Design of Pipelined Memory Systems for Decoupled Architectures

by

Koujuch Liou

Computer Sciences Technical Report #617

October 1985

DESIGN OF PIPELINED MEMORY SYSTEMS FOR DECOUPLED ARCHITECTURES

by

Koujuch Liou

Computer Sciences Department
University of Wisconsin–Madison
Madison, Wisconsin 53706

October 1985

ABSTRACT

Design of memory systems for decoupled architectures is the theme of this dissertation. The decoupled architecture uses hardware queues to architecturally decouple memory request generation from algorithmic computation. This results in an implementation that has two separate instruction streams that communicate via hardware queues. Thus, performance is improved through parallelism and efficient memory referencing.

Techniques for increasing memory bandwidth and algorithms for servicing memory requests are incorporated into the memory system designs within these two constraints: (1) the operands placed in the hardware queue must be in the correct order, and (2) the needed operands are the only operands that can be placed in the hardware queue.

Techniques such as pipelining, interleaving, servicing requests out of arrival order, and cache memory are investigated. Two strategies for servicing memory requests are studied: (1) to service requests according to their priorities, and (2) to minimize the total request service time. For the first strategy, the priority of each request type is derived from the characteristics of memory reference and possible bottleneck during decoupled computations. The second strategy results in a request scheduling policy, *Free-Module-Request-First*, that is proven to be able to minimize the total request service time.

A sequence control scheme must be used with the Free-Module-Request-First scheduling policy in order to deliver the memory outputs to the hardware queue in the correct order. This sequence control scheme is also used to track cache hits and misses, so that a data cache can be implemented in the memory system without difficulty.

The designed data cache can not only support flexible fetch and replacement cache algorithms, it can also detect memory access hazards and short-circuit the Read-After-Write requests. Therefore, the penalty of memory access hazards can be greatly reduced.

The combination of the designed data cache and the pipelined interleaved memory system using Free-Module-Request-First scheduling policy results in a high-performance memory system, that is capable of servicing memory requests nearly no conflict delay under the particular workload defined in the trace files.

ACKNOWLEDGEMENT

Professor James R. Goodman has been an exceptional advisor to guide my graduate career with patience and wisdom. His enthusiasm to share his knowledge with me, to think seriously about my research, and to read carefully my revisions have made the final form of my thesis tighter and neater than it would have been. The constructive comments and valuable conversations of Professors Andrew R. Pleszkun and Mary K. Vernon did much to clarify many points in this thesis; I am indebted to them and to Professors Charles N. Fischer and Charles R. Kime for their efforts with my thesis.

Special thanks must also be given to the members of the PIPE research group for their assistance in the simulation study, and to the colleagues in the Academic Computing Center for their support during my graduate study. Also, I would like to thank Michael Caligiuri for his time in reading and correcting my manuscripts. Their friendship has made my stay in Madison a pleasant time.

Words are not adequate to express my deep gratefulness to my parents, my parents-in-law and my wife for their love and support. Especially my wife, Gloria, her thoughtfulness, patience and encouragement have turned many distressful moments to joyful memories.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Several different systems using the technique of decoupled computation have been recently designed [CoSt81, SmiJ82, Ples82, SPKG83]. The CSPI MAP-200 array processor [CoSt81], the Decoupled Access/Execute (DAE) architecture [SmiJ82, SmiJ84], the Structured Memory Access (SMA) architecture [Ples82, PlDa83], and the PIPE (Parallel Instructions and Pipelined Execution) architecture [SPKG83, GHLP85] all share the same design concept: memory request generation is architecturally decoupled from algorithmic computations, resulting in improved performance through parallelism and efficient memory referencing. For example, Weiss and Smith [WeSm84] demonstrated that by decoupling data access from execution, it was possible to implement a computer architecture with minimal design complexity and to provide much of the performance improvement offered by complex issuing methods, such as issuing instructions out of order or issuing multiple instructions simultaneously. Decoupling also allows considerable memory access delay to be overlapped with other operations. The SMA machine of Pleszkun [Ples82, PlDa83] reduced addressing overhead by providing special access mechanisms in the memory access processor to generate references efficiently for blocks of instructions and several data types. Pleszkun's results showed that the SMA machine reduced the number of memory references to between 1/5 and 2/5 of those required by a conventional VAX computer.

Unlike the von Neumann architecture which consists of one central processing unit (CPU), the decoupled architecture consists of two processors: the *Access Processor* and the *Execute Processor.* Likewise, a single task in the von Neumann architecture is now separated into two subtasks: the *access task* and the *execute task.* Both Access and Execute processors cooperate in executing these two subtasks in parallel and communicate via *hardware queues.* The Access Processor calculates all memory addresses and makes all memory data references for the Execute Processor. Fetched operands are placed in an input queue of the Execute Processor. The Execute Processor carries out the algorithmic operations required on a given operand stream. The results computed by the Execute Processor will be paired with the addresses calculated by the Access Processor, and then stored into main memory. The Access Processor can overlap memory transactions by sending more requests than the memory system can immediately satisfy, and the memory, in turn, may provide more data than the Execute Processor can immediately utilize.

In order for decoupled computation to proceed smoothly, the memory system should be able to service memory requests efficiently and provide operands in order and before they are needed by the Execute Processor. Therefore, the successful implementation of a memory system is key to the improved performance obtained through decoupled computation.

The design of the memory system for a decoupled architecture is the topic of this dissertation. The goal of this research is to design a memory system that can service memory requests efficiently in order to sustain the improved performance through decoupled computation. Techniques to increase the performance of memory systems will be investigated, such as pipelining, interleaving, multiple-words-fetched-per-request, servicing requests out of arrival order, and cache memory. Request scheduling policies will be derived from two strategies: (1) to service requests according to their priorities, and (2) to minimize the total request service time. Then, memory systems with the derived request scheduling policies and the performance improvement features will be designed and evaluated through trace-driven simulations.

A general understanding of the features and performance levels of a decoupled architecture is important to this research. Because the PIPE architecture has all the necessary features for performing decoupled computations, a look at the PIPE architecture can show us clearly how the decoupled computation is performed, and how memory transactions are overlapped. The limitations and problems in the memory system will be listed along with general strategies and approaches to address them. Details of the proposed memory system will then follow in the remaining chapters of the thesis.

1.1. The PIPE Architecture

The PIPE architecture (Parallel Instructions and Pipelined Execution) [SPRG83, GHLP85] is a pipelined, decoupled architecture designed in the VLSI environment. It uses identical processors for the Access and Execute processors that communicate via hardware queues (Figure 1.1). PIPE uses a simple instruction set with the goal of issuing one instruction per CPU cycle in each of the processors. Each processor issues memory requests to fetch its own instruction stream, and is capable of fetching its own operands[1]. Each processor has registers and hardware queues to store information and a program counter to track its progress. In addition, there is an instruction cache on each processor chip.

Decoupled computation allows the PIPE architecture to explore a new level of parallelism, which is lower than the multiprogramming environment, without precluding traditional multiprogramming. This new level of parallelism permits some code scheduling to be performed dynamically (at run-time), and reduces the burden of static code scheduling done by the compiler.

There are three considerations that guide the design of the PIPE architecture: (1) the von Neumann bottleneck [Back78], (2) the Flynn limit [Flyn66], and (3) efficient code generation by a compiler. The von Neumann bottleneck refers to the communication path between the processing unit and the main storage unit. This path is a potential bottleneck because instruction fetch and instruction execution contend for its available bandwidth. The Flynn limit is the observation first stated in [Flyn66] that at most one instruction can pass through the instruction fetch/decode path per clock period. Code scheduling is required to order the instruction codes for a pipelined architecture, so that the amount of overlapped operations on the pipeline stages can be maximized. Thus, PIPE is designed with three important features: (a) the prepare-to-branch/exit [Scho71] program control scheme, (b) memory Load/Store operation through hardware

[1]This capability is not used in the Execute Processor.

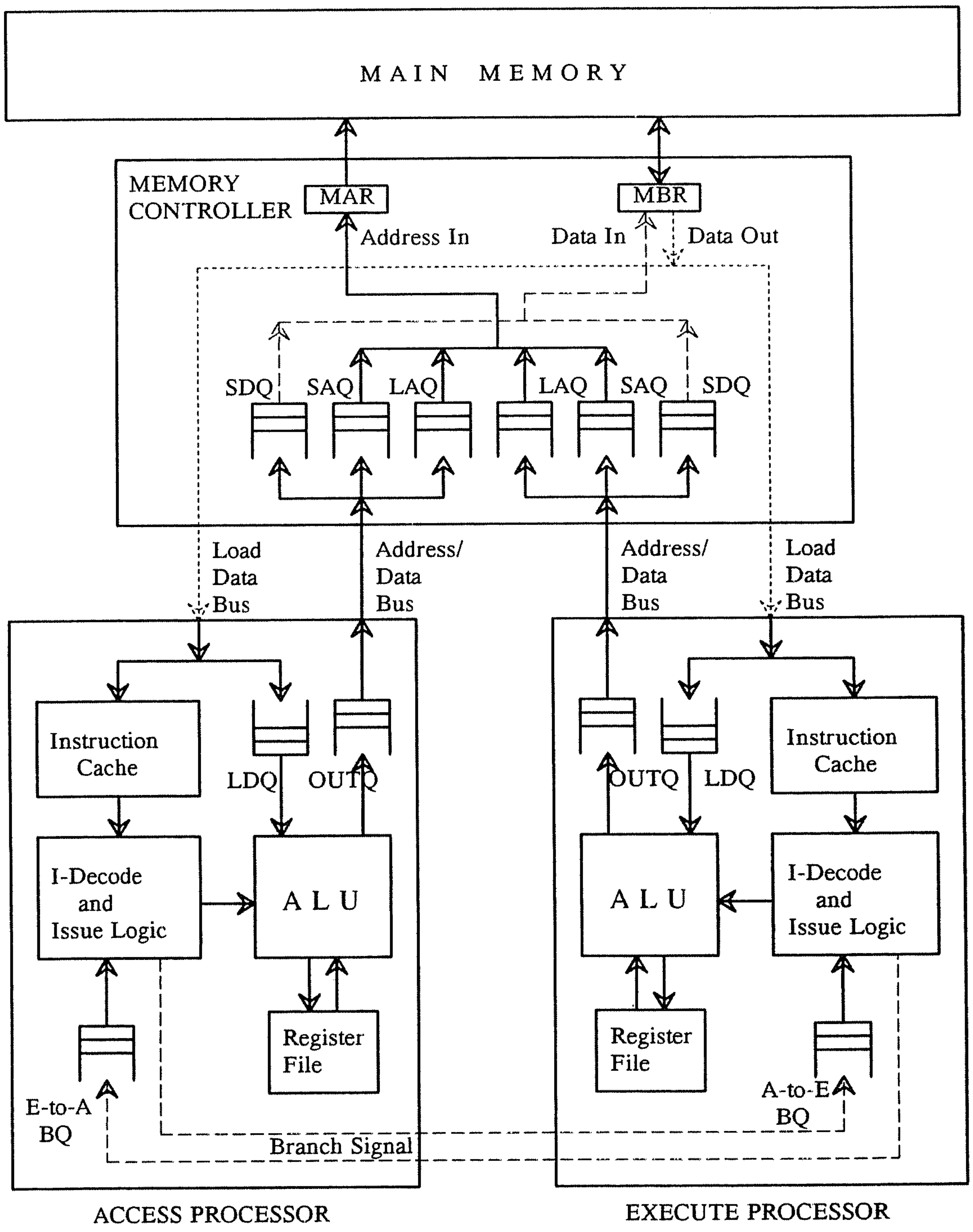


Figure 1.1 Block Diagram of a PIPE Architecture

queues, and (c) instruction cache. These features combine to reduce the severity of the von Neumann bottleneck. While features (a) and (b) provide flexibility in compiler code generation, they also result in simple pipeline interlock circuitry, increasing the possibility of issuing one instruction per CPU cycle. For the Flynn limit, PIPE uses two processors to execute two instruction streams simultaneously, so that the number of instructions passing through the instruction fetch/decode paths can be greater than one per clock period.

As shown in Figure 1.1, there are several queues in the PIPE architecture; each is implemented in the hardware. The *Branch Queue* (BQ) is used to pass control information (i.e., branch decisions) between the Access Processor and the Execute Processor. Memory requests are temporarily stored in the *Output Queue* (OUTQ) and are later sent to the memory system as soon as the memory system is prepared to accept the requests. The *Load Data Queue* (LDQ) stores the operands fetched from the memory. The *Load Address Queue* (LAQ) stores the load address requests that fetch operands for the corresponding LDQ of a processor. The *Store Data Queue* (SDQ), and the *Store Address Queue* (SAQ) in the memory system store the store address and the store datum, respectively.

The load operation of the PIPE architecture places a load request into the OUTQ, this load request is then sent to the memory system to fetch the datum from the target address into the LDQ. Another operation retrieves the datum from the LDQ when the datum is needed. The load instruction that starts the load operation is considered to be completed once the load request is placed into the OUTQ. The store operation of the PIPE architecture consists of a store address operation and a store data operation. The store address operation puts an address into the SAQ, and the store data operation puts a datum into the SDQ. When both the SAQ and the SDQ are not empty, the address and the datum from the heads of both queues are paired and sent to the memory as a store operation. The load/store scheme of the PIPE architecture provide the compiler flexibility to schedule other instructions between the separated load data and retrieve data operations or the separated store address and store data operations, so that the effective delays among the instructions with data dependencies can be reduced. In addition, the separation dissociates the completion of a load/store instruction from the completion of its corresponding memory operation. Therefore, it is possible for the compiler to generate efficient codes which will utilize the computing resource effectively and achieve the issue rate of one instruction per CPU cycle.

The load request of a processor that fetches and delivers data to the LDQ of the other processor is called an *alternative load request.* An *alternative store request* consists of two operations issued by two different processors: (1) an alternative store address operation to put an address into the SAQ of one processor, and (2) an alternative store data operation to put a datum into the SDQ of the other processor. Subsequently, the address and the datum from the SAQ and the SDQ of different processors will be paired in the memory.

In order to increase the memory bandwidth and to facilitate the flow of memory requests through the von Neumann bottleneck, each PIPE processor has two uni-directional buses. One is dedicated to memory input for sending addresses and data to the memory system, and the other is dedicated to memory output for receiving data from the memory system. Tags are used to distinguish between the different types of items appearing on the bus. All the data paths are uni-directional so that memory requests and fetched data can be overlapped.

1.2. Constraints on the Memory System

While the use of hardware queues contributes several advantages to a decoupled architecture, it also places constraints on the memory system. The constraints are: (1) the data must be placed in the queue in the correct order based on the program algorithm, and (2) the required data is the only data allowed in the queue. Three techniques to increase memory bandwidth will be investigated: cache memory [SmiA82], multiple-words-fetched-per-request, and servicing memory request out-of-order [BoGr67]. Each of these techniques must work with a bookkeeping scheme, whose responsibility is to reorder the fetched data while complying with the constraints.

1.3. Design Issues of the Memory System

Design techniques are employed to construct an efficient memory system within the constraints stated above. Three issues related to the use of queues to buffer memory requests and fetched data in the design of the memory system are described below. Each issue is accompanied by a possible design alternative.

(1) Multiple memory requests can be simultaneously waiting for service in the memory system. An efficient request scheduling algorithm may be needed to maximize the throughput and/or meet the urgency of memory requests.

(2) If the load address and the store address requests are stored in separate queues, the load and store request can be serviced out of arrival order. Thus, memory access hazards -- Read-After-Write, Write-After-Read, and Write-After-Write hazards -- must be checked before a load or a store request is serviced. When fetch and store requests are accessing the same memory location, the memory system may detect a Read-After-Write or Write-After-

Read access hazard before the store datum arrives. If a Read-After-Write access hazard is detected, the memory system should short-circuit the store datum to the fetch request, so that a memory fetch cycle can be avoided. The impact of access hazard on the performance of memory system will be investigated, as well as the performance improvement through short-circuiting Read-After-Write data.

(3) Each PIPE processor has two buses dedicated to input and output, respectively. The memory system can be designed to receive two inputs per clock period, one from each processor, or one input per clock period. The performance difference for these two alternatives will be investigated. If the performance difference is substantial, alternative multiplexing schemes for receiving one input per clock must also be studied.

The memory controller can be considered as another processor in a decoupled architecture. Its responsibility is to service requests efficiently and to provide operands before they are needed, so that decoupled computation can proceed smoothly without excessive memory wait time. During decoupled computation, it is possible that the Execute Processor falls behind the Access Processor. It is also possible that the memory controller cannot meet the demand from both Access and Execute processors. If the Execute Processor is running behind and becomes the bottleneck of decoupled computation, the memory controller should service the request of the slower processor first. Thus, it will reduce the memory delay for the slower processor and speed up the progress of decoupled computation. If the memory controller cannot meet the demand of requests from the processors, the memory controller should service requests as fast as it can and minimize its effects as the bottleneck of decoupled computation. Thus, the memory request scheduling policy can be designed with two strategies: (1) to service requests according to their priorities, or (2) to minimize the total request service time.

Since there are different ways to organize the queue space, to store memory requests, to schedule memory requests, to design multiplexing schemes, and to implement bookkeeping schemes, this research will study alternative schemes to address each issue. These schemes will each be evaluated, and their tradeoffs analyzed. A selection of the most efficient scheme is then proposed.

A review of the previous work related to the performance of interleaved memory models, memory access conflicts and access hazards, memory system designs of decoupled and non-decoupled architectures, will be discussed in the next chapter. After a study of memory reference characteristics for decoupled computation is presented in chapter 3, several request scheduling policies will be proposed. The design and the simulation of the pipelined, interleaved memory systems for these scheduling policies will be presented in chapters 4 and 5, respectively. Chapter 6 will discuss the design of a data cache to further enhance the performance of the memory system.

CHAPTER 2

A Survey of High Performance Memory System Designs

The memory system is a major bottleneck of a von Neumann architecture [Back78]. Matching the memory bandwidth with the speed of the central processing unit has long been an important factor in the design of computing systems.

Latency is the time it takes for a processor to fetch information from the main memory. When the memory system receives a request, three steps are required to complete the service. First, the address of the request is decoded; then, the addressed memory location is selected. Finally, the read or write of a word at this location will take place. The total fixed time to carry out these three steps is called a *memory bank busy time.* A memory module can only accept one request per memory bank busy time. If there are parallel memory modules in the memory system and the stream of memory references are staggered to enough different memory modules, the apparent memory throughput can be much higher than the reciprocal of memory bank busy time.

Memory *access conflicts* and *access hazards* are two important factors that prevent the interleaved memory system from achieving its maximum bandwidth.

A memory access conflict occurs when a memory request is accessing a busy memory module, or when two or more memory requests attempt to access the same memory module in the same clock period. When an access conflict occurs, one of the memory requests can be stored in conflict buffers and processed in the next memory cycle. A memory access hazard exists when more than one memory request references the same memory location, and an incorrect sequence of memory operations can result in using wrong data or storing wrong information into memory. There are three kinds of access hazards: the *Read-After-Write* (RAW) hazard, the *Write-After-Read* (WAR) hazard and the *Write-After-Write* (WAW) hazard. Techniques for reducing memory access conflicts and resolving access hazards will be discussed in the following sections.

A discussion of the characteristics and the theory of interleaved memory systems is necessary for designing a high performance interleaved memory system. This is followed by examples of methods used to reduce memory access conflicts and resolve memory access hazards in current memory systems.

2.1. Theory of Interleaved Memory System

Consider an interleaved memory system with m memory modules of n memory words per module and in which each memory module performs operations independently. The memory word is the unit of a memory operation, and not necessarily the same width as the CPU word. Interleaving techniques can increase memory bandwidth, allowing the use of slow, inexpensive main memory to match the speed of the processing unit, and providing a large memory at low cost.

In a conventional memory system, each memory module contains *n* consecutive memory words and the word at address *i* is in module *i div n.* The amount of memory in the system can be increased by simply adding more memory modules. When a memory module malfunctions, a contiguous block of memory is inaccessible; the rest of memory can still be accessed. Because instructions are stored in adjacent memory words, the likelihood that a segment of program code is stored in one memory module would be high. In such a case, memory conflicts are frequent and the latency for instruction fetches will be as long as the memory bank busy time. However, this memory organization can be used for some systems where reliability, reconfigurability, or cost is the main concern.

In the interleaved memory system, the consecutive *n* memory addresses are located across the *m* memory modules -- the word at address *i* is in module *i mod m.* This low-order interleaving scheme can eliminate most memory conflicts, resulting in latency close to the access time (the time required by a memory module to provide the stored information). This technique is used in many high performance computer systems, such as the IBM 360/91 [AnSp67, BoGr67], the CDC6600 [Thor70], and the CRAY-1 [Russ78]. The low-order interleaving scheme is assumed throughout the rest of the dissertation.

Since the first appearance of interleaved memory systems in STRETCH [Kuck78] and ILLIAC II [UIll57], memory models [Hell67, BuCo70, CoBu71, Ravi72, Bhan75 and ChKL77] with different assumptions have been introduced to evaluate the performance of interleaved memory systems. The effective memory bandwidth, which is defined to be the average number of memory accesses per memory cycle, is used to measure the performance in each memory model.

Chang and others [ChKL77] established models to study the data dependency of the address streams and the performance of interleaved memory models with conflict queues. Their results indicate that the performance of a properly designed memory system can be a linear function. Hellerman's model [Hell67], on the other hand, established a square root function of the number of memory modules.

2.1.1. Memory Models without Conflict Queues

Hellerman's model considers memory requests as a stream of addresses in the range of 1 through m, where m is the number of memory modules. His assumptions are that no queuing of requests are permitted on busy modules, and that there is no data dependency between successive memory references. Addresses are examined in order until the first duplicate memory module number is found. These first k requests to distinct addresses are then processed in parallel. He shows that the probability of encountering a string of exactly k distinct integers with the $(k + 1)$st a repetition of one of the k others is

$$P(k) = \frac{k\,(m - 1)!}{m^k\,(m - k)!}$$

The average length of address sequences, which is the effective bandwidth, is the sum of the products of the probabilities and k:

$$Bandwidth = \sum_{k=1}^{k=m} k\,P(k)$$

When $1 \le m \le 45$, Hellerman found a good numerical approximation of the above equation to be $m^{0.56}$, or approximate $\sqrt{m}$. The error is no more than 4.3

percent. This means that the effective memory bandwidth of an interleaved memory system is a *square root* function of the interleaving factor. Knuth and Rao [KnRa75] also show in a closed form that the effective bandwidth of Hellerman's model is

$$Bandwidth = \left(\frac{\pi m}{2}\right)^{1/2} - \frac{1}{3} + \frac{1}{12}\left(\frac{\pi}{2m}\right)^{1/2} + O(\, m^{-1}\,)$$

This confirms that Hellerman's bandwidth is asymptotic to $\sqrt{m}$.

In Ravi's memory model [Ravi72], p memory requests are generated during each cycle. This model can be considered as a multiprocessor system of p processors and m memory modules. Ravi derives a formula to compute the average number of *distinct* integers in a group of p integers chosen uniformly from the integers 0 to m - 1. He takes this value to be the effective bandwidth in the steady state and is computed as

$$Bandwidth = \sum_{k=1}^{k=t} k \, \frac{k!S(p,k)\binom{m}{p}}{m^p}$$

where $t = min(m,p)$, $S(p,k)$ is a Stirling number of the second kind, and $k!S(p,k)$ is the number of ways to put p distinct requests into k distinct memory modules with each modules holding at least one request. Ravi's formula was reduced to a very simple closed form [ChKL77], that is

$$Bandwidth = m\left[1 - \left(1 - \frac{1}{m}\right)^p\right].$$

It is shown that, given a constant p/m ratio, the effective bandwidth in the steady state is a *linear* function of either m or p. The assumption in Ravi's model that the conflicting requests are simply ignored and not re-submitted does not correspond to any real machine.

Another memory model similar to Ravi's model but of different assumptions is introduced by Chang [ChKL77] to study the performance of interleaved memory in multiprocessor systems. Chang's model assumes that there are p independent address streams, each uniformly distributed, issued from p processors. In any given stream, an address can be accessed only after its predecessor has been accessed. Those processors not being serviced will reissue their addresses in the next cycle. Simulation techniques are used by Chang to find the steady-state bandwidth of their model. The results show that the effective bandwidth is linear in p but is 6 to 8 percent worse than Ravi's result. Chang believes that the difference is due to the difference in assumptions and the statistical error in the simulation. Chang's results are very close to Bhandarkar's results [Bhan75], where a similar model is solved with a Markov chain method.

2.1.2. Memory Models with Conflict Queues

Coffman, Burnett, and Snowdon [CoBS71] introduced a memory model with a conflict queue to buffer conflicting requests. They assumed that the addresses are not uniformly distributed but are determined by two parameters α and β. They assumes that the probability of a request accessing the next module in sequence (modulo m) will be α and the probability of addressing any other module out of sequence will be β, where $\beta = (1-\alpha)/(m-1)$.

The stream of memory requests from the request queue is examined and the conflicting requests are placed in the conflict queue. For each memory cycle, the conflict queue are first scanned and then the new requests in the request queue are scanned. This procedure continues until either the conflict queue is full or all the memory modules are busy. Because of memory conflicts, memory requests can be serviced out of their arrival sequence in this model. However, this model assumes that there are no data dependencies between requests and therefore out-of-order service is permitted. Their results indicate that handling requests out-of-order can greatly increase memory bandwidth. For example, if $\alpha = 0.25$ and $m = 16$, the bandwidth is about 4.5 when no conflict queue is used but jumps to 9.5 if a conflict queue of length 4 is used. Clearly, as the length of the conflict queue approaches infinity, the bandwidth approaches the interleaving factor m.

A memory model with conflict queues is also introduced by Chang [ChKL77] to investigate the effect of using queues to improve the memory bandwidth in a multiprocessor system. There are p processors and m memory modules in this model, each memory module has a conflict queue. When a memory module receives more requests than it can queue, then the processors which submitted the unqueueable requests are blocked and will submit their request during the next memory cycle. Since it is possible for a processor to have two or more requests satisfied in one memory cycle (e.g. its current request in one module plus some previously queued requests in other modules), Chang assumes that the processors are capable of accepting more than one returned result in one memory cycle. Simulation techniques are used to evaluate this model. This study indicates that adding queues to the memory system did improve the bandwidth of the system, and the effective bandwidth in the steady state is a *linear* function of r, where $r = p/m$. When $r = 1$, the simulation results showed an approximate 5 percent improvement over Ravi's model if the memory queue length is 1, and an approximate 10 percent improvement over Ravi's model if the queue length is 2.

In summary, the literature review of interleaved memory models suggests that if a conflict queue is not implemented in an interleaved memory system, (1) for a single request stream, the effective memory bandwidth approximates a *square root* of the interleaving factor; (2) for p request streams and m memory modules, the effective memory bandwidth is a *linear* function of either p or m, given a constant p/m ratio. If a conflict queue is used with an interleaved memory system that services a single request stream, the effective memory bandwidth is a *linear* function of the interleaving factor.

2.2. Methods to Reduce Memory Access Conflicts

Although interleaved memory systems can reduce the frequency of memory conflicts and balance the memory speed with the processor speed, the nondeterministic occurrence of memory conflicts can degrade the performance of a memory system when the request arrival rate is increased. Better memory systems design requires understanding memory reference patterns and program behavior.

Memory references may be partitioned into three categories: (1) instruction fetch, (2) scalar variable access, and (3) array variable access. These three categories of memory reference types and their characteristic access patterns can influence the approach to the design of interleaved memory systems. The memory addresses for instruction fetches are sequential until there is a successful branch. Scalar variables are frequently stored in blocks of memory spaces, but are referenced unpredictably. Array variables are usually referenced in a regular sequence. The addresses of these referenced elements can be in contiguous memory locations, or in locations with the same number of words separating two consecutively referenced elements. The distance between two consecutive array element references is called a *stride*.

Since the instruction stream comprises sequential words, multiple-words-returned-per-fetch and/or prefetching techniques may be used to fetch the instruction words. Many mainframe systems use wide processor-memory buses to fetch more than one memory word per access, with the expectation that the prefetched words would be used in following cycles. For example, two 32-bit words are fetched per memory access in the IBM SYSTEM/360 Model 91 [AnST67, BoGr67], and the fetched instruction words are stored in an *instruction buffer array*. Other computer systems with cache memories use the cache line to be the unit of memory access such that contiguous memory words are fetched into the cache memory in each memory fetch cycle. Instruction buffer arrays and cache memories are used not only to match the demand from the processing units, but also to reduce the number of memory requests, thus reducing memory conflict.

For array variable accesses, many supercomputers such as the CRAY-1 [Russ78], the CDC Star-100 [HiTa72], the CYBER205 [Linc82], and the Burroughs Scientific Processor BSP [KuSt82], use interleaving techniques and vector instructions to facilitate vector operations. The arrays must be carefully assigned to memory in order that conflict-free memory access is achieved.

The memory system of the CRAY-1 computer is 16-way interleaved. It prevents memory conflicts except in the case of memory accesses that step through memory with a multiple of 8-word increment. Vector instructions place a reservation on whichever functional unit they use, including memory, and on the input operand registers. Vector elements are brought into the register files and manipulated within the register files. Once issued, a vector instruction produces its first result after a delay equal to the functional unit latency. Subsequent results then continue to be produced at a rate of 1 per clock period. These results must be stored in a vector register. A separate instruction is required to store the final result vector to memory. In order to facilitate the operation flow of a vector functional unit, the memory system has to be idle before a vector stream is initiated. The memory system is reserved for the vector operation such that the vector elements can be accessed without interference by scalar accesses. Conflict-free memory references can thus be scheduled unless the stride is a multiple of eight. When a vector memory reference with a stride of a multiple of eight is detected, *speed control* is put into effect, and vector elements are referenced at a reduced but predictable rate.

The CDC Star-100 and the CYBER 205 are *memory-to-memory* vector machines. Unlike the CRAY-1 computer, the data in vector form are read from central memory and sent to the arithmetic units with the results returned to memory upon completion of the vector operation. These two systems can support up to three simultaneous vector streams: two load streams and one store stream. In the CYBER 205, the memory system is 128-way interleaved; it is organized into sixteen sections each of eight banks. The lowest order three bits of a memory address select the bank number of a section, which is selected by the next higher four bits. Instead of *slotting* I/O requests around the memory demands of the vector streams as in the input/output scheme of the STAR-100, the CYBER 205 designers chose to give I/O free entry to the central memory. In order to service two vector load streams, one vector store stream, and an I/O data stream simultaneously, the memory system of the CYBER 205 fetches eight consecutive memory words belonging to one vector and stores them in buffers. In the following clock period, eight consecutive memory words of another vector are fetched and stored into buffers. In the third clock period, eight buffered results can be stored into the memory. The fourth clock period is dedicated to I/O transfers. Then, the same four operation sequence is repeated. If the vector accesses are not of stride one, the needed elements must first be *gathered* from the buffered consecutive vector words before being used as input operands, and the results must be *scattered* into the consecutive memory words in the result buffers before being stored in the memory. A vector control unit will determine the timing of memory requests when a vector instruction is initiated, so that there are no memory conflicts.

The memory system for the Burroughs Scientific Processor was organized differently from those of the other vector machines. There were two separate memory units in the BSP system: the *control memory* and the *parallel memory*. The *control memory* was used to store portions of the operating system and all user programs as they were executed. It was also used to store data values that were operands for those instructions executed by the scalar processor unit. The *paral-*

lel memory was used only to hold data arrays for the *parallel processor.* There were 16 arithmetic processing units in the *parallel processor,* and 17 memory modules in the *parallel memory.*

The organization of the *parallel processor* and the *parallel memory* is the result of a study of parallel memory by Budnik and Kuck [BuKu71]. They demonstrated that, in general, accessing rows, columns, and diagonals of a square array without conflicts is impossible to accomplish if any *power of two* number of memory modules is used.

If the distance between the referenced array elements and the number of memory modules are relatively prime, however, then all the modules can be accessed simultaneously without conflict. The data of array elements are then stored such that there are conflict-free accesses to row, column, and diagonal vectors of length 16.

An input alignment network is needed to arrange the fetched array elements into proper order in this memory system, and an output alignment network is used to re-align the array elements before they are stored into the *parallel memory.* Also, modulo 17 arithmetic is required to compute the module number of an array element and the address within the module. For example, the module number M of $A(i,j)$ is

$$M = (j*I + i + base) \bmod 17$$

where I the row dimension of array $A(I,J)$ and *base* is the starting address of array $A(I,J)$.

2.3. Memory Access Hazard and Its Resolution Schemes

Since the ultimate performance of an interleaved memory system is limited by the memory modules themselves, the memory controller must optimize the data rate by efficiently scheduling the unserviced requests and controlling the processor-memory buses. The data rate can be increased by issuing memory operations for memory requests whenever the required modules become available, i.e., not in logical order. The memory controller must detect the access hazards and correctly sequence several stores, or stores and fetches, to the same memory location.

Compiler techniques can also be used to avoid memory access hazards. Given certain physical constraints, the compiler can detect the potential access hazards at compilation time, then schedule the code such that the second memory request will not be started until the first memory request to the same address is completed. However, because every store request can create a potential access hazard, it is not a simple task for the compiler to precisely detect the access hazards. For example, when a fetch instruction and a store instruction are generated for two elements of the same array variable that are referenced in an arithmetic statement, the compiler has to know the indices of these two elements in order to decide whether or not there is a potential access hazard between them. If the indices of these two array elements are indexed by two arithmetic expressions and the results of these two expressions cannot be computed in the compilation time, then the compiler can only assume that there is a potential access hazard between these two array elements. In addition, if the fetch and the store of the same array variable are initiated from different program statements or each of them are submitted by the code of the same loop segment but at different iterations of the loop during the run-time, the correct detection of access hazard can become very complex for the compiler. Also, worst-case behavior must always be assumed. For a cache memory system, this is a serious penalty because the compiler decides whether a potential access hazard may occur during the run-time based on the completion time of the memory request in conflict. Thus, the time required to fetch information from main memory for a cache miss must be assumed for each detection of potential access hazard.

Alternatively, hardware control can be used to detect and resolve the memory access hazards when requests are submitted at run-time. Three computer systems, the IBM SYSTEM/360 Model 91, the IBM SYSTEM/370 Model 168, and the CSPI MAP-200 array processor [CoSt81], will be used as examples to discuss how memory access hazards are resolved in non-decoupled architectures and a decoupled architecture.

2.3.1. The Storage System of the IBM SYSTEM/360 Model 91

The IBM SYSTEM/360 Model 91 [AnST67, BoGr67] is a high performance pipelined computer system. In order to achieve a smooth operation flow along the pipeline stages, the organizational techniques of storage interleaving, arithmetic execution concurrency, and buffering are utilized. The tasks of the pipeline stages are performed by three major processing units: the *Instruction Unit* (I-unit), the *Execution Unit* (E-unit), and the *Main Storage Control Element* (MSCE). The E-unit is further divided into two independent units: the fixed point execution unit and the floating point execution unit. A block diagram of the IBM SYSTEM/360

Model 91 is shown in Figure 2.1 with an illustration of buffer allocation and function separation.

The central control functions for the CPU are performed in the I-unit, which decodes and sets up instructions for execution by the E-unit. The I-unit accomplishes this by scanning each instruction, in the order presented by the program, and clearing all necessary interlocks for buffer allocations before releasing the instruction. In addition, when a storage reference is required by the operation, the issuing mechanism performs the necessary address calculations, initiates the memory requests, and establishes the routing by which the operand and operation will ultimately be merged for execution within the E-unit.

In designing the I-unit, many program situations were examined by the design group of the IBM SYSTEM/360 Model 91 [BoGr67]. They found that, while many short instruction sequences are nicely ordered, the trend is toward frequent branching. In addition, even with sophisticated execution algorithms, very few programs can actually cause answers to flow from the pipeline stages at an average rate in excess of one every two cycles. Inter-instruction dependencies, storage and other hardware conflicts, and the frequency of operations requiring multi-cycle execution all combine to prevent this rate from being achieved.

In order for the I-unit to consistently run ahead of the E-unit, the multiple-words-returned-per-fetch scheme is used in conjunction with instruction fetching. Additionally, various types of buffers are used to smooth the total instruction flow by allowing the initiating pipeline stages to proceed despite unpredictable delays further down the pipeline. Buffers for instruction fetch, operand fetch, operand store, operation, and address are used among the I-unit, the E-unit and the

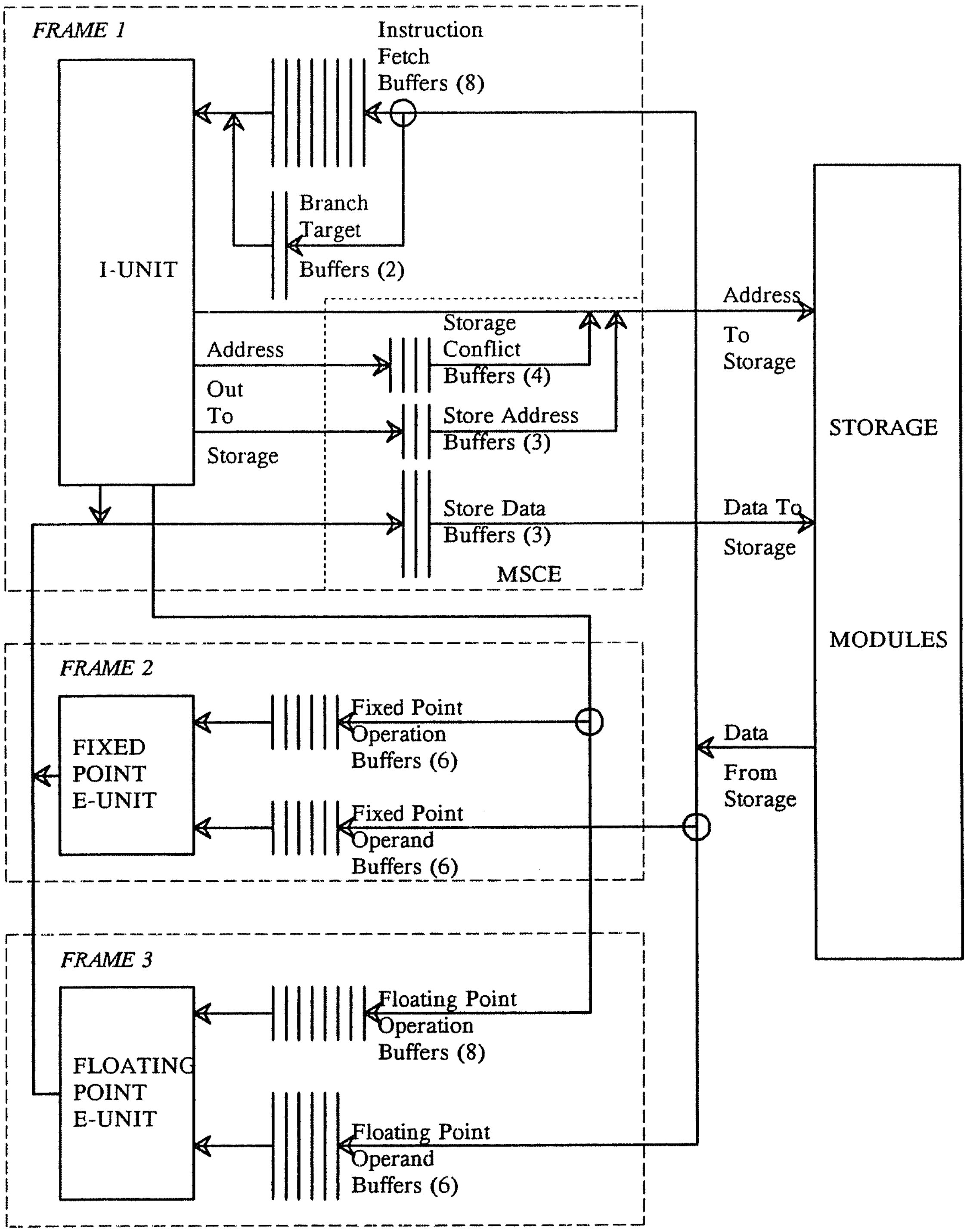

FRAME 1
Instruction Fetch Buffers (8)
Branch Target Buffers (2)
I-UNIT
Address Out To Storage
Storage Conflict Buffers (4)
Store Address Buffers (3)
Store Data Buffers (3)
MSCE
Address To Storage
Data To Storage
STORAGE MODULES
Data From Storage
FRAME 2
FIXED POINT E-UNIT
Fixed Point Operation Buffers (6)
Fixed Point Operand Buffers (6)
FRAME 3
FLOATING POINT E-UNIT
Floating Point Operation Buffers (8)
Floating Point Operand Buffers (6)

Figure 2.1 -- Buffer allocation and function separation within the IBM SYSTEM/360 Model 91 from [AnST67].

MSCE as illustrated in Figure 2.1. The I-unit will not issue an instruction unless the required buffers are available. For example, a free operand buffer will be assigned to an operand fetch request before the request is sent to the MSCE, so that there is always a buffer available to receive the operand fetched from the memory.

For each instruction, following the clearing of all interlocks, the decode decision determines whether or not to issue the instruction to the E-unit and initiate address generation for memory access, or to retain the instruction for sequencing within the I-unit. For example, after a storage-to-register (RX) instruction is decoded, the issuing of the RX instruction to the E-unit and the fetching of the operand together constitute a controlled splitting operation, where sufficient information is forwarded along both paths to effect a proper execution unit merge.

The block diagram of the MSCE is shown in Figure 2.2, where MC is the maintenance console and PSCE is the Peripheral Storage Control Element. There are different functional areas within the MSCE. Only those related to memory access hazard resolution will be discussed. Details about other functions of the MSCE can be found in [BoGr67].

