be executed at some time prior to executing the above code segment.

2.2.2. Instruction Prefetching

The instruction streams executed by each processor are segmented into a sequence of instruction blocks [Ples82]. Briefly, a *basic* instruction block is a maximal-length ordered set of instructions such that all entry points into the set are to the first instruction, all exit points from the set are from the last instruction, and all instructions within the block are executed sequentially. We have expanded on this definition by eliminating the requirement that the instructions in a block be strictly sequential; *complex* instruction blocks may contain branch instructions as long as the targets of the branches are also within the same block. With this generalization, complex instruction blocks may contain nested loops, two or more adjacent loops, reconvergent branch trees, or combinations of these constructs and sequential code. Any instruction block that does contain a loop, however, must be terminated by a conditional branch or other specialized instruction that initiates an instruction block purge operation in the instruction buffer. Instruction block handling is discussed in the next section. The upper bound on the number of instructions in a block is determined by the size of the instruction buffer that receives the block.

Conceptually then, a program can be considered to be a logical sequence of instruction blocks. Program control flow can be modeled by a directed graph, usually containing cycles, where each vertex represents an instruction block. In general, control flow from one instruction block to the next can be generalized to three cases. A block can be terminated by 1) any nonbranch instruction, or 2) an unconditional branch instruction, where, in both cases, there is only a single successor block, or 3) a conditional branch, in which case two possible successor blocks exist. Given that, from the program graph, the control flow of any program is relatively predictable, it would seem advantageous to exploit this inherent program structure. In the SMA architecture we implement a mechanism to initiate prefetching of instructions into a high-speed buffer (OIB). This is accomplished by including a *prefetch* instruction as part of the basic instruction set of the machine. Each MAP instruction block contains at least one *prefetch* instruction which initiates the fetching of its successor instruction blocks. Each MAP instruction block also contains the necessary *prefetch* instructions required to fetch the corresponding CP instruction blocks. We rely on the compiler to delineate instruction blocks and to insert *prefetch* instructions into the MAP instruction stream such that the correct instruction blocks are fetched prior to being demanded for execution by the processors. This method of buffering instructions is very efficient since the prefetching is *not* hueristic, and only those instructions that are in the immediate flow of program control are fetched.

The control unit identifies *prefetch* instructions and issues them to the Instruction Fetch Unit (IFU). Figure 7 illustrates the data flow of the IFU. The *prefetch* instruction specifies the starting address (Block Addr) and the length (Block Len) of instruction blocks. The starting address can be specified as immediate data in the second word of a two-word instruction, or as an offset, to be summed with the contents of an SBR, in a single-word instruction. The IFU generates sequential addresses corresponding to the words of the instruction block and places the addresses in the Instruction Fetch Queue (IFQ). The Memory Controller services the requests in the IFQ and routes the fetched instruction words to the OIB or the CP Instruction Buffer.

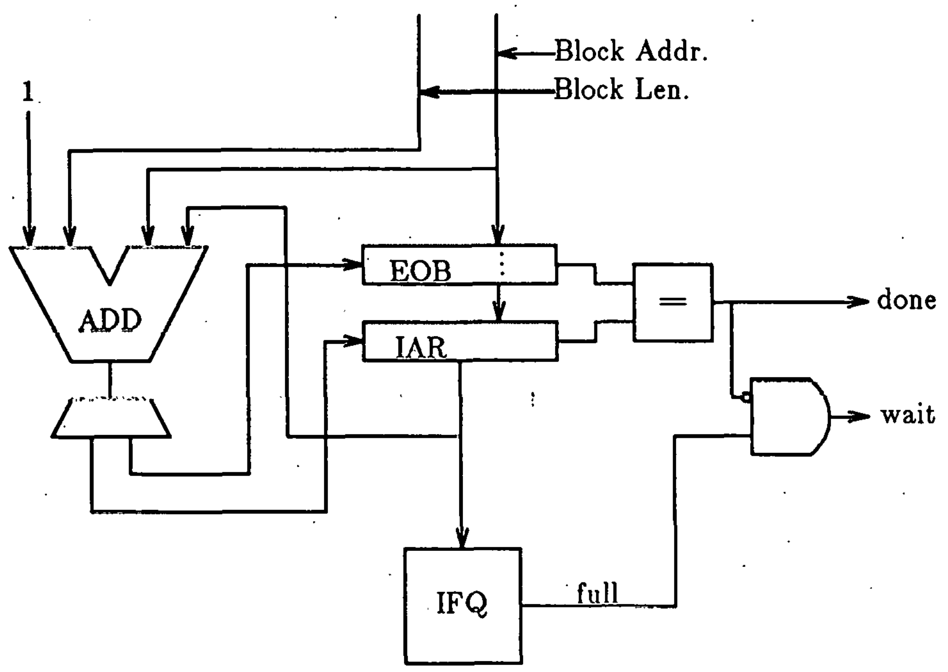

Figure 7. Instruction Fetch Unit (IFU).

Initially, the instruction block address is loaded into the Instruction Address Register (IAR), and the sum of the block address and the block length is loaded into the End-Of-Block Register (EOB). On successive clock cycles the contents of the IAR are transferred to the Instruction Fetch Queue (IFQ), and the result of the adder (IAR + 1) is gated back into the IAR. A comparison of the contents of the IAR and EOB indicates the completion of the address sequence by asserting the signal done. This signal indicates to the control unit that another *prefetch* instruction can be issued. The IFU is required to stop generating addresses temporarily whenever the IFQ becomes full. The wait signal indicates this condition.

Analysis of program graphs for a wide range of application programs has shown that, on the average, the size of basic instruction blocks is on the order of five or less [Kuck78].

This fact suggests that we can compute the EOB and increment the IAR with a simple 4 bit adder and carry propagation logic. This approach would result in an economical hardware implementation, but the maximum block length would be limited to sixteen. Another possibility would be to integrate the IFU with the AGU by sharing one of the AGU's existing 32 bit adders. The AGU would be inhibited whenever the IFU is active (done $=0$) and not stopped (wait $=0$). Our simulation results show that the performance degradation resulting from the AGU being interrupted to allow the IFU to fetch instructions is negligible ($<1\%$) when the instruction buffer (OIB) is large enough to contain all the instructions of inner loops for programs dominated by inner loop execution. Use of the full-precision AGU for IFU additions allows the instruction blocks to be larger, which is particularly desirable for the larger complex instruction blocks permitted here, without significant additional hardware cost.

2.2.3. Operand and Instruction Buffer

The Operand and Instruction Buffer (OIB) is a high-speed circular buffer used to store instructions and in-line operands prefetched by the IFU. The size of the OIB must be large enough to contain the instructions of reasonably large loops. For loops of size less than n (the size of the buffer), the OIB is able to trap the corresponding instructions, and the MAP can reexecute the loop repeatedly without refetching the instruction block. The OIB achieves a large hit ratio through the deterministic prefetching of instructions discussed in the last section. Since the size of instruction blocks is controlled by the compiler a moderate buffer size is feasible, and the OIB may be more economical than typical instruction cache implementations. The OIB is shown in Figure 8.

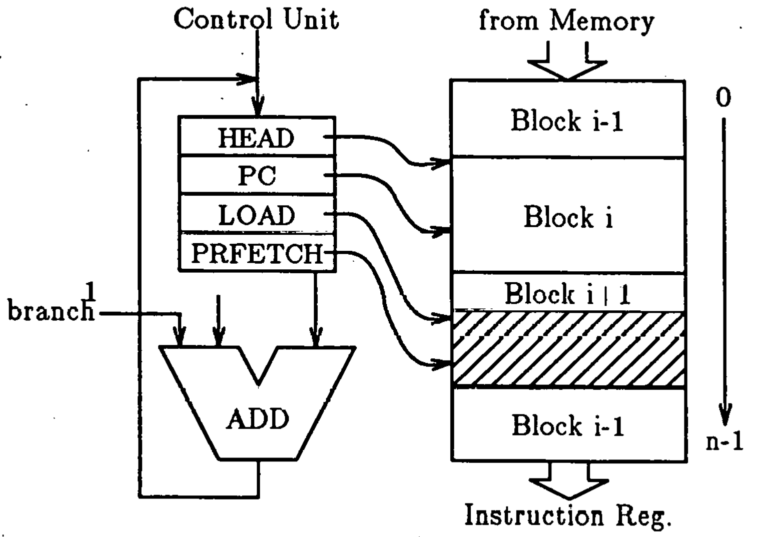

Figure 8. Operand and Instruction Buffer (OIB).

The PC contains the OIB address of the next instruction to be executed, and the HEAD register contains the OIB address of the first instruction of a loop mode block, while that block is in execution. For nonloop mode blocks, the PC and the HEAD always contain the same address, and both are incremented together as instructions are fetched. In either case, the HEAD points to the oldest valid instruction in the OIB. The LOAD register points to the OIB location where the next instruction received from the Memory Controller will be loaded, and the PREFETCH register points to the OIB location that is the target of the next instruction to be fetched by the IFU. The OIB supports one read operation and one write operation per cycle; therefore, instructions fetched by the IFU can be loaded into the OIB at the same time as instructions are fetched for execution by the AGU. The IFU loads the instructions of the block(s) that will follow the currently executing block; therefore, the LOAD and PREFETCH pointers generally remain ahead of the PC. Due to the prefetching mechanism, the next instruction to be executed by the AGU is usually contained in the OIB or, in the worst case, is in the process of being fetched.