There are three request queues in the MSCE, each of them associative for address matching: (1) The store address registers, where three registers hold the addresses of stores pending availability of store data. (2) The request stack where a set of four registers holds rejected requests from the processor pending availability of the memory module, and thus buffers the processor from storage conflicts. (3) The accept stack where a set of shift registers hold information on accepted requests in process. In addition, there are three store data buffers that hold store

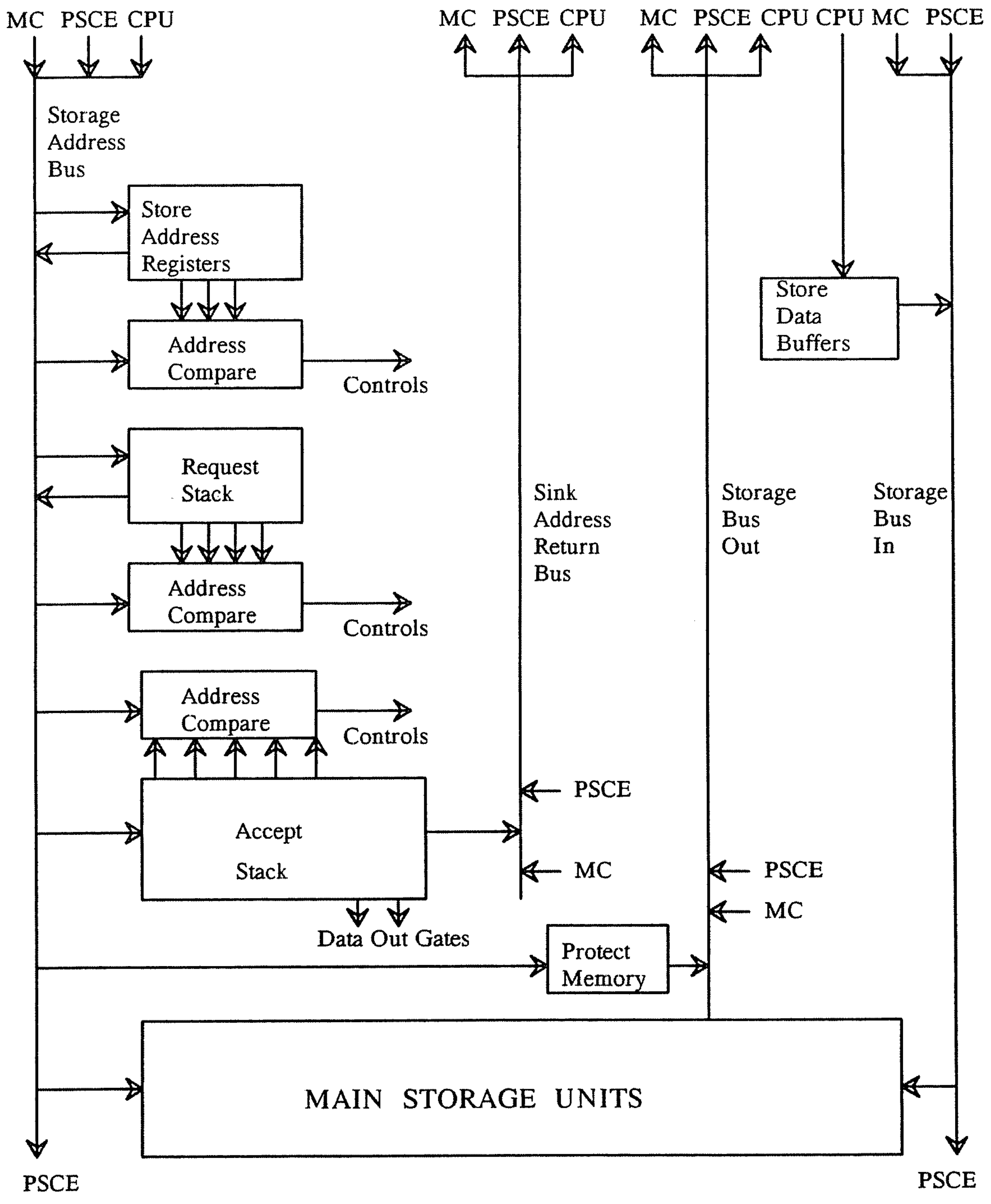


Figure 2.2 -- Block Diagram of MSCE Organization from [BoGr67].

data words from all areas of the processor pending availability of the memory module.

A memory request to the MSCE consists of an address, a return or sink address (to route the returning data), control bits (to define the operation precisely), and the data for store operations. When neither memory conflicts nor access hazards occur (and store data is available for store operation), the memory request that is transmitted through the storage address bus, can be gated into the accept stack and serviced immediately. However, when there are memory conflicts and/or access hazards, the memory request will be stored in the request stack or the store address registers depending on whether or not it is a fetch or a store request. For a store request, the store data is submitted by either the I-unit or the E-unit, while the store address is generated by the I-unit. Since the I-unit always decodes and sets up the execution of instructions ahead of the E-unit along the pipeline stages, the store address of the store operation will arrive at the MSCE ahead of the store data if the store data is the result of arithmetic operation produced by the E-unit. In this case, the store address is placed into the store address register waiting for the store data to arrive. There may also be conflicting store requests that are waiting for the memory modules to become available. Thus, the MSCE has to check whether an access hazard exists for each new request.

During each cycle, the MSCE controls determine the source (the peripheral storage control element, the maintenance console, or the central processing unit) that is allowed to make a request. The address of the request is then sent through the address bus. This address bus is also used to load the address of a store request into a store address register. The store address remains in the store address register until its data word is generated by the processor. A successful request is sent to the proper memory module, or to the PSCE if the extended main storage is requested. If a request is accessing main storage, it is also gated into the top of the accept stack with its control information. Any rejected processor request is stored in a position of the request stack for later recycling.

The priority order for the requesting sources used by the MSCE to decide which should be selected is:

(1) PSCE to (main) storage.
(2) Maintenance console to storage.
(3) Request stack to (main) storage for Multi-Access.
(4) Store address register to storage.
(5) Request stack to storage.
(6) Processor to storage.

When a fetch address passes through the address bus, it is compared with addresses in the store address registers, the request stack and the accept stack. A match with a store address register forces rejection of the request and the request is stored in the request stack. This is because the acceptance of the request would have caused an out-of-sequence fetch. A Read-After-Write access hazard is therefore detected and resolved. A match with an address in the accept stack implies that the desired word is being fetched by a previous request, and can be obtained when the previous request is completed without selecting a memory module or waiting for the memory module to become free. This is the *Multi-Access* scheme used to obtain data for a fetch following either a fetch or a store to the same memory address.

When a store address is sent to the MSCE, it is gated into a store address register and will wait for the store data to be available. Once the store address and the store data are available, the store request is gated to the address bus as soon as possible. The store address register is made available once again to the I-unit, which considers a store operation completed once it loads the store address register. If the accessed memory module is busy or the store address matches with an address in the request stack, (Write-After-Read or Write-After-Write hazard detected) the store request is gated into the request stack.

When a request address passes through the address bus, a match with an address in the request stack causes the request to be tagged for a future Multi-Access operation, and to be gated into another position of the request stack. The presence in the request stack of an outstanding request for a particular address causes the second request to be rejected, thus keeping the two in the proper sequence.

In summary, hardware interlocks in the form of address comparators (address bus vs. pending store address registers and pending requests in the request stack) are used to order stores to the same address, and to detect and re-order out-of-sequence store/fetch requests to the same address. Hence, memory access hazards in the IBM SYSTEM/360 Model 91 are detected and resolved.

The storage system of the IBM SYSTEM/360 Model 91 is a high performance memory system. However, its hazard detection and resolution scheme requires centralized interlock control and associative request buffers that can be too complex to be partitioned and implemented on VLSI chips.

2.3.2. Resolution of Access Hazards in the IBM SYSTEM/370 Model 168

The memory access hazard resolution scheme in the IBM SYSTEM/370 Model 168 [IBM76] is very different from the SYSTEM/360 Model 91. Instead of detecting and avoiding out-of-sequence (fetch/store) memory operations before the operations are started, the IBM SYSTEM/370 Model 168 detects the out-of-sequence fetch after the data is fetched from the storage system. The out-of-sequence fetch is ignored, and a fetch request has to be submitted again after the store is completed.

The IBM SYSTEM/370 Model 168 is a cache-based architecture. The high-speed cache permits the central processing unit to work with the 80-ns cache, rather than with a relative slower 4-way interleaved main storage system. Similar to the SYSTEM/360 Model 91, the major processing units of SYSTEM/370 Model 168 are: the I-unit, the E-unit, and the *Processor Storage Control Function* (PSCF). The I-unit fetches, decodes instructions, makes operand fetch requests and sets up instructions for execution by the E-unit. The PSCF controls requests for the processor storage from the I-unit, the E-unit, the maintenance console, and the I/O channels. As illustrated in Figure 2.3, the PSCF has these major sections: priority controls, translator and translation lookaside buffer (TLB), buffer (caches are called buffers in the IBM systems) invalidation address stack (BIAS), channel buffers, source/sink, and processor storage controls. Together, these units provide the following functions:

(1) Determine the priority of I-unit and E-unit requests.
(2) Determine the priority of the CPU and the channel for processor storage requests.

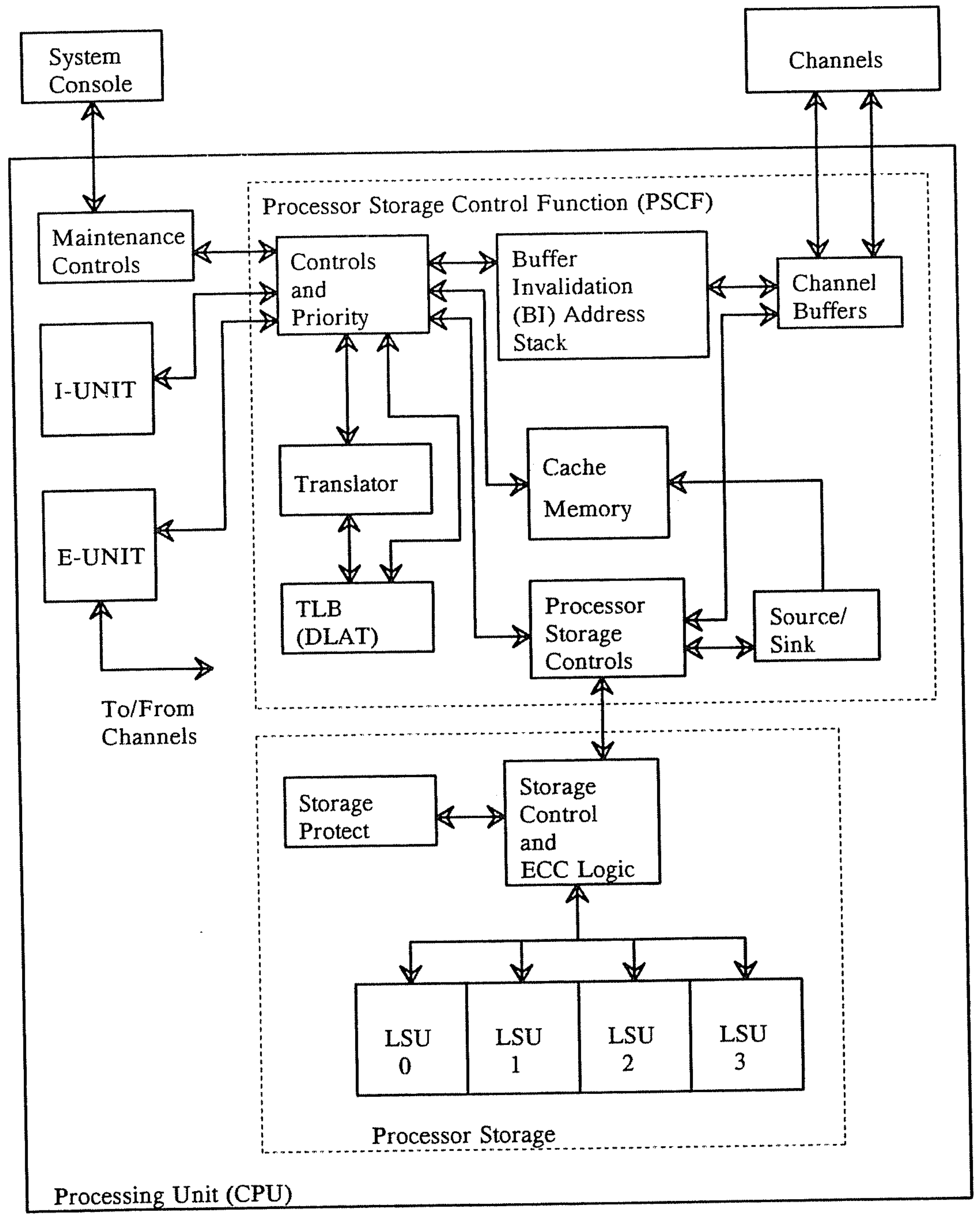


Figure 2.3 -- Block Diagram of the PSCF Organization of IBM SYSTEM/370 Model 168 from [IBM75].

(3) Translate logical addresses to physical addresses for CPU requests. Recently translated addresses are held in the TLB (also called the DLAT), so that the translation process can be accelerated.

(4) Buffer recently accessed instructions and data in the cache memory. The CPU gets its information from the cache memory about 90% of the time.

Only those areas related to memory access hazard will be discussed in the following paragraphs.

Four address registers are used by the I-unit to make memory requests via the Processor Address Register (PAR). These are Instruction Address A-register (IAA), Instruction Address B-register (IAB), Operand Address 1 Register (OA1), and Operand Address 2 Register (OA2). The PAR contains the address sent to the PSCF priority circuits for a storage operation. A word of storage, i.e. 64 bits, is obtained for each storage fetch request. IAA and IAB are used to fetch instructions. One of them fetches the main stream of instructions; the other, which is used for an execute or a branch instruction, fetches the target stream of instructions. OA1 and OA2 are used to fetch and/or store operands, and each of them has its own operand buffer for storing the operand from memory or the operand to be stored into memory. The OA1 and OA2 and their operand buffers can be controlled by either the I-unit or the E-unit. The storage addressing flow for a memory request from the CPU and from other request sources is illustrated in Figure 2.4 with the request lines that feed the PSCF priority resolvers arranged from top to bottom in that priority order. For the requests from the CPU, the store requests are given higher priority than the fetch requests, and the priority of operand fetch is higher than instruction fetch. Table 2.1 lists the priority assignment of requesting sources to the PSCF.

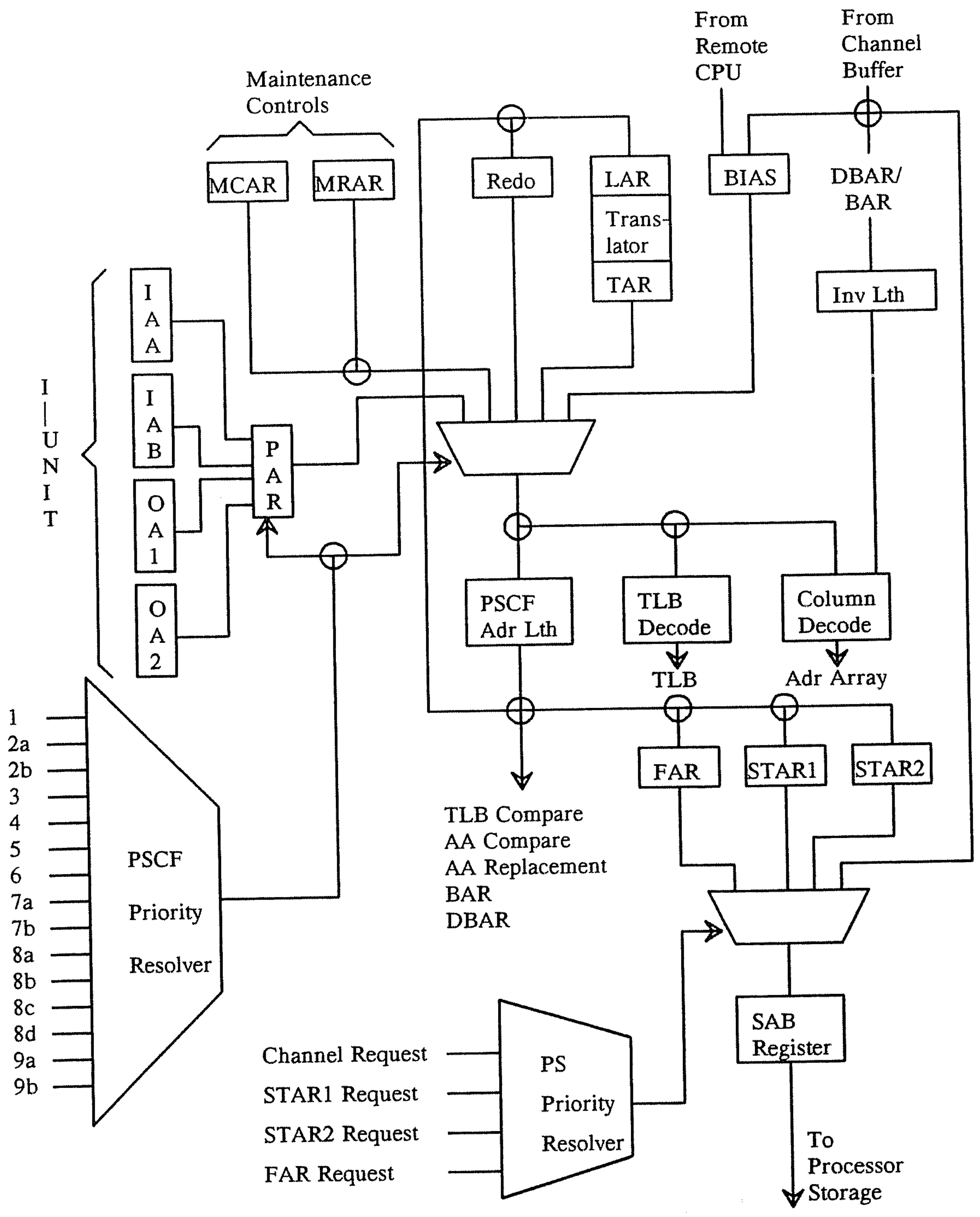


Figure 2.4 -- Storage Addressing Flow within the PSCF of the IBM SYSTEM/370 Model 168 from [IBM75].

Priority Assignment of Storage Requests in IBM SYSTEM/370 Model 168	
Priority	Function
1	Prefetch Request for Cache Memory
2a	Invalidate Latch Request
2b	Buffer Invalidation Address Stack (BIAS) Request
3	Buffer Add/Delete
4	Translation Lookaside Buffer (TLB) Update
5	Translator Request
6	Redo Request
7a	CPU Store Request from OA1 Operand Address Buffer
7b	CPU Store Request from OA2 Operand Address Buffer
8a	CPU Fetch Request from OA2 Operand Address Buffer
8b	CPU Fetch Request from OA1 Operand Address Buffer
8c	CPU Fetch Request from IAB Instruction Address Buffer
8d	CPU Fetch Request from IAA Instruction Address Buffer
9a	Maintenance Console Store Request
9b	Maintenance Console Fetch Request

Table 2.1

Fetch requests are issued from the I-unit and store requests are issued from the E-unit (in the IBM SYSTEM/360 Model 91, a store datum can be issued from either the I-unit or the E-unit depending on the instruction format,) with the I-unit handling the interlocking between requests. The operand address register, being used as the destination register of a store operation, becomes busy when an address is loaded by the I-unit. This address register remains busy until the instruction that issues the store operation is completed and the store request is accepted by the PSCF. Once a store request is made, the E-unit cannot initiate another store request until it is signaled by the PSCF. Since there are only two operand address registers (OA1 and OA2), at most two store addresses can exist at the same time. Instructions are released to and executed by the E-unit according to the order presented in the program; thus out-of-sequence stores never occur and *Write-After-Write* hazards are precluded.

When a fetch instruction follows a store instruction, the fetch request by the I-unit may precede the store request by the E-unit. To prevent old data from being used, the I-unit compares the addresses in the source/destination address registers (OA1 and OA2). If a match occurs, then an out-of-sequence fetch is detected. This out-of-sequence fetch is ignored and a fetch is re-submitted after the store is completed. Thus the *Read-After-Write* hazard is detected and resolved.

For the cache memory of the IBM SYSTEM 370 Model 168 uses write-through policy and blocks memory requests when a cache miss occurs. Thus, no access hazards can be introduced by cache operations.

The access hazard detection and resolution scheme of the IBM SYSTEM 370 Model 168 is less complex than that of the IBM SYSTEM 360 Model 91.

However, the scheme of the IBM SYSTEM 370 Model 168 is not suitable for a VLSI decoupled architecture for two reasons: (1) off-chip communication is required to re-submit the memory request that fetches a wrong operand from main memory, and (2) it can create ordering problem for storing the fetched operands correctly into the data queues.

2.3.3. Resolution of Access Hazards in CSPI MAP-200

The MAP-200 [CoSt81] is a peripheral array processor and a decoupled architecture. There are three major processing units in the MAP-200: the *Addresser* (APS), the *Arithmetic Processing Unit* (APU), and the *Memory Transfer Controller* (MTC). As illustrated in Figure 2.5, these three processing units are connected by the address and data queues: the *Read Address FIFO* (RAF), the *Write Address FIFO* (WAF), the *Input Queue* (IQ), and the *Output Queue* (OQ).

The APS computes the load and store addresses and places them into the RAF and the WAF, respectively. The fetched operands are stored into the IQ and used by the APU. When a computed result is to be stored into the memory, it is placed in the OQ by the APU. As soon as the MTC detects that there is a load address in the RAF and there is space in the IQ, a fetch request will be issued to the memory. Alternatively, a store address in the WAF and data in the OQ will cause the MTC to issue a memory store operation. Obviously, the instruction that will send a memory request to the RAF should be blocked when the RAF is full. The size of the RAF and the flow control of handling the RAF full condition is not described in [CoSt81].

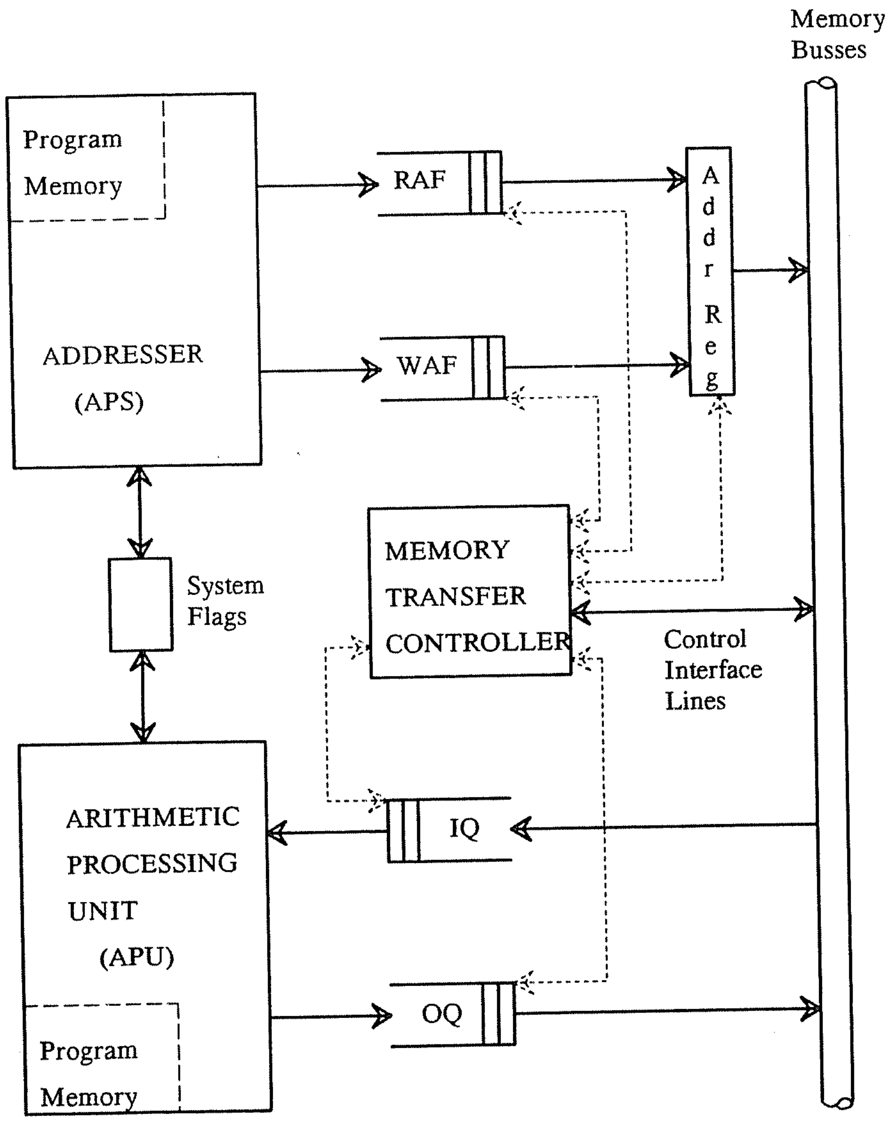


Figure 2.5 -- Block Diagram of MAP-200 Organization from [CoSt81].

The MAP-200 system is designed to assign fetch requests higher priority than the store requests, because fetches tend to be needed before stores. Cohler and Storer investigated the performance using this priority scheme over a wide range of algorithms. Their results did not produce a situation where giving a higher priority to the store request than the fetch request would have improved performance. However, they did find examples where, in giving output (writes) preference, throughput is improved.

Out-of-sequence memory operations will occur when a fetch request after a store request is made to the same memory location and the requests are waiting for service simultaneously. This is because the fetch request has been given priority over the store request. An access hazard detection and resolution scheme is required to prevent the memory requests from being serviced out-of-sequence. A software solution is chosen by the MAP-200 system to solve the problem.

For this purpose, the APS has an instruction that allows it to wait on the WAF to become empty. The APS can use this instruction to ensure that a fetch request will not be issued until the store request in the WAF is transferred to memory. Thus, whenever there is a potential access hazard, the APS will not place a fetch request into the RAF until the WAF is empty. However, since every store creates a potential access hazard, the compiler for the MAP-200 needs to be efficient in detecting access hazards; otherwise, the load and store operations will be serialized and the parallelism obtained through decoupled computation will be reduced.

2.4. Summary and Discussion

The review of previous research and existing memory systems led to the formulation of the following features to be considered for the proposed memory system for the decoupled architecture in this dissertation.

The first important feature is that the memory system must be interleaved, so that memory bandwidth can be increased to match the speed of processing units. Second, the use of conflict queue to buffer conflicting requests can make the effective bandwidth of the interleaved memory system to be a *linear* rather than a square root function of the interleaving factor. However, the use of conflict queues creates a sequencing problem when the order of data fetched from memory is not the same as the order of its original arrival (which, of course, is precisely when the conflict queue is used). Thus, a bookkeeping scheme must be used to correct this problem. On the other hand, if the memory request are serviced in the arrival order, the operands can be fetched and stored in data queue in the correct order without re-ordering. In addition, access hazard detection is not necessary. Thus, the control logic for servicing requests in the arrival order can be very simple. The performance difference of servicing request in order and out of order will be investigated in this research.

Third, multiple-words-returned-per-fetch techniques can be used to match the memory bandwidth with the processor speed. The only shortcoming of multiple-words-returned-per-fetch scheme is that data can be fetched but not used. Since the needed data is the only data allowed in the operand queue, the fetched data must be screened so that only the needed data is placed in the operand queue (an exception is made for an instruction fetch request, where a complete cache line is fetched).

Memory access hazards must be checked in order to prevent out-of-sequence fetch/store to the same memory location from occurring. Categorization of the techniques to detect and resolve memory access hazards in existing memory systems are as following:

(1) **Prevention**: memory requests are scheduled in a way that fetch and store requests do not exist simultaneously; thus, access hazards will never occur. The prevention scheme used in the decoupled architecture, MAP-200, results in a memory model where no conflict queue is used. This memory model is similar to Hellerman's model, whose effective memory bandwidth approximates the square root of the interleaving factor. The prevention scheme implemented in the Stunt Box [Thorn70] of the CDC 6600 actually uses a conflict queue of size 4 to buffer *only* either fetch or store requests at any moment. No mixed fetch and store requests can exist in the conflict queue simultaneously. This prevention scheme is the underlying scheme in the First-Come-First-Serve scheduling policy. This scheduling policy will be studied in detail.

(2) **Detection and prevention**: memory requests are serviced as soon as possible, but access hazards, if detected, must be resolved before a request is serviced. This scheme is used in the storage system of the IBM SYSTEM/360 Model 91, and can be applied to memory system for decoupled architecture. It will therefore be further investigated in this dissertation.

(3) **Detection and abortion**: memory requests are serviced as soon as possible and the address of the fetched data must be compared with the previous

pending store request. If a match is found, the fetched data is discarded and a fetch request must be submitted after that store request is completed. This scheme is used in the storage system of IBM SYSTEM/370 Model 168, and can create a complex problem which is the correction of the order of the fetched data within the queue. Thus, it is not further investigated.

Pipelining [RaLi77, ClLP81] has proven to be very effective in improving throughput with a small increase in hardware complexity. During a decoupled computation, the Access Processor can continuously send memory requests to the memory system. It is conceivable that the pipeline stages in the memory system are full most of the time. The interlock controls for the pipelined memory system can be implemented without much complexity as compared to the interlock controls of the pipelined processing unit. The interlock control of pipelined memory system only needs to compare memory addresses for checking the availability of a memory module and detecting memory access hazards. Once the memory request moves to a pipeline stage, it can be processed there and wait for moving to next stage. There are no situations that require the pipeline to be flushed and the operations in progress must be either restarted or discarded. While, it is much more complex to design a pipelined processing unit that can avoid the necessity of flushing the operations on the pipeline stages, e.g., the instructions being executed are wrong due to an incorrect branch decision made earlier. Thus, pipelining techniques for improving the performance of the memory system also will be used in this research.

In the following chapter, the memory reference pattern of decoupled computation is analyzed. Next, investigation of efficient request scheduling policies, the organization of the memory system pipeline, and the structure of request buffers between the pipeline stages will be studied in preparation for the memory system design. The impact of memory access hazards on performance and alternative access hazard detection/resolution schemes will be measured through simulations. Finally, a new organization of data cache combined with a memory access hazard detection scheme will be described, and its performance will be evaluated.

CHAPTER 3

An Analysis of Memory Reference Characteristics in Decoupled Computations

Understanding of the memory reference characteristics in decoupled computation is the key to designing a practical memory system that facilitates the flow of memory requests in a decoupled architecture. In this chapter, the first thirteen loops of the Lawrence Livermore Laboratories benchmark program [McMa72, RiSc84], are used to study memory reference patterns and their influence in decoupled computation. From an analysis of decoupled computation, the desirable features and the request scheduling policies for the memory system will be proposed and evaluated in chapter 5 and 6.

Although the memory system designs are intended for the PIPE architecture, the discussion is applicable to any architecture where hardware queues are used at the processor-memory interface. Since the PIPE architecture is implemented in the VLSI environment, the constraints of a VLSI system, such as the off-chip communication and the partitioning problems, will be considered.

3.1. Request Types and Potential Deadlock in Request Scheduling

The Memory Controller of the PIPE architecture can receive six different types of memory requests: (1) instruction fetch (IFa) request for the Access Processor, (2) instruction fetch (IFe) request for the Execute Processor, (3) load address (LA) request for fetching operands to the Load Data Queue of the Access Processor, (4) alternative load address (ALA) request for fetching operands to the Load Data Queue of the Execute Processor, (5) store address (SA) request for storing data generated by the Access Processor, and (6) alternative store address (ASA) request for storing data generated by the Execute Processor. In addition, there are store data (SD) from the Access and the Execute processors which are to be paired with SA and ASA requests, respectively. The IF request provides the beginning address of an instruction cache line, and the LA and SA request provide the address of a memory word being accessed. The Memory Controller must fetch four instruction words from four different memory modules for each IF request.

Except for the instruction fetch requests, there is a *correct* order of servicing requests defined by the order of arrival. The operands fetched for the LA requests must be delivered to the Load Data Queue of the Access Processor in the arrival order of LA requests. Similarly, the operands fetched for the ALA requests must be delivered according to the arrival order of ALA requests. The pairing of store addresses and store data must also be done according to their arrival order. For example, the n-th arrival of an ASA request must be paired with the n-th arrival of SD from the Execute Processor. If there are load and store requests to the same memory location, the correct order of servicing these load and store requests is the arrival order of these load and store requests. A store operation occurs logically at the time the store address arrives.

The Memory Controller can either store all six different types of requests in a single request queue and service the requests in First-Come-First-Serve order, or store them in different request queues according to their request types and service the requests according to a priority of requests. When the load and store requests are placed in separate queues, it will introduce memory access hazards. Thus, it requires an access hazard detection and resolution scheme. There are different ways of storing requests in separate queues, and different ways of assigning priorities to request types. For example, the load requests can be stored in one queue and the store requests stored in another queue. Then, the Memory Controller can service requests either using the Read-Request-First scheduling policy which gives the load request a higher priority than the store request, or using the Write-Request-First scheduling policy which gives the store request a higher priority than the store request. Methods of dividing requests into separate queues and assigning a priority to each request type are discussed in detail in chapter 4 where strategies of request scheduling are discussed. In the following paragraphs, the potential deadlocks that can occur due to the organization of request queues, and the minimum number of request queues required in order to avoid deadlock are discussed.

There are two types of potential deadlocks if requests are serviced in the First-Come-First-Serve order: (1) permanent blocking on LA requests, and (2) permanent blocking on IF requests. Both types of blocking are caused by a SA request waiting for a store datum that will never arrive.

The first type of deadlock is caused by an incorrect arrival sequence of the LA requests and the SA requests. For example, in the process of computing A = B + C, the Memory Controller has to complete two read operations and one write operation for storing the sum of B and C into A. The SA request for A can be sent to the Memory Controller in three different orders: *Case a*: before the LA requests for B and C, *Case b*: after the LA requests for B and C, or *Case c*: between the LA requests for B and C. The Memory Controller should deliver the values of B and C to the Load Data Queue, and wait for the store datum of the SA request of A to arrive. Thus, the order of request arrival of *Case a* and *c* will cause a deadlock if there is only one request queue to store both the LA and the SA requests. This type of deadlock can be alleviated through two methods: (a) The compiler generates the access code for the Access Processor such that the processor submits the LA and the SA requests as in *Case b*. This method can be implemented easily by the compiler; however, it can severely limit the amount of codes that the compiler could have scheduled to improve the speedup of decoupled computation. (b) There are two separate queues for storing the LA and the SA requests, respectively; each queue must have associative search capability in order to detect memory access hazards -- Read-After-Write and Write-After-Read hazards.

The second type of deadlock may occur when the IF and the LA requests are stored in the same request queue. For example, in the decoupled computation for computing these two arithmetic statements A = B + C and D = A + E, the Access Processor has the access code to submit requests. The order of such requests is: ALA request of B, ALA request of C, ASA request of A, ALA request of A, ALA request of E, and ASA request of D. However, the Execute Processor does not have the execute code to compute B + C and A + E. The Execute Processor, therefore, submits an IFe request to fetch the needed execute code from main memory. If this IFe request arrives at the Memory Controller later than the ALA request of A, then this IFe request cannot be serviced until the ALA request of A is serviced. Thus a deadlock has occurred, because a Read-After-Write hazard exists between the ASA request of A and the ALA request of A. Since it is impossible to predict the occurrence of instruction cache misses, and since the IF and LA requests need not be serviced according to the arrival order of IF and LA requests, a simple solution is to use separate queues for the IF requests and the LA requests. If self-modifying codes are allowed, the Memory Controller must detect access hazards between the IF and the SA/ASA requests. In addition, the existence of self-modifying codes requires the processor to check whether or not a store request is updating an instruction which is in the instruction cache. For example, the I-unit must check whether or not a store request is updating an instruction in the instruction array buffer of the IBM SYSTEM 360 Model 91 [AnST67, BoGr67]. Since there is an instruction cache on each PIPE processor and the store address requests are issued from the Access Processor, it can be very complex to design an instruction cache that allows self-modifying code to exist. Thus, self-modifying codes are not considered in this research, and access hazard detection for the IF requests will not be considered in the memory system designs discussed in this dissertation.

In summary, the memory requests must be organized into at least four request queues: (1) one IF Request Queue for storing IF requests from both

Access and Execute processors, (2) one LA/SA Request Queue for storing load addresses of LA and ALA requests and store addresses of SA and ASA requests (assuming that requests are received from the Access Processor in correct sequence, so that a deadlock will not occur), and (3) & (4) two Store Data (SD) Queues for storing store data from the Access and the Execute processors, respectively. Further dividing the LA and SA requests into separate queues gives the compiler much greater flexibility in optimizing memory accesses.

3.2. Some Observed Characteristics and Their Implications

During decoupled computation, two instruction streams are executed by the Access and the Execute processors simultaneously. If both the Access and the Execute processors were to make their memory requests independently of each other, it is likely that two memory requests would arrive at the Memory Controller at the same time. If there are some variables that are shared by both processors, continual interaction between processors through the memory system or through some other means will occur. There are three characteristics that have been observed during trace-driven simulations. In the following sections, each of these characteristics is described and its implications on the memory system designs and trade-offs are discussed.

3.2.1. Simultaneous Memory Input Arrivals

Two memory inputs -- request addresses or store data -- one from each processor can arrive in the same clock period. This may occur frequently if the instruction cache on the Execute Processor has a low hit ratio. Since off-chip communications are expensive in a VLSI system, the Memory Controller should accept whatever information is received, i.e., retransmission should never be necessary for the memory transactions (except perhaps because of transmission errors).

3.2.1.1. Problem With More Than One Arrival Per Clock

In order to receive memory inputs from both processors simultaneously, the Memory Controller must have two input buses. Each input bus is dedicated to receive memory inputs from a processor. The Memory Controller must decode the received memory inputs, and decide which request queues should store the memory inputs. If both requests are instruction fetches, these two memory inputs are for the same request queue. Thus, the hardware queue must be capable of receiving two entities in the same clock period. Since it is very difficult to design a hardware queue which is capable of accepting two entities in the same clock period, an alternative is to process one of the two received memory inputs, place the other one in a buffer, and service the latter in the subsequent clock periods. However, two more requests may arrive on the following clock period. Thus, the control logic for the Memory Controller to simultaneously receive two memory inputs must be very complex.

3.2.1.2. Possible Solutions

If the memory input arrival rate is restricted to one per clock, a multiplexing scheme can be used. In this approach, the required input pins for the Memory Controller can be reduced by half. In addition, only one data path is required to route the new arrivals, which cannot be processed immediately, to request queues. Thus, restricting arrivals to one per clock can greatly simplify the input logic of the Memory Controller.

Four multiplexing schemes are proposed and evaluated through trace-driven simulations: (a) time-division multiplexing, (b) AP-First multiplexing, (c) EP-First multiplexing, and (d) round-robin multiplexing.

In the time-division multiplexing scheme, clock cycles are divided into two groups of time slots, even and odd cycles. The Access Processor can only send a memory transaction out during an even clock cycle, whereas the Execute Processor can issue a memory transaction only during an odd clock cycle. In this scheme, the peak bandwidth for each processor is reduced by 50%. Each processor can check its clock cycle, and decide whether or not it can send a memory transaction out. Off-chip communication is not needed for the implementation of time-division multiplexing scheme.

A static priority scheme, which assigns a fixed priority to each processor, is used to decide the use of the memory input bus for the AP-First or EP-First multiplexing schemes. The arbitration process can be overlapped with the transmission of the previous memory transaction. Thus, there is no bandwidth penalty in this scheme, and no time delay for the lower priority processor, except when conflicts occur. Because it requires one clock to send a signal from one processor to the other processor, and one clock for a processor to decode the received signal, the higher-priority processor must send an arbitration signal to the other processor two clocks before it sends a memory transaction to the memory system. Thus, the allocation of the memory input bus is determined the clock before it is used. For the AP-First multiplexing, the Execute Processor can issue a memory transaction only when the Access Processor is not sending memory transactions. The Access Processor has to signal the Execute Processor, in each clock, as to whether or not a memory transaction is to be sent two clocks following the current clock cycle. Similarly, in the EP-First multiplexing scheme, the Access Processor can issue a memory transactions only when the Execute Processor is not sending memory transactions. The Execute Processor also has to signal the Access Processor in each clock about the use of memory input bus. For both multiplexing schemes, the high-priority processor can request the use of the memory input bus only if the Memory Controller has buffers to receive the memory transactions.

In the round-robin multiplexing scheme, the assignment of the memory input bus is determined by the memory transaction which last used the memory input bus. If both processors were to send memory transactions out during the same clock cycle, the processor which last used the memory input bus has to surrender its privilege. As in the static priority multiplexing scheme, the arbitration process overlaps the transmission of the previous memory transaction in order to avoid peak bandwidth reduction. In each clock, each processor has to send an arbitration signal to the other processor as to whether or not it intends to use the memory input bus two clock periods hence. In the case of a conflict, the loser uses the third clock period. It requires one clock to send an arbitration signal, one clock to transmit the signal from one end to the other end, and another clock for the receiving end to decode the signal. Each processor requires a finite state machine (FSM) to track: (1) the last use of the memory input bus and (2) the arbitration signals sent in the last two clocks, so that the processor can determine whether or not it can use the memory input bus. Because the FSMs on both processors are always in the same state, and a processor knows whether or not the other processor intends to use the memory input bus in the next clock cycle, only one processor will decide to send out a memory transaction. Thus, no bus contention will ever occur.

The performance of these four input multiplexing schemes, and the performance degradation due to the use of input multiplexing will be evaluated through trace-driven simulations. Several memory models with different request scheduling policies are proposed and discussed in chapter 4. From the proposed memory models, the two best memory models (evaluated from the simulation results in chapter 5) will be chosen for evaluating the performance of input multiplexing schemes. The simulation models and the techniques of performance measurements are described in details in chapter 5.

3.2.1.3. Simulation Results

A memory model with First-Come-First-Serve/Read-Request-First (FCFS/RRF) scheduling policy and a memory model with Free-Module-Request-First/Read-Request-First (FMRF/RRF) scheduling policy were used. The possible time delay of sending a bus reservation signal was not taken into account during the simulations of the static priority multiplexing and the round-robin multiplexing schemes. For the simulations of computing the first twelve LLL loops, the number of occurrences and the percentage of total simulation time that both the Access and Execute processors simultaneously send memory inputs to the Memory Controller are summarized in Table 3.1. The performance degradation due to the use of input multiplexing schemes for the FCFS/RRF and FMRF/RRF memory models are summarized in Table 3.2. The workload of memory input arrivals and simulation results for each of the first twelve LLL loops are given in Appendix V. The workload ranges from 0.0% to 64.1% of the time that there are two memory inputs simultaneously arriving at the Memory Controller.

The results from the two simulations of the memory models using the FCFS/RRF scheduling policy and the FMRF/RRF scheduling policy were similar. Both results show that the best-to-worst ranking of the multiplexing schemes were (1) the round-robin and EP-First schemes, (2) the AP-First scheme and (3) the time-division scheme.

The performance of the round-robin and EP-First multiplexing schemes were similar and the performance degradation due to the use of these two multiplexing schemes was insignificant. In addition, their results were nearly as good as and sometimes better than the simulation model when no multiplexing scheme was used. The simulation results of the 8-way interleaved memory model using FMRF/RRF scheduling policy show that the use of round-robin or EP-First multiplexing actually completed the simulation in a shorter time than when no multiplexing scheme was used. This is because the store address and the store data were both available earlier when no multiplexing was used than when multiplexing schemes were used. During LLL loop 10 simulations in the 8-way interleaved model, there were store requests at the end of current loop iteration and load requests at the beginning of next loop iteration accessing the same memory module. When no multiplexing scheme was used, the store data for the store requests of current loop iteration arrived at the memory system before the load

Simulation Results of Simultaneous Input Arrivals

Scheduling Policy	Interleaving Factor	Simulation Time	Number of Occurrence	Percentage of Simulation Time
FMRF/RRF	4	73,950	2,755	3.7%
	8	54,050	4,583	8.5%
	16	52,182	5,727	11.0%
	32	52,024	5,624	10.8%
	64	51,989	5,648	10.9%
FCFS/RRF	4	109,633	1,312	1.2%
	8	88,552	3,367	3.8%
	16	55,356	5,522	10.0%
	32	52,220	5,694	10.9%
	64	52,038	5,664	10.9%

Table 3.1 -- The number of occurrence and the percentage of total simulation time that both the Access and Execute processors simultaneously send memory inputs to the Memory Controller.

Performance Degradation of Memory Models						
Scheduling Policy	Multiplexing Scheme	Interleaving Factor				
		4	8	16	32	64
FMRF/RRF	Time-Division	0.4%	9.2%	12.4%	12.6%	12.6%
	AP-First	0.9%	1.6%	3.0%	3.4%	3.4%
	EP-First	0.04%	-0.4%	0.8%	0.6%	0.6%
	Round-Robin	0.3%	-0.5%	0.4%	0.4%	0.4%
FCFS/RRF	Time-Division	0.5%	0.9%	9.2%	12.3%	12.6%
	AP-First	0.5%	0.0%	5.5%	2.6%	2.7%
	EP-First	0.2%	0.05%	0.5%	0.7%	0.5%
	Round-Robin	0.5%	0.0%	0.4%	0.5%	0.4%

Table 3.2 -- Comparison of performance degradation due to the restriction of receiving one request per clock period.

time in the EP-First multiplexing simulations. This was because the Execute Processor generated the store data at the maximum rate of one every two clocks (assumed operands are provided before being used); thus, there were time slots for the Access Processor to use the memory input bus.

3.2.1.4. Conclusions

The processor-to-memory bandwidth is potentially reduced by a maximum of 50% for each processor in the time-division multiplexing scheme. However, the performance degradation in the memory system is far less than this because the processors do not send requests to the memory system at the rate of one per clock. Thus, if the processor-to-memory bus is efficiently utilized (as in the round-robin or EP-First multiplexing schemes), the performance degradation due to the use of input multiplexing scheme can be insignificant.

Table 3.2 indicates that, in the time-division multiplexing scheme, the performance degradation increases as the interleaving factor increases. This is because the throughput of a memory system increases as the interleaving factor increases. When the throughput increases, the memory wait time for processors decreases and the total processing time of decoupled computation decreases. Consequently, the average request arrival rate increases. Therefore, it is important to have an efficient multiplexing scheme when the throughput of the memory system is high. Conversely, the multiplexing scheme will have negligible impact on the memory system when the throughput is low.

During decoupled computation, the Access Processor must issue ALA requests for the Execute Processor, and issue an ASA request for each store requests of next loop iteration arrived; thus, the store operations of current loop iteration were started before the fetch operations of next loop iteration. While, when the round-robin or the EP-First multiplexing were used, the store data for the store requests of current loop iteration arrived at the memory system later than the load requests of next loop iteration; thus, the fetch operations of next loop iteration were started before the store operations of current loop iteration. The operands for the Execute Processor were delivered earlier when the round-robin or the EP-First multiplexing was used than when no multiplexing was used. Thus, the total simulation time for LLL loop 10 in 8-way interleaved model was longer for no multiplexing than for the round-robin or EP-First multiplexing (see Appendix V).

The simulation results in Appendix V indicate that the EP-First multiplexing was better than the AP-First multiplexing in the LLL loop 10 simulations, and the AP-First multiplexing was slightly better than the EP-First multiplexing for the rest of LLL loops simulations. This discrepancy was due to the starvation that occurred in the Execute Processor if AP-First multiplexing was used in LLL loop 10 simulations. In each iteration of LLL loop 10, the Access Processor sent ten ALA and ten ASA requests to the memory system, and the Execute Processor sent ten store data to the memory system. Because the Access Processor sent requests to the memory system at a rate close to one per clock, it occupied the memory input bus most of the time. The Execute Processor was not given enough time slots to send store data out, and filled its output queue. Thus, the Execute Processor was blocked frequently in the AP-First multiplexing simulations. Conversely, the Execute Processor did not occupy the memory input bus most of the datum generated by the Execute Processor. The number of memory transactions issued from the Access Processor is always greater than that from the Execute Processor. The Execute Processor must wait for the operands to arrive before generating store data, and it will never generate store data at the rate of one per clock for a sustained period. Thus, starvation will not occur in the Access Processor in the EP-First multiplexing scheme. Because the Execute Processor does not generate store data at the rate of one per clock, and because the round-robin multiplexing scheme gives equal opportunity to both processors, the performance of the EP-First and the round-robin multiplexing schemes can be the same. The decision of selecting one of these two multiplexing schemes will be determined by the implementation of a PIPE processor. For the round-robin multiplexing scheme, both the Access and Execute processors must have the same finite state machine for bus arbitration. While, the bus arbitration logic for the Access and Execute processors will not be the same for the EP-First multiplexing scheme. The control logic for the round-robin multiplexing scheme is more complex than the EP-First multiplexing scheme. The possible delay for sending reservation signal can occur to both processors for the round-robin multiplexing scheme, but it will occur only to the Execute Processor for the EP-First multiplexing scheme. The selection of a multiplexing scheme must trade off these criteria and realize the best solution in the design of a decoupled architecture.

3.2.2. Flow Control Problems

During decoupled computation, the Access Processor continuously issues alternative load address requests for the Execute Processor, while the Execute

Processor accepts the fetched operands and performs computations on the operands. If the Execute Processor cannot process the operands as fast as they arrive, the Load Data Queue in the Execute Processor will fill. Similarly, if the memory system cannot handle the arriving stream of requests efficiently, the Memory Controller has to prevent the Access Processor from sending further requests. Therefore, a flow control protocol is indispensable in regulating the transactions between the processors and the Memory Controller.

A Start-and-Stop flow control scheme is proposed, and it is used throughout the trace-driven simulations. In this scheme, a processor sends requests to the Memory Controller until a *Stop* signal is received from the Memory Controller, and it resumes sending requests when a *Start* signal is received. Similarly, the Memory Controller delivers fetched operands to the processors according to the control signals received from the destination processors. It requires three clock cycle to complete the three steps: (1) the Memory Controller detects a queue full condition and sends the *Stop* signal out, (2) the *Stop* signal passes through the memory-to-processor bus, and (3) the processor receives and decodes the control signal. Thus, the Memory Controller must reserve three request buffers for receiving the possible three requests sent by the processor during these three clock cycles.

In each clock cycle, each receiving end must send a *Start* or a *Stop* signal to the transmitting end depending on its current state. At the beginning of the flow control process, the receiving end must have at least four empty buffers. The receiving end sends a *Start* signal in each clock until an entity is received from the transmitting end. Then, the receiving end must send a *Stop* signal to the transmitting end. After a *Stop* signal is sent to the transmitting end, the receiving end must send a *Start* signal to the transmitting end if an entity is removed from the buffers. Otherwise, the receiving end must send a *Stop* signal to the transmitting end. Thus, for this Start-and-Stop flow control scheme, a Load Data Queue size of *at least four* in the processor and a request queue size of *at least four* in the Memory Controller are required to permit continuous operation.

There are two reasons to consider a queue of length greater than four: (1) continuous flow of requests/operands, and (2) sufficient length to allow non-deadlocking code scheduling. As the queue length increases, the transactions between the processors and the memory system can be transmitted smoothly without unnecessary holdup, and the Access Processor can send more load address requests to fetch operands before the operands are needed. Thus, the performance can increase as the length of queue increases. However, with a larger queue size, a longer clock cycle (or more clock cycles) will be needed to complete a queue remove/insert operation. In addition, the performance gain will not increase as the length of queue grows beyond a certain length. Thus, it is important to optimize the size of Load Data Queue in the processor and the number of request buffers in the Memory Controller.

Trace-driven simulations [HsPG84, GoYo85] show that the performance gained through the use of a longer queue in decoupled computation levels off when the size of the Load Data Queue is about four. Simulations on 4-way and 16-way interleaved memory systems with memory request queue sizes of 4, 8, 16 and 32 were also performed in this study. The results indicate that, except for LLL loop 10 simulations, the performance difference for request queue sizes of 8, 16 and 32 is insignificant. (For the LLL loop 10, twenty load address and ten store address requests are issued per loop iteration, and there are memory conflicts among these requests. The memory request queue can be filled up easily if the request queue size is less than or equal to 16. Therefore, the longer the request queues in the memory system, the better performance the memory system.) Thus, the queue size for the Load Data Queue on each processor does not need to be larger than four, and the request queue size on the Memory Controller does not need to be larger than four for most application programs.

3.2.3. Shared Variables

There are variables that are used for algorithmic computations and also used for address generation and/or branch decision evaluation in some programs. These variables, called *shared variables*, will be shared by the Access and the Execute processors during decoupled computations. The occurrence of shared variables depends very much on the programming style and the construction of the compiler. For example, when the second LLL loop is coded in PASCAL (see Appendix I), the loop counter increment statement (K := K + 5;) is within the *while*-loop body of LLL2. If the compiler considers this increment statement as an algorithmic statement and makes the Execute Processor compute this increment statement, then the loop index K becomes a shared variable. If the *while*-loop of LLL2 is coded in a *do*-loop in FORTRAN, the increment of the loop counter will become part of the *do* statement, and the compiler will not make the Execute Processor compute the increment of the loop counter any more (since the increment of the loop counter is only related to branch evaluation). Thus, the loop index K will not be a shared variable. Alternatively, if the PASCAL compiler can recognize that the loop counter K is only used for address generation and branch decision evaluation in LLL2, it will not make the Execute Processor compute the increment of the loop counter either. Then, the loop index K will not be a shared variable.

There are programs that cannot avoid having shared variables whenever the programs are decoupled. For example, the array elements *p[1,ip]* and *p[2,ip]* of the 13th LLL loop (see LLL13 in Appendix I) are shared variables. *P[1,ip]* and *p[2,ip]* are used to compute array element addresses of arrays *b, c, y, z, e, f,* and *h,* and the values of *p[1,ip]* and *p[2,ip]* are recomputed from other array elements during each loop iteration. Thus, *p[1,ip]* and *p[2,ip]* are shared variables whenever LLL13 is decoupled. The loop unrolling and software pipelining techniques cannot prevent the *p[1,ip]* and *p[2,ip]* from becoming shared variables.

3.2.3.1. Problems Resulting from Shared Variables

When there are shared variables in a decoupled computation, the Access and the Execute processors may run in a tightly-coupled mode where each processor must wait for the other processor to compute current values of shared variables. Thus, the Access Processor cannot run ahead of the Execute Processor and provide operands for the Execute Processor in advance. If no efficient way to transfer the values of the shared variables between the Access and the Execute processors is available, the likelihood of a realizable speedup through decoupled computation will be limited. If the performance of decoupled computation is heavily degraded due to the shared variables, it may actually run faster in a single processor.

In a general decoupled architecture, the value of a shared variable can be transferred between processors by two methods: (a) from a processor to the other processor, or (b) first from a processor to memory, then to the other processor. A processor-to-processor bus is required to implement method (a). Because off-chip communication pins are limited in a VLSI system, method (a) requires that some pins of the processor-to-memory bus be assigned to the processor-to-processor bus if there are no extra pins available. In this case, the processor-memory bandwidth is reduced, and extra clock cycles are needed for a processor-memory transmission. If method (b) is used, as in the PIPE architecture, each transfer of the shared variable value between processors requires the shared variable value to be stored in memory and then fetched from memory. Both the store and fetch requests are initiated by the Access Processor; thus, a Read-After-Write access hazard can occur for each transfer of a shared variable value. A Read-After-Write hazard occurs only if the store and the fetch to the same memory location exist in the memory system simultaneously.

Because it is difficult to have a processor-to-processor bus for transferring data between processors in the VLSI-based decoupled architecture, the implementation of method (a) is not considered in this research. Solutions for the Read-After-Write access hazard resulting from method (b) are proposed and evaluated through trace-driven simulations. The solutions, trade-offs, and simulation results are presented in the following sections.

3.2.3.2. Solutions for the Shared Variable Problem

Because the Access Processor is responsible for making memory requests for data accesses, the store address and load address requests for each transfer of a shared variable value can be sent to the Memory Controller in consecutive clock cycles. Thus, Read-After-Write access hazards can occur frequently. There are two methods to reduce the number of memory accesses for shared variable references. These two methods are described in the next two sections.

3.2.3.2.1. Detecting Read-After-Write Hazard and Bypassing Store Data

When the Memory Controller detects the Read-After-Write hazard, the Memory Controller should bypass the store datum to the load address request, and combine the store and the fetch requests for a shared variable into a single memory store operation. Thus, the memory fetch cycle for the Read-After-Write request is avoided, and only one memory store operation is needed for a transfer of shared variable value between processors.

If the store operation for a shared variable is completed before the load address request for that shared variable arrives, there will be no Read-After-Write hazard. Obviously, the Memory Controller cannot bypass the store datum to the load address request. In this case, store and fetch operations are needed for the transfer of shared variable between processors.

3.2.3.2.2. Short-Circuiting Read-After-Write Request

In this method, a data cache or write buffers are used to saved the store datum of a store request in the anticipation that the following load requests will access the same memory location as the store request. Thus, the store datum can be passed to the load requests if the store datum is still in the data cache or write buffers. In this way, the memory fetch operation for a shared variable reference can be avoided. This technique is called *short-circuiting* the Read-After-Write request.

In chapter 6, a data cache with the ability to detect Read-After-Write access hazards and short-circuit Read-After-Write requests will be designed and incorporated into the Memory Controller. In the following sections, the impact of shared variables on the performance of decoupled computation is discussed. Trace-driven simulations for memory models with data cache or write buffers were used to evaluate the performance improvement of short-circuiting Read-After-Write requests for shared variables, and results will be presented.

3.2.3.3. Analysis of Delay Due to Read-After-Write Access Hazards

A fetch request is sent to the memory system to fetch information that will be needed soon. The purpose of a store request is to store information that will not be needed until a subsequent fetch request occurs. Thus, the Memory Controller should service the fetch request before the store request in order to shorten the total computation time. However, when a Read-After-Write or Write-After-Read access hazard occurs between a store and a fetch, these two requests must be serviced in the correct order. Otherwise, the fetch request will fetch a wrong datum from memory. The Memory Controller also must detect and resolve Write-After-Write access hazards.

During decoupled computations, the Access Processor continuously overlaps memory transactions in order to provide operands before they are needed. Thus, it is likely that more than one fetch request for the same Load Data Queue can exist simultaneously in the Memory Controller. Because operands are placed in the Load Data Queue in their request arrival order, a fetch request that has a Read-After-Write access hazard can delay the service of the fetch requests that arrive at the memory system later. If the Memory Controller services the non-blocking requests out of arrival order, then the operands fetched for the requests that arrive later than the Read-After-Write requests can be ready for delivery earlier than servicing fetch requests in the arrival order. Therefore, the delay of delivering fetched operands introduced by Read-After-Write hazards can be minimized, and the performance degradation due to Read-After-Write hazards can be reduced. This analysis is verified in the simulation results presented in the next section, where the FMRF/RRF memory model with 8 write buffers can service requests out of arrival order and perform better than the AP-First or EP-First memory models.

3.2.4. Simulation Results

Trace-driven simulations using the trace file generated from the decoupled code of LLL loop 13 were performed to study the impact of shared variables on the performance degradation and evaluate the two proposed solutions: (1) bypassing store data and (2) short-circuiting Read-After-Write requests. In the decoupled code of LLL loop 13, the current values of array elements were saved in registers for future use whenever it was possible. Thus, only one Read-After-

Write access hazard for p[1,ip] and p[2,ip], respectively, would occur in each loop iteration. Coincidentally, the load address of h[i2,j2] was the same as the store address of h[i2,j2] of the previous loop iteration. Therefore, 383 Read-After-Write access hazards occurred in the 128 loop iterations of each LLL13 trace-driven simulation.

Four memory models with different request scheduling were used to evaluate the proposed solutions for Read-After-Write access hazards:

(1) AP-First gives a higher priority to the load request than the store request. Within the load or the store requests, the request of the Access Processor is given a higher priority than the request of the Execute Processor.

(2) EP-First gives a higher priority to the load request than the store request. Within the load or the store requests, the request of the Execute Processor is given a higher priority than the request of the Access Processor.

(3) FMRF/RRF with 8 first-in-first-out write buffers per memory module. Each module has 8 write buffers to save the store address/data belonging to the module. The 8 write buffers in each module functions as a data cache that allocates cache entries for writes only and replaces cache entries with FIFO policy. Before servicing a load address request, the Memory Controller checks if the address of load address request matches any store address saved in the write buffer. The store datum is passed to the load address request if a match occurs; otherwise, the Memory Controller starts a read operation for the load address request.

(4) FMRF with a data cache per memory module. Each data cache is used to cache the memory requests accessing a memory module. A cache line consists of *M* cache words from *M* data caches, where *M* is the interleaving factor (the organization of data caches within the Memory Controller is described in details in chapter 6). Two different organizations of data cache sizes are used to obtain the simulation results presented in this section: (a) 128 cache words per module, and (b) a total of 128 cache words in the Memory Controller. For organization (a), the total number of cache words in the Memory Controller increases as the interleaving factor increases. For organization (b), the number of cache words per module decreases as the interleaving factor increases.

Trace-driven simulations for these four memory models were performed on interleaving factors from 1 to 64. Simulations for the memory models of AP-First and EP-First with infinite interleaving factor were also performed. When the interleaving factor was infinite, each memory location constituted a memory module; thus, memory conflicts never occurred, and only memory access hazards prevented the interleaved memory system from achieving its maximum throughput.

A *contention-free memory* was defined for the purpose of evaluating the impact of Read-After-Write access hazards on the performance degradation. In the contention-free memory, conflicts never occurred and access hazards were ignored, but it required the same amount of time (10 CPU cycles) as the AP-First and the EP-First memory models to complete a memory operation. Thus, the difference of simulation times between the contention-free memory and the AP-First (or the EP-First) memory model of infinite interleaving factor indicated the performance degradation due to Read-After-Write access hazards. The simulation results are shown in Tables 3.3 and and plotted in Figure 3.1.

For the same interleaving factor, the total simulation time for the AP-First model was shorter than the EP-First model. This result indicated that servicing the LA request for the Access Processor before the ALA request for the Execute Processor can let the address computation task proceed faster; thus, the total simulation time was shorter if the LA request is given a higher priority than the ALA request.

There were 383 Read-After-Write access hazards in each simulation. These 383 Read-After-Write requests constituted about 20% of the total number of load

Total Simulation Times of LLL13					
Interleaving Factor	Scheduling Policies				
	AP-First	EP-First	FMRF with Data Cache 128 Wds/Mod	FMRF with Data Cache 128 Wds Total	FMRF with Write Buffer 8 Buffers/Mod
1	29,139	30,536	22,805	22,805	25,988
2	22,586	23,162	18,907	20,135	19,580
4	21,057	21,601	18,126	19,577	18,247
8	20,316	20,840	13,577	19,505	17,547
16	18,681	19,197	8,701	20,642	15,810
32	18,477	19,009	6,863	20,796	14,971
64	18,405	18,925	6,371	21,463	13,825
Infinity	10,837	10,847	--	--	--
Total Simulation Time of LLL13 in the *Contention-Free Memory* = 9,440					

Table 3.3 -- Total simulation times of LLL13 simulations

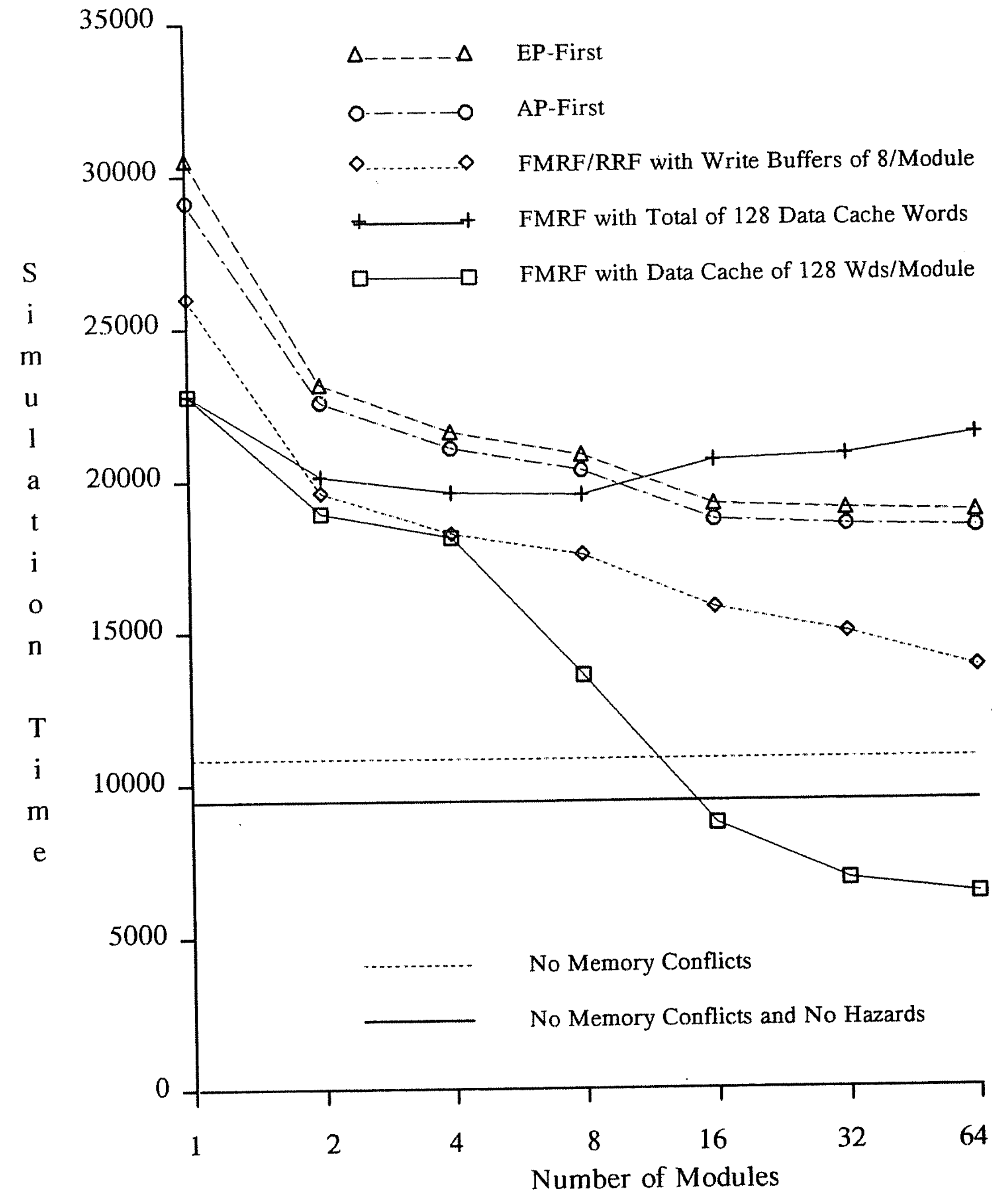


Figure 3.1 -- Performance comparison of memory models in handling Read-After-Write hazards in LLL13 simulations.

address requests. The store data were bypassed to all these 383 Read-After-Write requests in the FMRF/RRF model with 8 write buffers per module. The speedup of the FMRF/RRF model with write buffers over the AP-First model is shown in Table 3.4. The speedup was roughly constant for interleaving factors up to 16;

Interleaving Factor	Simulation Time		Speedup
	AP-First	FMRF with Write Buffer 8 Buffers/Mod	
1	29,139	25,988	1.12
2	22,586	19,580	1.15
4	21,057	18,247	1.15
8	20,316	17,547	1.16
16	18,681	15,810	1.18
32	18,477	14,971	1.23
64	18,405	13,825	1.33

Table 3.4 -- Speedup using write buffers to bypass Read-After-Write data.

however, the speedup increases when the interleaving factor was greater than 16. This indicated that access hazards prevent the AP-First (or EP-First) model from servicing memory requests even when the needed memory modules were free. Thus, the performance gain for the AP-First (or EP-First) leveled off when the interleaving factor was greater than 16. However, the performance gain still increased for the FMRF/RRF model with write buffers when the interleaving factor increased from 16 to 64.

The performance difference between the AP-First (or EP-First) model with infinite interleaving factor and the contention-free memory was about 15%. This indicated that the performance degradation due to Read-After-Write access hazard was about 15% for the LLL13 simulations.

The simulation results of the FMRF models with different data cache sizes illustrate that a data cache of suitable size can effectively short-circuit Read-After-Write requests. In the simulation model for FMRF with data cache, the Memory Controller fetched the entire cache line for each read requests. Thus, when the interleaving factor was 64, 64 words from different modules were fetched (if any cache misses) into data caches for each load address request. If the data cache size was small, the data stored in the data cache would be replaced frequently. Consequently, the Memory Controller used many memory cycle times to fetch data into data caches, but those fetched data may have been replaced before they were used. As illustrated in the simulation results, the performance of the memory model of FMRF with 128 cache words per module increased as the interleaving factor increased, and it was even better than the contention-free memory[1] when the interleaving factor was equal or greater than 16. However, the performance of the memory model of FMRF with total of 128 cache words decreased when the interleaving factor increased after 8 (hit ratios decreases as the interleaving factor increases).

In order to find a suitable size of cache in each Module Controller, simulations for the FMRF Memory Controller with different total cache sizes from 128 cache words to 2048 cache words were performed. The simulation results are shown in Table 3.5 and plotted in Figure 3.2. The relation of the hit ratio and the number of cache words per module are shown in Table 3.6. Figure 3.2 illustrates that the total simulation time of LLL13 decreased as the total number of cache words increased. For the cache sizes investigated, it took the least time when the interleaving factor was 8. Table 3.6 indicates that, for the same total cache size, the hit ratio was the highest when the interleaving factor was 8 for the total cache size that was equal to or greater than 512 words. The hit ratio was the highest when the interleaving factor was 4 for the total cache size that was 128 or 256 words. These results showed that an interleaving factor of 8 was the best configuration for the data cache organization used in the simulations. It also suggests that a larger cache size would do better for a larger interleaving factor, i.e., that 256 words per module is not enough. Further simulation results for the FMRF memory model with data cache will be presented in chapter 6.

[1]Note the contention-free memory always requires 10 CPU cycles to service a memory request, while the cache memory requires 3 CPU cycles on a hit.

Simulation Times of LLL13 in FMRF Memory Model with Data Cache					
Interleaving	Total Number of Data Cache Words				
Factor	128	256	512	1024	2048
1	22,805	21,765	21,737	19,678	14,618
2	20,135	18,907	18,853	15,665	9,134
4	19,577	18,238	18,126	13,985	7,790
8	19,505	18,105	17,877	13,577	7,180
16	20,642	19,669	19,197	14,798	8,701
32	20,796	20,296	19,470	14,990	8,721
64	21,463	21,243	19,713	15,651	11,255

Table 3.5 -- Total simulation times of LLL13 simulations with different total cache sizes from 128 to 2048 words in the FMRF Memory Controller.

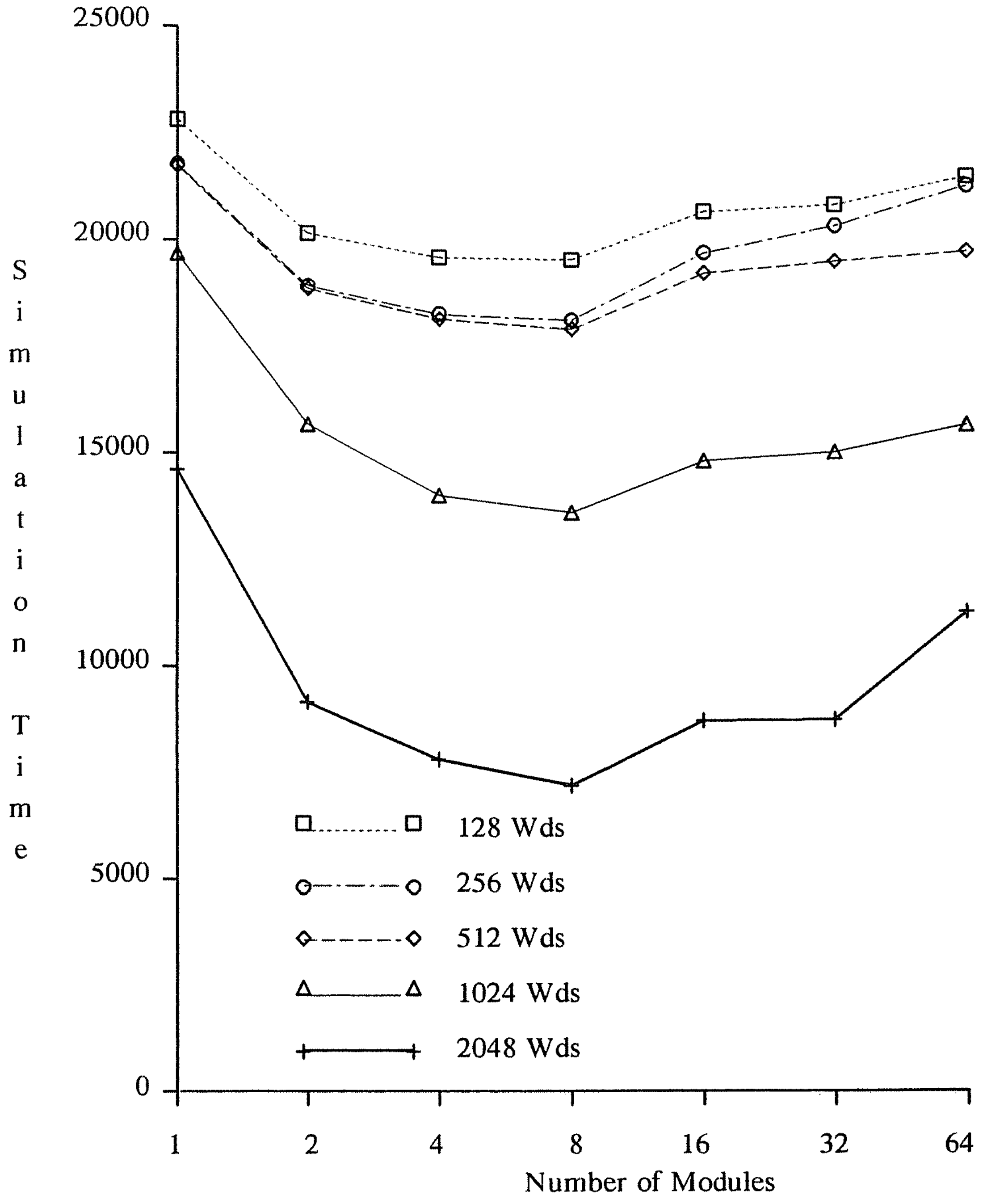


Figure 3.2 -- Total simulation times of LLL13 in FMRF memory modules with total cache sizes from 128 to 2048 words in the Memory Controller.

Simulation Results of LLL13										
Interleaving Factor	Hit Ratio and Number of Cache Words Per Module									
1	24.6%	128	37.0%	256	37.2%	512	59.7%	1024	79.5%	2048
2	27.3%	64	39.8%	128	40.2%	256	62.2%	512	89.2%	1024
4	28.1%	32	40.7%	64	41.4%	128	66.9%	256	93.7%	512
8	27.4%	16	40.2%	32	41.8%	64	67.8%	128	95.8%	256
16	20.0%	8	25.5%	16	28.6%	32	57.7%	64	85.2%	128
32	12.4%	4	22.3%	8	27.2%	16	56.5%	32	84.6%	64
64	6.2%	2	14.6%	4	22.6%	8	52.1%	16	74.2%	32

Table 3.6 -- Relation of the hit ratio and the cache words per module of LLL13 simulations in the FMRF memory model with data cache.

3.3. Summary and Discussion

The potential deadlock states in request scheduling have been identified. The instruction fetch requests and the load address requests must be stored in separate queues in order to avoid deadlock. The memory requests must be organized into a minimum of four request queues: (1) one IF Request Queue for storing IF requests from both Access and Execute processors, (2) one LA/SA Request Queue for storing load addresses of LA and ALA requests and store addresses of SA and ASA requests (assuming that requests are received from the Access Processor in correct sequence, so that deadlock will not occur), and (3) & (4) two Store Data (SD) Queues for storing store data from the Access and the Execute processors, respectively.

Three characteristics of memory references in decoupled computations were observed: (1) two memory inputs can be issued to the memory system in the same clock period, (2) overflow of hardware queues can be caused by continuous occurrence of memory requests, and (3) there is a shared variable problem that leads to Read-After-Write access hazards. Problems resulting from these three characteristics have been discussed, and possible solutions were proposed and evaluated.

The round-robin and the EP-First multiplexing schemes were shown to be similar from the results of the trace-driven simulations, and they were the best of the four proposed multiplexing schemes that restrict the memory input arrival rate to one per clock period. The performance degradation due to the use of these two multiplexing schemes is negligible.

A Start-and-Stop flow control scheme is proposed to regulate the transactions between the processors and the memory system. By using this scheme, both the processor and the Memory Controller must reserve three empty buffers before a *Stop* signal is sent. After a *Stop* signal is sent, a *Start* signal must be sent if a new empty buffer is made available; otherwise, a *Stop* signal must be sent. Thus, the sizes of the Load Data Queue on a processor and the request queue on the Memory Controller each needs to be at least 4. The simulation studies show that the size of the Load Data Queue on a processor does not need to be larger than 4, and the request queue size in the Memory Controller does not need to be greater than 4.

For the shared variable problem, short-circuiting Read-After-Write requests has been shown to be effective in reducing the penalty resulting from Read-After-Write access hazards between shared variable references. In the simulation studies of LLL13, the FMRF/RRF memory model with eight write buffers in each memory module can bypass the store data to the Read-After-Write requests, and improve the performance by more than 15% over the AP-First or EP-First memory models when the interleaving factor is greater than one. The FMRF memory model with a data cache in each memory module can short-circuit the store data to the Read-After-Write requests and improve the performance significantly if the total cache size is equal to or larger than 1 K (128 words per module and the interleaving factor is equal or greater than 8). However, the performance of the FMRF memory model with data cache in each module can degrade if the cache size is not large enough (less than 16 words per module in the LLL13 simulations).

CHAPTER 4

Request Scheduling and Memory System Design

The literature review of previous work in memory system design has led to the conclusion that pipelining, interleaving and buffering conflicting requests can be used to increase the throughput of the memory system. By using conflict queues, memory requests will be serviced out of arrival order. Thus, memory access hazards must be detected and resolved. Determining request priorities and designing a request scheduling policy with the ability of buffering conflict requests are the themes of this chapter. Memory systems based on the proposed request scheduling policies will be designed, and tradeoffs of the different characteristics in the design of the memory system will be discussed.

4.1. Memory Request Scheduling Strategies

Scheduling policies are critical to minimizing the total request service time in the memory system. Therefore, a basic understanding of the types of scheduling policies and the way each behaves under different memory organizations is necessary. There are two questions to memory request scheduling strategies for servicing six different types of requests: (1) within single type of memory requests, what order are the requests to be serviced? and (2) among request types, which request type is to be serviced next?

In the following sections, a memory module scheduling policy which can minimize the request service time for requests of a single type is derived, motivated by example and substantiated by formal proof. Then, strategies of servicing requests from different types are proposed. Finally, their performance implications and trade-offs are discussed.

4.1.1. Request Scheduling For a Single Request Type

There are two major categories of scheduling policies possible for requests of a single type with multiple servers: (1) to service the requests according to arrival sequence, and (2) to service the requests out of arrival sequence. The data fetched from the memory in the former type of scheduling policy will be returned in the same order as their arrival sequence. The data fetched from the memory in the latter type of scheduling policy must be reordered according to their arrival sequence before delivery from the memory system. The first category is termed First-Come-First-Serve (FCFS). An example of the second category, *Free-Module-Request-First* (FMRF), will be studied and shown to minimize the request service time for every request.

The Memory Controller using a First-Come-First-Serve policy services the oldest request pending in the memory if and only if the needed memory module is free, and it issues at most one memory operation in each clock period. The Memory Controller using a Free-Module-Request-First policy consists of M

Module Controllers, where M is the number of memory modules. Each Module Controller is allocated to service the requests accessing a single memory module, and issues the memory operation for the oldest request for that module whenever its associated memory module is free. Thus, the Module Controllers can issue more than one memory operation in the same clock period. All the Module Controllers must coordinate among themselves to deliver memory outputs to destination queues in the correct sequence (only one per clock, obviously).

The queueing model, in Figure 4.1, illustrates the queueing and servicing disciplines of the First-Come-First-Serve and Free-Module-Request-First policies. In Figure 4.1, the requests of the same type are further divided into separate queues according to the memory module number. In the First-Come-First-Serve policy, the request at the head of the queue is serviced when the server is free and it is the oldest request in the set of queues. In the Free-Module-Request-First policy, the requests at the heads of the queues are serviced as soon as the servers are free; however, the servers must coordinate among themselves in order to deliver the results of the serviced requests through the output bus one at a time according to the request arrival order. We assume that it does not require any additional time to reorder the memory outputs into the request arrival order.

There is an interesting intermediate possibility of servicing requests, that is, to use a centralized Memory Controller, which issues only one memory operation per clock period, but out of arrival sequence. The fetched memory outputs are then assembled by the centralized Memory Controller and delivered to their destination queues in the correct order. This request scheduling scheme completes memory requests in a longer time on the average than the Free-Module-Request-

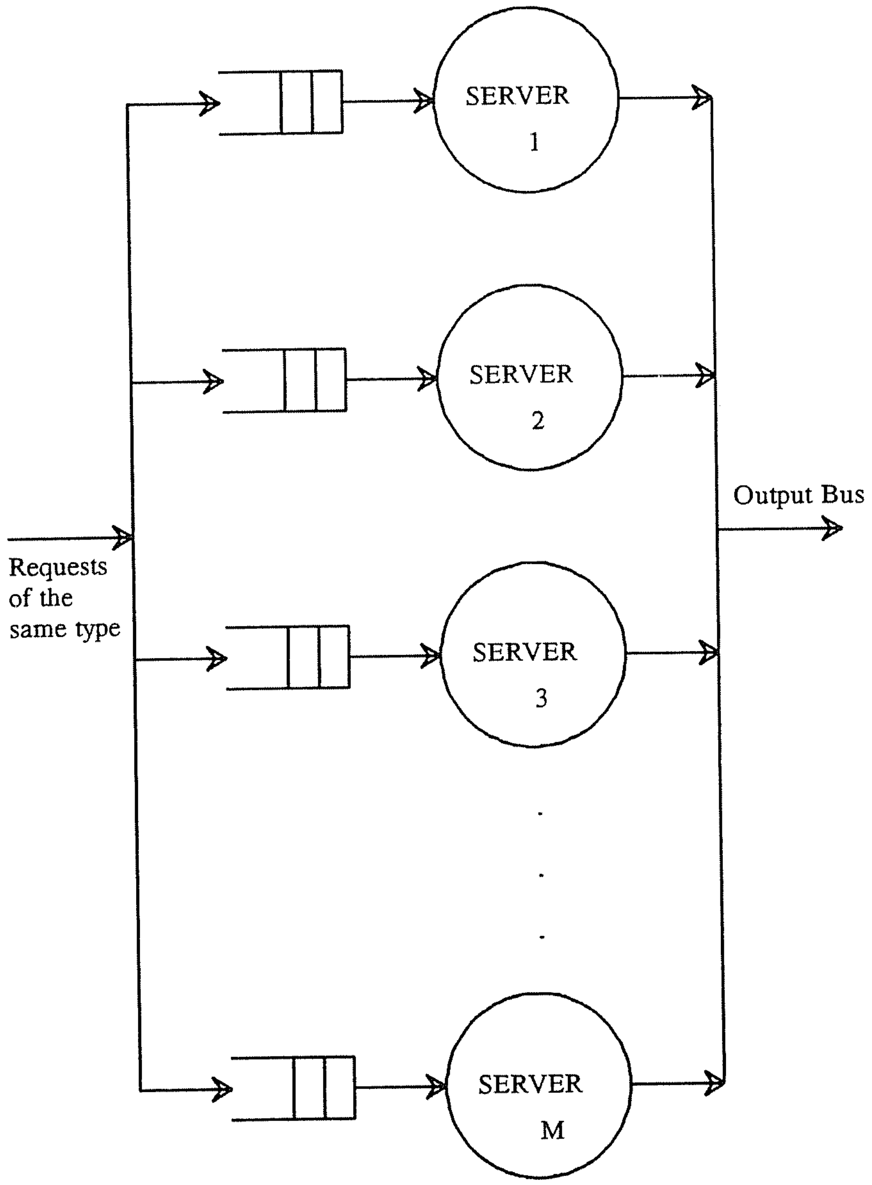


Figure 4.1 -- A queueing model for servicing requests of the same type with multiple servers. Each server services requests accessing the same memory module.

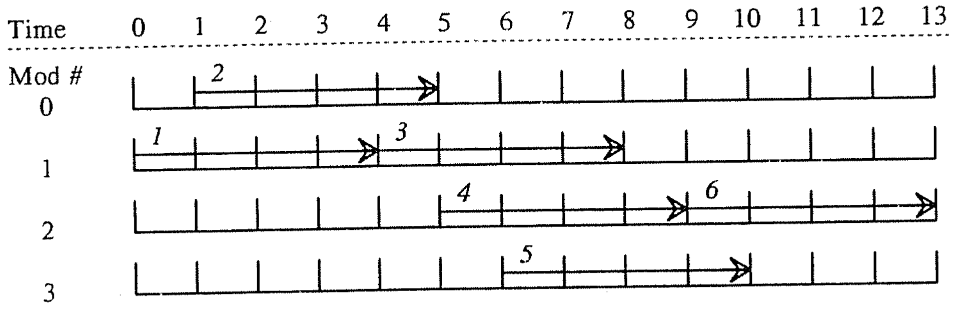
First policy, and is no easier to implement. It will not be further discussed. In the following examples, the request service sequence of the Free-Module-Request-First policy is illustrated by comparing its total service time with the First-Come-First-Serve policy.

Example 4.1 :

Assume that the memory system is 4-way interleaved, the memory bank busy time is 4 CPU clock cycles, and read requests arrive one per clock period at module 1, 0, 1, 2, 3 and 2 sequentially. Figure 4.1 illustrates the difference between the First-Come-First-Serve and the Free-Module-Request-First scheduling policies.

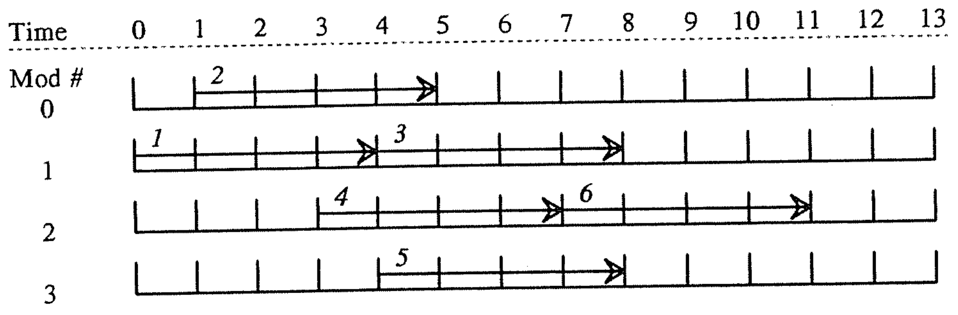
The first and the second requests arrive at module 1 and 0, respectively, and are serviced immediately. The third request conflicts with the first request at module 1; thus the third request has to wait until the first request has completed at time 4. Up to this point, both the First-Come-First-Serve policy and the Free-Module-Request-First policy provide the same results. But, when the 4th and 5th requests arrive at modules 2 and 3, the two scheduling policies differ. The Free-Module-Request-First policy can service the 4th and 5th requests on the same clock period that they arrive. The First-Come-First-Serve policy cannot service the 4th request until the 3rd request is issued, and the 5th request will not be serviced until after the 4th request is serviced. In the case of Free-Module-Request-First policy, the 4th request will be completed before the 3rd request, and the 5th request will be completed on the same clock period as the 3rd request. Since the outputs have to be sent out in the order of request arrivals, the 4th request output

FIRST-COME-FIRST-SERVE POLICY



Arrival	1	0	1	2	3	2								
Issue	1	0	.	.	1	2	3	.	.	2				
Output	.	.	.	.	1	0	.	.	1	2	3	.	.	2

FREE-MODULE-REQUEST-FIRST POLICY



Arrival	1	0	1	2	3	2						
Issue	1	0	.	2	1,3	.	.	2				
Output	.	.	.	.	1	0	.	.	1	2	3	2

Figure 4.2 -- Assume 4-way interleaving and a memory bank busy time of 4 CPU cycles. At time 0, 1, 2, 3, 4, and 5, read requests arrive at module 1, 0, 1, 2, 3 and 2, respectively. It takes 13 CPU cycles for the First-Come-First-Serve scheduling policy to complete service, while the Free-Module-Request-First scheduling policy needs only 11 CPU cycles.

and the 5th request output have to follow the 3rd request output. Thus, total service time to process these five memory requests is the same for both scheduling policies. However, after the 5th request, the Free-Module-Request-First policy can complete the 6th request earlier than the First-Come-First-Serve policy, if the module referenced by the 6th request is 2 or 3. In this example, the 6th request, that arrives at module 2, will be completed at the 11th clock period for the Free-Module-Request-First policy, and completed at the 13th clock period for the First-Come-First-Serve policy.