Three status bits are associated with each location in the OIB: The *valid* bit indicates whether the OIB location contains a valid instruction (cf. Full/Empty bit), the *loop* bit indicates whether the OIB location contains an instruction that is part of a block that contains a loop, and the *last* bit marks the OIB locations which contain the last instruction of a block. The OIB sets the *valid* bit corresponding to the location of each new instruction as it is received from memory and any OIB location that has its *valid* bit set cannot be loaded with a new instruction. To ensure that valid instructions are not overwritten, the *valid* bit of the OIB location addressed by the PREFETCH register is checked prior to each instruction fetch request issued by the IFU. The IFU waits if the *valid* bit is set. Instructions are purged from the OIB, and the corresponding OIB locations become available to receive new instructions from memory, when their *valid* bits are reset. For blocks containing sequential instructions, *valid* bit resets occur one instruction at a time as each instruction is executed, and for loop mode instruction blocks, an entire block is invalidated (purged) in one operation by resetting the *valid* bits of all the OIB locations that contain the block and then setting the HEAD register equal to the PC. When the instructions of a loop mode block have been executed the required number of times and are no longer needed, the HEAD and the PC are used to generate a mask vector which is ANDed with the vector of current *valid* bits. The result resets the old *valid* bits to reflect the fact the locations occupied by the block are now invalid and can be overwritten. This block purge operation is initiated by instructions that terminate loop mode instruction blocks and other special instructions used specifically for instruction handling. For example, an unsuccessful conditional branch that is the last instruction of a loop mode block (marked by the *last* and *loop* bits being set) causes control to exit the loop, purge the loop instruction block, and proceed to the first instruction of the next block. The use of both a HEAD register and *valid* bits may appear to be redundant; however, the *valid* bits are required for the purpose of determining whether an instruction which is the target of a forward branch (e.g., unconditional jump or subroutine call) is resident in the OIB. When jumping forward it is difficult to determine whether the PC has jumped past the LOAD register and points to an OIB location that has not yet been loaded with the desired instruction. The *valid* bits are also convenient for permitting a quick validity test used, for example, in stalling the IFU prefetch operation. The HEAD register is used simply to determine the starting location of block purge operations.

The setting of the *loop* and *last* bits for each instruction is determined by control logic in the IFU. Loop mode blocks are determined by the compiler and indicated to the IFU by a flag in the *prefetch* instruction, and the last instruction of a block is known to the IFU when the final address of an instruction sequence is generated. This information is relayed to the Memory Controller which sets the appropriate bits for each instruction before they are sent to the OIB. When the *loop* bit of an instruction is 0, indicating that it is contained in a non-loop mode block, the *valid* bit of that location is immediately reset, and the HEAD and the PC are both incremented, when the instruction is fetched by the PC for execution. When the *loop* bit of an instruction is set, indicating that it is contained in a loop mode block, only the PC is incremented, unless a block purge operation is initiated. To illustrate the use of the status bits and the overall control of the OIB, we examine the three following cases. These examples are representative of most situations encountered in the control of the OIB.

Consider an instruction block that contains strictly sequential instructions to be executed once. As the address of each instruction of the block is generated by the IFU, the *valid* bit of the target OIB location, i.e., the location addressed by the PREFETCH register, is checked. If the *valid* bit is set, this OIB location already contains an instruction that is still required by the MAP. Thus, the IFU must wait for the AGU to execute the instruction and reset the valid bit. If the *valid* bit is reset, the IFU generates the instruction address and initiates the memory request by placing the address in the IFQ. The *valid* bit of the OIB location is set when the instruction is loaded from memory. The *loop* bit of each instruction of the block is reset indicating that no instruction in the block will be executed more than once. When each instruction of the block is fetched for execution, the *valid* bit of the OIB location addressed by the PC is checked to determine if the instruction is present. If the *valid* bit is not set, the execution unit waits for the IFU to fetch the instruction and set the *valid* bit; if the *valid* bit is set, the instruction is loaded into the instruction register in the control unit, and the *valid* bit is reset. When an instruction is successfully fetched by the PC, the PC and the HEAD are incremented. (Only one increment is performed and the result is stored in both registers.) In effect, by resetting the *valid* bit, the OIB location just fetched is vacated, and the IFU is free to load a new instruction in that location.

Now consider an instruction block containing a loop. As the IFU fetches and loads the instruction block, both the *valid* bit and the *loop* bit of each instruction are set. In this case, however, the OIB locations where the instruction block resides cannot be marked invalid after they are executed since at least some of the instructions may be reexecuted. The control unit, therefore, does not reset the *valid* bit of instructions that have their *loop* bit set. The HEAD register maintains the OIB address of the first location of a loop mode block while it is being executed. As long as the *valid* bits remain set, the IFU cannot overwrite the current block. As stated in the previous section, loop mode instruction blocks must have a conditional branch (or a special "purge" instruction) as the final instruction. Conditional branch instructions that are successful and are the last instruction of a loop mode block cause control to transfer back to a location within the block. Since the placement of instruction blocks in the OIB may be different from their relative location in the object module as stored in memory, all branches must be relative to the PC and not larger than the size of the OIB. The branch target displacement can always be determined by the compiler since, from the program graph and instruction prefetching, the compiler determines what instruction blocks will reside in the OIB at any given time, as well as their relative locations in the OIB. Conditional branch instructions that are unsuccessful and are the last instruction of a loop mode block cause control to proceed sequentially into the next block and cause the current block to be purged. Purging the instruction block is accomplished by reseting the *valid* bit of each OIB location between the HEAD and the PC, including the location addressed by the HEAD, then storing the contents of the PC in the HEAD.

In the two cases discussed above, there was only one possible successor instruction block and this block was located in the OIB immediately following the current block. Therefore, transferring control from the current block to the successor block simply involved incrementing the PC and, perhaps, purging the last instruction block. Loops too large to be contained in the OIB must be handled as two or more serial (nonloop mode) blocks. Some instruction in each block initiates a prefetch for the next block, and the *loop* bit of each instruction of the next block is reset by the IFU. The final block of the loop must prefetch both the first block in the loop, for the case when the loop terminating conditional branch is successful, and the next sequential block after the loop, when the terminating conditional branch is unsuccessful. A prefetch instruction is inserted in the final block such that the first block of the loop will always be prefetched and will be located in the OIB immediately following the last block of the loop. This is accomplished by placing the *prefetch* instruction for the first block of the loop *before* the conditional branch that terminates the loop. The *prefetch* instruction for the block that will be executed when the branch is unsuccessful is placed *after* the conditional branch; therefore, it is executed only when the loop has been exhausted. When the loop is exhausted, the first block of the loop has already been prefetched, and consequently must be jumped over and purged. Therefore, the final block of the loop must contain two instructions following the loop terminating conditional branch: a prefetch for the next sequential block after the loop and an unconditional branch to jump over the first block of the loop. Any nonloop mode forward branch will cause a purge operation which removes the skipped code from the OIB. (Note that backward branches within the OIB are supported only when the branch and its target are within the same loop mode instruction block.) Thus, the first block, which is not needed in this case, will be invalidated. Note that a successful branch, which is conceptually a branch *back* to the beginning of the loop, is physically a branch *forward* in the OIB. The actual branch distances and directions can be determined at compile time once the program graph is constructed, and the size of loop mode and nonloop mode blocks are determined. Also, in this case where the instruction block has two possible successor blocks, the final (current) block and the first block of the loop, and at least one instruction of the next block after the loop should all fit in the OIB simultaneously. This is required so that, for either branch outcome, the OIB location that is the target of the branch is outside of the current block; otherwise, the control unit might jump to an OIB location that is still marked valid, but is not the correct next instruction.

Making the OIB reasonably large (e.g., 1K instructions) and limiting the maximum size of an instruction block to be less than half the size of the OIB (e.g., 256 instructions) is a simple conservative guideline that eliminates any possibility of deadlock caused by instruction handling. Using this guideline it is always possible either to execute an existing instruction or to load a new instruction. In general, the compiler must limit the size of instruction blocks such that either the AGU or the IFU will be able to operate. The IFU will become blocked by the OIB only if the OIB is full, in which case the AGU must be able to execute, and eventually purge, instructions. If the AGU is blocked waiting for instructions to arrive, then there must be available space in the OIB so the IFU can operate. A larger buffer also makes the compilation problem simpler since desirable instruction blocks typically will not have to be artificially trimmed to fit in the buffer.

2.3. Computation Processor

The Memory Access Processor, discussed in the previous section, is designed to stream operands to the computation section of the SMA architecture, i.e., the Computation Processor (CP), at the maximum possible rate. To accrue the full benefit of this high memory data transfer rate, the CP must be capable of processing input operands at a rate comparable to that at which the MAP is able to deliver them. These requirements imply the need for multiple pipelined arithmetic and logic function units. In order to best evaluate the effectiveness of the MAP, we have chosen to model the CP of the SMA architecture after the scalar computation section of one of the fastest existing scalar processors, the Cray-1.