From *Example 4.1*, we see that the Free-Module-Request-First policy takes advantage of free memory modules and completes memory requests as soon as possible. The Free-Module-Request-First policy is superior when memory conflicts occur frequently; therefore, it is important to understand the characteristics of memory interference in the decoupled architecture.

Example 4.2 :

To further illustrate the difference of the Free-Module-Request-First policy and the First-Come-First-Serve policy under different memory conflict frequencies, examples of memory reference sequences with different memory bank busy times from a decoupled code of LLL2 (see Appendix II) are used.

For the following example, assume that there is neither a bus conflict nor an instruction cache miss on the Access Processor and that it takes one clock to complete a one-parcel (16-bit) instruction and two clocks for a two-parcel (32-bit) instruction. The memory arrival sequence of LLL2 can be derived by timing the execution of its access code. With this request arrival sequence, the performance difference of the First-Come-First-Serve and the Free-Module-Request-First policies can be evaluated for different memory configurations.

Assume a memory system that is 4-way interleaved. Since there are 1000 elements in the TP, X and Z arrays of LLL2, unless special storage arrangements are made to prevent these three arrays from storing in consecutive memory locations, it is likely that their base addresses will be in the same memory module. Without loss of generality, assume that the array base addresses are in module 3 and the first read request arrives at time 0. For each loop iteration, the progress of memory read requests with a memory bank busy time of 6 and 4 CPU clock cycles is illustrated in Figures 4.2 and 4.3, respectively. These two time diagrams demonstrate that the Free-Module-Request-First scheduling policy can benefit from frequent memory conflicts and a sufficiently long memory bank busy time. In the First-Come-First-Serve scheduling policy, more memory conflicts occur when the memory bank busy time is 6 CPU cycles than when the memory bank busy time is 4 CPU cycles (the number of memory conflicts is counted as the number of memory cycles when the Memory Controller cannot service any pending requests.) The total service time for the Free-Module-Request-First policy is shorter than that of the First-Come-First-Serve policy when the memory bank busy time is greater than 4 CPU cycles; however, the total service time is the same for both policies when the memory bank busy time is 4 CPU cycles or less. This is because the 3rd request arrives before the 2nd request is serviced if the memory bank busy time is greater than 4 CPU cycles. Thus, for the First-Come-First-Serve policy, the 3rd request cannot be serviced when it arrives, and

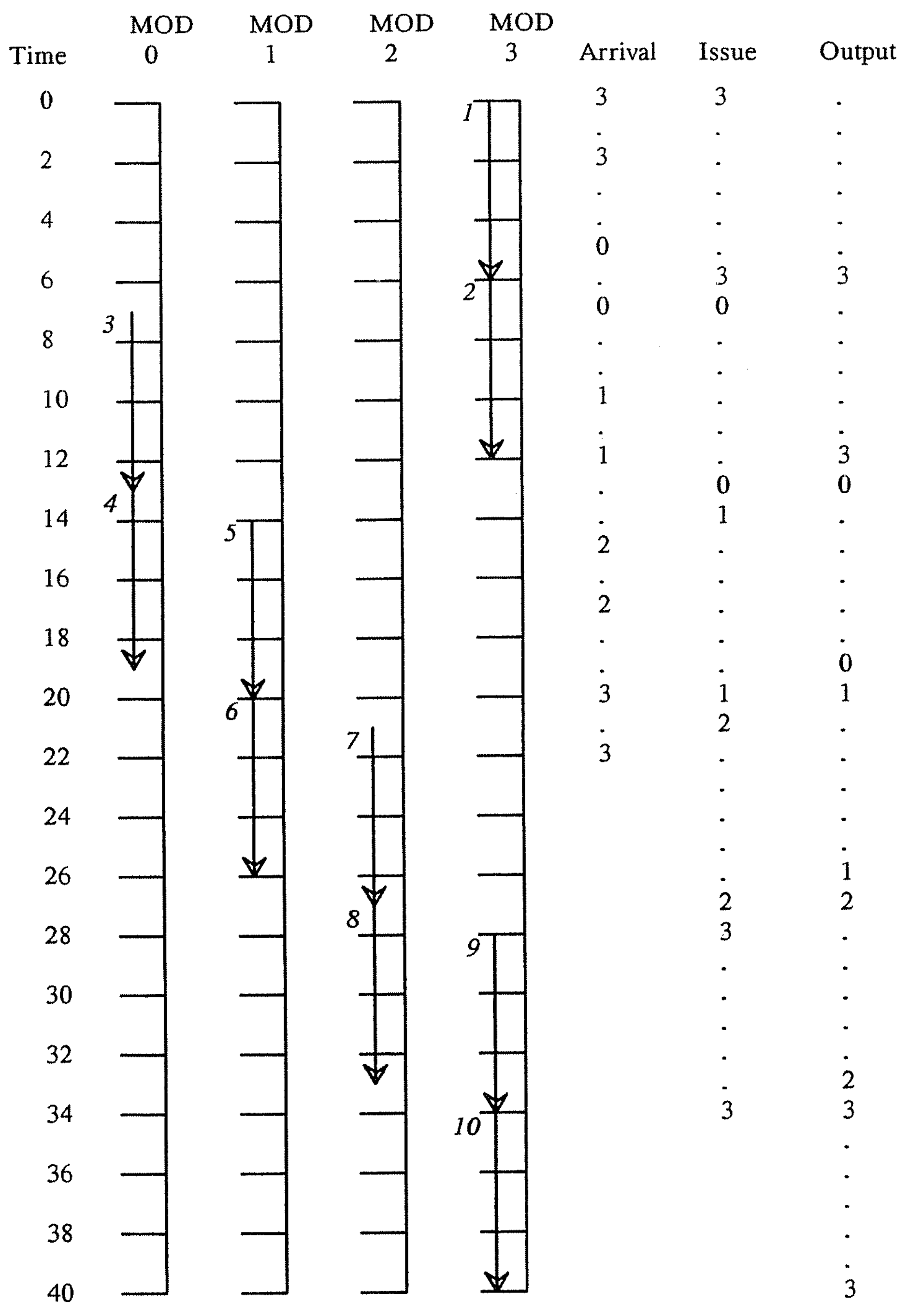


Figure 4.3.A -- Assume 4-way interleaving and a memory bank busy time of 6 CPU cycles. At time 0, 2, 5, 7, 10, 12, 15, 17, 20 and 22, read requests arrive at module 3, 3, 0, 0, 1, 1, 2, 2, 3 and 3, respectively. It takes 40 CPU cycles for the First-Come-First-Serve scheduling policy to finish.

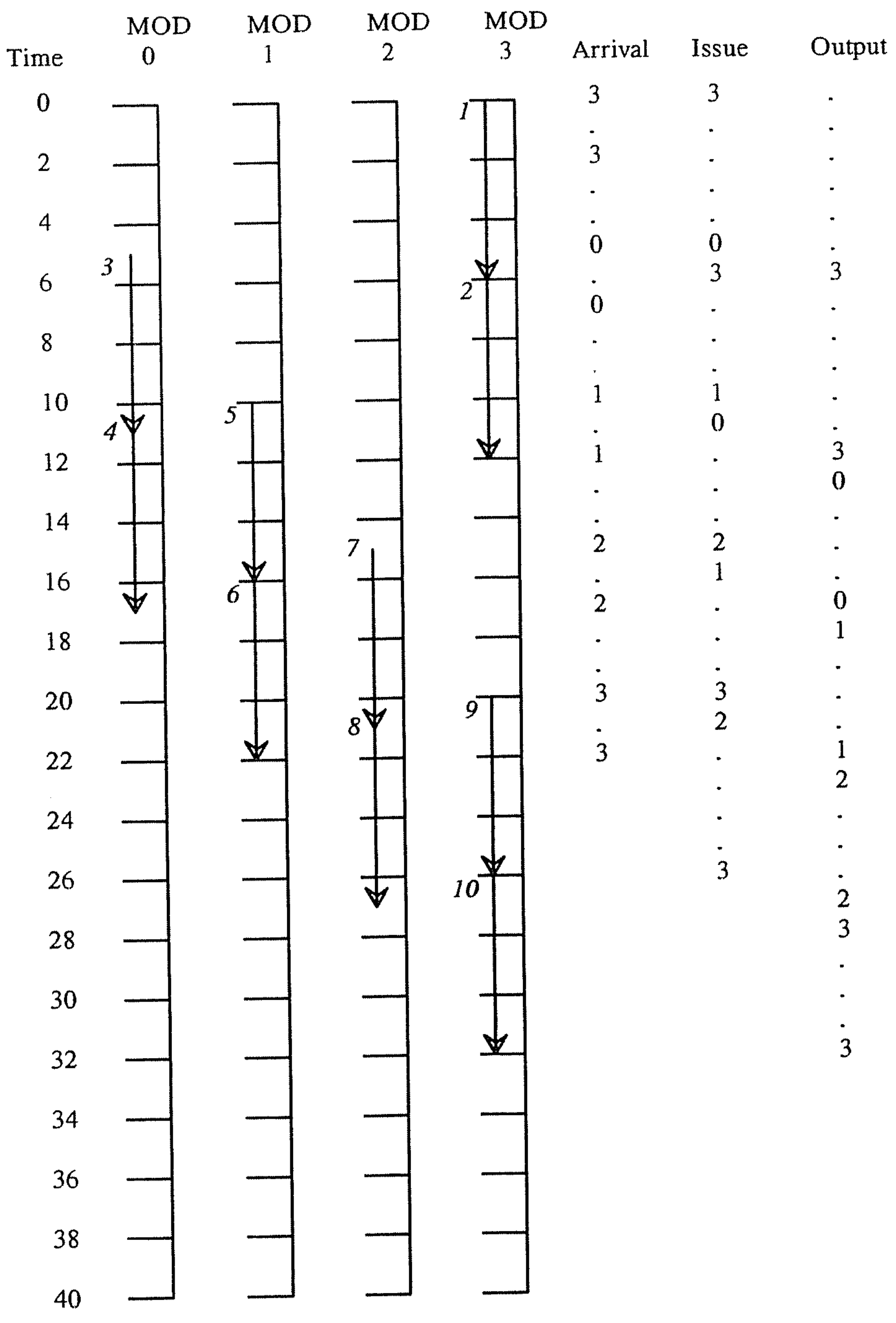
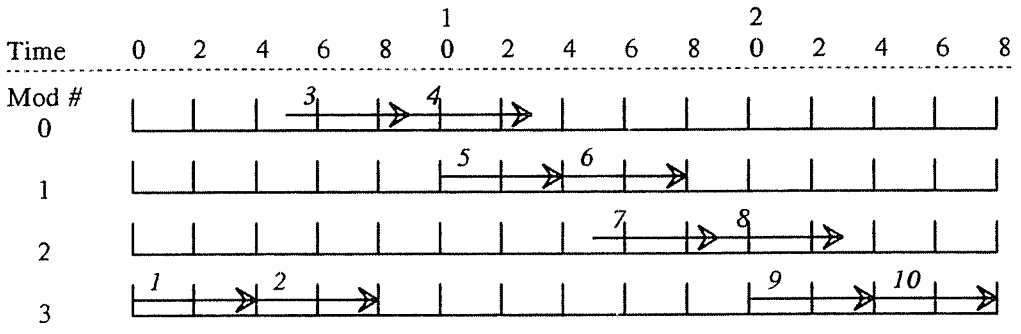


Figure 4.3.B -- Assume 4-way interleaving and a memory bank busy time of 6 CPU cycles. At time 0, 2, 5, 7, 10, 12, 15, 17, 20 and 22, read requests arrive at module 3, 3, 0, 0, 1, 1, 2, 2, 3 and 3, respectively. It takes 32 CPU cycles for the Free-Module-Request-First scheduling policy to finish.

FIRST-COME-FIRST-SERVE POLICY

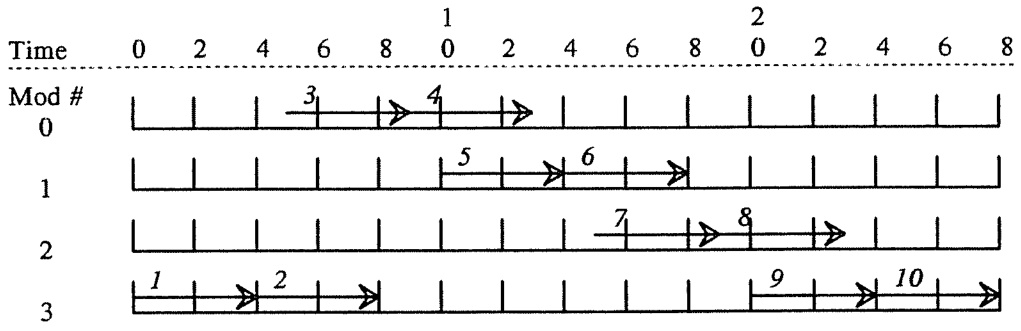


```
Arrival   3 . 3 . . 0 . 0 . . 1 . 1 . . 2 . 2 . . 3 . 3

Issue     3 . . . 3 0 . . . 0 1 . . . 1 2 . . . 2 3 . . . 3

Output    . . . . 3 . . . 3 0 . . . 0 1 . . . 1 2 . . . 2 3 . . . 3
```

FREE-MODULE-REQUEST-FIRST POLICY



```
Arrival   3 . 3 . . 0 . 0 . . 1 . 1 . . 2 . 2 . . 3 . 3

Issue     3 . . . 3 0 . . . 0 1 . . . 1 2 . . . 2 3 . . . 3

Output    . . . . 3 . . . 3 0 . . . 0 1 . . . 1 2 . . . 2 3 . . . 3
```

Figure 4.4 -- Assume 4-way interleaving and a memory bank busy time of 4 CPU cycles. At time 0, 2, 5, 7, 10, 12, 15, 17, 20 and 22, read requests arrive at module 3, 3, 0, 0, 1, 1, 2, 2, 3 and 3, respectively. It takes same amount of time for the Free-Module-Request-First and the First-Come-First-Serve scheduling policies to finish.

it delays the starting service time of the later arrivals.

In this example, the beginning addresses of arrays X and Z are located in the same memory module when the interleaving factor is 4 or 8 (because the 1000 array elements of X and the 1000 array elements of Z are stored in consecutive memory locations) and not located in the same module when the interleaving factor is greater than 8. Thus, the number of memory conflicts between the array references of X and Z decreases when there are more than 8 memory modules[1], and the performance difference of First-Come-First-Serve and Free-Module-Request-First policies will be insignificant.

4.1.1.1. Theorem of Free-Module-Request-First Scheduling Policy

> *Theorem 1* : Given a stream of request arrivals and that the memory outputs of the read requests have to be delivered in the same sequence as the their arrivals, the Free-Module-Request-First request scheduling policy always completes service for each request in the shortest possible time.

Proof: For the queueing model in Figure 4.1, the FMRF scheduling policy starts the service for the requests in each queue as soon as the needed server is free, and the operands fetched for the read requests will be ready for delivery as soon as the server completes the service of the read requests. If the requests in each queue are not serviced as soon as the needed server is free, then there is an idle server between the end of a memory operation and the beginning of the following memory operation. Due to the delay of memory operations, the operand fetched

[1]A compiler can achieve the same effect by careful alignment of array addresses.

for each read request can not be ready for delivery at an earlier time than if the read request is serviced by the FMRF scheduling policy.

Because the operands fetched for the read requests must be delivered in the arrival order, there is no benefit for a server to service the requests in its request queue out of arrival order. For the FMRF scheduling policy, the server of a particular module completes the service for each of its requests at the earliest possible time. The fetched operands are then ready for delivery at the earliest possible time. Because it does not require extra clock periods for the servers to reorder the fetched operands into the arrival order, the FMRF scheduling policy can deliver the fetched operands at the earliest possible time. Thus, the FMRF scheduling policy always completes the service for each request in the shortest possible time.

End of Proof.

4.1.2. Request Scheduling Among Request Types

During decoupled computations, the Memory Controller can receive six different types of requests: (1) instruction fetch (IFa) request for the Access Processor, (2) instruction fetch (IFe) request for the Execute Processor, (3) load address (LA) request for fetching operands to the Load Data Queue of the Access Processor, (4) alternative load address (ALA) request for fetching operands to the Load Data Queue of the Execute Processor, (5) store address (SA) request for storing data generated by the Access Processor, and (6) alternative store address (ASA) request for storing data generated by the Execute Processor. In addition, there are store data (SD) from the Access and the Execute processors to be paired with

SA and ASA requests, respectively.

In chapter 3, it was concluded that the IF and the LA/ALA requests must be stored in separate queues in order to avoid deadlock, and that the load and the store address requests can be stored in the same queue as long as the decoupled code is correct. The Memory Controller need not check for access hazards (Read-After-Write, Write-After-Read and Write-After-Write hazards) if the load and the store address requests are stored in the same LA/SA queue.

4.1.2.1. Considerations of Separating Load and Store Requests

There are three reasons, described below, for dividing an LA/SA Request Queue into an LA Request Queue and an SA Request Queue.

(1) A store operation can be issued only if the store address and the store datum are both available. If the head of LA/SA Request Queue is an SA request, and the store datum for this SA request is not yet available, this SA request will block the other LA requests from being issued until its store datum arrives.

(2) By storing the LA and the SA requests in separate queues, the Memory Controller can service the LA request before the SA request as long as an access hazard does not exist; thus, the operands can be delivered to the processors earlier.

(3) The potential deadlock states (discussed in chapter 3), that can be introduced by the combination of an inappropriate sequence of load and store with the use of a single LA/SA Request Queue for the LA requests and the SA requests will not occur.

In Chapter 3, it was concluded that by giving a higher priority to the LA request than the SA request, the total processing time of decoupled computation can be shorter than servicing the LA and SA requests in order of arrival. If LA requests are to be serviced before SA requests, the Memory Controller must check for Read-After-Write access hazards before a LA request can be serviced. It was also concluded that the short-circuiting technique to bypass a store datum to the Read-After-Write request is effective in reducing the total request service time. Because short-circuiting Read-After-Write data can generate memory outputs for a Load Data Queue in an order that is different from their original request arrival sequence, the Memory Controller must assemble the memory outputs into the correct sequence before delivering them.

Alternatively, if the Memory Controller blocks the service of a LA request when an access hazard is detected and services the LA request when the hazard is resolved, then memory outputs will be generated in the correct order and the Memory Controller can deliver memory outputs whenever the processor is ready to receive them. Therefore, the control logic for the Memory Controller can be simpler.

Thus, it is concluded that handling loads before stores improves performance, while aggravating the hazard detection problem. In order to evaluate the trade-offs, the performance difference of handling loads and stores in order and handling loads before stores is evaluated through trace-driven simulations in chapter 5. A data cache with the ability to detect/resolve access hazards is designed in chapter 6, and its performance is evaluated.

4.1.2.2. Priority of Instruction Fetch and Load Address Requests

In the PIPE architecture, the instruction cache lines are fetched on demand. Thus, IF requests are sent to the Memory Controller only when the instruction words are needed. Each IF request fetches four parcels from four consecutive memory locations in four different memory modules. When there are IF and LA requests waiting for service simultaneously, the Memory Controller has to decide whether the request from the IF Request Queue or the LA Request Queue should be serviced first. The IF request is serviced before the LA request for two reasons:

(1) The IF request queue is usually empty because an instruction cache miss does not occur frequently, and it can never be longer than 2 (for the PIPE architecture).

(2) A PIPE processor does only limited prefetching of instructions, never fetches instruction cache lines that will not be used. Thus, the instruction issue logic likely will cease in only a few clock periods when a cache miss occurs. On the other hand, the operands being fetched probably will not be required for a longer time if reasonable code scheduling has been performed.

Thus, the IF request is more urgent than the LA request.

4.1.2.3. Three Steady-State Models in Decoupled Computation

In the previous sections, we established the priorities between the LA and the SA requests, and between the IF and the LA requests. The LA requests are further divided into the LA requests for the Access Processor and the ALA requests for the Execute Processor, so that the Memory Controller can select the next service for the load address request of a processor whose requests are considered to be more urgent than those of the other processor. The priority of the LA and the ALA requests can be determined based on these three steady-state situations:

(1) The Access Processor is not able to run ahead and is the bottleneck of decoupled computation, i.e. the LA request queue is empty.

(2) The Execute Processor has a full Load Data Queue and is the bottleneck of decoupled computation.

(3) The Memory Controller is not able to meet the demand of requests and is the bottleneck of decoupled computation, i.e. the LA and/or SA request queues are full.

4.1.2.3.1. Access Processor is the Bottleneck

The workload of address generation is heavier than that of algorithmic computation when the Access Processor is the bottleneck of decoupled computation. In this situation, the LA request should be given a higher priority than the ALA request. Thus, the memory wait time for the Access Processor can be reduced, and consequently the total processing time can be reduced.

The Load Data Queue of the Execute Processor will be empty most of the time, because the Access Processor is not able to run ahead, and because the Memory Controller services the LA request before the ALA request. The ALA request queue in the Memory Controller may become full when LA requests are always serviced before ALA requests. When the ALA request queue is full, the flow control scheme will stop the Access Processor from sending further requests until the ALA request queue is not full. The Memory Controller will service ALA requests either when the ALA request queue is full or when there is no LA request. The Execute Processor can continue its processing when operands arrive, and it will not be blocked all the time.

In conclusion, the Memory Controller should service the LA request before the ALA request if the Access Processor is the bottleneck. Flow control can prevent the Memory Controller from servicing the LA request all the time without servicing the ALA request queue. Therefore, the Execute Processor will not be blocked.

4.1.2.3.2. Execute Processor is the Bottleneck

The workload of algorithmic computation is heavier than that of address generation when the Execute Processor is the bottleneck of decoupled computation. In this situation, the Load Data Queue of the Execute Processor will be full most of the time. It does not make any difference whether the LA request is serviced first or the ALA request is serviced first[2] as long as the Load Data Queue of the Execute Processor is never empty. Thus, a higher priority can be given to the LA request than to the ALA request.

4.1.2.3.3. Memory Controller is the Bottleneck

When the Memory Controller cannot service memory requests efficiently, one or both of load request queues on the Memory Controller may be full most of the time, and neither of the Load Data Queues on the Access and Execute processors will be full. The decoupled computation works best if the Access Processor is able to race ahead of the Execute Processor. As long as the Memory Controller is the bottleneck, the order in which the LA or ALA requests are serviced is unimportant. However, servicing the ALA request first is likely to result in a full LA request queue and an empty ALA request queue because the Access Processor may not be able to issue load requests. The only reason for servicing ALA

[2]In fact, ALA requests will not be serviced if the Load Data Queue of the Execute Processor is full.

requests first is that a shared variable may block the Access Processor. At this point, the LA request queue will drain.

If load requests are always serviced before store requests, the store request queue may become full. When the store request queue is full, the flow control will stop the processor to send more requests; thus, load requests will be completely serviced and then store request will be serviced. In addition, if the destination data queue of load requests is full, the service for load requests will be deferred. Then, store requests will be serviced.

Thus, from previous analysis, it is concluded that the load request should be serviced before the store request, and the LA request should be serviced before the ALA request.

4.2. Pipelined Memory System Design

The basic model of our pipelined memory system and its pipeline stages will be described and a detailed design of the Memory Controller using Free-Module-Request-First policy will be illustrated.

4.2.1. Basic Model of A Memory Controller

The block diagram of a *basic* pipelined memory system model is illustrated in Figure 4.5. The major sections of the basic pipelined memory system are the input logic, the issue logic, the output logic, the Memory Operation Status Table, and the memory modules. All but the last of these sections are in a centralized Memory Controller for the First-Come-First-Serve memory system, while they are *distributed* among the Module Controllers for the Free-Module-Request-First

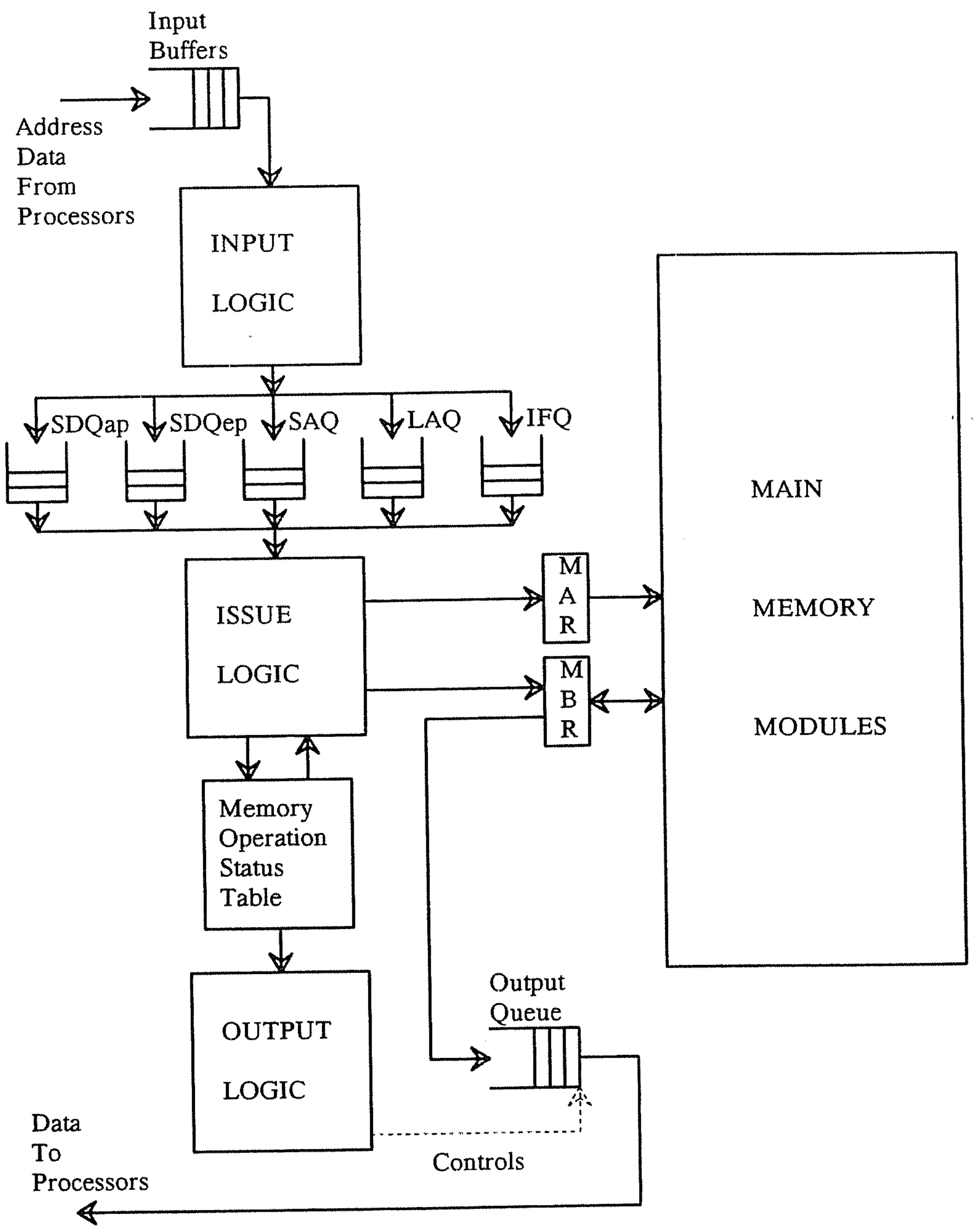


Figure 4.5 -- Basic Model of the Pipelined Memory System

memory system. The organization of a Module Controller for the Free-Module-Request-First memory system is similar to the Memory Controller for the First-Come-First-Serve memory system. Detailed design of a Module Controller and algorithms for distributed sequence control between Module Controllers for Free-Module-Request-First policy are described after the primary functions of the Memory Controller are illustrated.

4.2.1.1. Primary Functions of a Memory Controller

The pipeline organization of the Memory Controller is shown in Figure 4.6, where the steps to service a request are listed along with the pipeline stages. Some of these steps are needed only for Free-Module-Request-First policy, e.g., the sequence number assignment, and are further described in the later part of this chapter.

The general operation of the memory pipeline begins with the input received from the processors and stored in the input buffers. The input logic removes an entity (request address or data) from the input buffer, decodes the entity, then transfers the entity to the request queue along with descriptive information about the entity. Each of the LAQ and SAQ must have the capability to detect access hazards before a memory request is selected for service. (An efficient access hazard detection and resolution scheme will be presented in chapter 6.) The issue logic selects a request from the request queue according to the availability of a memory module, and then issues the memory operation. The Memory Operation Status (MOS) Table contains the information about the status of memory modules, and the activity of every memory operation in progress. The issue logic

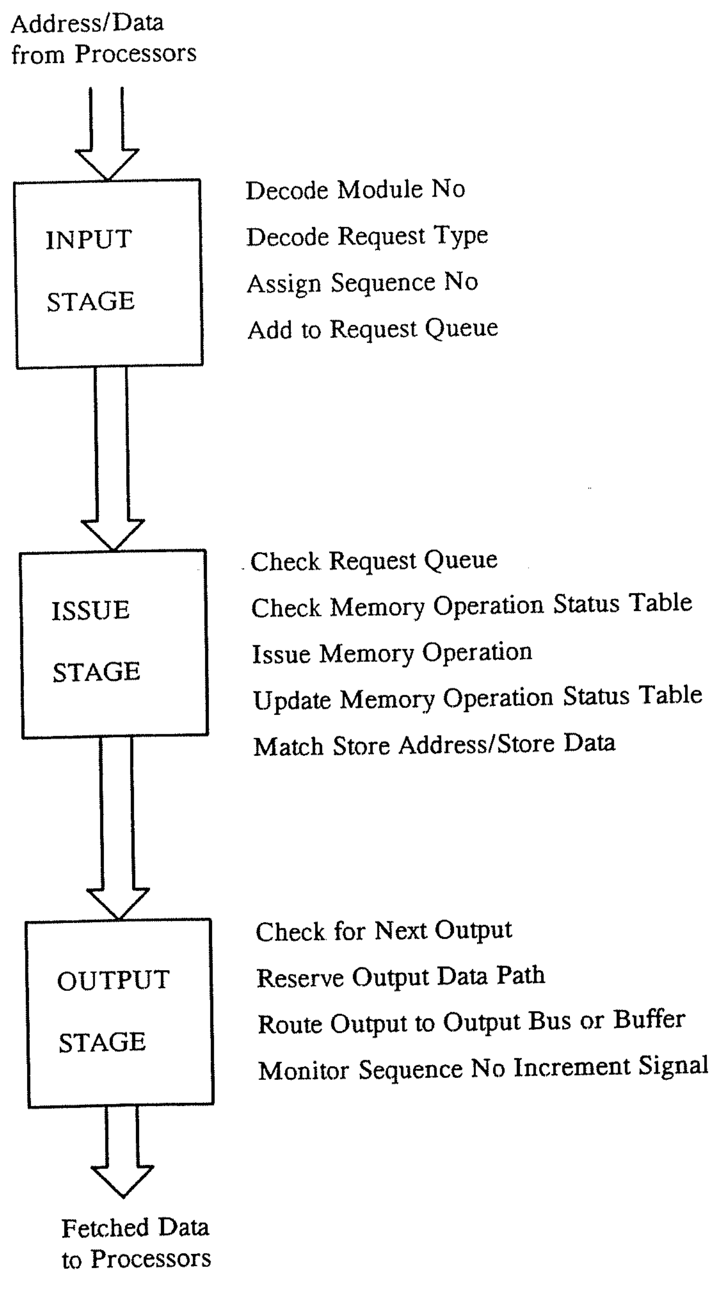


Figure 4.6 -- Memory Pipeline Stages and Their Functions

must store information about a memory operation into the MOS Table whenever a memory operation is started. The output logic checks the MOS Table for information about a completing memory operation, so that the data fetched from the memory can be delivered to the correct destination. The output queue holds memory outputs if the Load Data Queues in the processors are full. The output queue also assembles the out-of-order data if the Free-Module-Request-First scheduling policy is used.

4.2.1.2. Organization of Memory Operation Status Table

The Memory Operation Status Table contains the availability status of each memory module as well as progress information for every memory operation. The scheduling policy used by the Memory Controller will determine the content of the progress information. Basically, the MOS Table consists of M groups of N registers, where M is the number of memory modules and N is the number of registers needed to store the availability status of a module and the progress information of a memory operation. The main registers are:

(1) BUSY register (B): a (C+1)-bit shift register contains the availability status of a memory module and the progress status of a memory operation in the module, where C is the memory bank busy time.

(2) Destination register (DST): a 1-bit register contains the ID of the processor that will receive the data fetched from memory.

(3) Read/Write register (R/W): a 1-bit register indicates whether the memory operation in progress is a read or write operation.

(4) Request type register (I/D): a 1-bit register indicates whether it is an instruction fetch or data fetch operation if a read operation is in progress.

Other registers are included depending on the design of the memory system. For example, a sequence number register (SEQ NO) is used to store the sequence number of a fetch request serviced in the Free-Module-Request-First scheduling policy, and registers that contain information specific to a data cache.

All bits of the B register of a memory module are set to zeros when the memory module is free. When a memory operation is started in a memory module, the Memory Controller stores information in the corresponding group of registers in the MOS Table: (1) all $C+1$ bits of the B shift register are set to 1, (2) the descriptive information of this memory operation is stored into their respective registers. As the memory operation proceeds, the contents of the i-th bit of the B register is shifted to the $(i+1)$-th bit, zero is stored into the bit 0 of the B shift register, and the content of the C-th bit of the B register is lost. The B register is a reservation table [BrDa77]. It is used by the Memory Controller to check whether or not the module is busy, and it is also used by the Memory Controller to determine when a memory operation will be completed. Thus, the Memory Controller uses the contents of the B register to plan activities for the memory operation before the operation is completed. The Memory Controller uses the contents of the DST, R/W, and I/D registers to determine the destination of the information fetched by the current memory operation (if it is a read operation).

Since the Memory Controller knows whether or not the memory operation is completing, whether or not the Load Data Queues on the processors are full (from the flow control message received from the processors), and whether or not there are any outputs in the output queue ready to be delivered, it can correctly reserve a data path for the completing memory operation. The data path which routes the fetched data into the output queue will be reserved when: (a) there is a data item in the output queue which has to be delivered before the completing memory output, or (b) the Load Data Queue on the processor which will receive the completed memory output is full. Otherwise, the Memory Controller reserves the data path which routes the completing memory output to the output bus, and delivers the memory output to the final destination without delay (but in the correct sequence).

4.2.2. VLSI Implications and Constraints on Memory System Design

The number of available pins limits the number and the bandwidth of off-chip communication paths in a VLSI system. The larger the number of communication paths, the smaller the average communication bandwidth per path. The design process must trade off these two conflicting criteria and realize a solution which best meets the needs of decoupled computation. In order to reduce the traffic across the processor-memory paths, the Memory Controller should receive whatever information is sent from the processors, and the processors should only request the Memory Controller to fetch needed information.

The high density and low replication costs of VLSI circuits allows complex control logic to be implemented on a VLSI chip, and then be produced in large quantity at low cost. This feature favors the design of a distributed Memory Controller, whose control functions are distributed among the Module Controllers. Each Module Controller controls the memory operation activities within a memory module. Since the procedure of servicing requests in each memory module is the same for all memory modules, a single design of a Module Controller chip can be used to build a distributed Memory Controller for the entire memory system. In addition, the amount of data cache implemented on the Module Controller chip is limited by the available resource of a chip. Thus, the total amount of data cache implemented in the memory system is larger for a distributed Memory Controller than for a centralized Memory Controller.

4.2.3. Memory System Design Considerations

There are several considerations which will guide the design of the Memory Controller for the decoupled architecture in the VLSI environment:

(1) The control circuitry and the organization of request queues should be kept as simple as possible.

(2) The communication bandwidth between the Memory Controller and the processors should be appropriate for decoupled computation.

(3) The structure and organization of request queues should be suitable for the Memory Controller to efficiently select the next request to service and to avoid deadlock within the Memory Controller.

(4) Increased bandwidth should be emphasized whenever latency and memory bandwidth considerations conflict. This consideration is the result of previous findings that decoupling allows the memory system to be optimized for bandwidth rather than access time [WeSm84, GHLP85].

In chapter 3, it was concluded that the round-robin or EP-First multiplexing schemes can efficiently utilize the processor-to-memory data path in a decoupled computation. Also, analysis indicated that the size of the request queue must be at least four in order to have a smooth flow of flow control, and simulation showed that it need not be larger than eight. In the following sections, the partitioning of the Memory Controller into the Module Controllers for the Free-Module-

Request-First scheduling policy, and the sequence control scheme for the correct delivery of memory outputs are described. In order to evaluate the memory system design using trace-driven techniques on a closed-loop model that consists of the PIPE processor simulators and the memory system simulator, some of the design features for the memory system will be specific to the PIPE architecture. The features specific to the PIPE architecture are not important.

4.2.4. Implementation of Free-Module-Request-First Policy

In order to implement the Free-Module-Request-First scheduling policy, the memory operation of an outstanding request in each memory module must be started as soon as the memory module becomes free. Therefore, the control functions of the Memory Controller of an *m*-way interleaved memory system using the Free-Module-Request-First scheduling policy must be distributed among *m* Module Controllers. Also, the MOS Table is distributed such that each Module Controller has its own group of N registers to track the memory operation status within its module. The organization of the PIPE architecture with a 4-way interleaved Memory Controller using the Free-Module-Request-First scheduling policy is illustrated in Figure 4.7, and the multiple pipeline organization for the Memory Controller is illustrated in Figure 4.8. A block diagram for the Module Controller is illustrated in Figure 4.9.

Each Module Controller has request buffers to store the requests accessing the memory module controlled by the Module Controller. The request buffers within a Module Controller are organized into several request queues according to request types. The Module Controller selects a request from its request queues,

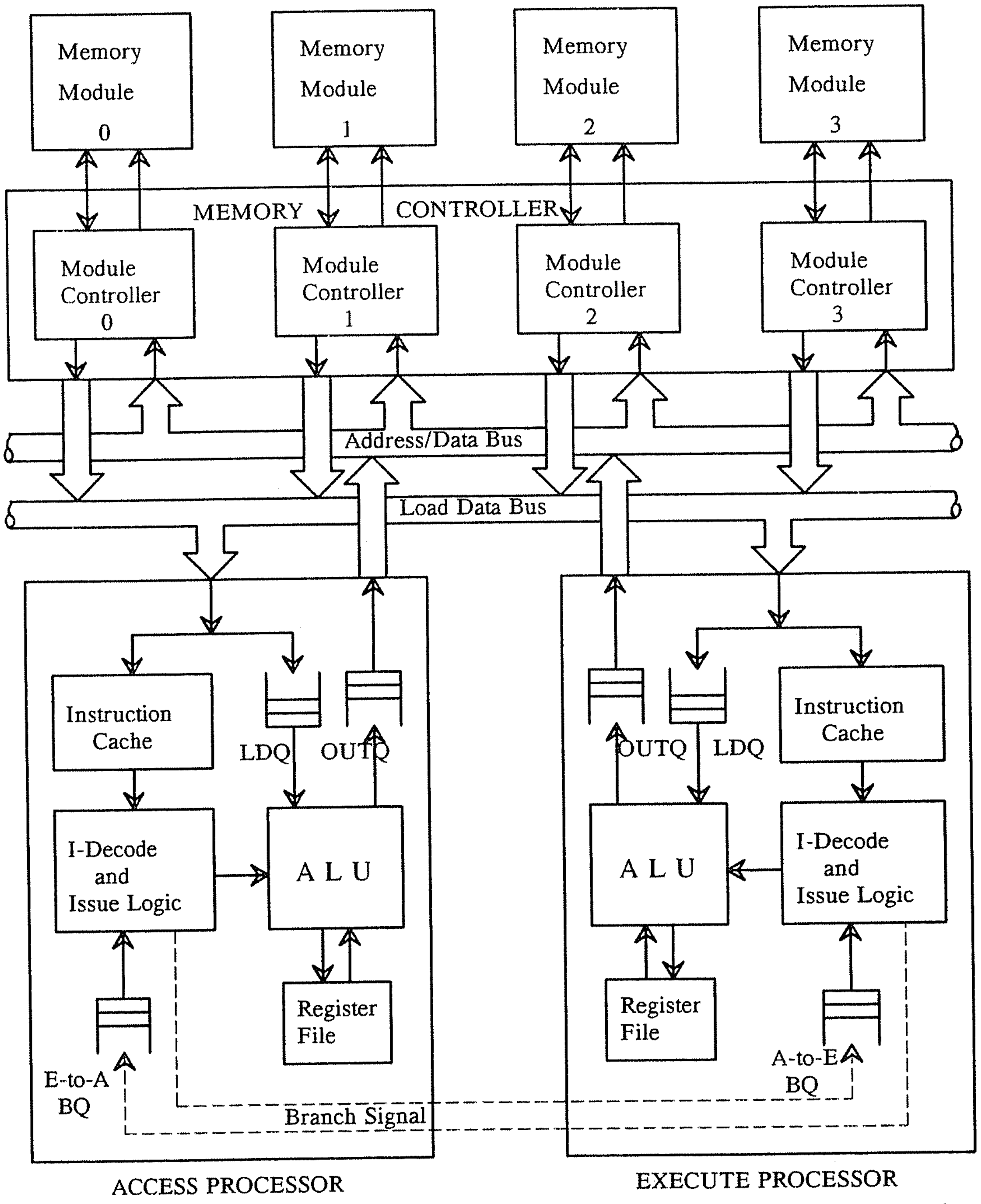


Figure 4.7 -- Organization diagram of the PIPE architecture with a 4-way interleaved Memory Controller using Free-Module-Request-First scheduling policy.