Figure 9 diagrams the scalar function units of the Cray-1 in the context of the SMA architecture, more specifically, with the data flow of the CP. In the Cray-1 architecture, each arithmetic and logic operation is implemented as an independent pipelined function unit. Separate pipelines exist for both floating-point and integer operations. All function units can operate simultaneously, and each can accept a new pair of operands every clock cycle. Similarly, in each cycle the control unit can issue one instruction to any function unit,

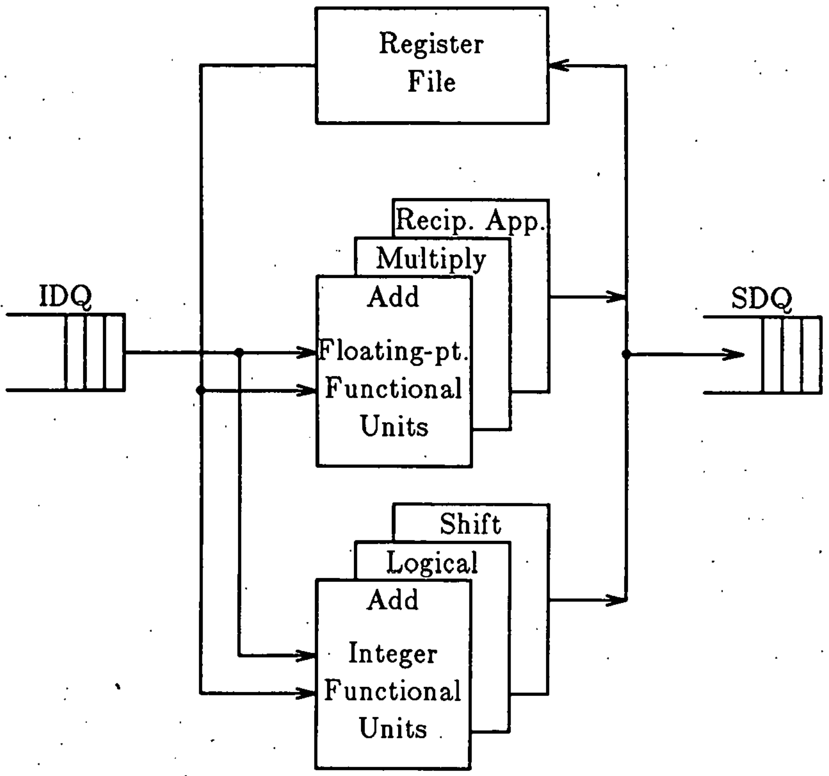

Figure 9. Computation Processor Data Flow.

except when data dependencies force a stall. The sources of operands are the register file and/or the IDQ (whose head is addressed as a pseudo-register), and the result destination is either the register file or the SDQ (whose tail is addressed as a pseudo-register). For each instruction issued, the control unit places a reservation on the destination register. When the instruction is completed and the result is stored, the register is freed for use as a source for subsequent instructions. An instruction whose source or destination register is already reserved is delayed from issuing until the register is freed by the completion of a previously issued instruction.

2.4. SMA Software

Shown in Figure 10(b) is an SMA assembly language program (MAP and CP code) used to perform matrix multiplication as described by the C language algorithm in Figure 10(a). The SMA program is used as input to the SMA simulator (discussed in Chapter 3), and accurately reflects the instruction set of a realistic SMA implementation. Contrasted with the SMA code is the corresponding VAX[1] code produced by an optimizing C compiler provided with the Unix[2] operating system. What is interesting in this comparison is that the inner loop of the VAX assembly code consists of 20 instructions, whereas the inner loop of the SMA program consists of 9 instructions (5 in the MAP and 4 in the CP). This disparity is due to the data structure address calculation overhead which is relegated to software in the VAX. Note that most of the overhead in the SMA implementation (i.e., initializing the AGU

```
mmult(A,B,C)
int A[N][N], B[N][N], C[N][N];
{
register i, j, k;
   for (i=1; i<=N; i++) {
      for (j=1; j<=N; j++) {
         C[i][j] = 0;
         for (k=1; k<=N; k++) {
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
         }
      }
   }
}
```

Figure 10(a). Matrix Multiplication Algorithm.

[1]VAX is a trademark of Digital Equiptment Corporation.

[2]Unix is a trademark of Bell Laboratories.

Inst.		VAX	Code		SMA	Code	Comments
					MAP	CODE	
1		movl	$1,r11		pref	Init,10,0	Prefetch MAP inst.
2	L3:	movl	$1,r10	Init:	pref	blk1,8,1	Prefetch all CP code.
3	L2:	mull3	$400,r11,r0		load	sbr0, scalar area	Load Scalarbase reg.
4.		addl2	12(ap),r0		load	sdt0, (sbr0)	sdt0 ←base of A.
5.		ashl	$2,r10,r1		load	sdt2, 1(sbr0)	sdt2 ←base of B.
6.		addl2	r1,r0		load	sdt4, 2(sbr0)	sdt4 ←base of C.
7.		clrl	(r0)		load	apt0, 3(sbr0)	apt0 ←ptr to x0.
8.		movl	$1,r9		load	apt1, 5(sbr0)	apt1 ←ptr to x1.
9.	⎧L1:	mull3	$400,r11,r0		load	apt2, 7(sbr0)	apt2 ←ptr to x2.
10.	⎪	addl2	4(ap),r0		load	apt3, 9(sbr0)	apt3 ←ptr to x3.
11.	⎪	ashl	$2,r9,r0		setup	x0, 11(sbr0)	Index for i.
12.	⎪	addl2	r1,r0		pref	L3,14,1	Prefetch second block.
13.	⎪	mull3	$400,r9,r1$	L3:	setup	x1, 14(sbr0)	Index for j.
14.	⎪	addl2	8(ap),r1		comp	sdt1, sdt0, apt0	sdt1 ←2D base of A.
15.	⎪	ashl	$2,r10,r2		comp	sdt5, sdt4, apt0	sdt5 ←2D base of C.
16.	⎪	addl2	r2,r1	L2:	setup	x2, 17(sbr0)	Index for k.
17.	⎪	mull3	(r1),(r0),r0		setup	x3, 20(sbr0)	Other index for k.
18.	⎨	mull3	$400,r11,r1	⎧L1:	comp	sdt3, sdt2, apt3	sdt3 ←2D base of B.
19.	⎪	addl2	12(ap),r1	⎪	fetch	sdt1, apt2	Fetch A(i,k).
20.	⎪	ashl	$2,r10,r2	⎨	fetch	sdt3, apt1	Fetch B(k,j).
21.	⎪	addl2	r2,r1	⎪	inc	x2	Inc k.
22.	⎪	addl2	(r1),r0	⎩	inc	x3, L1	Inc other k & branch.
23.	⎪	mull3	$400,r11,r1		store	sdt5, apt1	Store C(i,j).
24.	⎪	addl2	12(ap),r1		inc	x1, L2	Inc j.
25.	⎪	ashl	$2,r10,r2		inc	x0, L3	Inc i.
26.	⎪	addl2	r2,r1		End:	ret	
27.	⎪	movl	r0,(r1)				
28.	⎩	acbl	$100,$1,r9,L1		CP	CODE	
29.		acbl	$100,$1,r10,L2	blk1:	clr	r0	r0 ←0.
30.		acbl	$100,$1,r11,L3	⎧loop:	mov	r1, IDQ	r1 ←IDQ.
31.		ret		⎨	mul	r1, IDQ	r1 ←r1 × IDQ.
32.				⎪	add	r0, r1	r0 ←r0 + r1.
33.				⎩	bfq	loop	Branch to loop.
34.					mov	SDQ, r0	SDQ ←r0.
35.					bfq	blk1	Branch to blk1.
36.					bfq	blk1	Branch to blk1.

Figure 10(b). Matrix Multiplication. VAX and SMA assembly language. (Brackets demarcate the inner loops.)

tables) is outside of the loop bodies and is therefore incurred only once.

The following describes the details of the SMA program. The operating system causes the first instruction of the MAP program to be fetched and executed. The first instruction is a *pref* instruction which initiates the fetching of the first instruction block of the MAP program. This instruction block is located at symbolic address Init, contains 10 instructions, and is not a loop mode block (designated by the 0 flag in the instruction). The first instruction in the first block of the MAP program initiates the fetching of the first instruction block of the CP program. All eight instructions of the CP program are contained within a single block starting at symbolic address blk1. The block contains three nested loops and, therefore, is designated a loop mode block.

The second instruction in the MAP program loads the first scalar base register (sbr0) with the base address of a data area in memory which contains the structure definition and access pattern information. This information is generated at compile time but is not shown here for simplicity. The subsequent *load* instructions set up the specific SDT and APT registers. This information is stored in a set of locations which is some small offset from the contents of sbr0. There are three two-dimensional matrices being accessed; therefore, six SDT registers are used. Sdt0, sdt2, and sdt4 are loaded with the absolute base addresses of arrays A, B, and C, respectively. These three SDT entries are used to compute the base addresses of the second dimensions of each structure later in the program. Three *comp* instructions are used for this purpose, and they essentially initialize the three additional SDT registers (sdt1, sdt3, and sdt5) to the base addresses of the second dimension of each array. These latter SDT registers are the ones specified in the actual *fetch* and *store* requests. Note that the *comp* instructions must be located inside the loops since the second-dimension base addresses are periodically recomputed. Next, four APT registers are initialized; all contain offsets of zero and a pointer to a given index register.

The second *pref* instruction starts the IFU prefetching the second MAP instruction block which contains the three nested loops starting at symbolic address L3. The four *setup* instructions are equivalent to initializing a loop count variable before beginning a loop. The *setup* instructions load the index registers with the Current Value, Final Value, and Step Size (stride) for the loop indices *i*, *j* and *k*. The index register for *i* is set up only once, and those for *j* and *k* are set up repeatedly since they correspond to nested loops. There are actually two *k* index registers (x2 and x3) since, in the source program, *k* indexes both the second dimension of the A matrix and the first dimension of the B matrix with different strides (see Figure 10(a)). Similarly, four APT registers are used instead of three. In computing addresses for the elements of matrix A, apt2 is used, and for matrix B, apt3 is used. The index registers containing *i* and *j* can be shared between matrices A and C, and B and C, respectively, since they each index along dimensions with the same length and stride for the two matrices they access. Index register sharing is accomplished by specifying similar APT registers in *fetch*, *store*, or *comp* instructions (cf. instructions 13 and 14, for example).

At this point the MAP enters the innermost *for* loop designated by label L1. Thc *comp* instruction computes the second dimension base address of the B matrix. This base address must be recomputed for every inner loop iteration because it is a function of the *k* index. This is a result of the fact that the algorithm accesses the *columns* of B, so the base address of dimension two of array B changes for every iteration. (A clever programmer could devise a way to avoid this recomputation in the inner loop; however, we wish to keep this example relatively straightforward.) The following two instructions *fetch* the required operands by computing the addresses of $A_{i,k}$ and $B_{k,j}$. After the two addresses are computed and placed in the RAQ, the k indexes are incremented and x3 is tested against its Final Value.

The result of the test causes the control unit to branch to the symbolic address L1 if the test is successful, and to the next sequential instruction if not. The test also involves sending a branch signal to the CP's BRQ so that the CP can determine, by execution of the *bfq* instruction, whether to reexecute its inner loop or to continue sequentially. When k reaches its Final Value, control proceeds to the next sequential instruction, and the address of $C_{i,j}$ is computed (*store* instruction) and placed in the SAQ. The actual memory write will be initiated when the corresponding inner product is computed in the CP—accumulated in r0 and placed in the SDQ.

The CP code is rather straightforward. The first instruction simply initializes a register which is used as the accumulator for partial products. The next four instructions form the inner loop which computes inner products. The values in the IDQ are, alternately, the values of a row of the A matrix and the values of a column of the B matrix. Each pair of input values are multiplied, and the product is summed with the contents of r0. When k reaches its Final Value in the AGU, the CP is instructed to exit the inner loop and continue with the next sequential instruction. The next sequential instruction moves the inner product, accumulated in r0, to the SDQ so it can be stored in memory. The CP then executes a conditional branch to determine whether to reenter the loop to compute another inner product. The CP continues in this manner until the last inner product has been computed and placed in the SDQ, i.e., until the j loop and the i loop have both been exhausted in the MAP program and the last two *bfq* instructions in the CP code determine that no more input operands will arrive.

CHAPTER 3.

SMA SIMULATION AND PERFORMANCE EVALUATION

In order to perform a precise evaluation of the SMA architecture described in the previous chapter, we have developed a discrete-event register transfer level simulator for the machine. By accurately simulating the execution of programs on the SMA architecture, we have been able to observe the performance of the system and, in particular, the AGU. Recall that our primary objective in the design of the SMA is to issue instruction and operand fetch requests to memory at a rate capable of supplying input to high performance pipelined functional units with a minimum of memory wait time. Through simulation we are interested in obtaining the percent utilization, percent nonutilization (i.e., blocked and/or idle), and throughput of the main system components (i.e., AGU, CP function units, and memory). Simulation results show that the performance of the MAP hardware presented in Chapter 2 is more than sufficient for streaming operands to the CP at rates which achieve high utilization of the CP's function units. Also of interest are the effects that memory access time and queue length have on the total execution time since these parameters are easily modified without affecting the organization of the machine. Finally, the total execution time, as measured by the number of cycles required to execute benchmark programs, is used to compare the performance of the SMA architecture with that of the Cray-1 scalar unit.

The following section presents an overview of the SMA simulator. Sections 3.2 and 3.3 present simulation results concerning the utilization and throughput of each of the SMA subsystems, and the effects of memory access time and queue length on SMA performance, respectively. In the final section we present a performance comparison of the SMA architecture and the Cray-1 which, architecturally, represents the current state-of-the-art in scalar processing. For all the simulation results presented here, the CP of the SMA architecture was parameterized to perform instruction issue and computation at the rate characteristic of the Cray-1 scalar unit [Cray77]. Performance statistics and comparison information were derived from simulation of the first twelve Lawrence Livermore loops.

3.1. The SMA Simulator

Input to the SMA simulator is a program similar to that shown in Figure 10(b). The simulator essentially interprets and executes a defined assembly language. The simulator reads a file containing an SMA program and loads the instructions into its memory. From this point, the simulator fetches instruction blocks and executes instructions in a manner characteristic of an actual SMA implementation. All computations and register transfers required by an actual implementation are carried out by the simulator in the proper sequence with the specified timing.

The timing delays of various components of the system are parameterized (e.g., floating-point and integer arithmetic operations, memory access time, AGU propagation time, etc.). It is assumed that the delay of each stage of the AGU pipeline is equivalent to the time required to perform one integer addition. The AGU pipeline propagation time is controlled by the integer addition parameter, and is twice the delay of the integer arithmetic unit in the CP. For the simulation results presented in this chapter the integer addition parameter was set to one and, therefore, the number of cycles required by the AGU to produce a single operand address was two. No additional delay for multiplexing or bussing was accounted for. The AGU is fully pipelined and, therefore, is capable of producing addresses on consecutive cycles. For example, two *fetch* instructions can be issued on consecutive cycles, and the resulting addresses they compute are produced on consecutive cycles, after an AGU propagation time of two. Some MAP instructions cause more than one address to be produced by the AGU (e.g., *load* Index Register) so a MAP instruction cannot always be issued to the AGU every cycle, even when there are no data dependencies or prefetch operations.

As in the Cray-1, the CP of the SMA architecture contains multiple arithmetic and logic function units. In the simulations we ran, only floating-point instructions were executed in the CP; therefore, it was sufficient to simulate just the floating-point function units of the Cray-1, namely, a floating-point add/subtract unit, a floating-point multiplication unit, and a reciprocal approximation unit (see Figure 9). Only the first two of these units were utilized in our simulations. The addition unit and the multiplication unit are fully pipelined, and each can accept one new operation per clock cycle. The add unit delivers results in six cycles, and the multiply unit delivers results in seven cycles. In the Cray-1, the number of cycles is equal to the number of stages in each of the pipelines. The two pipelines operate independently. When an instruction is issued, its destination register is marked reserved until the instruction is completed and the result is stored in the register. An instruction is delayed from issuing until none of its source registers are reserved by previously issued instructions and, if the IDQ is a source, it must be nonempty. CP instructions are always issued in order, as in the Cray-1.

For simplicity in the simulation, the memory unit is modeled as possessing infinite interleaving; every memory word is contained in its own bank and, therefore, all memory references are conflict free. This aspect of the SMA simulator does not model a feasible machine; however, an adequate degree of interleaving should make conflict degradation minimal for the SMA at a modest cost. The memory unit services one request per cycle, and the result is delivered to the destination after a delay defined by the memory access time parameter (11 cycles for the Cray-1). Note that the memory system can accept requests on consecutive memory cycles even though prior requests have not been completed, resulting in a memory that behaves like a perfectly pipelined 11 stage function unit.

The service priority of the memory address queues are as follows:

1) IFQ (Instruction fetch),

2) RAQ (Operand fetch),

3) SAQ/SDQ pair (Operand store).

Pipelined computers are susceptible to hazards and the SMA architecture is no exception. A read-after-write (RAW) hazard occurs in the SMA when the AGU issues a read request for a data item whose address appears in the SAQ, waiting to be written. As discussed in Section 2.1, the SMA simulator assumes that operands contained in the SDQ can be forwarded to the CP before they are written to memory. This forwarding operation minimizes the effect of RAW hazards which significantly improves the performance of the SMA for benchmark programs that contain certain linear recurrences and data dependencies. Note that write-after-read hazards do not present any problem due to the fact that read requests have higher priority that write requests. Also, write-after-write hazards do not occur due to the queuing and servicing of write requests in order.

The lengths of all hardware queues and instruction buffers are variable. Hence, we are able to monitor the performance of the system as a function of some of the machine parameters.

The simulator reports a number of performance statistics for each run:

1) Total number of clock cycles required to execute the program.
2) Throughput of the MAP, CP, and memory.
3) Percent utilization of the MAP, CP, and memory.
4) Percent of clock cycles that the MAP, CP, and memory is blocked.
5) Reasons for function unit blockage (and percent blocked per reason).
6) Mean queue lengths.