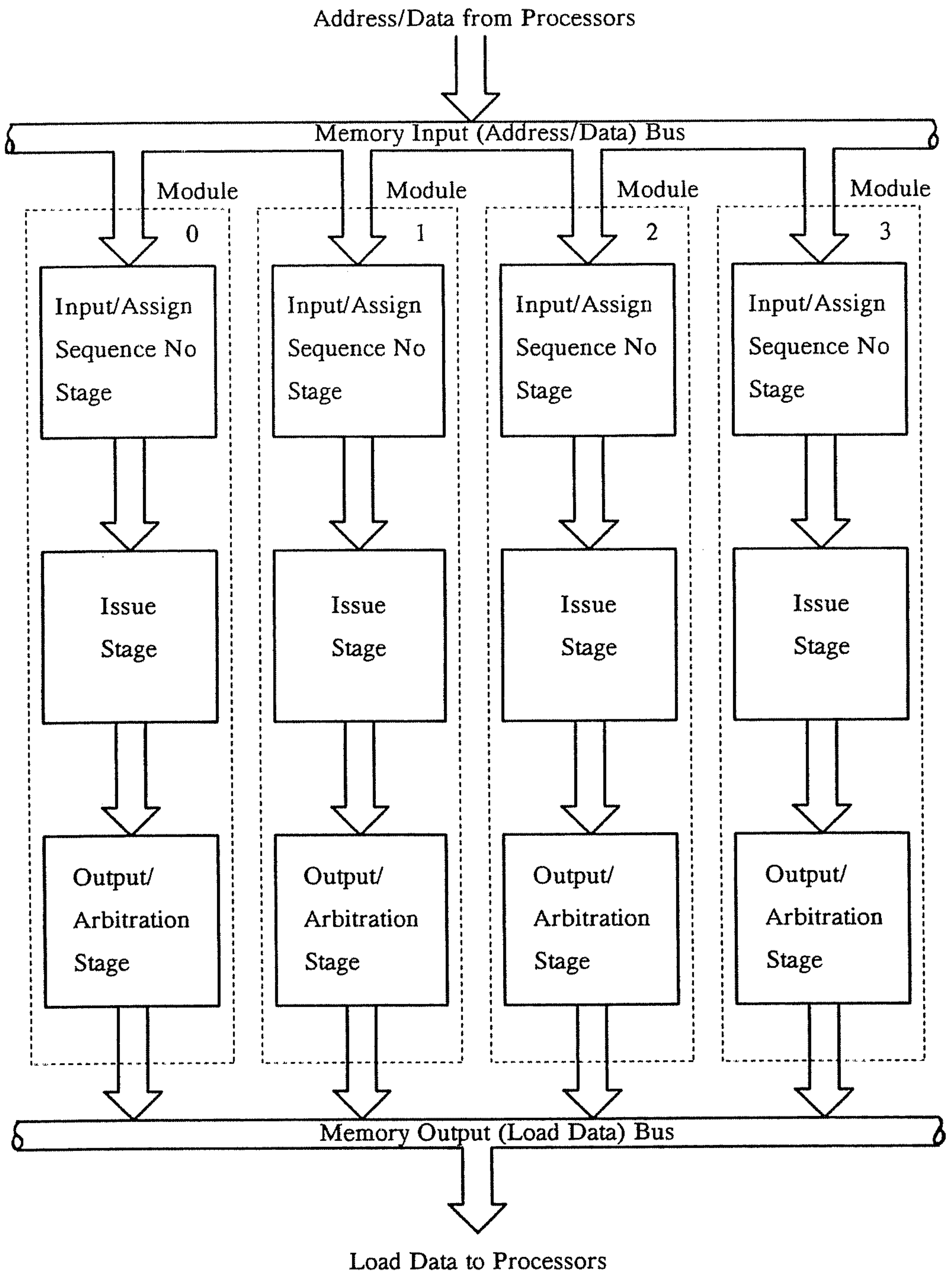

Address/Data from Processors
Memory Input (Address/Data) Bus
Module 0
Module 1
Module 2
Module 3
Input/Assign Sequence No Stage
Issue Stage
Output/ Arbitration Stage
Memory Output (Load Data) Bus
Load Data to Processors

Figure 4.8 -- Pipeline organization of a 4-way interleaved Memory Controller using Free-Module-Request-First scheduling policy.

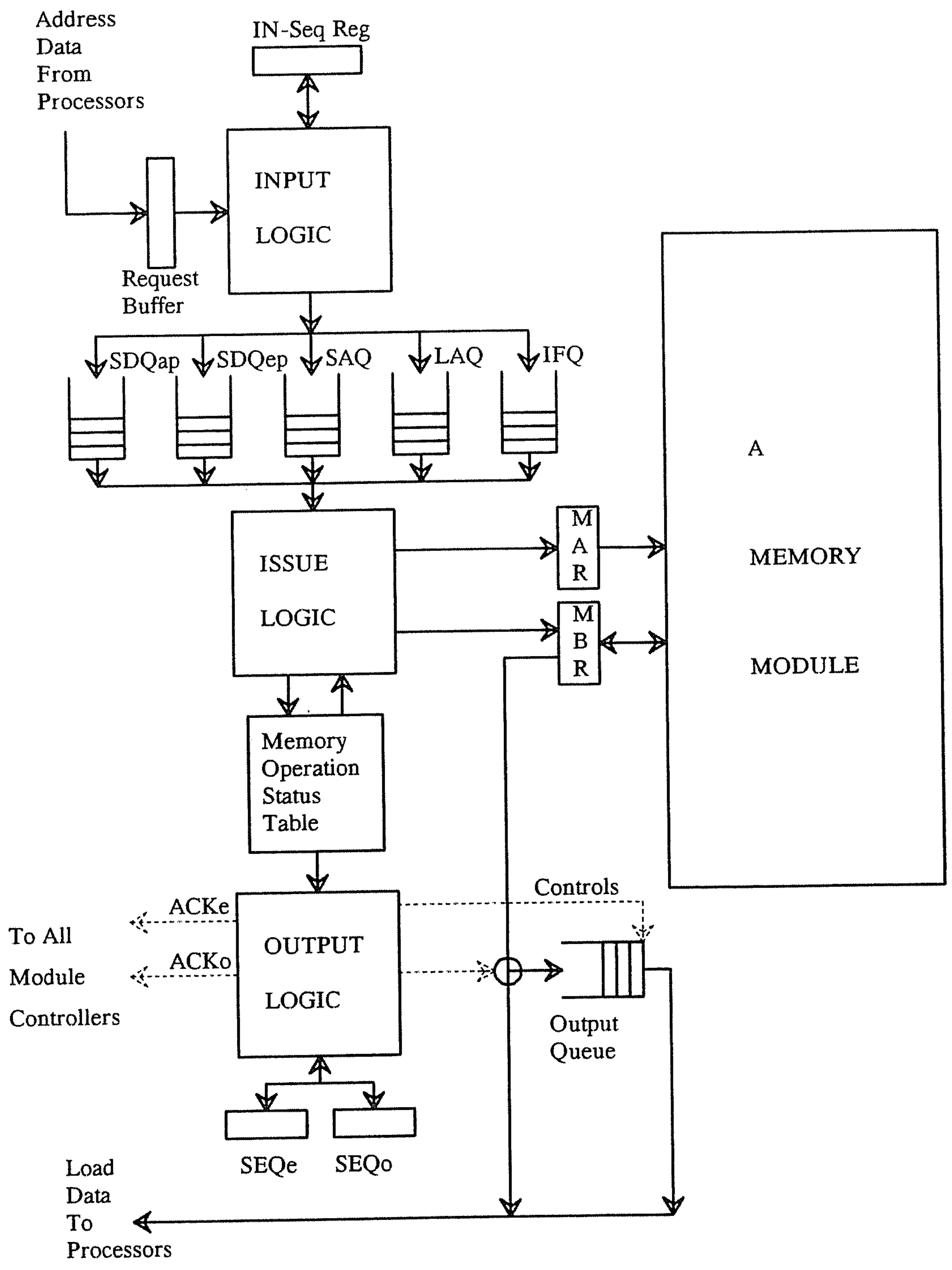


Figure 4.9 -- Block diagram of a Module Controller using the Free-Module-Request-First scheduling policy.

and services the selected request as soon as the module is free. The output stages of all Module Controllers must cooperate in order to deliver memory outputs to the processors in the correct sequence.

In the following discussion, the term *Memory Controller* refers to the controller of the entire memory system, and the term *Module Controller* refers to the controller of a memory module.

In order to deliver memory outputs in the correct order, sequence numbers are assigned to read requests. The memory outputs of read requests are delivered according to their assigned sequence numbers. Each sequence number is *unique* among requests of the same type that are waiting or being serviced in the Memory Controller. Since a write operation does not generate memory outputs, it is not necessary to assign a sequence number to a write request.

When an IF request is broadcast to all Module Controllers, each Module Controller checks whether it has one of the four instruction words requested by the IF request (an IF request fetches a 4-word instruction cache line). The Module Controller that has one of the requested instruction words, generates an instruction word fetch request for itself to fetch the corresponding instruction word from its module. Within a single IF request, the Module Controller computes a sequence number from the ordering of the instruction word that is to be fetched by the Module Controller, and assigns the sequence number to the instruction word fetch request. Thus, the four instruction words fetched from four different Module Controllers can be delivered to the processor in the correct sequence. (Note this is always the same order.) From previous discussions, the instruction words fetched for an IF request are given higher priority than the operands fetched for the LA requests.

A sequence control scheme which uses the sequence numbers to track the order of request arrival, allows the Module Controllers to issue memory operations out of arrival order, and then deliver the memory outputs in the arrival order (according to the sequence numbers). This sequence control scheme also allows a data cache to be implemented in the Module Controller. Because an operand can be fetched from the data cache (cache hit) in a shorter time than from the main memory module (cache miss), the operands generated from cache hits and misses will not be in the same order as the original arrival order. The sequence control scheme can be used to assemble the operands fetched from cache hits and misses into the correct order. Thus, the reordering required by a data cache can be implemented on the Module Controller almost for free. The sequence control scheme is described in the following sections, and the detailed design of a data cache in the Module Controller is presented in chapter 6.

4.2.4.1. Distributed Sequence Control for the Free-Module-Request-First Policy

Before a memory request is stored in the request queue, the *sequence number assignment logic* tags the memory request with a sequence number. The task of sequence number assignment and memory output control must be synchronized among all Module Controllers. In order to simplify the discussion of the sequence control scheme, the sequence control for load address requests will be described first. Then, the sequence control for both the IF and the load address requests will be discussed.

If the LA and ALA requests are treated as two different types of load address requests, it will require separate sets of sequence numbers for the LA and the ALA requests, respectively. This approach will complicate the design of distributed sequence control. In addition, the Access Processor makes most of the memory requests for the Execute Processor and very few requests for itself. Thus, the LA and ALA requests will be considered as the same type of load address requests in the sequence control.

As illustrated in Figure 4.8, a common memory input bus is used for broadcasting memory requests to all Module Controllers at the input stage. At every output stage, a Module Controller has to coordinate with other Module Controllers to determine the use of the memory output bus for delivering memory outputs. Each Module Controller (Figure 4.9) uses its input sequence number register (IN-SEQ) to track the input sequence number for the next LA request received from the input bus. Its output sequence number registers (SEQe and SEQo) are used to track the output sequence number of the last memory output delivered (by itself or another Module Controller) through the output bus.

4.2.4.2. Distributed Sequence Number Assignment at Input Stage

When the Module Controller receives a LA request from the input bus, it checks whether the received request is accessing a memory location within the module (see Figure 4.10). If so, the LA request is tagged with the input sequence number from the IN-SEQ register. Otherwise, if the received request is not accessing the receiving module, the received LA request is discarded. In either case, the IN-SEQ register is incremented by one. Since the LA request is

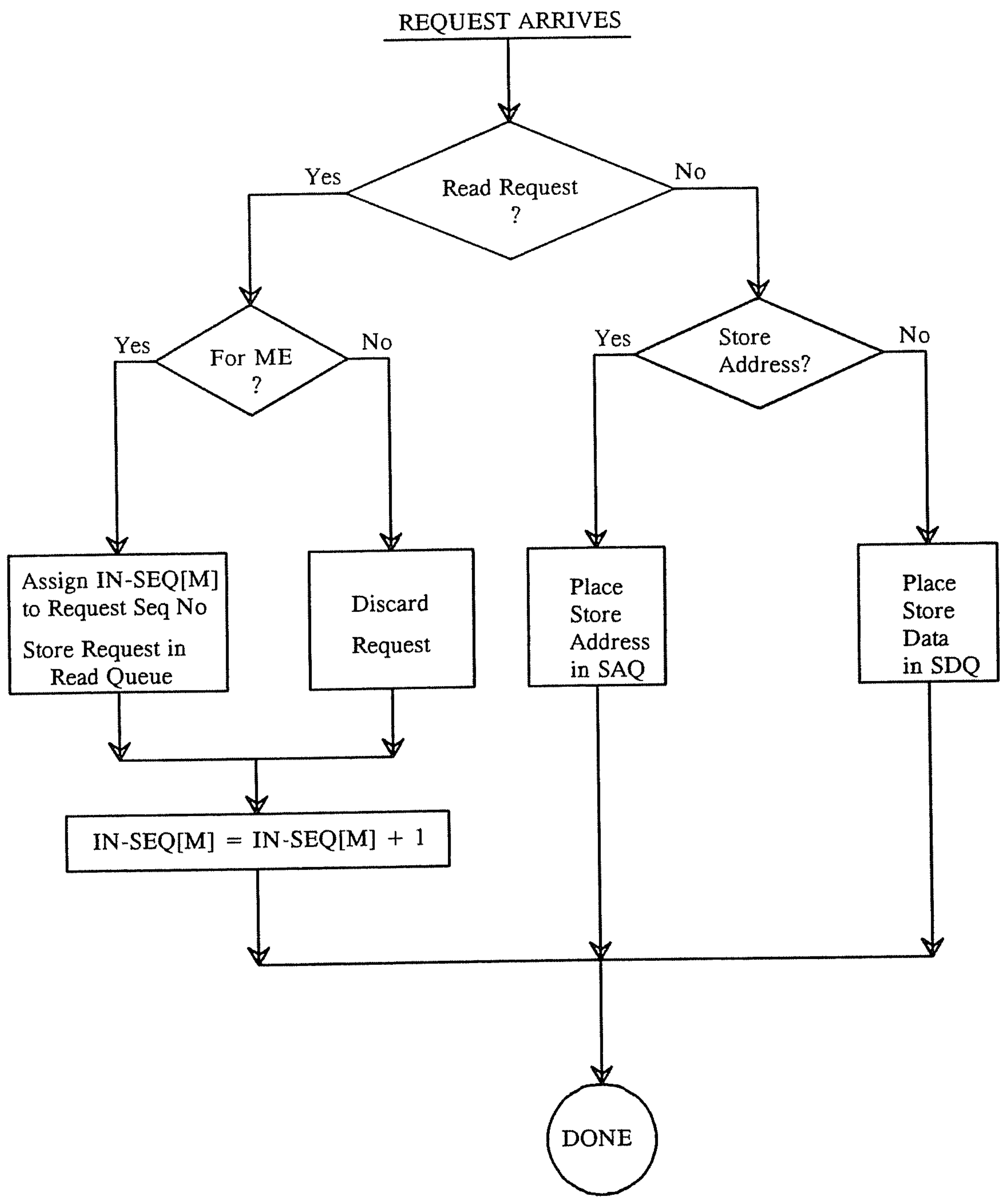

REQUEST ARRIVES
Read Request ?
Yes
No
For ME ?
Yes
No
Store Address?
Yes
No
Assign IN-SEQ[M] to Request Seq No
Store Request in Read Queue
Discard Request
Place Store Address in SAQ
Place Store Data in SDQ
IN-SEQ[M] = IN-SEQ[M] + 1
DONE

Figure 4.10 -- Input stage algorithm of memory system using Free-Module-Request-First scheduling policy.

broadcast to all the Module Controllers in the same clock period, the increments of all the IN-SEQ registers in all Module Controllers occur in the same clock period.

4.2.4.3. Distributed Store Address/Data Pairing at Issue Stage

When the Module Controller receives a store address that does not access itself, the Module Controller can either keep the store address or discard it. The decision depends on the method used to match the store address and the store data. If the Module Controller does not keep the store address, a counter must be used to indicate the number of store data that should be discarded between two correct pairings of store address and store data. This approach will require the Module Controller to maintain $N+1$ counters when there are N pending store addresses accessing itself. Alternatively, if the Module Controller stores all store addresses into the SA Queue (as in Figure 4.10), then the matching of store address and store data can be accomplished without maintaining a counter. The second approach is easier for the implementation and is used in the simulation model. The matching of the store address and the store datum is done by the issue stage and is described in Figure 4.11.

4.2.4.4. Distributed Output Sequence Control at Output Stage

During each clock period, the output logic of a Module Controller checks the MOS Table to determine if a memory operation is completing. The MOS Table provides the information of the type of operation that is completing and the number of clock periods left (from the BUSY shift register) before the operation

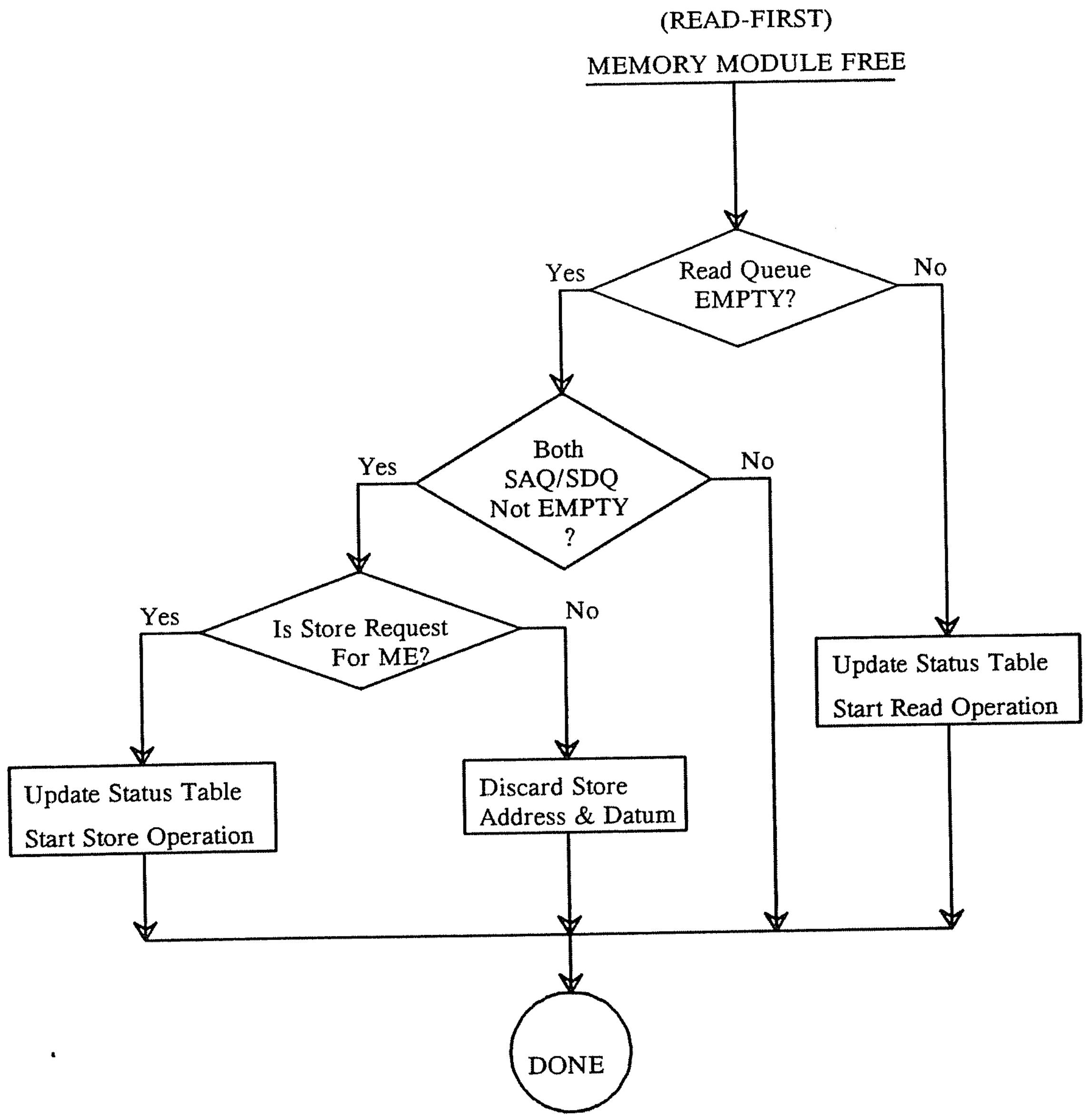


Figure 4.11 -- Issue stage algorithm of memory system using Free-Module-Request-First/Read-Request-First scheduling policy.

has completed. This status information allows the output logic (1) to compare the sequence number of the completing read operation with the next output sequence number and (2) to decide whether the fetched datum is to be sent directly out of the memory system or stored into the output queue. Since the read request within each Module Controller is serviced in the first-in-first-out order, the entry at the head of the output queue, if any, is the next waiting entry to be sent out of the Module Controller; otherwise, the output of the completing read operation is sent out. Therefore, each Module Controller can determine the next sequence number of a read request to be sent out of its module, and therefore bids for its turn to use the memory output bus. Because the sequence numbers are unique and the memory outputs are to be sent out in a pre-determined order, only one Module Controller can have the privilege of using the memory output bus in each clock period. Thus, bus collisions will never occur in this *distributed arbitration* method.

In the following discussions, the memory output of sequence number N will be called *memory output N*.

4.2.4.4.1. Interleaving the Memory Output Bus for Avoiding Delays

A Module Controller can deliver memory output $N+1$ only if it knows that memory output N has been delivered. When a Module Controller sends memory output N to the memory output bus, it has to notify other Module Controllers that memory output N has been delivered, so that the Module Controller that has memory output $N+1$ can send memory output $N+1$ out. Since it takes one clock to transmit a signal from one Module Controller to any other Module Controller, the Module Controller that has memory output *N+1* will not know that memory output *N* has been delivered until 2 clocks after memory output *N* is sent. Therefore, memory outputs will be sent through the memory output bus at the rate of *one per two clock periods.*

In order to send memory outputs at the rate of *one per clock period,* memory outputs are divided into two groups of even sequence numbers and odd sequence numbers. The Module Controller that sends memory output *2n* out, must notify other Module Controllers that memory output *2n* has been delivered; thus, the Module Controller that has memory output *2n+2* can send memory output *2n+2* out two clock periods after the memory output *2n* is sent. Therefore, the memory outputs of sequence numbers *2n, 2n+2, 2n+4, ...* are delivered through the memory output bus at the rate of one per two clock periods. Similarly, the memory outputs of sequence numbers *2n+1, 2n+3, 2n+5, ...* can be delivered through the memory output bus at the rate of one per two clock periods. By *interleaving* the use of the memory output bus between the memory outputs of even sequence numbers and the memory outputs of odd sequence numbers, memory outputs of sequence numbers *2n, 2n+1, 2n+2, 2n+3, 2n+4, 2n+5, ...* can be delivered through the same memory output bus at the rate of one per clock period. Care must be taken to assure that strict ordering is maintained. This is described in the following section.

4.2.4.4.2. Handshake Signals to Allow Continuous Output Flow

The idea of the output sequence control algorithm is that *only the next two memory outputs* can participate in bidding for the use of the output bus. In other words, memory output *n* can send a bidding signal for the use of the output bus only if memory outputs *n-2* and *n-3* have already sent bidding signals. Thus, for the memory outputs of even sequence numbers, memory output *2n* can send its bidding signal if memory output *2(n-1)* has already sent its bidding signal. The Module Controller cannot actually send memory output *2n* until it has seen the bidding signal for memory output *2n-1*. Similarly, for the memory outputs of odd sequence numbers, memory output *2n+1* can send its bidding signal if memory output *2(n-1)+1* has already sent its bidding signal. The Module Controller cannot actually send memory output *2n+1* until it has seen the bidding signal for memory output *2n*.

Ideally, a Module Controller could first send a bidding signal for each memory output, and then send the memory output through the memory output bus in the following clock period. Because two Module Controllers that have the next two memory outputs can send the bidding signals in the same clock period, both Module Controllers could send the next two memory outputs simultaneously in the following clock period if these two Module Controllers do not coordinate with each other. Thus, a restriction of sending memory outputs for the even and odd sequence numbers is needed in order to avoid bus contention between the even and odd memory outputs and assure correct ordering of operands.

In order to monitor the progress of memory outputs and participate in bidding for the use of the output bus, each Module Controller needs two different output sequence number registers: *SEQe* and *SEQo*, to track the next even and odd output sequence numbers, respectively. The least significant bit of a sequence number is not stored in the *SEQe* or *SEQo* registers, i.e., (*2n div 2*) is stored in *SEQe* if the next output sequence number for the the even group is *2n*, and (*(2n+1) div 2*) is stored in *SEQo* if the next output sequence number for the odd group is *2n+1*.

Let *SeqNo* be the sequence number of a memory output waiting to be sent out of a Module Controller, and $Seq = (SeqNo\ div\ 2)$. Let the boolean variables *AckEven* and *AckOdd* be the values of the following two boolean expressions:

$$AckEven = [(SEQo = Seq - 1)\ or\ (SEQo = Seq)\ or\ (SEQo = Seq + 1)]†,$$

and

$$AckOdd = [(SEQe = Seq)\ or\ (SEQe = Seq + 1)\ or\ (SEQe = Seq + 2)]†.$$

These two boolean variables are used to decide which two memory outputs are allowed to bid for the use of the output bus, and they ensure that *only the next two memory outputs* can participate in the bidding for the use of the output bus. An even *SeqNo* is one of the next two memory outputs if its computed *AckEven* value is true. Similarly, an odd *SeqNo* is one of the next two memory outputs if its computed *AckOdd* value is true. The Module Controller can send a bidding signal for an even *SeqNo*, only if ($Seq = SEQe$) and *AckEven* is true. Similarly, the Module Controller can send a bidding signal for an odd *SeqNo* only if ($Seq = SEQo$) and *AckOdd* is true. Then, in the subsequent clock periods, the memory output of even *SeqNo* can be sent out if its bidding signal has been sent ($SEQe = Seq + 1$) and *Condition E*: $[(SEQo = Seq)\ or\ (SEQo = Seq + 1)]$† is true. Similarly, the memory output of an odd *SeqNo* can be sent out only if its bidding signal has been sent ($SEQo = Seq + 1$) and *Condition O*: $[(SEQe = Seq + 1)\ or\ (SEQe = Seq +$

†Using modulo S arithmetic, where S is the maximum sequence number determined in the implementation of sequence control scheme.

2)]† is true. For an even *SeqNo*, *Condition E* is true if the bidding signal of (*SeqNo - 1*) has been received. For an odd *SeqNo*, *Condition O* is true if the bidding signal of (*SeqNo - 1*) has been received. *Condition E* and *Condition O* are used to ensure that the even memory output and the odd memory output will not be sent out in the same clock period. Thus, bus contention can never occur.

Two sequence control signals are needed to implement the distributed output sequence control: *ACKe* and *ACKo*. The sequence control signals, *ACKe* and *ACKo*, are used to bid for the use of the memory output bus and synchronize the increment of *SEQe* and *SEQo* registers, respectively. The sequence control signal is always sent at least one clock before the corresponding memory output is sent. It may be much earlier, if the sequence control signal for the previous memory output (in the sequence number ordering) is delayed. But the two sequence control signals, *ACKe* and *ACKo*, uniquely define the time slot when each memory output is sent. The algorithms of output sequence control for the even and odd sequence numbers are described in Figure 4.12.A and 4.12.B, respectively.

At the beginning of each clock period, the Module Controller checks in parallel for the occurrence of three events:

(1) the arrival of an *ACKe* signal,
(2) the arrival of an *ACKo* signal, and
(3) a memory output will soon be ready to be sent out.

The *SEQe* and the *SEQo* registers are incremented by one if (1) and (2) occur, respectively. If there is a memory output ready to be sent out and the memory output is one of the next two memory outputs, the Module Controller has to bid for its turn to use the memory output bus; otherwise, the Module Controller does

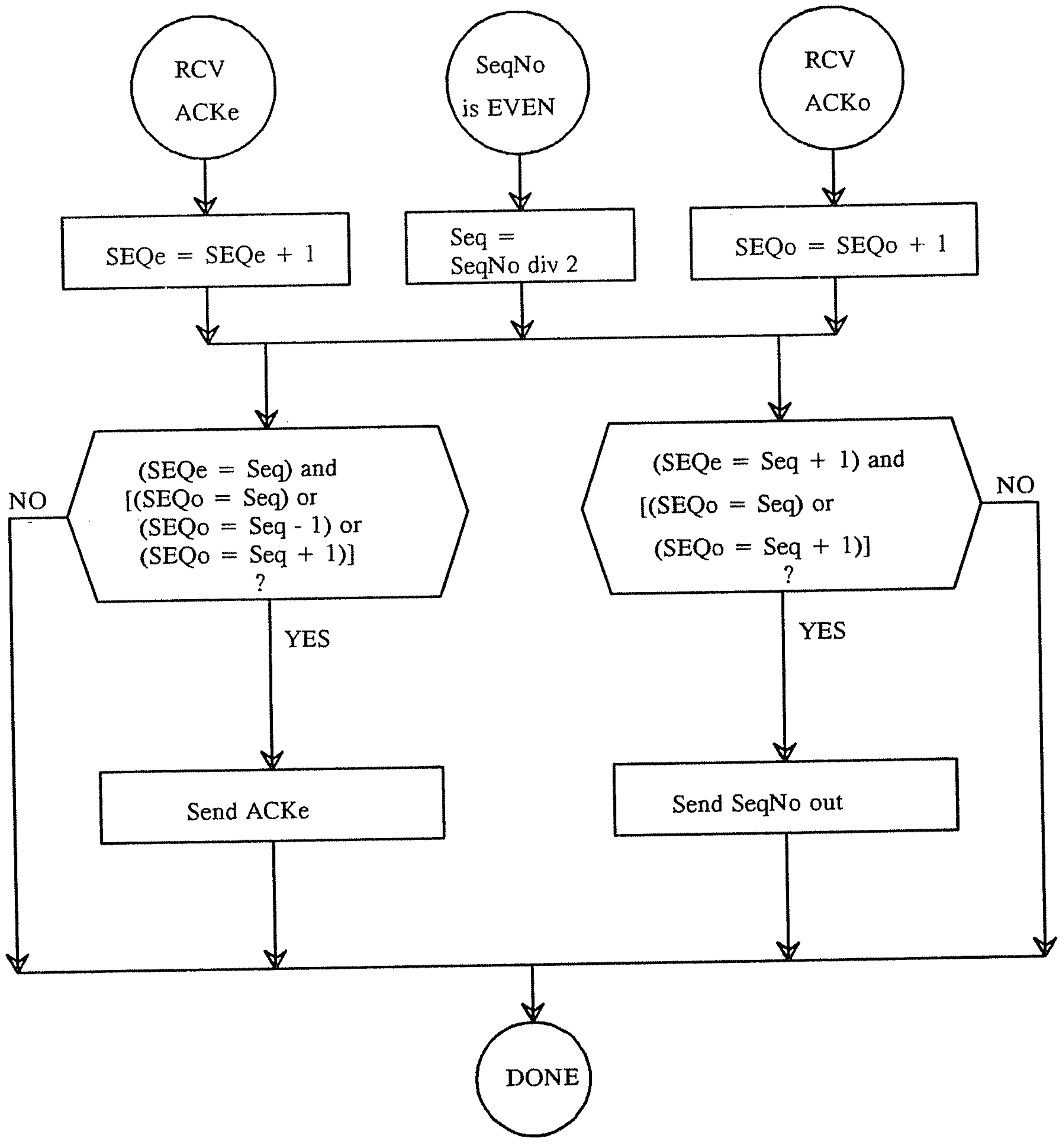

RCV ACKe
SeqNo is EVEN
RCV ACKo
SEQe = SEQe + 1
Seq = SeqNo div 2
SEQo = SEQo + 1
(SEQe = Seq) and [(SEQo = Seq) or (SEQo = Seq - 1) or (SEQo = Seq + 1)] ?
(SEQe = Seq + 1) and [(SEQo = Seq) or (SEQo = Seq + 1)] ?
NO
YES
NO
YES
Send ACKe
Send SeqNo out
DONE

Figure 4.12.A -- Output sequence control for sending a memory output of even sequence number. SeqNo is sequence number of the memory output to be sent out.

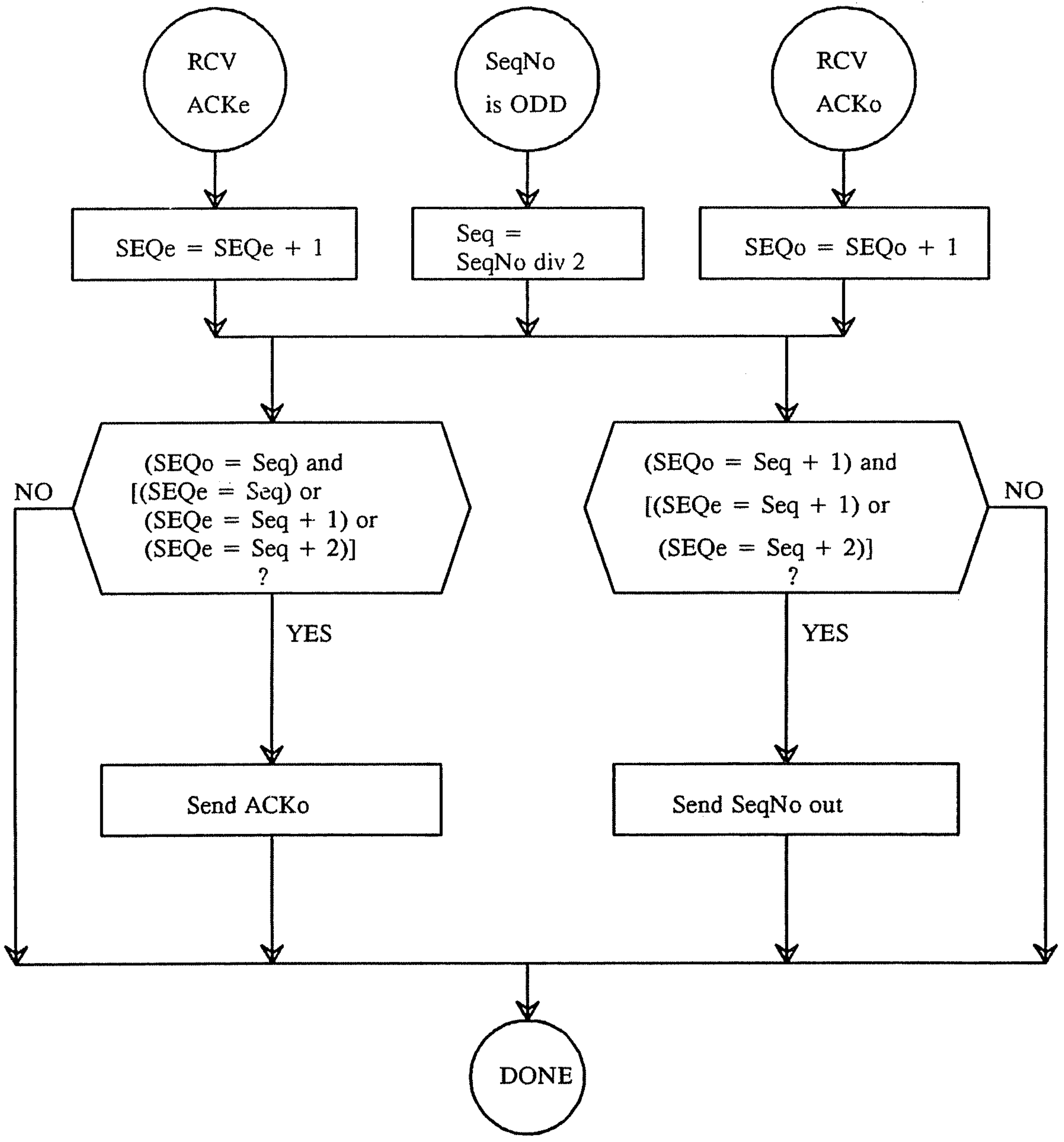

RCV ACKe
SeqNo is ODD
RCV ACKo
SEQe = SEQe + 1
Seq = SeqNo div 2
SEQo = SEQo + 1
(SEQo = Seq) and [(SEQe = Seq) or (SEQe = Seq + 1) or (SEQe = Seq + 2)] ?
(SEQo = Seq + 1) and [(SEQe = Seq + 1) or (SEQe = Seq + 2)] ?
NO
NO
YES
YES
Send ACKo
Send SeqNo out
DONE

Figure 4.12.B -- Output sequence control for sending a memory output of odd sequence number. SeqNo is sequence number of the memory output to be sent out.

nothing for the memory output in the current clock period.

Suppose the Module Controller is bidding for its turn to send an even memory output. As described in Figure 4.12.A, the Module Controller has to send an *ACKe* out. Then, on a subsequent clock period, the content of *SEQe* is incremented by one (it takes one clock period for a Module Controller to detect that a bidding signal was sent by itself, while it takes two clock periods for a Module Controller to detect that a bidding signal was sent by another Module Controller,) and the Module Controller can send the even memory output through the output bus when *Condition E* is true. Similarly, if the Module Controller is bidding for its turn to send an odd memory output, it has to follow the procedures described in Figure 4.12.B. The events and procedures described in Figures 4.12.A and 4.12.B are checked and performed once in every clock period. Because only one Module Controller can send an *ACKe* signal in any clock period, the increments of all *SEQe* registers among all Module Controllers are synchronized. Similarly, the increments of *SEQo* register are synchronized.

The memory output sequence control can be further explained by the following example.

Example 4.3 :

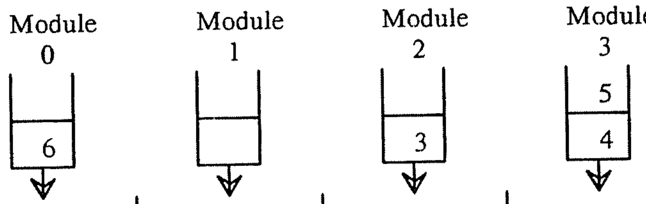
The memory system is 4-way interleaved. At clock 10, the memory outputs of sequence numbers 0 and 1 have been delivered, and memory output 3 is ready to be delivered; memory output 2 is being fetched from module 1 and it will not be ready until clock 12, while memory outputs 4, 5 and 6 are ready before clock 10. The memory output 3 is in the output queue of module 2, the memory outputs 4 and 5 are in the output queue of module 3, and the memory output 6 is in the output queue of module 0. Since the last memory output seen by all Module Controller is sequence number 1, the next two memory outputs are sequence numbers 2 and 3. The timing diagram which describes the sequence of delivering memory outputs is illustrated in Figure 4.13. In Figure 4.13, the symbol ACK3 indicates that the *ACKo* signal of memory output 3 is sent, and the symbol OUT3 indicates that the memory output 3 is sent. Other symbols can be explained similarly. The two numbers of the pairs in Figure 4.13 indicate the contents of registers *SEQe* and *SEQo* in each Module Controller. Initially, it is (1,1) in every Module Controller.

The sequence of memory output operations of the Module Controller for module 2 is as follows:

Clock 10 : Sends the *ACKo* for memory output 3.
Clock 11 : Increments *SEQo* by 1.
Clock 12 : Does nothing because *Condition O* is not true.
Clock 13 : Receives the *ACKe* of memory output 2, increments *SEQe* by 1, *Condition O* is true, sends out memory output 3.
Clock 14 : Does nothing.
Clock 15 : Receives the *ACKe* of memory output 4, increments *SEQe* by 1.
Clock 16 : Receives the *ACKo* of memory output 5, increments *SEQo* by 1.
Clock 17 : Receives the *ACKe* of memory output 6, increments *SEQe* by 1.

The sequence of memory output operations of the Module Controller for module 3 is as follows:

Clock 10 : Does nothing.
Clock 11 : Does nothing.
Clock 12 : Receives the *ACKo* of memory output 3, increments *SEQo* by 1,



TIME	Module 0	Module 1	Module 2	Module 3
10	(1,1) ---	(1,1) ---	(1,1) ACK3	(1,1) ---
11	(1,1) ---	(1,1) ACK2	(1,2) ---	(1,1) ---
12	(1,2) ---	(2,2) OUT2	(1,2) ---	(1,2) ---
13	(2,2) ---	(2,2) ---	(2,2) OUT3	(2,2) ACK4
14	(2,2) ---	(2,2) ---	(2,2) ---	(3,2) OUT4,ACK5
15	(3,2) ACK6	(3,2) ---	(3,2) ---	(3,3) OUT5
16	(4,3) OUT6	(3,3) ---	(3,3) ---	(3,3) ---
17	(4,3) ---	(4,3) ---	(4,3) ---	(4,3) ---

Figure 4.13 -- The timing diagram of sending memory outputs 2, 3, 4, 5, and 6 through the memory output bus. Memory outputs 3, 4, 5 and 6 are ready for delivery at clock 10, and memory output 2 will not be ready until clock 12. The two numbers of the pairs are the current values of *SEQe* and *SEQo* in each Module Controller.

Clock 13 : Receives the *ACKe* of memory output 2, increments *SEQe* by 1, sends the *ACKe* for memory output 4. (In this clock period, both the *ACKe* and *ACKo* for memory outputs 4 and 5, respectively, can be sent according to the algorithm. If the Module Controller for module 3 sent the *ACKe* and *ACKo* in this clock period, it could detect both signals and send memory outputs 4 and 5 simultaneously in clock 14. Thus, it is necessary to restrict that a Module Controller can only send either an *ACKe* or an *ACKo* according to the order of memory outputs in a clock period. Therefore, the *ACKo* of memory output 5 cannot be sent in this clock period.)

Clock 14 : Increments *SEQe* by 1, sends out memory output 4, sends the *ACKo* for memory output 5.

Clock 15 : Increments *SEQo* by 1, sends out memory output 5.

Clock 16 : Does nothing.

Clock 17 : Receives the *ACKe* of memory output 6, increments *SEQe* by 1.

The memory output sequences in the other Module Controllers can be explained similarly. At clock 17, the contents of *SEQe* and *SEQo* in all Module Controllers are (4,3). The (4,3) indicates that next memory output is for sequence number 7, and memory outputs 7 and 8 can bid for the use of the output bus.

4.2.4.4.3. State Transition Diagram of Output Sequence Control

The state transition diagram which describes the action of merging memory outputs through the output bus is shown in Figure 4.14. An input-symbol/output-symbol notation is associated with each transition arrow to indicate the occurrence of actions between each state transition. The input-symbol indicates the memory output that comes to bid for the output bus, and the output-symbol indicates the type of signal sent out. The occurrence of output actions is a function of the present state, and the occurrence of input actions determines the

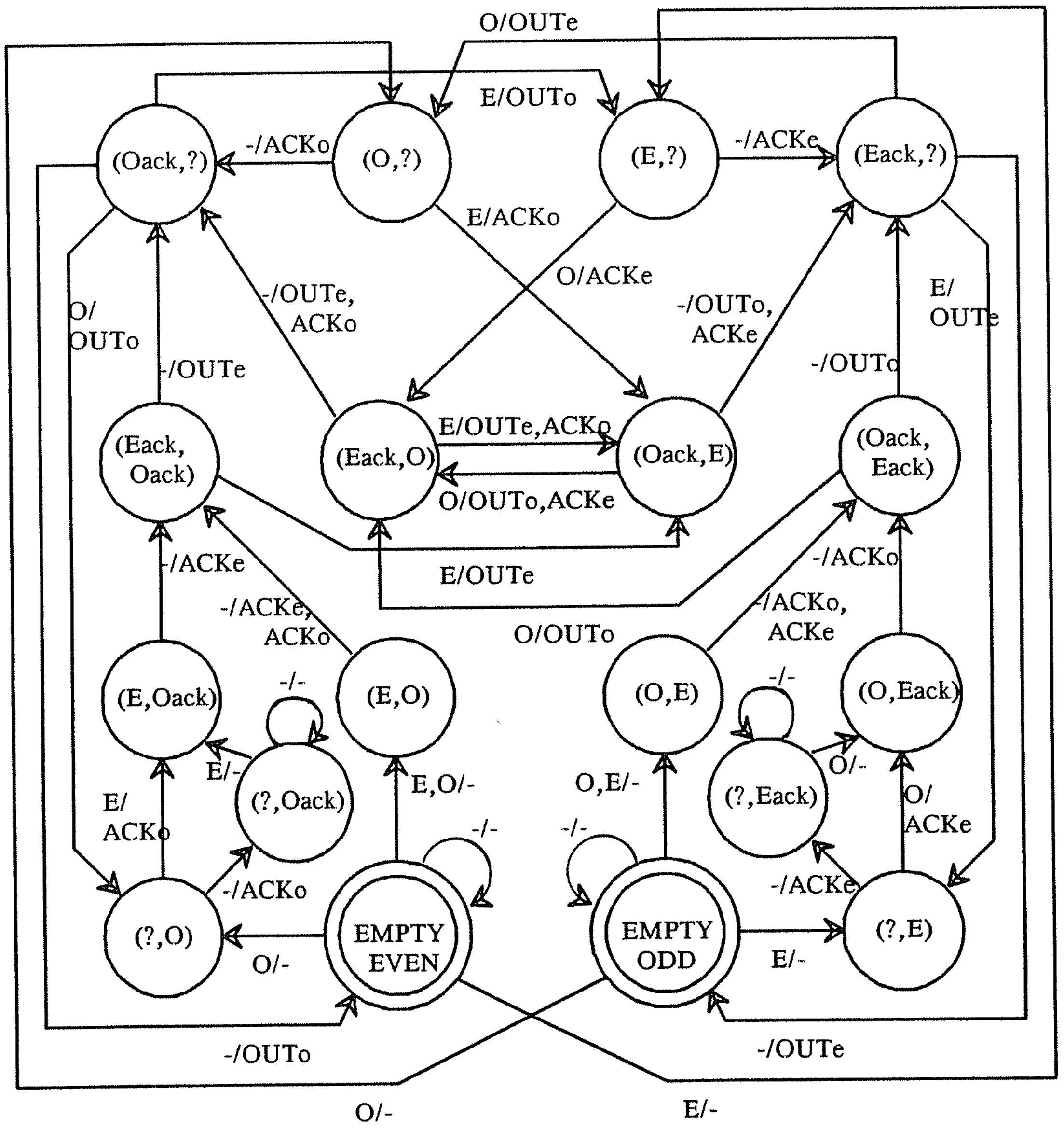


Figure 4.14 -- State transition diagram of memory output sequence control for Free-Module-Request-First scheduling policy.

next state where the transition is made. For example, the output signal *ACKe* will be sent out if the present state is *(E,?)*, while the input action determines whether the next state is *(Eack,?)* or *(Eack,O)*. If an odd memory output (O) comes to bid for the use of the memory output bus, then the next state will be *(Eack,O)*; otherwise, the next state will be *(Eack,?)*. The '-' input symbol indicates that no input or output action occurs. The meanings of the input symbols, the output symbols, and the state symbols are described in the following paragraphs.