Function unit throughput is defined as the percentage of all clock cycles in which an instruction (or operation in the case of the memory unit) is successfully issued. This figure is also equivalent to the rate at which instructions are completed. In the case of the AGU, the output of the pipeline is actually greater than the pipeline throughput because some instructions (e.g., *load*, *setup*, etc.) cause more than one address to be computed. Function unit utilization is recorded as the percentage of total clock cycles that a unit is active (i.e., at least one computation in progress for pipelined units), or is inactive due to being blocked, but has work pending. In general, a function unit becomes blocked as a result of a dependency in the instruction stream, when one of the queues or buffers that supply input to the unit is empty, or when one of the queues that accepts output from the unit becomes full. The AGU becomes blocked when an OIB miss occurs, when an instruction requires input from a table (or the BRQ, in case of the *bfq* instruction) that has not yet been populated (i.e., a data dependency is present), or when an address queue that is the destination of an instruction is full. In any of these cases, the AGU must be idle for one or more cycles. For the memory, blockage can occur when the IDQ is full, or when either the CP instruction buffer or the OIB is full. The memory unit is not considered to be blocked on cycles when the address queues (i.e., RAQ, SAQ, and IFQ) are all empty. The CP becomes blocked when an instruction buffer miss occurs, when data dependencies in the instruction stream exist, when a full SDQ is an instruction's destination, or when an empty IDQ is an instruction's source. In fact, this last statistic—the percent of cycles that the CP is blocked due to the IDQ being empty—is perhaps the single most important performance metric since it indicates whether the MAP is accomplishing the task for which it was designed: namely, to prefetch operands such that the memory access wait time experienced by the CP's function units is minimal.

3.2. SMA Performance

In this section we examine the utilization and throughput of the various SMA subsystems. We are primarily interested in determining whether the address generation hardware of the AGU is sufficiently powerful to supply a CP, which has the computational capability of the Cray-1 scalar unit, with operands at a rate which provides superior utilization and throughput by minimizing memory access wait time. The most relevant function units in regard to the overall performance of the machine are the AGU, the CP function units, and the memory. For the simulation results presented in this section, the instruction buffers were of length length 128, each of the AGU tables had 16 entries, and all queues were of length 4. Code segments corresponding to the inner loops of all the benchmark programs were contained entirely in the OIB (the largest containing 106 words), and at most 10 of the 16 AGU table entries were used during the simulations. As a result, the effect of instruction fetching and AGU table loading was an insignificant percentage of the overall execution time (<1%); hence, we do not provide an analysis of the performance of the IFU or OIB.

Shown in Table 1 are the function unit utilization and throughput statistics derived from the SMA simulator for the first twelve Lawrence Livermore loops. A good description of the nature of these loops is found in [HsPG84]. These statistics indicate a fairly good balance of activity among the units. The utilization figures for each unit are all very high. This is a result of the fact that a function unit that is blocked but does have instructions waiting to issue, or has at least one active computation in its pipeline, is considered to be utilized. The percentage of execution time that each unit is blocked from issuing instructions seems to be a bit alarming; however, an average instruction issue rate of 0.425 instructions per cycle in the CP is actually quite acceptable after the frequency of data dependencies in the CP instruction stream is taken into account. Furthermore, the CP has a somewhat higher instruction throughput rate than the AGU since the inner loops of the CP programs contain

Table 1. SMA Function Unit Utilization and Throughput.

Loop	AGU Util.	AGU Blocked	AGU T'put	CP Util.	CP Blocked	CP T'put	Memory Util.	Memory Blocked	Memory T'put
1	0.926	0.826	0.161	0.965	0.708	0.258	0.994	0.891	0.132
2	0.991	0.703	0.284	0.971	0.594	0.379	0.993	0.750	0.265
3	0.301	0.694	0.299	0.996	0.599	0.300	0.997	0.330	0.201
4	0.878	0.535	0.349	0.899	0.620	0.281	0.984	0.617	0.293
5	0.997	0.589	0.404	0.964	0.593	0.373	0.999	0.494	0.377
6	0.688	0.652	0.331	0.928	0.682	0.320	0.998	0.348	0.267
7	0.996	0.830	0.170	0.984	0.677	0.407	1.000	0.848	0.154
8	0.988	0.809	0.178	0.994	0.699	0.386	0.997	0.798	0.174
9	0.997	0.808	0.189	0.984	0.531	0.615	0.999	0.842	0.173
10	0.998	0.623	0.374	0.984	0.423	0.794	1.000	0.000	0.359
11	0.765	0.553	0.446	0.888	0.555	0.451	1.000	0.169	0.337
12	0.686	0.697	0.302	0.889	0.499	0.539	1.000	0.000	0.204
Avg.	0.851	0.693	0.291	0.934	0.598	0.425	0.997	0.507	0.245

roughly 30 percent more instructions, on the average, than the corresponding MAP programs for the benchmarks that we ran. Note that the AGU throughput represents instructions issued per clock, rather than addresses generated per clock. Thus, multiple address instructions as well as certain no-address instructions (e.g., *setup*) are each counted once when the AGU throughput is calculated. Also, the "Blocked" statistics are not directly available as such within the simulator and in some cases the estimate of Blocked time is slightly high. This fact accounts for the apparent anomaly where occasionally the sum of Blocked and Throughput slightly exceeds 1.00.

The percentage of execution time that each unit is blocked, and the reasons why, provide better understanding of the behavior of the overall system throughput and its limitations. Thus a closer look at blockage is in order. Table 2 presents a breakdown of the function unit blockage rates and their respective causes for the AGU and the CP. These figures were derived from simulation of the twelve Lawrence Livermore loops, with the queue lengths all set to four. In these simulations, almost all memory unit blockage was caused by the IDQ becoming full. While the IDQ is full, the CP is supplied with operands and memory blockage is not a serious concern. Therefore, no further details are given for the memory unit.

The AGU can become blocked due to either a data dependency in the MAP instruction stream, or the RAQ or SAQ becoming full. (The AGU can also become blocked by an OIB miss during an instruction fetch; however, this was never the case during these simulations due to the dominance of loop mode execution.) The MAP address queues fill up as a result of the memory unit not being able to service read requests due to the IDQ being full (refer to Figure 2). The IDQ, in fact, is full 51 percent of the time, on the average, as indicated by "Memory Blocked" in Table 2. When the IDQ is full, the memory unit becomes blocked

Table 2: SMA Function Unit Blockage and Causes.

Loop	AGU Blocked	Data Depend.	RAQ Full	SAQ Full	CP Blocked	Data Depend.	IDQ Empty	SDQ Full	Memory Blocked
1	0.826	0.065	0.001	0.760	0.708	0.704	0.004	0.000	0.891
2	0.703	0.003	0.701	0.000	0.594	0.589	0.005	0.000	0.750
3	0.694	0.000	0.694	0.000	0.599	0.596	0.003	0.000	0.330
4	0.535	0.082	0.018	0.449	0.620	0.554	0.066	0.000	0.617
5	0.589	0.003	0.125	0.463	0.593	0.589	0.004	0.000	0.494
6	0.652	0.310	0.195	0.150	0.682	0.667	0.005	0.000	0.348
7	0.830	0.004	0.826	0.000	0.677	0.677	0.000	0.000	0.848
8	0.809	0.000	0.521	0.288	0.699	0.698	0.001	0.000	0.798
9	0.808	0.000	0.808	0.000	0.531	0.530	0.001	0.000	0.842
10	0.623	0.000	0.000	0.623	0.423	0.389	0.034	0.000	0.000
11	0.553	0.236	0.000	0.317	0.555	0.552	0.003	0.000	0.169
12	0.697	0.315	0.000	0.382	0.499	0.499	0.000	0.000	0.000
Avg.	0.693	0.085	0.324	0.286	0.598	0.587	0.011	0.000	0.507

which, in turn, causes the address queues to back up, and hence the AGU becomes blocked. Note, however, that a full queue can be read from and written to on a given cycle, and therefore even under the full queue condition there may not be a blocked unit. The rate at which each address queue becomes full varies considerably among the loops, which indicates that these numbers are very application dependent. Table 2 shows that data dependencies account for only a small portion of AGU blockage relative to the amount of time that address queues are full, except for loops 6, 11 and 12. We expected this to be the case since the AGU, because of its unique design, requires little interaction with memory to compute operand addresses, and MAP instructions are relatively independent of each other, thereby reducing the number of dependencies.

The vast majority of CP blockage, on the other hand, is caused by data dependencies. In fact, 98 percent of the time that the CP is blocked from issuing instructions is caused by data dependencies; only two percent of the time that the CP is blocked is due to the IDQ being empty. Note that the dependency problem is inherent in the application code and causes blockage in any machine organization with heavily pipelined function units. We made no effort to improve this aspect of the machine's performance. We can conclude from Table 2 that instruction execution in the CP is rarely impeded by memory access wait time, and therefore, the CP is performing at near its maximum rate, namely, the ideal peak rate minus the data dependency degradation. This analysis of the results of Table 1 and Table 2 clearly indicate that the MAP is performing sufficiently well, and is perhaps even over-designed for cases where one of the address queues is full the majority of the time.