In the following discussion, the next two memory outputs that can bid for the use of the output bus are termed the *first output* and the *second output*, respectively. For example, at clock 10 of *Example 4.3*, the *first output* is memory output 2, and the *second output* is memory output 3.

The input symbols are:

O : a memory output of odd sequence number comes to bid for the output bus.
E : a memory output of even sequence number comes to bid for the output bus.

The output symbols are:

ACKo : the *ACKo* signal is sent either for an odd memory output.
ACKe : the *ACKe* signal is sent either for an even memory output.
OUTo : the memory output of odd sequence number is sent.
OUTe : the memory output of even sequence number is sent.

The state symbols are:

Empty-Even :	the next two memory outputs are not yet ready and the *first output* is even.
Empty-Odd :	the next two memory outputs are not yet ready and the *first output* is odd.
(E,O) :	the next two memory outputs are ready, the *first output* is even and the *second output* is odd.

(O,E) :	the next two memory outputs are ready, the *first output* is odd and the *second output* is even.
(?,O) :	the *first output* is even but not yet ready, and the *second output* is odd and ready.
(?,E) :	the *first output* is odd but not yet ready, and the *second output* is even and ready.
(O,?) :	the *first output* is odd and ready, and the *second output* is even but not yet ready.
(E,?) :	the *first output* is even and ready, and the *second output* is odd but not yet ready.
(?,Eack) :	the *ACKe* of the even *second output* is sent and the odd *first output* is not yet ready.
(?,Oack) :	the *ACKo* of the odd *second output* is sent and the even *first output* is not yet ready.
(Eack,?) :	the *ACKe* of the even *first output* is sent and the odd *second output* is not yet ready.
(Oack,?) :	the *ACKo* of the odd *first output* is sent and the even *second output* is not yet ready.
(Eack,O) :	the *ACKe* of the even *first output* is sent and the odd *second output* is ready.
(Oack,E) :	the *ACKo* of the odd *first output* is sent and the even *second output* is ready.
(E,Oack) :	the even *first output* is ready and the *ACKo* of the odd *second output* is sent.
(O,Eack) :	the odd *first output* is ready and the *ACKe* of the even *second output* is sent.
(Eack,Oack) :	the *ACKe* and the *ACKo* for both the even *first output* and the odd *second output* are sent.
(Oack,Eack) :	the *ACKo* and the *ACKe* for both the odd *first output* and the even *second output* are sent.

Since the first sequence number is 0, the output sequence control starts from the *Empty-Even* state and transfers to three different states: *(E,?)*, *(?,O)*, and *(E,O)*, depending on the three possible combinations of input symbols. It returns and stops at the *Empty-Even* or the *Empty-Odd* state when there are no more memory outputs to be delivered, and the next memory output is even or odd, respectively. For example, the state transition sequence for *Example 4.3* can be explained as follows:

Assume that (1) memory outputs 0 and 1 are delivered before clock 8, (2) memory outputs 4, 5, and 6 are ready before clock 10, (3) memory output 3 is ready at clock 10, and (4) memory output 2 is ready at clock 11.

Clock 8 :	at *Empty-Even* state.
Clock 9 :	memory output 3 comes to bid for the output bus, moves to state *(?,O)*.
Clock 10 :	*ACKo* is sent for memory output 3, memory output 2 comes to bid for the output bus, moves to state *(E,Oack)*.
Clock 11 :	*ACKe* is sent for memory output 2, moves to state *(Eack,Oack)*.
Clock 12 :	memory output of 2 is sent, memory output 4 comes to bid for the output bus, moves to state *(Oack,E)*.
Clock 13 :	memory output of 3 is sent, *ACKe* of memory output 4 is sent, memory output 5 comes to bid for the output bus, moves to state *(Eack,O)*.
Clock 14 :	memory output of 4 is sent, *ACKo* of memory output 5 is sent, memory output 6 comes to bid for the output bus, moves to state *(Oack,E)*.
Clock 15 :	memory output of 5 is sent, *ACKe* of memory output 6 is sent, no new memory output is ready, moves to state *(Eack,?)*.
Clock 16 :	memory output of 6 is sent, no new memory output is ready, moves to state *Empty-Odd.*

4.2.4.4.4. Output Sequence Control Between Instruction and Data Fetches

The sequence control scheme for the instruction fetch request requires another set of input sequence number registers (*IN-SEQi*), output sequence number registers (*SEQie* and *SEQio*) and output control signals (*ACKie* and *ACKio*) for sending the fetched instructions. The method of synchronizing the increment of instruction sequence numbers among all Module Controllers is the same as the method described in previous sections.

When an IF request is broadcast to all Module Controllers, only four Module Controllers (those having the four requested instruction words) will fetch the instruction words. A Module Controller can be servicing a LA request when it receives the IF request; thus, all the four Module Controllers may not fetch the four instruction words for an IF request at the same time. The Module Controllers must coordinate among themselves in order to deliver the instruction words in sequence.

Each Module Controller can be in one of these two states: *instruction output state* or *data output state*. The Module Controller can send a fetched instruction only if it is in the *instruction output state*, and can send a fetched datum if it is in the *data output state*. Initially, every Module Controller is in the *data output state*. The Module Controller changes to the *instruction output state* when either itself or another Module Controller is about to send a fetched instruction, and then changes back to the *data output state* when no more fetched instructions are to be delivered.

A *Preempt* signal line, which is an OR-line, is used to indicate whether the Module Controller should be in the *instruction output state* (when the *Preempt* line is high) or in the *data output state* (when the *Preempt* line is low). Thus, the

Preempt signal line is high as long as one Module Controller raises it, and it is low when all Module Controllers drop it.

The Module Controller must raise the *Preempt* signal line when it is preparing to deliver an instruction, and it must drop the *Preempt* signal line when it is not sending any instruction words. When the Module Controller detects a high *Preempt* signal line, all processes sending fetched data must be stopped until the *Preempt* signal line is dropped. Thus, all Module Controllers can change to the *instruction output state* after the *Preempt* signal line is raised.

After the *Preempt* signal line is raised, the instruction *ACKie* or *ACKio* signal (depending on whether or not it is of even sequence number or odd sequence number) is sent in the next clock. Then, the fetched instruction is sent according to the output sequence control algorithm described earlier. Because of the one clock bus delay time, the *Preempt* signal line must be raised two clocks before the instruction fetch operation is completed. Thus, the fetched instruction can be delivered without delay. Also, by raising the *Preempt* signal line two clocks before an instruction fetch is delivered, it allows all Module Controllers to change from the *instruction output state* to the *data output state* before an instruction fetch is sent through the output bus. Thus, simultaneous delivering of fetched instruction and data from different Module Controllers will not occur, and bus collisions between the instruction and the data fetches are avoid.

The Module Controller can compare the value of the next input sequence number of instruction fetches (from the content of the *IN-SEQi* register) and the value of the next output sequence number of the fetched instructions (from the contents of *SEQie* and *SEQio* registers) to determine if all the instruction fetches have been delivered. If the values of these two sequence numbers are the same, then there are no outstanding instruction fetch requests to be completed, and the Module Controller should drop the *Preempt* signal line. After all the Module Controllers have dropped the *Preempt* signal line, the Module Controllers change to the *data output state* simultaneously.

This output control mechanism can cause the delivery of fetched data to be delayed by one clock (the bus delay for the last Module Controller to drop the *Preempt* signal line) for each occurrence of a preemption. Except when the last instruction fetch and the next output of data fetch are located in the same Module Controller, the Module Controller knows that the last instruction fetch has been sent. Thus, the next output of data fetch can be sent in the following cycle without delay.

4.3. Summary and Discussion

The memory request scheduling strategy must answer two questions: (1) among all free memory modules, which modules are to be scheduled for memory operations? (2) among request types within a memory module, which request is to be serviced next? Addressing the first question, the *Free-Module-Request-First* scheduling policy was proven to minimize the total service time for the requests to the same queue. Addressing the second question, the alternatives of dividing request types were proposed and the priority of request types were established analytically. It was concluded that the instruction fetch request and the load data request must be stored in separate queues in order to avoid deadlock. This, however, aggravates the memory access hazard problem with which chapter 6 will deal. The order of priority for request types should be: (1) the instruction fetch request, (2) the load address request, and (3) the store address request. If the First-Come-First-Serve policy is used, then the load address requests can be divided into the LA request for the Access Processor and the ALA request for the Execute Processor. The LA request for the Access Processor should be given a higher priority than the ALA request for the Execute Processor. If the Free-Module-Request-First policy is used, the LA and ALA requests will be considered as the same type of load address requests.

By storing the load data and the store address requests in separate queues, the Memory Controller must check for memory access hazards before a memory request can be serviced.

A pipelined memory system for the First-Come-First-Serve and the Free-Module-Request-First scheduling policies has been outlined. Memory outputs fetched by the Memory Controller using First-Come-First-Serve are serviced in the same order as their original request arrival. Thus, operands can be delivered to the processors without resequencing. The use of Free-Module-Request-First policy requires a sequence control scheme for delivering memory outputs in their original arrival sequence. This sequence control scheme can be used to track cache hits and misses. Thus, the sequence control required by a data cache can be implemented on the Memory Controller almost for free. The data cache with the ability to detect access hazard and short-circuit Read-After-Write data is designed and evaluated in chapter 6.

In the next chapter, the performance difference between the First-Come-First-Serve and the Free-Module-Request-First scheduling policies and the performance improvement of storing load address and store address requests in separate queues are evaluated through trace-driven simulations. The memory access hazard detection and resolution scheme using an associative search technique is assumed in the simulation models investigated in chapter 5.

CHAPTER 5

Simulation and Evaluation of Memory Models

Trace-driven simulations in a closed-loop model of the PIPE architecture are used to investigate the scheduling policies and the design features considered in previous chapters and to evaluate the performance of different memory system models. The trace files, generated from the selected benchmark programs, must produce heavy workloads on the memory system in order to evaluate the performance of memory models under heavy workloads. The impact of memory access hazards on the performance of the memory system of a decoupled architecture is important and is a main concern for the remainder of the dissertation.

5.1. The Simulation Model

The timing determinants which are critical in the accurate measurement of the performance of a memory model are: (1) the memory request arrival rate and (2) the operand consumption rate. The characteristics of a memory reference pattern, such as memory access hazards, can also influence the simulation results. All these factors can only be modeled precisely in a closed-loop simulation model.

The closed-loop model in Figure 5.1 consists of the PIPE processor simulators[1] and memory system simulator. The trace file that is fed into the closed-loop model is generated by object code executed by the PIPE architecture interpreter[2]. Because the interpreter has no notion of time and the timing measurements of the interlock control of CPU pipeline stages, instruction cache misses, completion of memory requests, etc., the trace file will not reflect these characteristics. Instead, the trace file contains a scenario of instructions issued and the memory reference sequences of a program execution.

The interplay between the request arrival rate and the operand consumption rate will determine the results obtained from different scheduling policies. The closed-loop model assumes that, on average, the arrival rate is equal to the service rate. If the request arrival rate is high and the operand consumption rate is also high, the probability of more than one request waiting for service will be high. Conversely, if the request arrival rate is high, but the operand consumption rate is low, the flow control scheme will frequently stop the processors from sending requests to the memory system. Thus, the overall request arrival rate will be reduced. Obviously, if the request arrival rate is low, the Memory Controller will be frequently idle and the performance difference among scheduling policies will be minimal. Therefore, the greater the number of requests waiting for service simultaneously, the more distinctive the performance of each scheduling policy.

The frequency of occurrence of access hazards depends on the number of pending memory requests; the more pending memory requests the higher

[1]A software tool developed by Mr. Jian-Tu Hsieh.

[2]A functional interpreter for the PIPE architecture developed by Mr. P.B. Schechter.

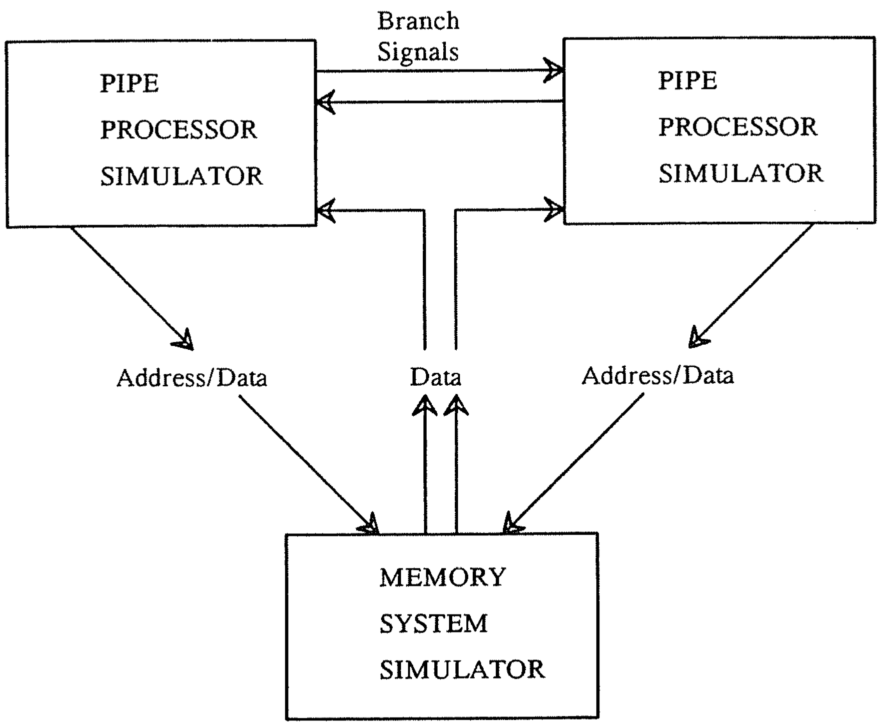


Figure 5.1 Closed-Loop Simulation Model of PIPE Architecture

probability of access hazards. Because the processor speed is more affected by the memory bandwidth than by the latency [WeSm84], access hazards are more likely to occur with slower memory service times. Since the occurrence of a memory access hazard depends on whether or not the fetch and store to the same location pending simultaneously, it is necessary to use a closed-loop model to simulate the computation steps—with precise timing of the memory access sequence. Thus, the impact of memory access hazards can be evaluated.

5.2. Selection of Application Programs for Trace-Driven Simulations

There are two considerations in selecting application programs for generating trace files:

(1) The decoupled code should allow the Access Processor to run ahead of the Execute Processor, such that continual memory requests can arrive at the Memory Controller.

(2) For the purpose of examining the Read-After-Write and Write-After-Read problems, the trace files should create realistic memory access hazards during decoupled computation.

The first twelve loops of benchmark programs from the Lawrence Livermore National Laboratories [McMa72, RiSc84] (LLL loops), were used in the simulation study. A listing of these twelve LLL loops translated into Pascal programs is in Appendix I. For the purpose of evaluating the performance of memory models under heavy workloads and studying the impact of memory access hazards on the performance of memory models, the decoupled codes for these twelve LLL loops were hand-optimized in order to generate trace files that satisfy the above considerations.

Since floating point arithmetic operations have not been defined in the PIPE architecture, the LLL loops selected for simulation study are modified to contain no floating point operations. The existence of floating point operations will only increase the computation time of the Execute Processor. It may increase the occurrence of access hazards, but will not change the memory access patterns issued from the Access Processor. Since the instruction execution speed of the Execute Processor can be controlled by the PIPE processor simulator, access hazards that would have been created by the floating point operation can be simulated by increasing the instruction execution time of the Execute Processor. Thus, the nonexistence of floating point operation is not a major concern in the simulation study.

By effective utilization of the Prepare-To-Branch and the load/store memory access schemes of the PIPE architecture, most decoupled codes for these twelve LLL loops can be generated such that the Access Processor can run ahead of the Execute Processor and overflow one of the request buffers.

In order to evaluate the impact of memory access hazards, two sets of trace files are created from the first 12 LLL loops: (1) files containing access hazards[3] and (2) files containing no access hazards. These two sets of trace files are used to compare the performance of memory models when there are access hazards and when there are no access hazards. There are two types of potential Read-After-Write access hazards: (a) among the array references within a loop and (b) among the array references between loop iterations. Both types of access hazards can be removed by saving the needed operands in the registers until the operands are no longer used. For the trace files of set (1), the potential access hazards of type (a) are removed, but type (b) remains, reflecting state-of-the-art compiler techniques.

5.3. Assumptions and Parameters of Simulation Models

The unit of simulation time is the CPU cycle, and various interleaving factors from from 4 to 64 are used. It is assumed that the memory bank busy time is

[3]There are potential Read-After-Write access hazards in LLL loops 4, 5, 6 and 11.

10 CPU cycles. Thus, at the issue stage of the Memory Controller, it requires 10 CPU cycles to complete a memory operation, and the memory operations can be started at the rate of one per 10 CPU cycles per memory module. Also, at least one CPU cycle is required at the input stage. The delay at the output stage varies. For the First-Come-First-Serve scheduling policy, the fetched operand is delivered to its Load Data Queue if the queue is not full. Otherwise, the fetched operand is stored in the output queue and remains there until its Load Data Queue is ready to receive the operand. For the Free-Module-Request-First scheduling policy, the fetched operand for a completing read operation is sent out of the memory system to its destination queue if the Load Data Queue is not full and the sequence number of the fetched operand is the same as the next output sequence number. Otherwise, the fetched operand is held in the output stage until it can be dispatched from the memory system. Thus, the minimum service time for a memory request is 11 CPU cycles.

It is assumed that it takes one CPU cycle to send a request to the Memory Controller, and one CPU cycle to deliver a fetched operand to the processor. The transmission time on the input and the output buses are the same regardless of the interleaving factors. Thus, the processors and the Memory Controller can send flow control signals to each other in one CPU cycle. However, it takes one CPU cycle to send it, and one more to recognize it.

A *Start*-and-*Stop* flow control scheme is used throughout the simulations. The PIPE processor sends requests to the Memory Controller until a *Stop* control signal is received from the Memory Controller and resumes sending requests when a *Start* control signal is received. Similarly, the Memory Controller has to deliver fetched operands to the processors according to the control signals received from the destination processors.

In order to correctly measure the performance of request scheduling policies, the simulation models will use a queue size that is large enough so that performance degradation due to insufficient queue spaces can be avoided. Therefore, the performance evaluation will not be biased. From chapter 3, we know that the performance of decoupled computation will not increase after the queue size is larger than eight. Therefore, it is assumed here that the size of Load Data Queue on the PIPE processors is eight, and the request queue size on the Memory Controller is 32, such that the probability of request queues become full is low.

In order to make a fair comparison of priority scheduling policies, it is assumed that the PIPE processor has an instruction cache size that is large enough to contain the entire loop body of every Lawrence Livermore Loop (actually, the hand-optimized decoupled codes of each of the twelve LLL loops can be stored into the 128-byte instruction cache of a PIPE processor.) Thus, simulation results will not be skewed by the nondeterministic instruction cache misses. A cache miss is assumed only the *first* time that any cache line is referenced.

5.4. A Method for Performance Evaluation

A *contention-free* memory system is used as the basis for evaluating the performance of different memory models. In the contention-free memory system, each address constitutes a separate interleaved memory module, so that memory conflicts occur only if the requests are accessing the same memory location (i.e. an Read-After-Read access). The pipeline organization of the contention-free memory system is the same as the pipeline organization of the memory models defined in chapter 4. There are two input buses to receive memory transactions from the Access Processor and the Execute Processor, respectively, so that the contention-free memory system can receive one input from each processor in the same clock cycle. Each memory request will take a fixed time (10 CPU cycles) to be completed, and the time used on each pipeline stage is the same as the parameters defined in the previous section. Only one memory request can occur per CPU cycle, and only one operand needs to be returned; thus, only one memory output bus is needed.

It is assumed that the contention-free memory system can resolve access hazards instantly. In other words, if a hazard is detected because the write address is waiting for the write data to arrive, the read request can be processed ahead of the write address request *before the write datum arrives.* In addition, a write request can be processed immediately after its write datum arrives. If there are two read requests to the same memory location, these two requests can be serviced in consecutive clock periods. Based on these assumptions, memory conflicts will never occur and there is no penalty for access hazards in the contention-free memory system. Consequently, the memory requests can be serviced *at the rate of their arrival.* Although this assumption allows incorrect sequencing of memory operations, it can be used as a scale for measuring the performance of different memory models under the workload defined by the trace files, and used as an index to evaluate the performance degradation due to memory access hazards. The performance of a memory model will be normalized by the following equation:

$$Performance = \frac{Total\ Simulation\ Time\ of\ Contention-Free\ Memory\ System}{Total\ Simulation\ Time\ of\ the\ Memory\ System\ being\ Measured}$$

The simulation time is measured by executing the trace file in the closed-loop model. The memory system of the closed-loop model is either a contention-free memory system or a memory model for which a request scheduling policy is to be evaluated.

5.5. Performance Evaluation of Memory Models

In chapter 4, it was shown that the Free-Module-Request-First scheduling policy can minimize the total service time for requests to the same queue. Also, it was concluded that the performance of a memory system can be improved if the load address and the store address requests within a module are stored in separate request queues.

The goals of this simulation study are: (1) to quantify the performance improvement of the Free-Module-Request-First scheduling policy over the First-Come-First-Serve scheduling policy, (2) to evaluate the performance improvement obtained through separating load data and store address requests, and (3) to evaluate the performance degradation of a memory model due to Read-After-Write access hazards. If the performance improvement of (2) is insignificant, load and store requests can be stored in a single request queue; thus, access hazard detection would not be needed. It will be shown that the performance degradation of (3) is significant, so a short-circuiting technique is considered, and the speedup through short-circuiting Read-After-Write requests will then be evaluated.

Memory models with four request scheduling policies are studied:

(1) FCFS/1Q scheduling policy: First-Come-First-Serve, one Load/Store Address Queue (LA/SA Queue) is used to store all the load address and the store address requests.

(2) FCFS/RRF scheduling policy: First-Come-First-Serve/Read-Request-First, the LA/SA Queue is divided into the Load Address Queue (LAQ) and the Store Address Queue (SAQ); requests from the LAQ are given higher priority than the requests in the SAQ.

(3) FMRF/1Q scheduling policy: Free-Module-Request-First, each Module Controller has one LA/SA Queue for storing load address and store address requests that are accessing the memory module.

(4) FMRF/RRF scheduling policy: the combination of the Free-Module-Request-First policy and the Read-Request-First policy, the LA/SA Queue within each Module Controller is divided into an LAQ and an SAQ; requests from the LAQ are given higher priority than the requests from the SAQ.

For the FCFS/RRF scheduling policy and the FMRF/RRF scheduling policy, the store address requests are serviced only when the LAQ is empty or the Load Data Queue on the processor is full. For the trace files of the first LLL loops, the Access Processor does not issue LA requests for itself. Thus, the LA and ALA requests are considered as the same type of load address request in the FCFS/RRF and FMRF/RRF scheduling policies.

For easy implementation in matching the store address/datum, two Store Data Queues (SDQ) are used to hold the store data from the Access Processor and the Execute Processor, respectively. Thus, a SA request can find its store datum correctly from its corresponding SDQ. For the FCFS/1Q and the FCFS/RRF scheduling policies, there is one pair of SDQs for the entire Memory Controller. For the FMRF/1Q and the FMRF/RRF scheduling policies, every Module Controller has a pair of SDQs to store all the store data received. The store data will

be paired with the addresses in the SAQ and a decision will be made as to whether or not they belong to the receiving module.

Since the requests are placed in the queues according to the order of their arrival (so that they are serviced in arrival sequence) the control logic for the FCFS/1Q policy is very simple. Access hazard checking is not necessary in the request selected for the next service. In addition, the memory outputs are in the same order as their original arrival sequence. Thus, the FCFS policy does not require sequence control.

Hazard detection and correct sequencing of memory operations should be done when load and store requests are stored into the LAQ and the SAQ, respectively. The presence of request queues with fully associative search capability for detecting memory access hazards is assumed. Whenever an access hazard is detected, memory operations are issued according to their arrival sequence. In this way the performance degradation due to access hazards can be evaluated.

In order to measure precisely the performance of a request scheduling policy, the performance difference resulting from the input multiplexing schemes[4] have to be minimized. The same input buses as in the contention-free memory system are used, and it is assumed that the memory system can receive one input from each processor in each CPU cycle. Different interleaving factors are used in simulations to study the performance degradation due to memory conflicts.

[4]In chapter 3, the simulation results showed that the performance degradation due to the use of an efficient multiplexing scheme, such as the round-robin or EP-First multiplexing scheme, is insignificant.

5.5.1. Evaluation of Memory Models with Read-After-Write Hazards

The simulation results using the set of trace files with Read-After-Write hazards are in Table 5.1 and 5.2, and the performance of the four request scheduling policies are plotted in Figure 5.2. The results showed that the FMRF/RRF scheduling policy offered the best performance. The performance of

Total Simulation Times of Memory Models with RAW Hazards in the Trace Files					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF/RRF	106,793	95,293	92,900	92,458	92,282
FMRF/1Q	144,969	128,438	111,769	109,283	108,565
FCFS/RRF	148,678	128,525	96,793	93,397	93,084
FCFS/1Q	180,370	157,988	116,138	111,666	110,777

Table 5.1 -- Total simulation time of the first 12 Lawrence Livermore Laboratory loops on four memory models. The set of trace files that have Read-After-Write access hazards are used in the simulations.

Performance of Memory Models Measured with RAW Hazards in the Trace Files					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF/RRF	0.541	0.606	0.622	0.625	0.626
FMRF/1Q	0.398	0.450	0.517	0.528	0.532
FCFS/RRF	0.388	0.449	0.597	0.618	0.620
FCFS/1Q	0.320	0.365	0.497	0.517	0.521

Table 5.2 -- Performance of the memory models using the first 12 Lawrence Livermore Laboratory loops. The set of trace files that have Read-After-Write access hazards are used in the simulations.

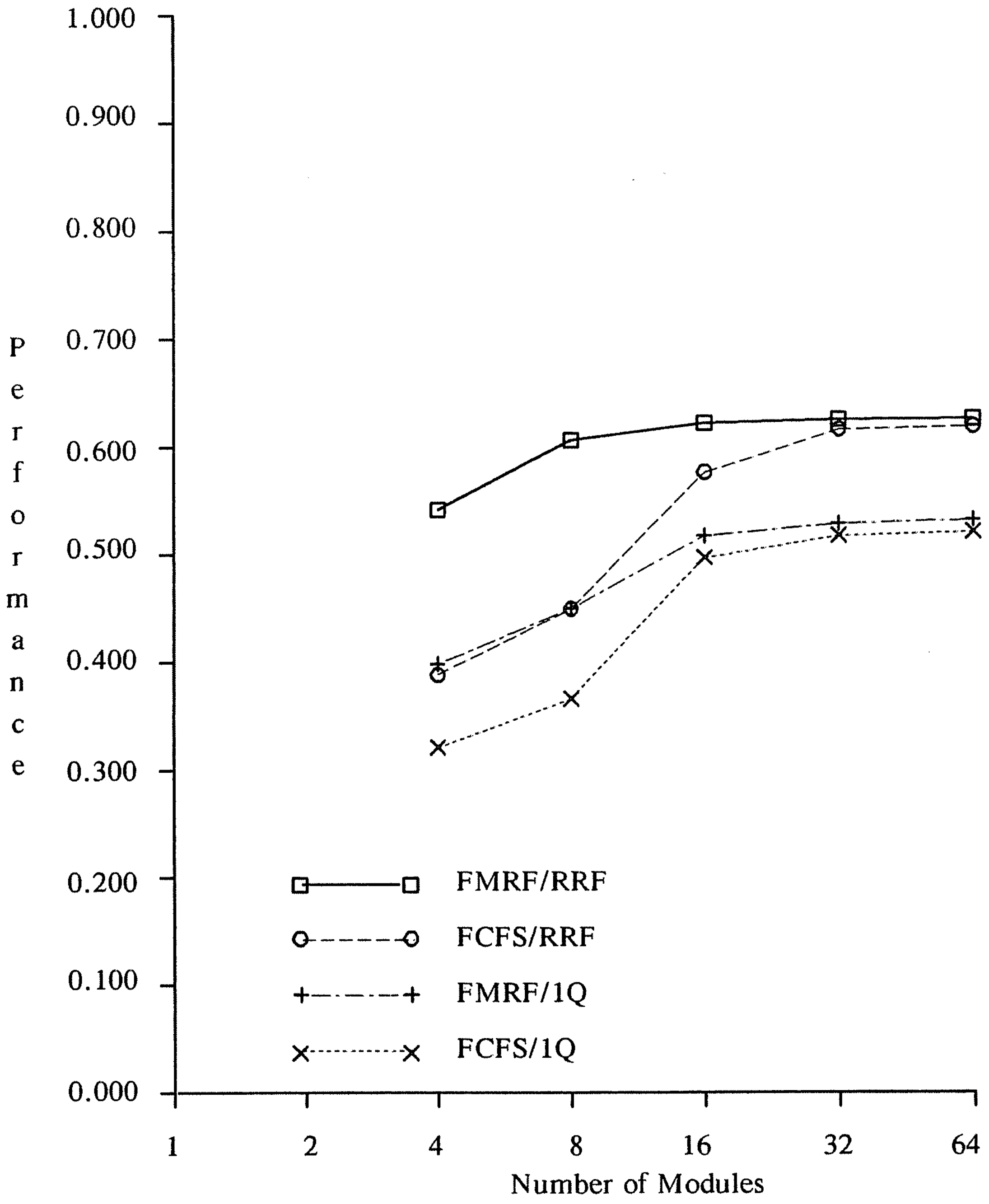


Figure 5.2 -- Performance comparison of memory models using the set of trace files where Read-After-Write hazards exist.

FCFS/RRF scheduling policy approximated the performance of FMRF/RRF policy when the interleaving factor was larger than 32 where memory conflicts did not occur frequently. When the interleaving factor was small and there were frequent memory conflicts, the Free-Module-Request-First scheduling policy could reduce the penalty of memory conflicts and utilize the memory modules more efficiently than the First-Come-First-Serve policy. When the interleaving factor was large, the RRF policies performed better than the 1Q policies. This indicated that the load address and the store address requests should be separated into different queues when the number of memory conflicts was small.

However, the maximum performance of the FCFS/RRF or the FMRF/RRF request scheduling policies was only about 60% of the contention-free memory system. This implies that memory access hazards significantly affect performance.

Further analysis of the trace-driven simulation results suggested that the Read-After-Write hazard among the memory requests within the trace files was the main factor that reduced the performance of every request scheduling policy. Since the memory access hazard was ignored in the contention-free memory system, the requests with access hazards had to be removed from the trace files[5] so that the comparison of these four request scheduling policies could be evaluated fairly.

---

[5]The decoupled codes of LLL loops 4, 5, 6 and 11 were modified to save the store datum of the store request, that created a Read-After-Write hazard previously, for the load request accessing the same location as the store request. Then, the store request was issued after its store datum was used by the load request. The modified decoupled codes were used to generate a new set of trace files, so that Read-After-Write access hazards will not occur during the simulations.

5.5.2. Evaluation of Memory Models without Read-After-Write Hazards

The results of trace-driven simulations using the set of trace files in which Read-After-Write hazards were eliminated are shown in Table 5.3 and 5.4 and plotted in Figure 5.3. As shown in Figure 5.3, the performance difference among scheduling policies was greater and the differentiation was clearer when

Total Simulation Times of Memory Models without RAW Hazards in the Trace Files					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF/RRF	73,950	54,050	52,182	52,024	51,989
FCFS/RRF	109,633	88,552	55,356	52,220	52,038
FMRF/1Q	113,590	98,419	80,799	78,678	78,124
FCFS/1Q	147,691	126,591	86,300	82,186	81,482

Table 5.3 -- Total simulation time of the first 12 Lawrence Livermore Laboratory loops on four memory models. The set of trace files that do not have Read-After-Write access hazards are used in the simulations.

Performance of Memory Models Measured without RAW Hazards in the Trace Files					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF/RRF	0.703	0.961	0.996	0.999	0.999
FCFS/RRF	0.474	0.587	0.939	0.995	0.998
FMRF/1Q	0.457	0.528	0.643	0.660	0.665
FCFS/1Q	0.352	0.410	0.602	0.632	0.638

Table 5.4 -- Performance of the memory models using the first 12 Lawrence Livermore Laboratory loops. The set of trace files that do not have Read-After-Write access hazards are used in the simulations.

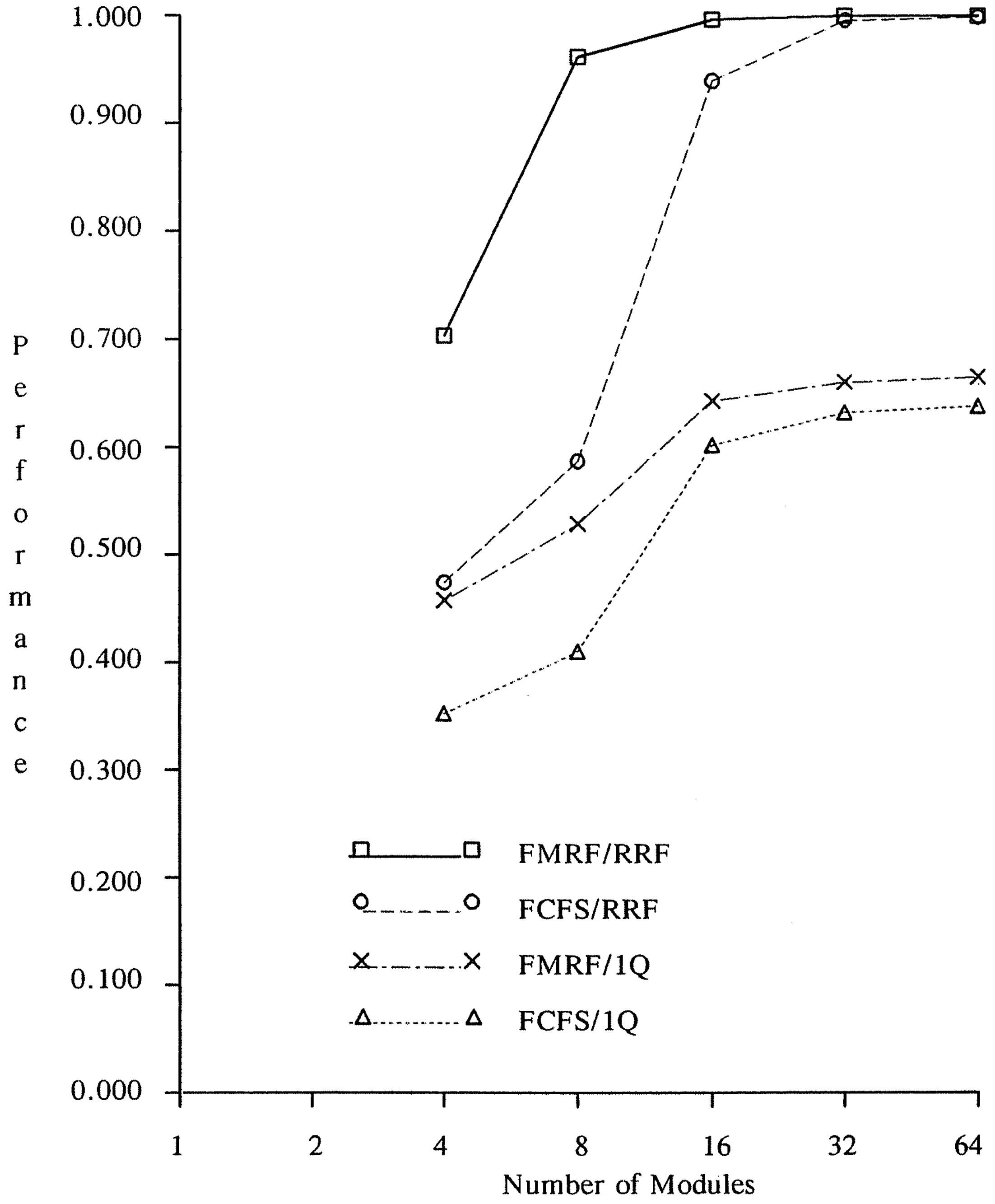


Figure 5.3 -- Performance comparison of memory models using the set of trace files where no Read-After-Write hazards exist.

the memory access hazards are removed. The following observations are suggested by the results of this simulation:

(1) When both the load address and the store address requests are placed into the single LA/SA Queue, as in the FCFS/1Q policy and the FMRF/1Q policy, the store address requests will block the load address requests from being issued. Performance peaks at about 65% performance of a contention-free memory system. When the load address and the store address requests are stored in the LAQ and the SAQ, respectively, the performance of FCFS/RRF and FMRF/RRF policies can approximate the performance of the contention-free memory system when the interleaving factor is equal or greater than 32. Thus, the performance of a memory system can increase about 50% if the load address and the store address requests are stored in separate queues.

(2) The FCFS/RRF policy can achieve a similar level of performance as the FMRF/RRF policy when the interleaving factor is 32 or greater, because the number of memory conflicts is small.

(3) Neither the FCFS/RRF policy nor the FMRF/1Q policy will perform well individually. The combination of these two policies into the FMRF/RRF policy can achieve much better result than either of them alone. Take for instance the cases where the interleaving factor is 8, the FCFS/RRF policy or the FMRF/1Q policy can achieve only 58.7% and 52.8% of the performance of contention-free memory system, respectively. But, the FMRF/RRF policy can achieve 96% performance of the contention-free memory system.

5.6. Summary and Discussion

For the trace files generated from the first LLL loops, the simulation results indicate that:

(1) The Free-Module-Request-First policy can significantly outperform the First-Come-First-Serve policy for servicing requests to the same queue when the interleaving factor is less than the memory bank busy time (10 CPU cycles). When the interleaving factor is equal to or greater than 32, memory conflicts do not occur frequently. Thus, the performance of the FMRF and the FCFS policies are almost the same.

(2) The separation of the load and the store requests can substantially improve the performance of memory system.
(3) Memory access conflicts can reduce the performance by almost 40% for the memory model using the FMRF/RRF policy under the workload of the first 12 LLL loops.

During the simulations of LLL loops 2, 3, 5 and 6 in the memory models with interleaving factors of 4 or 8, the array elements of the array variables with the same index are located at the same memory module. Thus, memory conflicts occurs frequently. If the compiler is able to detect that the array references can create frequent memory conflicts in an interleaved memory system, it should assign the array variables with different base addresses so that the number of memory conflicts in the run-time can be reduced. Then, the performance difference of the FMRF and the FCFS scheduling policies can be further reduced.

The Memory Controller with the FMRF/RRF scheduling policy has the ability to service the memory requests at their arrival rate if the access hazards are not considered. When the access hazards are taken into account, the FMRF/RRF policy can only achieve 62% performance of the contention-free memory system. If Read-After-Write access hazards resulting from the array references within the loop are considered, the performance of the memory model using the FMRF/RRF policy can decrease further. These results show that the memory access hazard is a bottleneck for computing the Lawrence Livermore Laboratory Loops in the decoupled architecture.

If the distributed control logic for the FMRF scheduling policy is more complex than the centralized control logic for the FCFS scheduling policy in the implementation, it is possible that the probability of failure for the memory system with the FMRF scheduling policy is higher than that for the memory system with the FCFS scheduling policy. The issue of fault tolerance for a memory system is not investigated in this research.

In order to make the FMRF/RRF scheduling policy practical, its implementation should be able to handle memory access hazards efficiently. Since the sequence control scheme of Free-Module-Request-First policy can be used to track cache hits and misses, a data cache with the ability to detect access hazards and the ability to short-circuit Read-After-Write requests is designed and evaluated in the next chapter.

CHAPTER 6

The Design of a Data Cache

Cache memories have been successfully implemented to store frequently referenced instructions and data in order to meet the demand of memory requests from a processor. Caches permit a processor to retrieve the information from the cache memory in a much shorter time interval compared to the main memory. Thus, a well-designed cache memory organization can enhance the performance of a high-speed computing system. A modification to a data cache will be shown to provide the ability for detecting and resolving memory access hazards, and its performance will be evaluated in this chapter.

6.1. Limitations of a Data Cache in the Decoupled Architecture

In decoupled computation, operand addresses are computed by the Access Processor. The Execute Processor is aware that these operands are in the Load Data Queue, but it is unaware of their memory addresses. If the Execute Processor were allowed to fetch operands from a cache memory, the operand addresses would have to be provided by the execute code so that the Execute Processor could use the addresses to fetch operands from the data cache.

Each PIPE processor has only one Load Data Queue for storing the operands fetched from memory. The order of operands placed into the Load Data Queue is critical for correct computation. If a data cache were implemented on each PIPE processor, the operands could be either fetched from the data cache (cache hit) or fetched from the main memory (cache miss). Since it requires a longer time interval to fetch data from the main memory than from the data cache, the PIPE processor would have to track the cache hits and misses in order to place the fetched operands into the Load Data Queue in the correct order. Also, the cache consistency problem [CeFe78] will be another difficult design issue when both the Access and Execute processors have their own data caches.

In addition, the speedup of cache hits following a cache miss cannot be realized. This is due to the fact that the operand not in the data cache must be fetched from the main memory. The operands fetched by those cache hits cannot be placed in the hardware queue until the operand for the cache miss is enqueued. Thus, a high hit ratio is an important factor for the design of a data cache for a decoupled architecture[1].

Due to these difficulties, the cache memory for the PIPE architecture is organized into an instruction cache for each processor and a data cache for each memory module. An instruction cache is located on each PIPE processor chip, and the data cache[2] is placed in the memory system. When the data cache is part of the memory system, the operands supplied by the cache must still be

[1]Note that the *bursty* nature of cache misses is advantageous here.

[2]Actually instructions may also be contained in the data cache.

transmitted to the processor chip, and the data cache cannot be used to reduce the processor-memory traffic [Good83]. However, this single data cache avoids the cache coherence problem (except for instructions) and provides a simple implementation of a data cache for the decoupled architecture.

The Memory Controller with the Free-Module-Request-First scheduling policy can service memory requests out of arrival order, and then coalesce the fetched operands into their original order when they are delivered. Thus, a data cache can be implemented in the Memory Controller. The Free-Module-Request-First Memory Controller can assume the responsibility of bookkeeping for the cache hits and misses without additional cost. A specially designed data cache in the Memory Controller not only provides flexible fetching schemes, it also detects memory access hazards and provides a short-circuit to reduce the penalty of access hazards.

In the following sections, the detailed design of this data cache in the Free-Module-Request-First Memory Controller will be discussed.

6.2. Organization of the Memory Controller with Data Cache

Because the address space of each memory module does not overlap with other modules, and because each Module Controller fetches its operands independently, the data cache will be partitioned among the modules.

A block diagram of the Module Controller with data cache is illustrated in Figure 6.1. The pipeline stages of the Module Controller are segmented into the input stage, the issue stage and the output stage. The functions of the input and the output stages are the same as the basic Module Controller using Free-

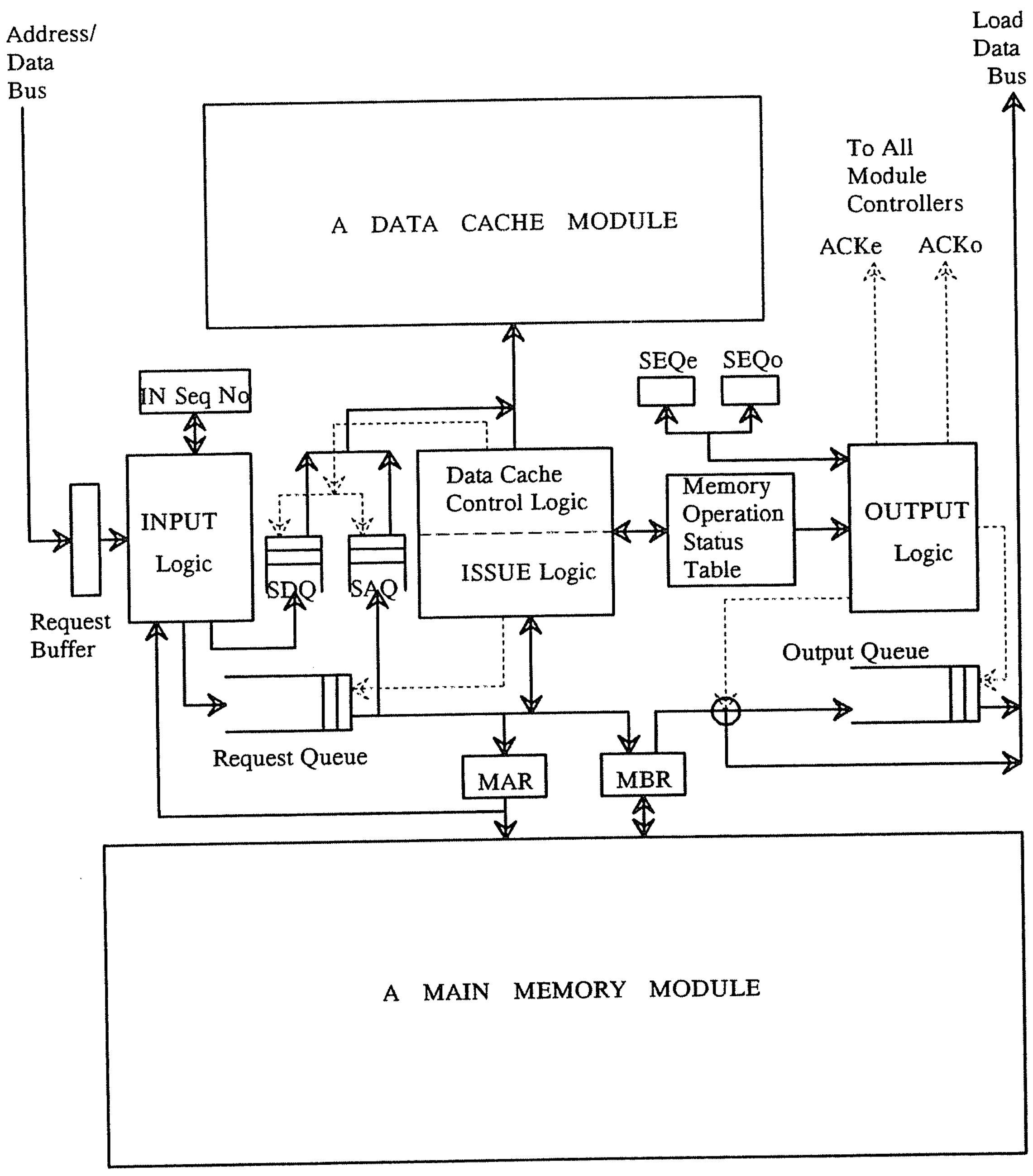


Figure 6.1 -- Block Diagram of the Module Controller with Data Cache

Module-Request-First scheduling policy. The issue stage has additional control logic (the data cache control logic) for accessing the data cache. The sequence number registers: IN-Seq, SEQo, and SEQe, and the control signals: ACKe and ACKo are the same as described in chapter 4.

A memory request, either load address or store address request, is placed in the Request Queue when it is received. Similarly, a store datum is placed in the Store Data Queue when it is received. The request at the head of the Request Queue will be removed and processed by the issue stage if the request can be serviced. In addition, the store address request is placed in the Store Address Queue, and is paired with the store data in the Store Data Queue. Detailed algorithms used by the issue stage will be discussed in later sections.

6.2.1. Organization of the Data Cache

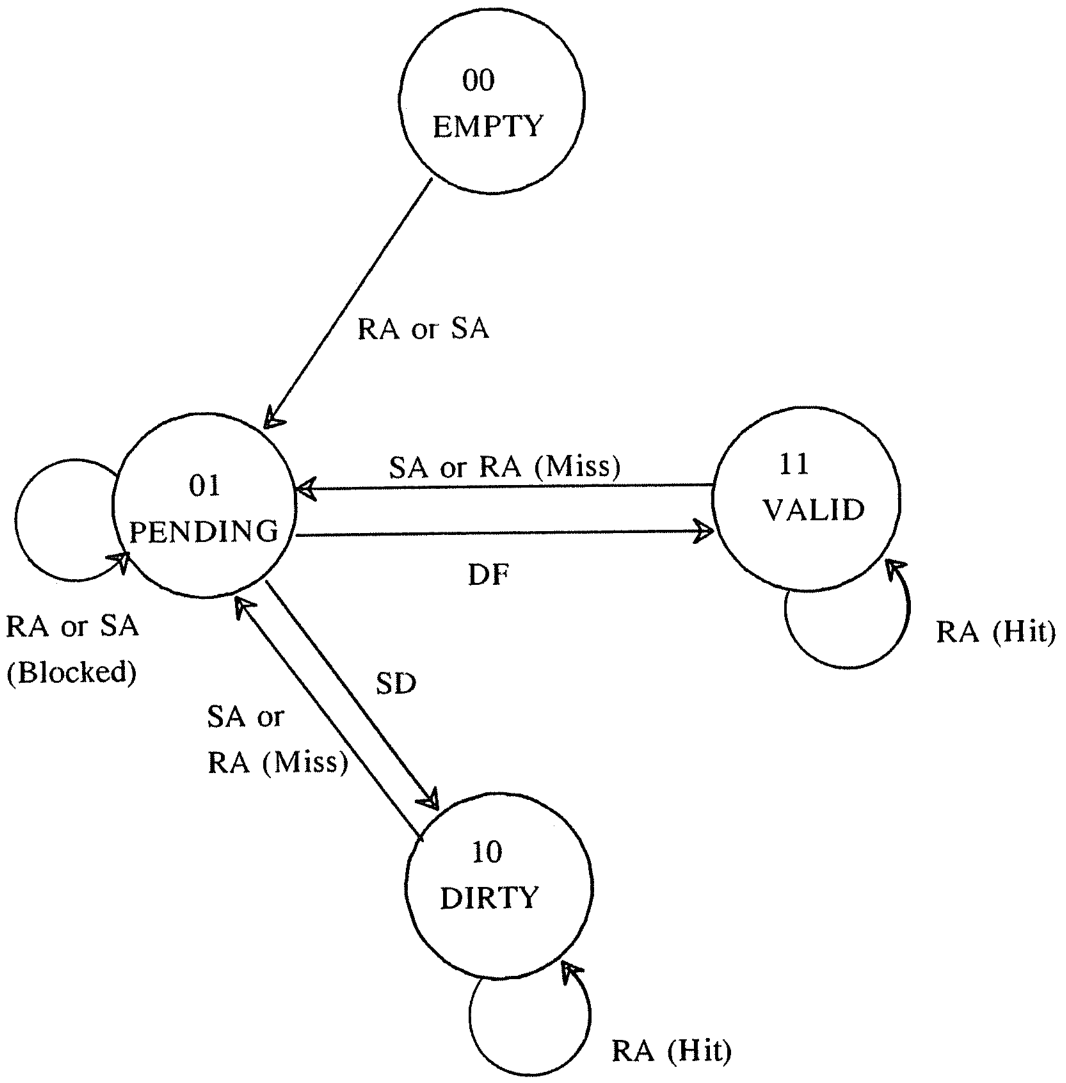
Each Module Controller has its own data cache to store the contents of the memory locations that are referenced by the requests accessing the module. The line size of a data cache in each Module Controller is the same as the memory word size.

A single data cache entry consists of three parts: the *tag*, the *address*, and the *data*. The tag indicates the status of a cache entry, and can take the form of one of four states: *Empty, Valid, Pending* and *Dirty*. The *Empty* entry does not have valid information in the address and in the data parts. The *Valid* entry contains the address of a memory location and the current value stored in that memory location. The *Pending* entry contains only a memory address and does not have valid data in the data part. The *Dirty* entry contains the address of a memory location and the current value of that memory location, but this value is not stored in that memory location, i.e., the memory location is stale. A *Pending* entry can belong to a *pending-write* request or a *pending-read* request. A *pending-write* request is a write request whose store address has arrived and has been stored in the data cache without associated data. Thus, it is waiting for the store data to arrive. A *pending-read* request is a read request being serviced, and its address has been stored in the data cache. Hence, it is waiting for the data to be fetched from the memory module. The state transition diagram of a data cache entry is illustrated in Figure 6.2, and the transition actions are explained in Table 6.1.

By insisting that a *Pending* entry not be replaced until its corresponding datum has been stored into the data part, a memory access hazard is detected if the memory request is mapped into a *Pending* entry. This feature allows the data cache to detect memory access hazards. Thus, the request queues need not be associative for detecting access hazards and can be simpler. In addition, the data cache provides a short-circuit for the Read-After-Write requests. Detailed discussion of this feature and its performance improvement to the memory system are described in section 6.2.3.

6.2.2. Selected Data Cache Algorithms

The address space of the entire data cache organization is illustrated in Figure 6.3, where a 4-way interleaved memory system is assumed. A *logical cache line* is defined which consists of the words of contiguous memory addresses across all data cache modules. Each word of a logical cache line is controlled by a Module Controller independently. Thus, the contents of a logical cache line may



RA	=	Read Address
SA	=	Store Address
SD	=	Store Data
DF	=	Data Fetched into Cache

Figure 6.2 -- State transition diagram of a data cache entry.

Current State	Input	Comment	Next State	Action
0 0 **Empty**	RA	Read address arrives	0 1	Start read/Update address
	SA	Store address arrives	0 1	Update address
	SD	Store data/addr arrives	0 0	ERROR
	DF	Data fetched from memory	0 0	ERROR
0 1 **Pending**	RA	Read address arrives	0 1	Try again next cycle
	SA	Store address arrives	0 1	Try again next cycle
	SD	Store data/addr arrives	1 0	If MATCH, update data;
			0 1	otherwise, ERROR.
	DF	Data fetched from memory	1 1	If MATCH, update data;
			0 1	otherwise, ERROR.
1 1 **Valid**	RA	Read address arrives	1 1	HIT, read cache.
			0 1	MISS, read memory/replace cache.
	SA	Store address arrives	0 1	Update address
	SD	Store data/addr arrives	1 1	ERROR
	DF	Data fetched from memory	1 1	ERROR
1 0 **Dirty**	RA	Read address arrives	1 0	HIT, read cache.
			0 1	MISS, write back/read memory.
	SA	Store address arrives	0 1	HIT, Update address.
			0 1	MISS, write back/update address.
	SD	Store data/addr arrives	1 0	ERROR
	DF	Data fetched from memory	1 0	ERROR

Table 6.1 -- State transition table for a data cache entry. The MATCH means that the address of a memory request is the same as the address stored in a *Pending* entry, and the ERROR action should never occur.

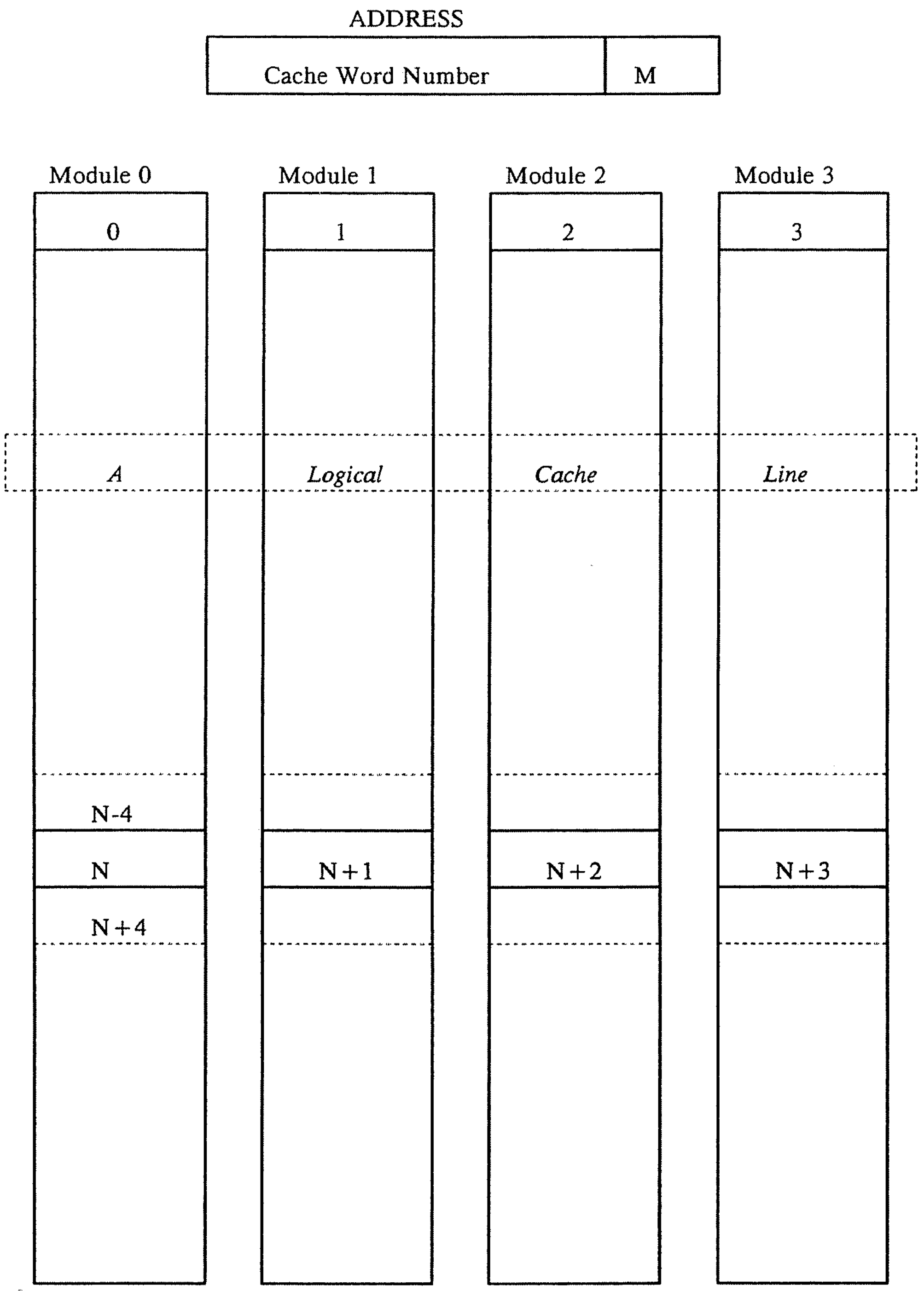


Figure 6.3 -- Address space of the data cache in 4-way interleaved memory system.

not be contiguous memory addresses. This feature allows different cache algorithms, and results in the definition of a cache line which is unlike that of the conventional cache memory. The study of different caching algorithms derivable from this feature is beyond the scope of this research. Only those selected for the implementation of memory access hazard detection and resolution scheme are discussed.

There are alternative ways to select the cache fetch algorithm, the placement algorithm, the replacement algorithm and the write-through or the write-back policies [Smith82]. Simple implementation is the main consideration in selecting the cache algorithms for evaluating the performance of the data cache. Although the selected algorithms may not represent the optimal solution for the data cache organization being studied, they can be used to evaluate the effectiveness of the solution to memory access hazards. The cache algorithms used for simulation study are:

(1) Logical cache lines are fetched on demand, and prefetching for adjacent logical cache lines is not attempted. However, the memory words of a logical cache line, which is referenced by a load address request, are prefetched by other non-busy modules if a cache miss occurs. Thus, the memory words of the same logical cache line may be fetched into the data caches in all modules simultaneously.

(2) A direct-mapped algorithm using the bit selection scheme, as shown in Figure 6.4, is used for both the placement and replacement algorithms. Thus, the operand fetched for a load address request will not replace a *Pending* entry.

(3) The write-back policy is used for three reasons: (a) cache consistency is not a problem in this cache organization, (b) fetch-on-write is not necessary because each cache word of a logical cache line can be placed and replaced independently, and (c) write-back policy results in fewer memory operations than write-through policy [Smith82]. Thus, the data part of a *Dirty* entry must be stored into the main memory when it is replaced by the cache

replacement algorithm.

6.2.2.1. Servicing a Read Request

Based on the module number of a read request address, the Module Controller can determine whether or not the request is *legitimate* to itself. If the module number of a received request address is not the same as the module number of the Module Controller, then it is an *illegitimate* request. The algorithm for servicing a read request is illustrated in Figure 6.5.

When a cache hit occurs for a legitimate request, the data is retrieved from the data cache and passed on to the output stage with its sequence number for delivery. When a cache miss occurs for a legitimate read request, the issue logic must start a read operation to fetch the needed data into the data cache. The fetched data is also passed to the output stage with its sequence number for delivery to its final destination. The cache entry to be replaced by the legitimate read request must be marked as a *Pending* entry before the read operation is

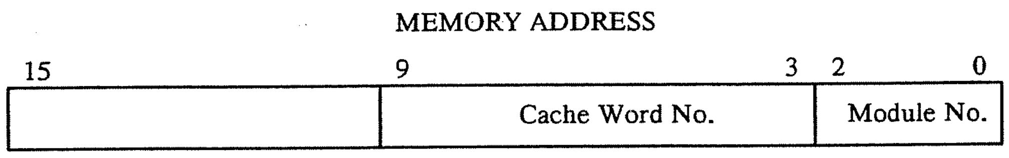


Figure 6.4 -- Bit assignment for selecting a cache placement/replacement word from the data cache of a module. Assumes 128-word data cache per module and 8-way interleaving.

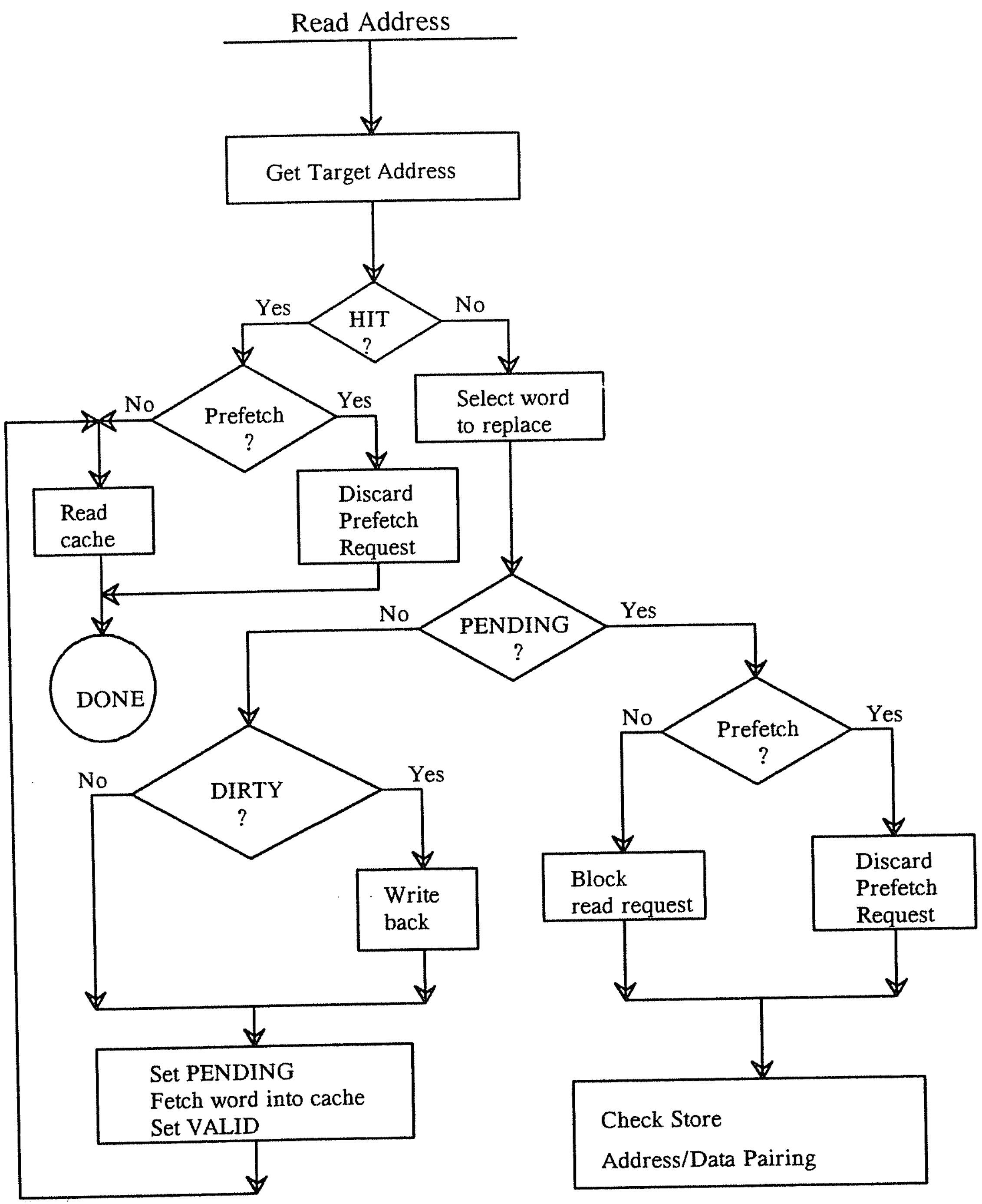


Figure 6.5 -- Algorithm for servicing a read request. The read request is checked whether or not it is legitimate. The read request for an illegitimate request is considered as a prefetch request, because it is fetching a memory word that is not needed when the read request arrives.

started, and then changed to a *Valid* entry when the fetched data is stored into the cache entry.

For an illegitimate read request, the issue logic has to obtain the address of the memory word that belongs to the same logical cache line as the memory word requested by the illegitimate read request. This word address can be obtained from the higher order bits, that indicates the word location within a module, of the illegitimate read request address. (For example, in Figure 6.4, bits 3-15 indicate the word location of a memory word within a module; the issue logic has to obtain the word address from bits 3-15 of an illegitimate read request address, and use the obtained word address to access the memory word within its module.) The issue logic then checks whether the data of the accessed word address is in the data cache. If it is in the data cache, this illegitimate read request is discarded and its service is completed. If the data of the computed address is not in data cache and the data cache entry selected for replacement is a *Pending* entry, this illegitimate read request is discarded and its service is completed. Otherwise, the entry selected for cache replacement is marked as a *Pending* and a prefetch for the computed address is started if the module is free. When the prefetching operation is started, the Memory Operation Status Table for this fetch is flagged as a prefetching operation, so that the fetched data is stored into data cache without being routed to the output stage. Once the prefetched data is stored into the cache, the cache entry is changed from *Pending* to *Valid*.

6.2.2.1.1. Variations of Servicing Read Requests

There are three alternative memory words that the Module Controller can fetch when the Module Controller receives an illegitimate read request: (1) the memory word on the addressed cache line, (2) the memory word on the cache line that is before the addressed cache line, or (3) the memory word on the cache line that is after the addressed cache line. If method 1 is used, the data cache operates the same way as the conventional cache where the unit of cache fetch is the cache line. When either method 2 or 3 is used, the number of words fetched is the same as the number of words within a cache line. But the fetched words may not all belong to the same cache line. Method 1 is used in the simulation study of this research, and methods 2 and 3 are not investigated.

6.2.2.2. Servicing a Write Request

From the performance of the FMRF memory model discussed in the previous chapter, we know that store address requests should not remain at the head of the queue and wait for the store data to arrive. Instead, the store address request is moved to the Store Address Queue where it will wait for store data if it has not yet arrived. Meanwhile, the address of the store request is stored into the selected cache entry, and this store address request becomes a pending-write request. The algorithms for servicing a store address request and matching the store address and store data are illustrated in Figure 6.6 and 6.7, respectively.

When a store datum arrives at the Module Controller, it is placed into the Store Data Queue regardless of whether or not it belongs to a store address request accessing the receiving module. It is likely that a store datum can arrive

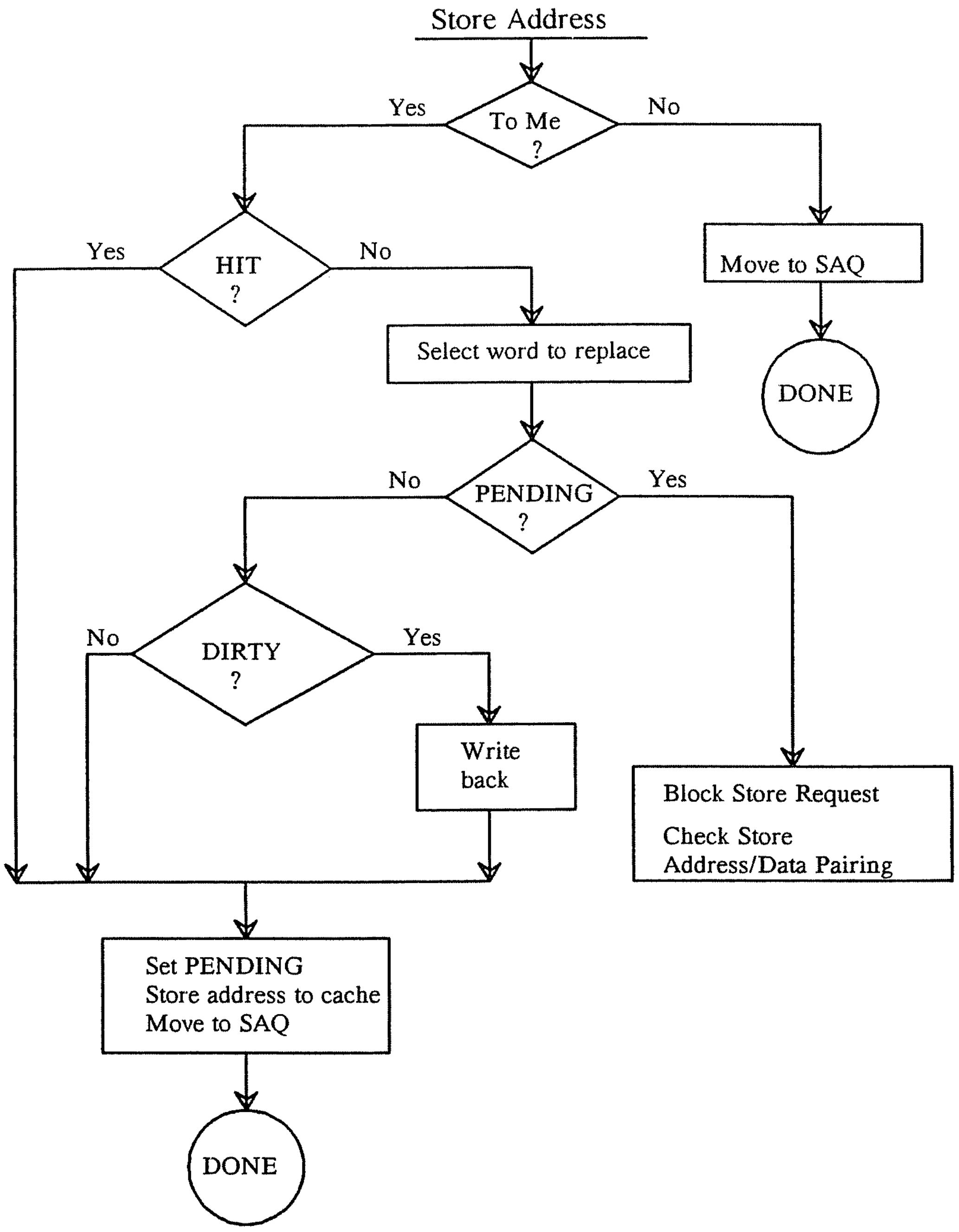


Figure 6.6 -- Algorithm for servicing a store address request.

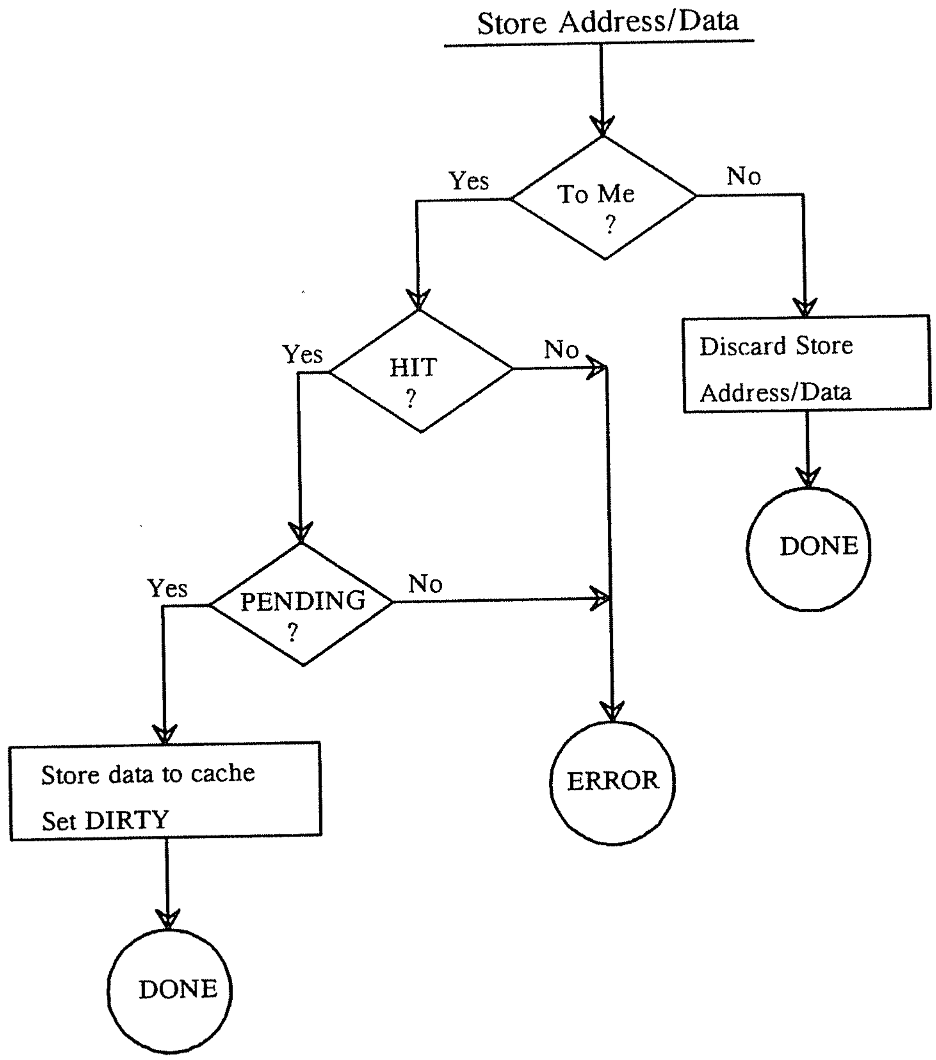


Figure 6.7 -- Algorithm for pairing store address and store data.

at the Module Controller earlier than its store address request. The store datum at the head of the Store Data Queue cannot be processed until there is a store address in the Store Address Queue. Similarly, the store address at the head of the Store Address Queue cannot be processed until there is a store datum in the Store Data Queue. The store datum reaching the head of the the Store Data

Queue will be paired with the store address at the head of the Store Address Queue. If this store address is not located at the memory module of the Module Controller, this store address/datum pair is discarded. Otherwise, the service for the store address/datum pair is started. The store address is used to locate the *Pending* entry in the data cache. Once the datum is stored into the data cache, the *Pending* entry will be converted to a *Dirty* entry that has the current data value of the store address.

Since the store address request always becomes a pending-write request when it is moved from the Request Queue to the Store Address Queue and since *Pending* entries are never purged, the paired store address and store datum will always find its pending-write entry in the data cache.

6.2.3. Access Hazard Resolution and Read-After-Write Short-Circuiting

Since multiple requests to a particular memory location are stored in the Request Queue in the order of arrival, they will be serviced in the arrival sequence. Thus, the correct sequence of memory operations is ensured. When the issue logic checks whether the next request can be processed, one of three situations can occur: (1) it is a cache hit, (2) it is a cache miss, or (3) it is mapped into a *Pending* entry.

Suppose that the next request is a read request. The issue logic fetches the data from the data cache if it is a cache hit. A memory read operation will be issued if it is a cache miss, and the data cache operation in the Module Controller is blocked until the memory read operation is completed. When the next request is mapped to a *Pending* entry of a pending-write request, a Read-After-Write access hazard occurs if the store address of the pending-write request is the same as the next read request. Otherwise, a memory access hazard does not occur. In either of these two situations, the next request has to wait at the head of the Request Queue until the *Pending* entry changes to a *Dirty* entry.

If the next request is a store address request and is mapped to a *Pending* entry of a pending-write request, a potential Write-After-Write access hazard is detected. The Write-After-Write hazard is resolved in the same way as the Read-After-Write access hazard by enforcing the store address request to remain at the head of the Request Queue (The first write need not be done, however).

The Write-After-Read hazard can occur if a store address of the next request is the same as the address of the read operation in progress, and it can be resolved in two ways. First, the Module Controller stops servicing requests when a cache miss occurs, so that Write-After-Read hazards never occur. This scheme is used in the simulation study. Second, the cache entry for the read request being serviced is always marked as a *Pending* entry and this *Pending* entry cannot be purged. Thus, the Write-After-Read hazard is resolved.

In this data cache design, the Read-After-Write request must be blocked at the head of Request Queue until the store data of the pending-write request is stored into the data cache. The Read-After-Write request is then guaranteed a cache hit, and the store data can be short-circuited to the Read-After-Write requests. Thus, the penalty of the Read-After-Write hazard can be reduced.

6.2.4. First-Come-First-Serve Memory Model with Distributed Data Cache

A centralized control First-Come-First-Serve memory system with distributed data cache can be implemented with one FCFS Memory Controller and *M* Module Controllers, where *M* is the interleaving factor. In this memory model, the Memory Operation Status Table is located at the FCFS Memory Controller, and the data cache and cache control tables are distributed among the Module Controllers. There is no distributed sequence control scheme among the Module Controllers. Therefore, the FCFS Memory Controller must service memory requests of the same type in the arrival order, so that the fetched operands can be delivered when they are fetched by the Module Controllers.

The FCFS Memory Controller services a memory request by broadcasting the request to all Module Controllers. All the Module Controllers will fetch the memory words of the logical cache line addressed by the received request. Thus, the memory words of the same logical cache line are fetched into the data caches of the Module Controller simultaneously. The fetched memory word for the received request will also be delivered to the processor.

The Module Controller must send a signal to the FCFS Memory Controller to indicate whether a cache hit or miss has occurred for the received memory request. A cache miss requires the Module Controller to fetch the needed information from main memory. Therefore, when a cache miss occurs, the FCFS Memory must stop sending requests to the Module Controllers until the needed information is fetched from main memory. Otherwise, the operands fetched for the cache hits following a cache miss will be delivered in the wrong order.

It requires (1) one clock period to send a request to the Module Controller, (1) one clock period for a Module Controller to look up the cache control table and decide whether it is a cache hit or miss, and (3) one clock period to send a signal to the FCFS Memory Controller about the cache hit or miss information. The FCFS Memory Controller can continuously send three requests to the Module Controllers before the FCFS Memory Controller knows that a cache miss has occurred. Because there is no sequence control scheme among the Module Controllers, the Module Controller cannot accept new requests when it is fetching information from main memory. Thus, the FCFS Memory Controller must keep a copy of each request sent to the Module Controllers until the FCFS Memory Controller knows that a hit or miss for that request has occurred. The submitted requests following the request of a cache miss must be resubmitted to the Module Controllers when the Module Controller is ready to receive new requests. In addition, the Module Controller must notify the FCFS Memory Controller when a memory write operation is required for a replaced dirty cache entry. Thus, the FCFS Module Controller can determine when the Module Controllers are ready to receive new requests.

The memory system using a FCFS Memory Controller and distributed data cache among the Module Controllers can also use the data cache to detect memory access hazards and short-circuit Read-After-Write requests. However, its implementation requires more off-chip communication and is more difficult than the distributed memory system using the FMRF policy with a data cache on each Module Controller. The performance of both memory systems will be evaluated through trace-driven simulation in the following section.

6.3. Performance Evaluation of the Data Cache

The goal of the trace-driven simulations is to evaluate the effectiveness of memory access hazard resolution and the performance improvement of the memory model with the data cache. The set of trace files of the first 12 Lawrence Livermore Loops that have Read-After-Write hazards (loops 4, 5, 6 and 11) is used. In order to have a fair comparison between the *contention-free* memory model and the memory models with data cache, the memory bank busy time of the *contention-free* memory model should be the same as the cache cycle time (three CPU cycles). Therefore, in the following discussion, the performance of a memory model will be normalized with the total simulation time of the *contention-free* memory model with memory bank busy time of three CPU cycles[3].

In the simulations, it is first assumed that the data cache of each Module Controller is the same size, i.e., 128 words of direct-mapped cache per module. The memory model with a larger interleaving factor will have a larger data cache than the memory model with a smaller interleaving factor. In later sections, simulations of an 8-way interleaved memory model with a data cache of different sizes per module are performed for investigating suitable cache sizes.

The simulation results in Tables 6.2 and 6.3, and Figure 6.8 show the performance comparisons of the memory model using the data cache and the other memory models. The results indicated that the data cache did improve the memory system significantly. In fact, it improved performance for the FMRF memory system from 60% to 94% and for the FCFS memory system from 44%

[3]The performance of the *contention-free* memory model is nearly the same for 3 and 10-CPU-cycle memory bank busy times.

to 75% of the performance of the contention-free memory system when the interleaving factor was 8. When the interleaving factor was greater than or equal to 32, the performance of the FMRF memory system with data cache was about 98% of the contention-free memory system, and the performance of the FCFS memory system with data cache was about 86% of the contention-free memory system. For the simulations of loops 4, 5, 6 and 11, where many Read-After-Write requests exist, the average performance using a data cache was 2.5 times that of the memory model without data cache (see Appendix III).

In order to determine the appropriate data cache size, simulations with different data cache sizes, from 1 to 256 words per module, were performed for the 8-way interleaved FMRF memory model. The result of simulations is in Table 6.4. The table shows that by merely incorporating a data cache in the memory model, even if there is only one cache word on each Module Controller, the enhanced model was better than when there was no data cache.

The performance of memory models with 2 or 4 cache words per module were slightly worse than the memory model that had only one cache word in each module. These results were due to the cache fetch and replacement algorithms used in the simulations. In the input stage, a prefetch was discarded if the address of prefetch request was mapped into a *Pending* entry. When the cache size was 1, all the addresses of memory requests were mapped into the same cache entry (since there is only one entry); thus most of the prefetches were discarded. When the cache size was 2 or 4, it was likely that the first few prefetches were mapped to separate entries and memory read operations were issued to prefetch the memory words into the data cache; but, the subsequent read requests

Total Simulation Times of Memory Models

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	73,896	60,823	59,044	58,088	58,028
FCFS with Data Cache	102,287	76,062	68,261	66,375	65,635
FMRF/RRF	106,793	95,293	92,900	92,458	92,282
FCFS/RRF	148,678	128,525	96,793	93,397	93,084
FMRF/1Q	144,969	128,438	111,769	109,283	108,565
FCFS/1Q	180,370	157,988	116,138	111,666	110,777

Total Simulation Times of *Contention-Free* Memory Models

Memory Bank Busy Time	Total Simulation Time
3 CPU Cycles	56,874
10 CPU Cycles	57,741

Table 6.2 -- Simulation results of memory models with a cache cycle time of 3 CPU cycles and a memory bank busy time of 10 CPU cycles. Simulation results for two types of *contention-free* memory models of memory bank busy times of 3 CPU cycles and 10 CPU cycles, respectively, are listed at the bottom of the table. The first 12 LLL trace files that have RAW hazards were used.

Performance of Memory Models					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.770	0.935	0.963	0.979	0.980
FCFS with Data Cache	0.556	0.748	0.833	0.857	0.867
FMRF/RRF	0.533	0.597	0.612	0.615	0.616
FCFS/RRF	0.383	0.443	0.588	0.609	0.611
FMRF/1Q	0.392	0.443	0.509	0.520	0.524
FCFS/1Q	0.315	0.360	0.490	0.509	0.513
Performance of *Contention-Free* Memory Model with Memory Bank Busy Time of 10 CPU Cycles = 0.985					

Table 6.3 -- Performance of memory models normalized with the total simulation time of the *contention-free* memory model with memory bank busy time of 3 CPU cycles. The first 12 LLL trace files that have RAW hazards were used.