3.3. Effects of Queue Length and Memory Access Time

The effects that queue length and memory access wait time have on the overall execution time of the SMA architecture are worth investigating since they can each be changed without modifying the basic organization of the machine. Intuitively, increasing the size of the address and data queues will help to smooth out perturbations in the flow of data items through the machine, and thus may help to increase the overall utilization and throughput of each subsystem. In the SMA architecture, queues basically allow the MAP and the memory to continue fetching operands before previously fetched operands have been consumed by the CP. If the machine operates in this state long enough, the IDQ may fill up, causing the MAP to become blocked. As we showed in the last section, this situation does occur and, in fact, poses no immediate performance problem since the CP is still able to run unimpaired for some time. Even during subsequent intervals where the CP is able to process operands faster than they can be delivered, the CP will not become blocked if the IDQ already contains several operands, and the MAP can resupply it before the CP empties the last operand. A limit therefore exists beyond which increased queue length will not provide any additional speedup. From a practical standpoint, the queues should be as small as possible without adversely affecting the execution time.

Memory access time also has an effect on the total number of cycles required to execute programs. This effect is particularly evident in applications where the MAP has difficulty staying ahead of the CP, i.e., where slip cannot be maintained. For example, slip is frequently lost in programs containing data dependent branches which are resolved in the CP, or programs in which the MAP must execute more instructions than the CP. In these situations, incrementally larger memory access time will have an increasingly pronounced effect on execution time. However, in programs where the MAP is able to maintain slip, we will show that the memory access time has a less significant effect on the total execution time. In general, subject to loss of slip, increasing the memory access time is beneficial since it allows reducing the system cost either by using a slower memory or by keeping the same memory and designing faster or more heavily pipelined function units, in which case performance can be increased by speeding up the system clock rate.

To observe the effects that queue length and memory access time have on execution time, we ran the several benchmarks and recorded the total execution time for queue lengths ranging from 1 to 8, and memory access times ranging from 2 to 12. Figure 11 shows graphs of the total execution time versus queue length for a range of memory access times for a matrix multiplication algorithm and for Lawrence Livermore loop 12. All the simulations displayed similar characteristics.

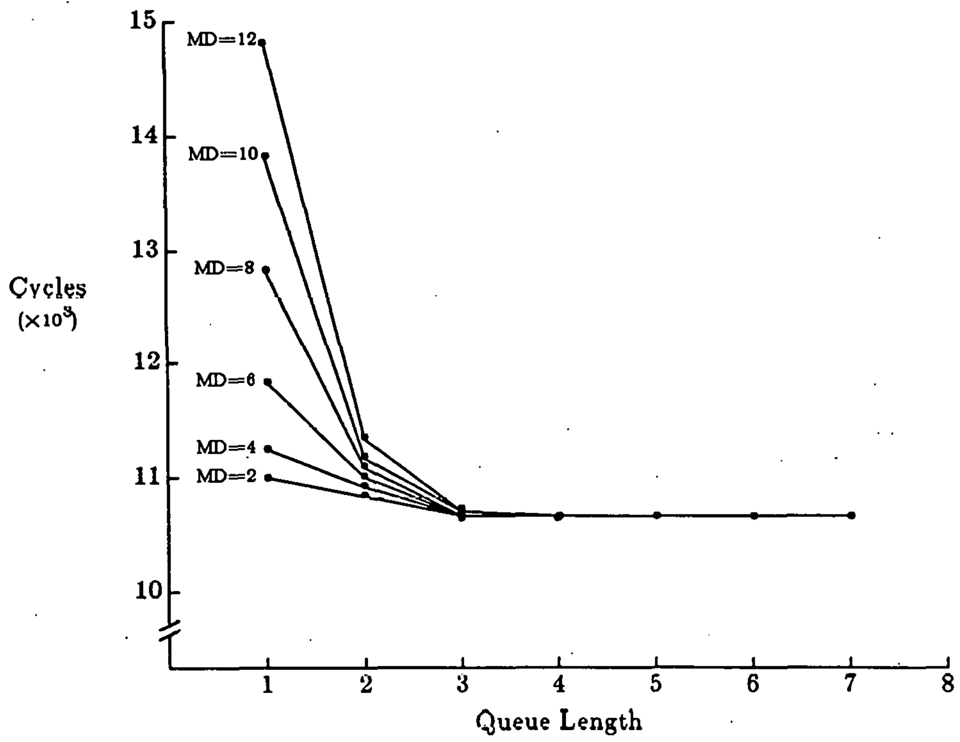

Figure 11(a). Execution Time vs. Queue Length for Matrix Multiplication.

These graphs show a striking performance improvement as the queue length is increased from one to two, particularly when the memory is slow. Very little speedup is achieved by increasing the queue lengths beyond two, or three when the memory access time is large. Queue lengths of one are quite detrimental when the memory is slow. Thus the SMA, given a modest amount of queuing, can tolerate a relatively slow memory with negligible performance degradation. It is important to point out, however, that in the simulations we ran (Livermore loops, Gaussian elimination, and matrix multiplication) all branch decisions were resolved in the MAP. For this type of program the SMA architecture tends to perform well because the MAP never loses slip, and therefore always remains ahead of the CP. Conceptually, the CP has to wait for the first stream of operands to arrive, and thereafter never endures the complete memory delay; a nonturbulent flow of input operands is always available to the CP. Table 2 substantiates this scenario by indicating that the CP experiences an empty IDQ only 1 percent of the time, on the average, for a memory access time of 11. This implies that a large memory latency is virtually transparent for these types of programs, and

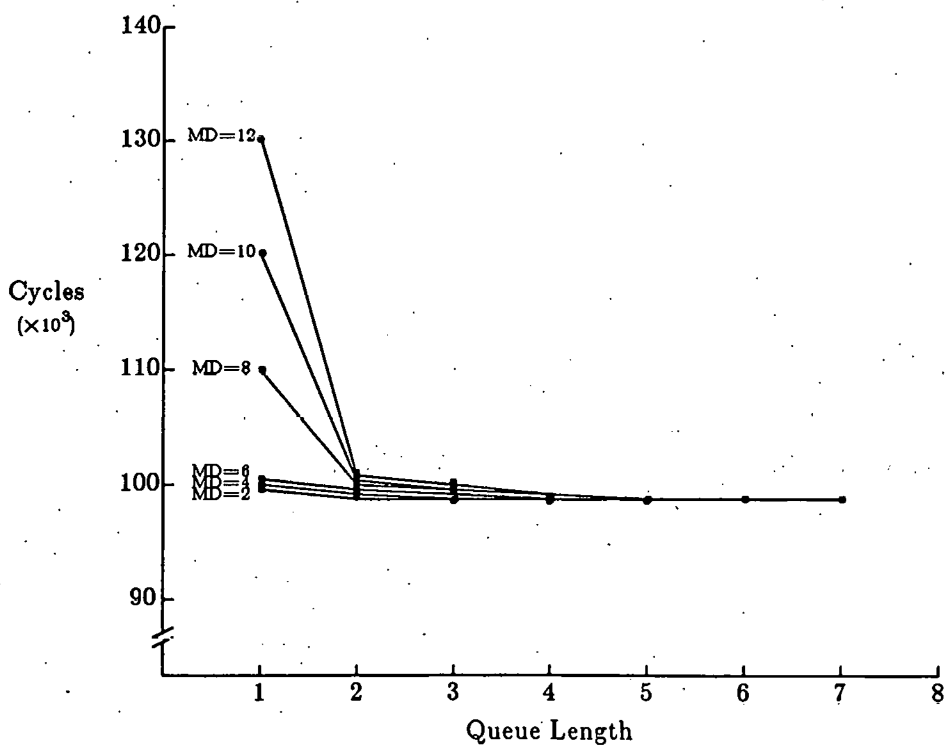

Figure 11(b). Execution Time vs. Queue Length for Lawrence Livermore Loop 12.

a faster memory would do little to improve performance.

We would expect the SMA to exhibit much different behavior for programs containing data dependent branches in the inner loops, however, because the MAP would lose its slip advantage and, in every loop iteration, the CP would experience the complete memory access time. The CP would then experience more blockage due to an empty IDQ, and this effect would be successively worse as memory access time is increased. Several other situations could also slow the MAP down. For example, in programs containing large loops or a substantial dynamic frequency of nonlooping code, instruction fetching could interfere significantly with the MAP's performance. The stream of operands to the CP would be intermittent resulting in more potential for CP blockage. Furthermore, in some applications, the MAP program may contain more instructions than the corresponding CP program which could also cause the CP to be blocked a greater portion of the time. In each of these cases, we would expect the asymptote for total execution time to be successively higher as memory access time is increased.

3.4. Cray-1/SMA Performance Comparison

We have examined the characteristic behavior of the SMA architecture and shown through simulation results that it is able to perform as expected on suitable benchmark programs. It is, however, also useful to compare the performance of the SMA architecture with other existing high performance computers for these benchmarks. The Cray-1 was chosen for comparison purposes because information on its architecture and operation is readily available, and it represents the foundation of the Cray-2 architecture which is perhaps the fastest existing scalar processor.

For this comparison, we are interested in obtaining the total number of cycles required by each machine to execute the benchmark programs. The execution times for the SMA architecture were taken from the simulator. As above, the SMA machine being simulated was parameterized to perform instruction issue and execution at the same rate as the Cray-1, instruction buffers were of length 128, AGU tables contained 16 entries, and all queues were of length 4. For our purposes, the conditions for instruction issuing on the Cray-1 can be summarized as follows:

1) The target function unit (i.e., the floating-point addition unit or the floating-point multiplication unit) must be free.

2) The source registers must be free (i.e., not reserved as the destination of a prior instruction).

3) The destination register must be free.

Instruction timings for the Cray-1 that are relevant to our simulations are the following:

- Floating-point addition takes 6 clock periods.
- Floating-point multiplication takes 7 clock periods.
- Branch resolution takes 2, 5 or 14 clock periods.
- Memory access takes 11 clock periods.

The function units are fully pipelined; therefore, instructions can be issued to the same unit on consecutive clock cycles, provided no data dependencies exist. Each instruction places a reservation on its destination register only, and this register is reserved until the result is stored, i.e., is reserved for the duration of the execution time of the instruction.

The execution times for the Cray-1 were derived analytically. The Lawrence Livermore loop kernels were each compiled using the Cray-XMP Fortran compiler version 1.13 with the automatic vectorizer turned off. From the Cray assembly language listings and knowledge of the Cray-1 instruction issue and execution timings, we were able to derive accurate timing estimates for each of the loops. From our analysis of the Cray timings, it was evident that the Cray Fortran compiler did an excellent job of interleaving computations and memory access instructions such that memory access wait time was minimized. The code, however, did contain an abundance of register transfer instructions, used mainly for address computation and loop control, that could have been avoided.

The SMA programs for the kernels were arranged to perform computations and memory accesses in the same order as the compiled Cray code. We took this approach to insure that the SMA had no special advantage, and so that the test would represent a comparison of the two machines' actual performance on this code, rather than the efficiency of a particular compiler or hand optimization. No special optimizations were added to the SMA code, but the code was designed to take advantage of the inherent features of the machine. For example, the source programs were divided into two instruction streams, and each one of the SMA processors' programs was significantly smaller than the Cray's single program. Also, the AGU tables were used to take advantage of register sharing and to minimize reloading. In addition, the architecture of the SMA allows it to perform some basic operations faster than the Cray. For example, branch instructions in the CP can complete in one clock cycle assuming that the corresponding branch flag is present in the BRQ at the time the branch instruction is executed. In effect, the SMA architecture is able to "turn" a loop in a single cycle when the MAP is ahead, whereas, in the Cray-1 evaluation five clock cycles are always required to resolve a successful branch.

Table 3 shows the analytically computed times for the Cray-1, and the simulated times for the SMA architecture for the Livermore suite. Some of the loops were run for the number of iterations specified in the Fortran code, and some loops where run for an arbitrary number of iterations (typically 1000 for singly nested loops, and 100 for doubly nested loops). The total execution time in seconds can be calculated by multiplying the number of clock cycles by 12.5 nanoseconds, the period of one machine cycle on the Cray-1. The floating-point execution rate, measured in millions of floating-point operations per second (MFLOPS), is then determined from knowledge of the total number of floating-point operations executed in the loop, and the total number of seconds required to run the loop. The speedups shown are simply the ratio of SMA MFLOPS to Cray-1 MFLOPS in that row of the table. The Avg. MFLOPS is the arithmetic mean of the MFLOPS figures for the 12 loops; i.e., it represents the MFLOPS that would be seen if each loop was run for the same amount of *time*. Note that in such an "average" job load, the SMA and the Cray-1 would have

Table 3. SMA/Cray-1 Performance Comparison.

Loop	**Cray-1**		**SMA**		Speedup
	Cycles	MFLOPS	Cycles	MFLOPS	
1	18000	8.89	12463	12.8	1.44
2	11800	12.20	8448	17.05	1.40
3	19000	8.40	10062	15.87	1.89
4	1568	5.00	881	8.90	1.78
5	20252	7.87	10772	14.79	1.88
6	45800	3.49	18206	8.79	2.52
7	836100	15.31	637318	20.08	1.31
8	164640	17.49	141003	20.42	1.17
9	837800	16.23	628282	21.65	1.33
10	814500	8.84	629155	11.44	1.29
11	224400	3.57	88784	9.01	2.53
12	253600	3.15	98291	8.14	2.58
Avg.		9.20		14.08	
H. mean		6.59		12.50	1.89

different job loads; i.e., each machine would execute a different number of floating-point operations. Therefore, this "Avg." weighting leads to a meaningless speedup. The H. mean (harmonic mean) MFLOPS for the 12 loops was computed by equalizing the number of floating-point operations performed by each loop. This calculation accurately represents a job load where each loop is run for the same fixed number, e.g. 1 million, of floating-point operations on each machine. The number of seconds required to execute each loop is computed from the MFLOPS figure for each loop. The H. mean MFLOPS is then computed from the sum of the seconds for each loop and the total number of floating-point operations chosen.

Table 3 shows a wide range of speedups for the various loops. The speedup computed from the harmonic mean of the MFLOPS is considerably greater than the speedup computed from the average MFLOPS. This is a result of the fact that, for loops where the MFLOPS tends to be lower, the Cray-1 performs proportionally worse than the SMA, and these loops tend to have a larger influence on the harmonic mean (cf. loops 6, 11 and 12). In general, the performance of the Cray-1 fluctuates more than the performance of the SMA. The range of performance across all loops is approximately 5.5 to 1 for the Cray-1, whereas, for the SMA the range is only 2.7 to 1. Thus the SMA provides more balanced performance for the entire job load.

It is interesting to consider the characteristics of loops that have a large speedup on the SMA, and conversely, those that do not. As mentioned above, the Cray Fortran compiler does a very good job of generating code that tends to hide the long memory access time of the Cray-1. This is accomplished by issuing fetch instructions far in advance of the instructions that will actually operate on those operands. Since the Cray-1 employs a single instruction stream, it must issue instructions that perform address calculations and various other overhead operations from the same instruction issue unit using a single stream. Wherever possible, these types of instructions are inserted in between memory access instructions and computation instructions, or between two computation instructions, that may have dependencies. Hence, many of the overhead instructions in the Cray program are issued on cycles that would otherwise be unused, and much of the memory access wait time is hidden by performing other necessary operations in the meantime. Unfortunately, the Cray compiler can only perform these optimizations when the loop in question contains a sufficient number of instructions of the proper types to work with. As loops get smaller, the number of possibilities for code rearrangement also becomes less. For small loops it becomes impossible to mask the memory access wait time, so it is here that we expect the SMA to perform particularly well relative to the Cray-1.

In Table 3, loop 8 shows the smallest speedup. Loop 8 also happens to contain the largest number of instructions of any in the suite (106 instructions in the inner loop). The Cray-1 requires 156 cycles to execute the inner loop. The instruction issue logic is idle for 4 cycles while the branch outcome is being resolved, 40 cycles are due to data dependencies, and only 6 cycles are idle due to memory access wait time. An instruction is issued on each of the remaining 106 cycles. Idle cycles due to memory access wait time amount to only 4 percent of the total execution time. In the SMA architecture, the inner loop of the CP program contains 59 instructions, and the inner loop of the MAP program contains 34 instructions. The CP can execute a single pass of its inner loop in 117 cycles, assuming the IDQ is always nonempty. The CP runs slower than the MAP, which requires only 34 cycles per loop iteration (i.e., no data dependencies are present) and, therefore, the CP performance bounds the total execution time of the SMA for this loop. Data dependencies in the CP block instruction issuing on 58 of the 117 cycles. The higher percentage of data dependency cycles for the CP, compared with the Cray-1, is a result of the reduced number of overhead instructions in the CP program. The number of cycles that the CP is blocked due to memory wait time cannot be determined from a static analysis of the code; however, simulation results reveal that, on the average, this number is less than one cycle per loop iteration. For this loop, the difference in memory access wait time between the Cray-1 and the SMA is on the order of 4 percent; therefore, the slightly better SMA performance is mainly a result of reduced overhead, rather than decreased memory access wait time. The Cray-1 thus performs well on this loop which accounts for the small SMA speedup.

On the opposite end of the spectrum, loop 12 shows the largest speedup. The Cray-1 requires 12 instructions and 25 cycles to execute its inner loop, whereas the CP requires only 5 instructions and 10 cycles. The Cray-1 and the CP both have the same 5 idle cycles due to data dependencies. For this loop, 4 cycles are idle due to memory access wait time on the Cray-1, which represents 16 percent of the total. In the SMA, however, the CP instruction issue unit is held up because of memory access wait time an average of less than one percent of the total execution time per loop iteration. The difference in the percent of memory access wait time per loop iteration in the Cray-1 over the SMA is thus 4 times greater for loop 12 than it is for loop 8. Furthermore, the Cray-1 requires 5 cycles to perform branch resolution, whereas, the CP requires only 1. The additional 4 branch cycles on the Cray-1 account for 16 percent of its loop execution time. For larger loops, however, these branch cycles will represent a much less significant percentage of the Cray-1's total execution time. Thus, address generation and other overhead coupled with the increased percentage of memory access wait time cause the Cray-1 to run considerably slower than the SMA architecture for this loop.

CHAPTER 4.

CONCLUSIONS

The Structured Memory Access architecture implementation presented in this thesis was formulated with the intention of alleviating two well-known inefficiencies that exist in current scalar computer architectures: address generation overhead and memory bandwidth utilization. Furthermore, the SMA architecture introduces an additional level of parallelism which is not present in current vector supercomputers, namely, overlapped execution of the access process and execute process on two distinct special-purpose, asynchronously-coupled processors. By using simulation results derived from representative benchmarks typical of intended SMA workloads, the Memory Access Processor was shown to expedite processor-memory traffic by efficiently computing instruction and operand addresses using special-purpose pipelined function units (i.e., the AGU and IFU), and at the same time, reducing the demand on memory bandwidth by requiring less interaction with memory to support the access process. Our simulation results showed that, for typical numerical programs, the MAP was capable of running slightly ahead of the CP, and consequently was able to issue operand fetch requests at a rate that rarely caused the CP to experience any memory access wait time. Memory access wait time accounted for only 1 percent of the total execution time, on the average, for the benchmark programs that were simulated.

It was further discovered that, for programs in which branch decisions are resolved solely in the MAP (i.e., a broad class of numerical programs), a large memory cycle time had a relatively minor effect on total execution time for processor queue lengths of three or more. This phenomenon is a result of the fact that once the stream of input operands to the CP is started, it is not interrupted (assuming no bank conflicts), and the long memory access wait time is seen only once by the CP. Thereafter, the MAP remains sufficiently ahead, and it appears to the CP as if most of its input operands were contained in registers (i.e., the head of the IDQ is rarely empty when accessed by the CP). Note that this is only true for programs where loop bounds are based on an index value or some other data item that is resident in the MAP. In these situations, the MAP is essentially able to perform perfect branch lookahead for the CP.

Comparison with the Cray-1 in nonvector mode showed that the SMA architecture's features do, indeed, provide improved performance in scalar processing over existing high performance scalar machines. Since the CP is rarely required to wait for operands to arrive from memory, the instruction issue rate is improved and, hence, function unit utilization is increased. The dual instruction stream feature enables each SMA processor's program to be significantly smaller than the conventional single instruction stream program and also frequently allows two instructions to be issued in a single cycle. Furthermore, the overhead associated with branch resolution is reduced in the SMA when these decisions are performed in the MAP, thus relieving the computation section of this chore. This overhead is particularly significant on the Cray-1 for small loops where branch resolution becomes a larger percentage of the total execution time. These factors account for the speedup shown by the SMA architecture over the Cray-1.

In all the simulations that we ran (the first twelve Lawrence Livermore loops, Gaussian elimination, and matrix multiplication) all branch resolution was performed by the MAP. Programs with this characteristic are best suited for fast execution on the SMA machine. Further analysis of the SMA architecture should also include simulation of programs that would be expected to run less efficiently on this machine, for example, a program containing data dependent branches to be resolved in the CP, or a data dependent branch to be resolved in the MAP, but which requires information from the CP to determine the outcome. In either of these cases, we expect the speedup over the Cray-1 in scalar mode to be small; however, we expect *any* program to execute at least as fast as the Cray-1 in scalar mode.

Results presented in Table 2 indicate that, for many programs, the Address Generation Unit hardware offers higher performance than is necessary. This is particularly true in simulations where the CP program contains a large number of data dependencies. When this is the case, the MAP has less difficulty staying ahead of the CP because the CP's instruction issue rate tends to be slightly lower, and consequently, the rate at which the CP consumes input operands is lower. The MAP is blocked over 50 percent of the time by full address queues in eight of the twelve Lawrence Livermore loop simulations. This fact suggests that a less complex hardware configuration for the AGU may be possible which for many programs would not compromise the overall performance of the machine. Another possible means of making more efficient use of the AGU may be to time-multiplex the MAP between two or more CPs, each running separate code or, perhaps, parallel segments of the same program.

Another obvious area for further investigation is that of examining the feasibility of performing vector operations on the SMA architecture. It would be straightforward to implement vector instructions on the SMA machine described herein. What remains to be determined is whether the machine would be capable of executing vector operations at a rate comparable to, or substantially faster than, existing vector machines. We believe that with enhancements to achieve comparable chaining, parallel execution, and peak memory bandwidth, the SMA architecture could provide performance comparable to state-of-the-art vector supercomputers on vectorizable code, and higher performance on scalar code.

REFERENCES

[Ande67]
Anderson, D. W., Sparacio, F. J., Tomasulo, R. M., "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development,* Vol. 11, No. 1, January, 1967, pp. 8-24.

[Cray77]
Cray Research. *CRAY-1 Computer System, CAL Assembler Version 1 Reference Manual,* Cray Research, Inc., Chippewa Falls, Wisconsin, 1977.

[Flyn72]
Flynn, M. J., "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers,* Vol. C-21, No. 9, September, 1972, pp. 948-960.

[GHLP85]
Goodman, J. R., Hsieh, J. T., Liou, K., Pleszkun, A. R., Schechter, P. B., Young, H. C., "PIPE: A VLSI Decoupled Architecture," *12th Annual International Symposium on Computer Architecture,* June, 1985, pp. 20-27.

[Hamm77]
Hammerstrom, D. W., Davidson, E. S., "Information Content of CPU Memory Referencing Behavior," *Fourth Annual Symposium on Computer Architecture,* March, 1977, pp. 184-192.

[HsPG84]
Hsieh, J., Pleszkun, A. R., Goodman, J. R., "Performance Evaluation of the PIPE Computer Architecture," Computer Sciences Technical Report No. 566, University of Wisconsin, Madison, Wisconsin, November, 1984.

[Kahh83]
Kahhaleh, B. Z., "Performance Modeling and Enhancement of the Structured Memory Access Architecture," CSG Report No. 23, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, December, 1983.

[Kuck78]
Kuck, D., *The Structure of Computers and Computations,* Vol. 1, John Wiley and Sons, New York, 1978.

[PaDi80]
Patterson, D. A., Ditzel, D. R., "The Case for the Reduced Instruction Set Computer," *Computer Architecture News,* Vol. 8, No. 6, October, 1980, pp. 25-33.

[PaSe81]
Patterson, D. A., Sequin, C. H., "RISC I: A Reduced Instruction Set VLSI Computer," *Eighth Annual Symposium on Computer Architecture,* 1981, pp. 443-457.

[Ples82]
Pleszkun, A. R., "A Structured Memory Access Architecture," CSG Report No. 10, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, August, 1982.

[PSKD86]
Pleszkun, A. R., Sohi, G. S., Kahhaleh, B. Z., Davidson, E. S., "Features of the Structured Memory Access (SMA) Architecture," *Proc. IEEE Compcon,* March, 1986, pp. 259-265.

[PlDa83]
Pleszkun, A. R., Davidson, E. S., "A Structured Memory Access Architecture," *International Conference on Parallel Processing,* August, 1983, pp. 461-471.

[SoDa84]
Sohi, G. S., Davidson, E. S., "Performance of the Structured Memory Access (SMA) Architecture," *Proc. 1984 International Conference on Parallel Processing,* August, 1984, pp. 506-513.

[Sohi83]
Sohi, G. S., "Memory Access Prediction, Execution Overlap and Branch Lookahead in the SMA Arichitecture," CSG Report No. 17, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, July, 1983.

[Smit82]
Smith, J. E., "Decoupled Access/Execute Computer Architectures," *Ninth Annual Symposium on Computer Architecture,* April, 1982, pp. 112-119.

[Smit84]
Smith, J. E., "Decoupled Access/Execute Computer Architectures," *ACM Transactions on Computer Systems,* Vol. 2, No. 4, November, 1984, pp. 289-308.

BIBLIOGRAPHIC DATA SHEET	1. Report No. CSRD-597	2.	3. Recipient's Accession No.

4. Title and Subtitle The Structured Memory Access Architecture: An Implementation and performance evalutation	5. Report Date August 1986
	6.
7. Author(s) Joseph Cyr	8. Performing Organization Rept. No. CSRD-597
9. Performing Organization Name and Address University of Illinois at Urbana-Champaign Center for Supercomputing Research and Development Urbana, IL 61801-2932	10. Project/Task/Work Unit No.
	11. Contract/Grant No. US NSF-DCR84-10110; DOE-DE-FG02-85ER25001
12. Sponsoring Organization Name and Address National Science Foundation, Washington, D.C.; and U.S. Department of Energy, Washington, D.C.	13. Type of Report & Period Covered Master's Thesis
	14.

15. Supplementary Notes

16. Abstracts

The Structured Memory Access (SMS) architecture implementation presented in this thesis is formulated with the intention of alleviating two well-known inefficiencies that exist in current scalar computer architectures: address generation overhead and memory bandwidth utilization. Furthermore, the SMA architecture introduces an additional level of parallelism which is not present in current pipelined supercomputers, namely, overlapped execution of the access process and execute process on two distinct special-purpose, asynchronously-coupled processors. Each processor executes a separate instruction stream to perform its specific task which, together, are functionally equivalent a conventional program. Our simulation results show that, for typical numerical programs, the access processor (MAP) is capable of achieving slip, i.e. running suficiently ahead of the execute processor (CP) so that operand fetch requests for data items required by the CP are issued early enough and rapidly enough for the CP rarely to experience any memory access wait time. In this manner the SMA tolerates long memory access time, albeit high bandwidth, paths to memory without sacrificing performance. Speedups relative to the Cray-1 in scalar mode often exceed two, due to dual processing and reductions in memory wait tiem.

17. Key Words and Document Analysis. 17a. Descriptors

Architecture
Performance-evaluation
Decoupled access-execute
Memory accessing
Pipelining

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Unlimited Distribution	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 70
	20. Security Class (This Page UNCLASSIFIED	22. Price