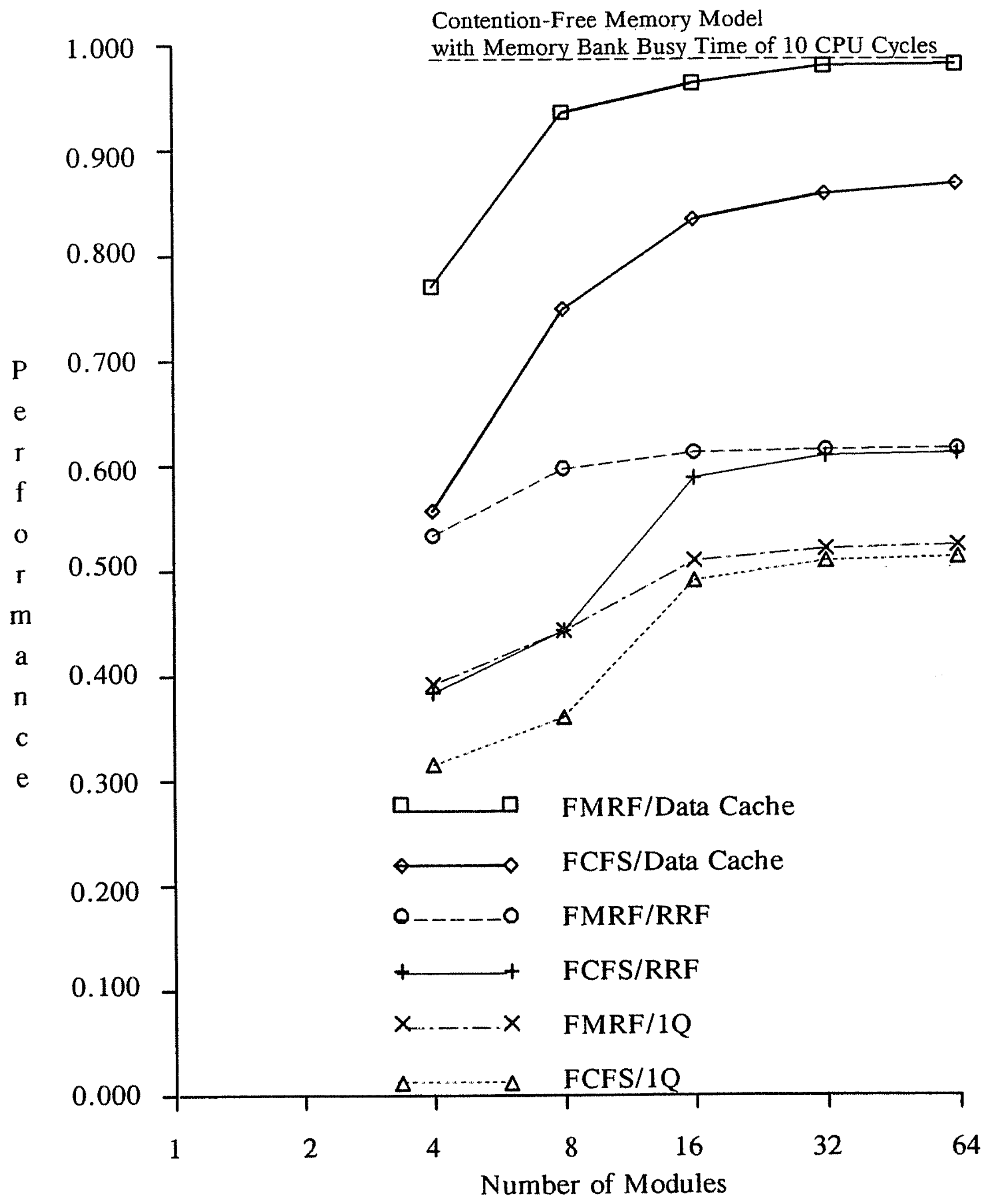


Figure 6.8 -- Performance comparison of memory models using the set of trace files where Read-After-Write hazards exist. The performance of each model is normalized with the total simulation time of the *contention-free* model with memory bank busy time of 3 CPU cycles.

Performance & Simulation Time of 8-Way Interleaved Memory Model with Data Cache		
Data Cache Words per Module	Performance	Simulation Time
0	0.597	95,293
1	0.736	77,225
2	0.681	83,562
4	0.712	79,920
8	0.773	73,608
16	0.857	66,386
32	0.895	63,568
64	0.906	62,767
128	0.935	60,823
256	0.935	60,797

Table 6.4 -- Comparison of 8-way interleaved memory models with different data cache sizes. The memory model of cache size 0 is the memory system using Free-Module-Request-First/Read-Request-First scheduling policy.

could be mapped into the entries where the prefetched operands were just stored. Therefore, there were numerous prefetching operations that used memory cycles, but the prefetched operands were replaced before they were used.

As the data cache size increased, the prefetched operands were more likely to remain in the data cache until called for. Thus, the performance increased as the cache size increased. For the 8-way interleaved memory system, the performance improvement was insignificant after the data cache size was greater than 128 words per module.

6.4. Summary and Discussion

A data cache for a decoupled architecture has been designed. By implementing the data cache in the Memory Controller, the sequence control scheme of the Free-Module-Request-First policy can track the cache hits and misses. Thus, the control for the data cache can be implemented in the Memory Controller without additional cost. In addition, the cache coherence problem is avoided.

The cache control logic is also designed to detect memory access hazards and short-circuit Read-After-Write requests, resulting in dramatic performance improvements for those trace files exhibiting Read-After-Write hazards. Thus, the request queues do not need associative search capability.

By implementing the data cache in the memory system with Free-Module-Request-First scheduling policy, the memory system can approach the performance level of the *contention-free* memory system under the workload of the Lawrence Livermore Loops used in this study. Since the data cache cycle is three CPU cycles in the simulation model, simulations for the *contention-free* memory system with a memory bank busy time of three CPU cycles were performed in order to obtain a fair comparison between the *contention-free* memory system and the FMRF memory system with data cache. The simulation results showed that the total simulation time is decreased only 1.5% for the *contention-free* memory system when the memory bank busy time is changed from ten to three CPU cycles. Thus, the results prove that the FMRF memory system with data cache is a high performance memory system, that is able to service requests with minimum memory conflict delays.

The data cache can also increase the performance of the FCFS memory system from 61% to 87% of the *contention-free* memory system when the interleaving factor is equal to or greater than 32. The performance of the FCFS memory system with data cache cannot be as good as the FMRF memory system with data cache due to the fact that the FCFS Memory Controller cannot service a new request when any of the Module Controllers are accessing main memory. Each cache replacement for a *Dirty* entry causes the FCFS Memory Controller to delay another memory bank busy time (10 CPU cycles). Thus, the delay increases as the number of memory write operations for *Dirty* entries increases, and the performance of the FCFS memory system with data cache degrades.

CHAPTER 7

Summary and Conclusions

7.1. Summary

The design of a high performance memory system for a decoupled architecture is the subject of this research. The analysis of memory reference characteristics during decoupled computations was seen to be an appropriate way to identify the requirements for the memory system. Memory reference characteristics were identified and strategies for scheduling overlapped memory requests were proposed and evaluated through trace-driven simulations.

The objective of the memory system design is to minimize the effective memory delay for decoupled computation through efficient request servicing. The strategy of request scheduling was designed with two goals: (1) to service memory requests according to their priorities, and (2) to minimize the total service time of memory requests. However, memory access hazards are the main factors that inhibit reaching the objective of the memory system for a decoupled architecture.

To achieve the first goal, the alternatives of dividing memory requests into different types were proposed, and the priority for each request type was assigned analytically. In addition, potential deadlock in request scheduling was identified, and solutions were proposed.

The achieving of the second goal resulted in much improved memory system performance. This contribution is primarily due to the sequence control scheme of Free-Module-Request-First scheduling policy, which allows requests to be serviced out of arrival order and coalesces the fetched operands into correct order when they are delivered. A theorem was derived to prove that the Free-Module-Request-First scheduling policy can indeed minimize the total service time of requests to a single queue.

The Free-Module-Request-First scheduling policy together with the Read-Request-First policy produced an acceptable performance improvement when the memory access hazard was not taken into account.

The embodiment of a specially designed data cache, able to resolve memory access hazards and to short-circuit Read-After-Write requests, combined with the Free-Module-Request-First scheduling policy, resulted in a memory model that can service the memory requests as efficiently as the *contention-free* memory model during the trace-driven simulations of the first 12 Lawrence Livermore Loops. Thus, the performance degradation due to memory conflicts and access hazards is greatly reduced and resulted in a high performance memory system.

7.2. Conclusions

The decoupled architecture uses hardware queues to architecturally decoupled memory request generation from algorithmic computation. This results in an implementation that has two separate instruction streams that communicate via hardware queues. Thus, performance is improved through parallelism and efficient memory referencing.

Techniques for increasing memory bandwidth such as pipelining, interleaving, servicing request out of arrival order, and cache memory are incorporated in the proposed memory system within two constraints: (1) the operands placed in the hardware queue must be in the correct order, and (2) the needed operands are the only operands that can be placed in the hardware queue.

Three characteristics of memory reference in decoupled computation were observed and solutions proposed: (1) two memory transactions can be issued to the memory system in the same clock period, (2) overflow of hardware queues caused by continuous overlap of memory requests, and (3) the shared variable problem that leads to Read-After-Write access hazards.

The round-robin and EP-First multiplexing schemes were shown to be the best of the four investigated multiplexing schemes that restrict the memory transaction arrival rate to one per clock period. The performance of the memory system using round-robin or EP-First is almost the same as the memory system which is capable of receiving one memory transaction from each processor per clock period.

A Start-and-Stop flow control scheme was proposed to regulate the transactions between the processors and the memory system, and prevent the hardware queues from overflowing. By using this scheme, the receiving end must reserve three empty buffers before a *Stop* signal is sent to stop the transmitting end from sending more memory transactions, and the *Start* signal is sent for accepting more arrivals when there are four empty buffers. Simulation studies showed that the performance gain levels off when the combined total queue length is greater than eight. Since the size of a Load Data Queue in the processor must be equal to or greater than four, the minimal queue size provides adequate length.

The technique of short-circuiting Read-After-Write requests was proposed and shown to be effective in reducing the penalty resulting from shared variables.

The sequence control scheme for the Free-Module-Request-First scheduling policy can be used to track cache hits and misses. Thus, a data cache with the ability to detect access hazards and short-circuit Read-After-Write requests almost for free was designed in the Free-Module-Request-First Memory Controller.

Architecturally, the request buffers of the memory system should be organized with four different types of queues: (1) Instruction Fetch Queue, (2) Load Address Queue, (3) Store Address Queue, and (4) Store Data Queue. The instruction fetch request and the load address request must be stored in different queues in order to avoid deadlocks. The store address request should be separated from the load address request so that load address requests will not be blocked by a store address waiting for its store datum. The store data from each of the Access and the Execute processors must be stored in separate Store Data Queues so that matching the store address and the store datum can be done efficiently. When memory requests are stored in these four types of queues, deadlock will never occur unless a processor attempts to submit a request to a full queue or an incorrect sequence of memory access instructions is scheduled in the decoupled codes.

7.3. Contributions

The memory *access conflict* and *access hazard* are two main factors that prevent an interleaved memory system from achieving its maximum throughput. The memory system proposed in this research has two important features. Each of them can reduce the penalties introduced by the access conflicts and access hazards:

(1) The *Free-Module-Request-First* scheduling policy issues memory operations out of arrival order by starting memory operations whenever the needed memory modules are free. Thus, the delay due to memory conflicts is minimized.

(2) The specially designed data cache *guarantees* a cache hit for each Read-After-Write request. Thus, the memory read operation for each occurrence of Read-After-Write hazard is avoided, and the total memory service time is reduced.

Traditionally, each memory read request must be accompanied by a tag which specifies the source/sink addresses of the read request. Thus, the fetched operand can be delivered to its destination (usually a register) correctly. The source/sink tagging scheme allows the memory system to issue memory operations out of arrival order (as in the storage system of the IBM System/360 Model 91) and achieve high performance at the cost of complex interlock controls for the processor-memory interface [AnST67, BoGr67]. In the memory system proposed for the decoupled architecture, a *sequence control scheme* (required by the Free-Module-Request-First scheduling policy) combined with the use of *hardware* *queues* provides an efficient memory access method for the load/store computer architecture. The sequence control scheme allows the memory system to issue memory operations out of arrival order and assumes the responsibility of delivering the operands to the processors in a correct order. Thus, the proposed memory system can achieve high performance with a much simpler processor-memory interface than that of the memory system using the source/sink tagging scheme.

The existing memory access hazard detection/resolution schemes can be classified into three categories (discussed in chapter 2): (1) the prevention scheme, (2) the detection and prevention scheme, and (3) the detection and abortion scheme. The second scheme is the best among these three schemes and is used in the storage system of the IBM System/360 Model 91. However, the obvious implementation of scheme 2 requires the memory request queues to have associative search capability in order to detect hazards. In addition, the memory system must sequence the fetch and store requests correctly so that the fetch and store to the same memory location can be serviced in the correct order. For the memory system proposed in this dissertation, an efficient memory access hazard detection/resolution scheme (which is a detection and prevention scheme) is designed and incorporated in the control logic of the data cache without additional cost. Moreover, the request queues need not have associative search capability. Thus, the implementation of this new hazard detection/resolution scheme can be very simple.

7.4. Further Research Plans

A memory system with a designed data cache and the Free-Module-Request-First scheduling policy can be used in multiprocessor systems where a common memory is shared by the processors. By placing the cache memory in the shared memory system, the coherence problem is avoided. When the processor does not have a local memory, the memory arrival rate of the shared memory will increase as the number of processors increases. Thus, the use of Free-Module-Request-First scheduling policy may be superior. The implementation of Free-Module-Request-First scheduling policy in multiprocessor systems should be investigated and performance measured.

The cache line of the designed data cache is different from the conventional cache memory. For the conventional cache memory, the cache line is the unit of transfer between the cache and the main memory. Each word of a cache line in our cache memory can be allocated and replaced independently. Thus, various fetch and replacement algorithms and prefetching schemes can be explored. The kinds of cache algorithms and performance results that obtain from this new cache organization should be further examined and reported.

REFERENCES

[AnST67] D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal of Research and Development,* pp. 8-24, January 1967.

[Back78] J. Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM,* Vol. 21, No. 8, pp. 613-641, August 1978.

[Baer80] Jean-Loup Baer, *Computer Systems Architecture,* Chapter 5, Computer Science Press, Rockville, Maryland, 1980.

[BaSm76] F. Baskett, and A. J. Smith, "Interference in Multiprocessor Computer Systems with Interleaved Memory," *Communications of ACM,* Vol. 19, No.6, pp. 327-334, June 1976.

[Bhan75] D. P. Bhandarkar, "Analysis of Memory Interference in Multiprocessors," *IEEE Trans. on Computers,* Vol. C-24, pp. 897-908, September 1975.

[BlCP62] L. Bloom, M. Cohen, and S. Porter, "Considerations in the Design of a Computer with High Logic to Memory Speed Ratio," *Proc. Gigacycle Computing Systems,* AIEE Special Publication 2-136, pp. 53-63, Jan. 29 - Feb. 2, 1962.

[BoGr67] L. J. Boland, G. D. Grantio, A. V. Marcotte, B. V. Messina, and J. W. Smith, "The IBM System 360 Model 91: Storage Systems," *IBM Journal Research & Development,* Vol. 11, pp. 54-68, January 1967.

[Bons69] P. Bonseigneur, "Description of the 7600 Computer System," *Computer Group News,* pp. 11-15, May 1969.

[BrDa77] F. A. Briggs, and E. S. Davidson, "Organization of Semiconductor Memories for Parallel-Pipelined Processors," *IEEE Trans. on Computers,* Vol. C-26,5, pp. 162-169, February 1977.

[BuCo70] G. J. Burnett, and E. G. Coffman, Jr., "A Study of Interleaved Memory Systems," *Proceeding of AFPIS 1970 SJCC,* Vol. 36, pp. 467-474, 1970.

[BuCo73] G. J. Burnett, and E. G. Coffman, Jr., "A Combinatorial Problem Related to Interleaved Memory Systems," *J. Ass. Comput. Mach.,* Vol. 20, pp. 39-45, January 1973.

[BuCo75] G. J. Burnett, and E. G. Coffman, Jr., ''Analysis of Interleaved Memory Systems Using Blockage Buffers,'' *Commun. Ass. Comput. Mach.,* Vol. 18, pp. 91-95, Feburary 1975.

[BuKu71] P. P. Budnik, and D. J. Kuck, ''The Organization and Use of Parallel Memories,'' *IEEE Trans. on Computers,* Vol. C-20, pp. 1566-1569, December 1971.

[CeFe78] L. M. Censier, and P. Feautrier, ''A New Solution to Coherence Problems in Multicache Systems,'' *IEEE Trans. on Computers,* Vol. C-27, No. 12, pp. 1112-1118, December 1978.

[Char81] A. E. Charlesworth, ''An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family,'' *IEEE Computer,* Vol. 14, No. 9, pp. 18-27, September 1981.

[ChKL77] D. Chang, D. J. Kuck, and D. H. Lawrie, ''On the Effective Bandwidth of Parallel Memories,'' *IEEE Trans. on Computers,* Vol. C-26, pp. 480-490, May 1977.

[ClLP81] D. W. Clark, B. W. Lampson, and K. A. Pier, ''The Memory System of a High-Performance Personal Computer,'' *IEEE Trans. on Computers,* Vol. C-30, No. 10, pp. 715-732, October 1981.

[CoBS71] E. G. Coffman, G. J. Burnett, and R. A. Snowdon, ''On the Performance of Interleaved Memories with Multiple Word Bandwidths,'' *IEEE Trans. on Computers,* Vol. C-20, pp. 1570-1573, December 1971.

[CoSt81] E. U. Cohler, and J. E. Storer, ''Functionally Parallel Architecture for Array Processors,'' *IEEE Computer,* Vol. 14, No. 9, pp. 28-36, IEEE, September 1981.

[Davi75] E. S. Davidson, et al., ''Effective Control for Pipelined Computers,'' *Proceeding of COMPCON,* Spring 1975, pp. 181-184, February 1975.

[Flyn66] M. J. Flynn, ''Very High-Speed Computing Systems,'' *Proceeding of the IEEE,* Vol. 54, No. 12, pp. 1901-1909, December 1966.

[Fost68] C. C. Foster, ''Determination of Priority in Associative Memories,'' *IEEE Trans. on Computers,* Vol. C-17, pp. 788-789, August 1968.

[Good83] J. R. Goodman, ''Using Cache Memory to Reduce Processor-Memory Traffic,'' *Proceeding of 10th International Symposium on Computer Architecure,* IEEE, 1983.

[GHLP85] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young ''PIPE: A VLSI Decoupled Architecture,'' *Preceeding of 12th International Symposium on Computer Architecture,* IEEE, June 1985.

[GoYo85], J. R. Goodman, and H. C. Young, ''Code Scheduling Methods for Some Architectural Features in PIPE,'' Tech. Report #579, Computer Sciences Department, University of Wisconsin-Madison, February 1985.

[Hell67] H. Hellerman, *Digital Computer System Principles,* pp. 228-229, McGraw-Hill Inc., 1967.

[HiTa72] R. G. Hintz, and D. P. Tate, ''Control Data STAR-100 Processor Design,'' *Proc. IEEE COMPCON 1972,* pp. 1-4, September 1972.

[HsPG84] J. T. Hsieh, A. R. Pleszkun, and J. R. Goodman, ''Performance Evaluation of the PIPE Computer Architecture,'' Tech. Report #566, Computer Sciences Department, University of Wisconsin-Madison, November 1984.

[IBM75] IBM, *IBM SYSTEM/370 Model 168 Theory of Operation/Diagrams Manual,* Vol. 1 & 4, IBM, Poughkeepsie, N.Y., 1975.

The Structure of Computers and Computations John Wiley & Sons, Inc., Volume 1, pp. 408, 1978.

[KuRa75] D. E. Knuth, and G. S. Rao, ''Activity in an Interleaved Memory,'' *IEEE Trans. on Computers,* Vol. C-24, pp. 943-944, September 1975.

[KuSt82] D. J. Kuck, and R. A. Stokes, ''The Burroughs Scientific Processor (BSP),'' *IEEE Trans. on Computers,* Vol. C-31, No. 5, pp. 363-376, May 1982.

[LaVo82] D. H. Lawrie, and C. R. Vora, ''The Prime Memory System for Array Access,'' *IEEE Trans. on Computers,* Vol. C-31, No. 5, pp. 435-442, May 1982.

[Linc82] N. R. Lincoln, ''Technology and Design Tradeoffs in the Creation of a Modern Supercomputer,'' *IEEE Trans. on Computers,* Vol. C-31, No. 5, pp. 349-362, May 1982.

[McMa72] F. H. McMahon, ''Fortran CPU Performance Analysis,'' Lawrence Livermore National Laboratories, 1972.

[Mead70] R. M. Meade, ''On Memory System Design,'' *Proceeding of AFPIS 1970 FJCC,* pp. 33-43, 1970.

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

[PlDa83] A. R. Pleszkun and E. S. Davidson, ''A Structural Memory Access Architecture,'' *1983 International Conference on Parallel Processing,* (Bellaire Mich., Aug. 23-26) IEEE, New York, 1983.

[RaLi77] C. V. Ramamoorthy, and H. F. Li, ''Pipelined Architecture,'' *Computing Surveys,* Vol. 9, No. 1, pp. 61-102, March 1977.

[RaTW78] C. V. Ramamoorthy, J. L. Turner, and B. W. Wah, ''A Design of a Cellular Associative Memory for Ordered Retrieval,'' *IEEE Trans. on Computers,* Vol. C-27, pp. 800-815, September 1978.

[Ravi72] C. V. Ravi, ''On the Bandwidth and Interference in Interleaved Memory Systems,'' *IEEE Trans. on Computers,* Vol. 21, pp. 889-901, August 1972.

[RaWa81] C. V. Ramamoorthy, and B. W. Wah, ''An Optimal Algorithm for Scheduling Requests on Interleaved Memories for a Pipelined Processor,'' *IEEE Trans. on Computers,* Vol. C-30, pp. 787-799, October 1981.

[RiSc84] J. P. Riganati, and P. B. Schneck, ''Supercomputing,'' *IEEE Computer,* Vol. 17, No.10, pp. 97-113, October 1984.

[Russ78] R. M. Russell, ''The CRAY-1 Computer System,'' *Commumication ACM,* Vol. 21, No. 1, pp. 63-72, ACM, January 1978.

[Ryma82] J. Rymarczyk, ''Coding Guidelines for Pipelined Processors,'' *SIGARCH Computer News,* Vol. 10, No. 2, pp. 12-19, ASPLOS, March 1982.

[Schor71] H. Schorr, ''Design Principles for a High-Performance System,'' *Symposium on Computers and Automata,* pp. 165-173, Polytechnic Institute of Brooklynn, April 1971.

[Shive82] R. R. Shively, ''Architecture of a Programmable Digital Signal Processor,'' *IEEE Trans. on Computers,* Vol. C-31, No. 1, pp. 16-22, January 1982.

[SiBN82] D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples,* McGraw-Hill, Inc., 1982.

[SmiA82] A. J. Smith, ''Cache Memories,'' *Computing Surveys,* Vol. 14, No. 3, pp. 473-530, September 1982.

[SmiJ82] J. E. Smith, ''Decoupled Access/Execute Computer Architectures,'' *Proceeding of the 9th Annual Symposium on Computer Architecture,* pp. 112-119, IEEE, May, 1982.

[SmiJ84] J. E. Smith, ''Decoupled Access/Execute Computer Architectures,'' *ACM Trans. on Computer Systems,* Vol. 2, No. 4, pp. 289-308, November 1984.

[SPKG83] J. E. Smith, A. R. Pleszkun, R. H. Katz, and J. R. Goodman, ''PIPE: A High Performance VLSI Architecture,'' Technical Report # 512, Computer Sciences Department, University of Wisconsin-Madison, September 1983.

[Tang76] C. K. Tang, ''Cache System Design in the Tightly Coupled Multiprocessor System,'' *AFIPS Proceeding of NCC,* Vol. 45, pp. 749-753, 1976.

[Thor70] J. E. Thornton, *Design of a Computer: The Control Data 6600,* Scott, Foresman & Company, Glenview, Ill., 1970.

[Toma67] R. M. Tomasulo, ''An Efficient Algorithm for Exploiting Multiple Arithmetic Units,'' *IBM Journal of Research & Development,* Vol. 11, No. 1, pp. 25-33, January 1967.

[UIll57] University of Illinois at Urbana-Champaign, ''On the Design of a Very High-Speed Computer,'' Report No. 80, University of Illinois at Urbana-Champaign, Digital Computer Lab., October 1957.

[WeSm84] S. Weiss and J. E. Smith, ''Instruction Issue Logic in Pipelined Supercomputers,'' *IEEE Trans. on Computers,* Vol. C-33, No. 11, pp. 1013-1022, November 1984.

[Wilk65] M. V. Wilkes, ''Slave Memories and Dynamic Storage Allocation,'' *IEEE Trans. Electronic Computers,* pp. 270-271, April 1965.

[YePD83] P. C. Yeh, J. H. Patel, and E. S. Davidson, ''Shared Cache for Multiple-Stream Computer Systems,'' *IEEE Trans. on Computers,* Vol. C-32, No. 1, pp. 38-47, January 1983.

[YoGo84] H. C. Young, and J. R. Goodman, ''A Simulation Study of Architectural Data Queues and Prepare-to-branch Instruction,'' *Preceeding, IEEE Internaltional Conference on Computer Design,* pp. 544-549, October 1984

Appendix I

The first thirteen loops of the benchmark program from the Lawrence Livermore National Laboratories are used to create trace files for the simulation study conducted in this research. Since the floating point instructions have not been defined in the PIPE architecture, the variables of these thirteen loops are *integerized*. A listing of these loops translated in Pascal is shown in this appendix.

The object codes of these thirteen loops are generated in two ways: (1) To use the PIPE Pascal compiler to compile the Pascal programs and generate the object codes. (2) To use the PIPE Assembler to translate the PIPE assembler codes into the object codes. Then, the object codes are executed by the PIPE interpreter to create trace files.

LLL1: HYDRO EXCERPT

```
program LLL1 (input, output);
var
        k, q, r, t : integer;
        x, y, z : array [ 1 .. 420 ] of integer;
begin { LLL1 }
   q := 0;
   for k := 1 to 400 do
        x[k] := q + y[k] * (r * z[k+10] + t * z[k+11]);
end. { LLL1 }
```

LLL2: MLR, INNER PRODUCT

```
program LLL2 (input, output);
var
        tp, x, z : array [1 .. 1000] of integer;
        k : integer;
begin { LLL2 }
   k := 1;
   while (k <= 996) do begin
        tp[k] := z[k] * x[k] + z[k+1] * x[k+1] +
                z[k+2] * x[k+2] + z[k+3] * x[k+3] +
                 z[k+4] * x[k+4];
        k := k + 5;
   end;
end. { LLL2 }
```

LLL3: INNER PRODUCT

```
program LLL3 (input, output);
var     k, q    : integer;
        z, y    : array [1..1000] of integer;
begin { LLL3 }
   q := 0;
   for k:= 1 to 1000 do
        q := q + z[k] * y[k];
end. { LLL3 }
```

LLL4: BANDED LINEAR EQUATIONS

```
program LLL4 (input, output);
var     j, l, lw: integer;
        x       : array [1..275] of integer;
        y       : array [1..870] of integer;
begin { LLL4 }
        l := 7;
        while (l <= 107) do begin
                lw := l;
                j := 30;
                while (j <= 870) do begin
                        x[l-1] := x[l-1] - x[lw]*y[j];
                        lw := lw + 1;
                        j := j + 5;
                end;
                x[l-1] := y[5] * x[l-1];
                l := l + 50;
        end;
end. { LLL4 }
```

LLL5: TRI-DIAGONAL ELIMINATION, BELOW DIAGONAL

```
program LLL5 (input, output);
var     i       : integer;
        x, y, z : array [1..1000] of integer;
begin { LLL5 }
   i := 2;
   while (i <= 1000) do begin
        x[i  ] := z[i  ] * (y[i] - x[i-1]);
        x[i+1] := z[i+1] * (y[i+1] - x[i]);
        x[i+2] := z[i+2] * (y[i+2] - x[i+1]);
        i := i + 3;
   end;
end. { LLL5 }
```

LLL6: TRI-DIAGONAL ELIMINATION, ABOVE DIAGONAL

```
program LLL6 (input, output);
var     i,j      : integer;
        x, z     : array [1..1001] of integer;
begin { LLL6 }
   j := 3;
   while (j <= 999) do begin
        i := (999+1) - j + 3;
        x[i  ] := x[i  ] - z[i  ] * x[i+1];
        x[i-1] := x[i-1] - z[i-1] * x[i  ];
        x[i-2] := x[i-2] - z[i-2] * x[i-1];
        j := j + 3;
   end;
end. { LLL6 }
```

LLL7: EQUATION OF STATE EXCERPT

```
program LLL7 (input, output);
var     m        : integer;
        r, t     : integer;
        u        : array [1..126] of integer;
        x, y, z : array [1..120] of integer;
begin { LLL7 }
   for m := 1 to 120 do
        x[m]= u[m+3]     + r*( z[m  ] + r*y[m  ] ) +
             t*( u[m+3] + r*( u[m+2] + r*u[m+1] ) +
             t*( u[m+6] + r*( u[m+5] + r*u[m+4] )));
end. { LLL7 }
```

LLL8: P.D.E. INTEGRATION

```
program LLL8 (input, output);
var
       a11, a12, a13, a21, a22, a23, a31, a32, a33 : integer;
       nl1, nl2, kx, ky, sig : integer;
       u1, u2, u3 : array [1..4,1..22,1..2] of integer;
       du1, du2, du3 : array [1..22] of integer;
begin { LLL8 }
   nl1 := 1;
   nl2 := 2;
   for kx := 2 to 3 do
     for ky := 2 to 21 do begin
         du1[ky] := u1[kx,ky+1,nl1] - u1[kx,ky-1,nl1];
         du2[ky] := u2[kx,ky+1,nl1] - u2[kx,ky-1,nl1];
         du3[ky] := u3[kx,ky+1,nl1] - u3[kx,ky-1,nl1];

         u1[kx,ky,nl2] := u1[kx,ky,nl1] + a11*du1[ky] +
                          a12*du2[ky] + a13*du3[ky] +
                          sig * ( u1[kx+1,ky,nl1] -
                                 2*u1[kx,ky,nl1] +
                                 u1[kx-1,ky,nl1] );

         u2[kx,ky,nl2] := u2[kx,ky,nl1] + a21*du1[ky] +
                          a22*du2[ky] + a23*du3[ky] +
                          sig * ( u2[kx+1,ky,nl1] -
                                 2*u2[kx,ky,nl1] +
                                 u2[kx-1,ky,nl1] );

         u3[kx,ky,nl2] := u3[kx,ky,nl1] + a31*du1[ky] +
                          a32*du2[ky] + a33*du3[ky] +
                          sig * ( u3[kx+1,ky,nl1] -
                                 2*u3[kx,ky,nl1] +
                                 u3[kx-1,ky,nl1] );

     end;
 end. { LLL8 }
```

LLL9: INTEGRATE PREDICTORS

```
program LLL9 (input, output);
var     c0, bm22, bm23, bm24, bm25, bm26, bm27, bm28 : integer;
        i       : integer;
        px      : array [13,100] of integer;
begin { LLL9 }
   for i := 1 to 100 do
        px[1,i] := bm28*px[13,i] + bm27*px[12,i] + bm26*px[11,i] +
                   bm25*px[10,i] + bm24*px[ 9,i] + bm23*px[ 8,i] +
                   bm22*px[ 7,i] + c0*(px[ 5,i] + px[6,i]) + px[3,i];
end. { LLL9 }
```

LLL10: DIFFERENCE PREDICTORS

```
program LLL10 (input, output);
var
        i : integer;
        ar,br,cr: integer;
        cx : array [1..5,1..6] of integer;
        px : array [1..14,1..6] of integer;
begin { LLL10 }
   for i := 1 to 6 do begin
        ar       :=      cx[5,i];
        br       := ar -       px[5,i];
        px[5,i]  := ar         ;
        cr       := br -       px[6,i];
        px[6,i]  := br         ;
        ar       := cr -       px[7,i];
        px[7,i]  := cr         ;
        br       := ar -       px[8,i];
        px[8,i]  := ar         ;
        cr       := br -       px[9,i];
        px[9,i]  := br         ;
        ar       := cr -       px[10,i];
        px[10,i] := cr               ;
        br       := ar -       px[11,i];
        px[11,i] := ar         ;
        cr       := br -       px[12,i];
        px[12,i] := br               ;
        px[14,i] := cr -       px[13,i];
        px[13,i] := cr               ;
   end;
end. { LLL10 }
```

LLL11: FIRST SUM.

```
program LLL11 (input, output);
var     k       : integer;
        x, y    : array [1..1000] of integer;
begin { LLL11 }
   for k := 2 to 1000 do
        x[k] := x[k-1] + y[k];
end. { LLL11 }
```

LLL12: FIRST DIFF.

```
program LLL12 (input, output);
var     k       : integer;
        x, y    : array [1..1000] of integer;
begin { LLL12 }
   for k := 1 to 999 do
        x[k] := y[k+1] - y[k];
end. { LLL12 }
```

LLL13: 2-D PARTICLE PUSHER

```
program LLL13 (input, output);
var
        ip              : integer;
        i1, j1, i2, j2  : integer;
        p               : array [1..4, 1..512] of integer;
        b, c, h         : array [1..64, 1..8] of integer;
        e, f            : array [1..192] of integer;
        y, z            : array [1..1001] of integer;
begin {LLL13}
        for ip := 1 to 128 do begin
                i1 := p[1,ip];
                j1 := p[2,ip];
                p[3,ip] := p[3,ip] + b[i1,j1];
                p[4,ip] := p[4,ip] + c[i1,j1];
                p[1,ip]:= p[1,ip] + p[3,ip];
                p[2,ip] := p[2,ip] + p[4,ip];
                i2 := p[1,ip];
                j2 := p[2,ip];
                p[1,ip] := p[1,ip] + y[i2+32];
                p[2,ip] := p[2,ip] + z[j2+32];
                i2 := i2 + e[i2+32];
                j2 := j2 + f[j2+32];
                h[i2,j2] := h[i2,j2] + 1;
        end;
end. {LLL13}
```

Appendix II

A Decoupled Code of Lawrence Livermore Loop 2

(1) Access Code:

```
1        ENTER       1      -1000         / setup loop counter
2        LDBR    0   4               / load branch addr AL0
3   AL0: ALDLN       1      3060          / fetch z[k] for E-proc
4        ALDLN       1      2060          / fetch x[k] for E-proc
5        ADDIM       1      1             / inc loop counter
6        ALDLN       1      3060          / fetch z[k+1] for E-proc
7        ALDLN       1      2060          / fetch x[k+1] for E-proc
8        ADDIM       1      1             / inc loop counter
9        ALDLN       1      3060          / fetch z[k+2] for E-proc
10       ALDLN       1      2060          / fetch x[k+2] for E-proc
11       ADDIM       1      1             / inc loop counter
12       ALDLN       1      3060          / fetch z[k+3] for E-proc
13       ALDLN       1      2060          / fetch x[k+3] for E-proc
14       ADDIM       1      1             / inc loop counter
15       ALDLN       1      3060          / fetch z[k+4] for E-proc
16       ALDLN       1      2060          / fetch x[k+4] for E-proc
17       ADDIM       1      1             / loop counter has been inc by 5
18       PBRLT       1      0      2      / prepare to branch to AL0
19       ASTLN       1      1026          / store addr for tp[k]
20       HALT
```

(2) Execute Code:

```
1        XOR      5  5      5      / clear temp result
2        LDBR     0  24            / load branch addr EL0
3   EL0: MULI           3   7      7       / z[k]*x[k]
4        ADDI     5  3      5      / add temp result
5        MULI     3  7      7      / z[k+1]*x[k+1]
6        ADDI     5  3      5      / add temp result
7        MULI     3  7      7      / z[k+2]*x[k+2]
8        ADDI     5  3      5      / add temp result
9        MULI     3  7      7      / z[k+3]*x[k+3]
10       ADDI     5  3      5      / add temp result
11       MULI     3  7      7      / z[k+4]*x[k+4]
12       ADDI     5  3      5      / add temp result
13       IPBRQ       0      0      2       / check branch queue boolean
14       MOVNFF      7      5      0       / store result tp[k]
15       XOR      5  5      5      / clear temp result
16       HALT
```

Appendix III

Simulation results of the memory models using the set of trace files generated from the hand-coded decoupled codes of the first twelve Lawrence Livermore Loops. There are Read-After-Write access hazards in loops 4, 5, 6 and 11 for this set of trace files.

The Read-After-Write access hazards in LLL loop 4, 5, 6 and 11 are removed by better register allocation in their access codes. These new decoupled codes are also used to generate another set of trace files where no Read-After-Write access hazards exist, and they are used to obtain the simulation results in Appendix IV.

The performance of a memory model is computed by normalizing its simulation time with the simulation time of a *contention-free* memory model (described in chapter 5). In some simulation results, the performance of the FMRF/RRF memory model is greater than 1.0 (better than the *contention-free* memory model). This is because the instructions, that issue load address requests for the Execute Processor, was delivered to the Access Processor earlier in the FMRF/RRF memory model than in the *contention-free* memory model. Thus, during the first loop iteration, the Execute Processor received the operands from the FMRF/RRF memory model earlier than from the *contention-free* memory model. So, the simulation time of the FMRF/RRF memory model is shorter than that of the *contention-free* memory model.

Performance & Simulation Time of Memory Models Using LLL Loop 1

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.988	1.002	1.003	1.005	1.005
	3,355	3,309	3,306	3,300	3,300
FMRF/RRF	0.729	0.997	1.000	1.002	1.002
	4,550	3,324	3,314	3,307	3,307
FMRF/1Q	0.303	0.304	0.511	0.511	0.511
	10,933	10,887	6,491	6,491	6,489
FCFS with Data Cache	0.641	0.900	0.904	0.904	0.904
	5,174	3,683	3,667	3,667	3,667
FCFS/RRF	0.649	0.995	0.995	0.999	0.999
	5,108	3,331	3,331	3,317	3,317
FCFS/WRF	0.618	0.993	0.997	0.999	0.999
	5,364	3,339	3,325	3,317	3,317
FCFS/1Q	0.303	0.304	0.480	0.480	0.480
	10,947	10,918	6,907	6,905	6,901

Performance & Simulation Time of Memory Models Using LLL Loop 2					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.997	1.001	1.001	1.001	1.001
	7,523	7,493	7,493	7,493	7,493
FMRF/RRF	0.995	0.997	1.000	1.000	1.000
	7,539	7,521	7,500	7,500	7,500
FMRF/1Q	0.770	0.824	1.000	1.000	1.000
	9,735	9,101	7,500	7,500	7,500
FCFS with Data Cache	0.727	0.903	0.963	0.963	0.963
	10,319	8,304	7,786	7,785	7,785
FCFS/RRF	0.607	0.609	0.999	0.999	0.999
	12,345	12,316	7,509	7,509	7,509
FCFS/WRF	0.607	0.609	0.999	0.999	0.999
	12,345	12,316	7,509	7,509	7,509
FCFS/1Q	0.496	0.497	1.000	1.000	1.000
	15,125	15,096	7,500	7,500	7,500

Performance & Simulation Time of Memory Models Using LLL Loop 3

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.957	1.000	1.002	1.002	1.002
	6,357	6,084	6,070	6,070	6,070
FMRF/RRF	0.995	0.997	1.000	1.000	1.000
	6,117	6,105	6,086	6,085	6,085
FMRF/1Q	0.995	0.998	1.001	1.001	1.001
	6,114	6,095	6,076	6,075	6,075
FCFS with Data Cache	0.753	0.962	0.963	0.963	0.963
	8,085	6,327	6,317	6,317	6,317
FCFS/RRF	0.502	0.503	0.999	1.000	1.000
	12,120	12,105	6,094	6,085	6,085
FCFS/WRF	0.502	0.503	0.999	1.000	1.000
	12,120	12,105	6,094	6,085	6,085
FCFS/1Q	0.502	0.503	1.000	1.000	1.000
	12,120	12,105	6,080	6,085	6,085

Performance & Simulation Time of Memory Models Using LLL Loop 4					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.594	0.858	0.952	1.001	1.002
	11,424	7,902	7,120	6,774	6,765
FMRF/RRF	0.477	0.526	0.556	0.570	0.578
	14,209	12,880	12,207	11,896	11,730
FMRF/1Q	0.443	0.490	0.519	0.534	0.543
	15,291	13,835	13,065	12,688	12,499
FCFS with Data Cache	0.514	0.694	0.765	0.794	0.809
	13,203	9,766	8,859	8,543	8,387
FCFS/RRF	0.460	0.502	0.525	0.537	0.543
	14,744	13,495	12,906	12,635	12,484
FCFS/1Q	0.439	0.486	0.517	0.533	0.541
	15,435	13,940	13,106	12,719	12,523

Performance & Simulation Time of Memory Models Using LLL Loop 5					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.528	0.810	0.934	0.934	0.934
	9,089	5,929	5,144	5,144	5,144
FMRF/RRF	0.418	0.419	0.419	0.419	0.419
	11,504	11,460	11,455	11,455	11,455
FMRF/1Q	0.418	0.419	0.419	0.419	0.419
	11,495	11,470	11,457	11,451	11,451
FCFS with Data Cache	0.382	0.521	0.656	0.766	0.792
	12,560	9,215	7,317	6,273	6,063
FCFS/RRF	0.285	0.286	0.419	0.419	0.419
	16,842	16,800	11,451	11,451	11,451
FCFS/1Q	0.285	0.286	0.419	0.419	0.419
	16,859	16,823	11,458	11,458	11,459

Performance & Simulation Time of Memory Models Using LLL Loop 6

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.705	0.999	1.000	1.001	1.001
	7,752	5,472	5,469	5,462	5,462
FMRF/RRF	0.476	0.477	0.478	0.478	0.478
	11,495	11,459	11,432	11,431	11,431
FMRF/1Q	0.362	0.362	0.478	0.478	0.478
	15,127	15,096	11,440	11,431	11,431
FCFS with Data Cache	0.470	0.680	0.815	0.870	0.899
	11,626	8,041	6,709	6,287	6,082
FCFS/RRF	0.281	0.281	0.478	0.478	0.478
	19,476	19,440	11,450	11,439	11,439
FCFS/1Q	0.267	0.268	0.478	0.478	0.478
	20,475	20,432	11,433	11,432	11,432

Performance & Simulation Time of Memory Models Using LLL Loop 7

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.965	0.989	0.994	0.999	1.007
	2,264	2,209	2,197	2,186	2,169
FMRF/RRF	0.624	0.987	0.998	1.004	1.004
	3,499	2,213	2,188	2,175	2,175
FMRF/1Q	0.376	0.490	0.534	0.605	0.605
	5,813	4,459	4,088	3,607	3,607
FCFS with Data Cache	0.768	0.803	0.805	0.805	0.808
	2,843	2,721	2,713	2,713	2,703
FCFS/RRF	0.390	0.598	0.750	0.999	0.999
	5,603	3,650	2,913	2,186	2,186
FCFS/WRF	0.367	0.599	0.750	0.999	0.999
	5,957	3,649	2,913	2,186	2,186
FCFS/1Q	0.296	0.371	0.478	0.586	0.586
	7,368	5,892	4,568	3,730	3,730

Performance & Simulation Time of Memory Models Using LLL Loop 8					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.758	0.949	0.981	0.993	0.999
	2,995	2,394	2,315	2,286	2,274
FMRF/RRF	0.451	0.816	0.961	0.981	0.978
	5,035	2,782	2,363	2,315	2,322
FMRF/1Q	0.264	0.392	0.452	0.509	0.543
	8,593	5,790	5,028	4,464	4,184
FCFS with Data Cache	0.444	0.568	0.607	0.624	0.651
	5,115	3,998	3,739	3,640	3,491
FCFS/RRF	0.281	0.462	0.654	0.971	0.983
	8,068	4,916	3,470	2,340	2,310
FCFS/WRF	0.280	0.450	0.621	0.909	0.990
	8,118	5,044	3,659	2,499	2,293
FCFS/1Q	0.244	0.355	0.437	0.489	0.523
	9,307	6,391	5,201	4,645	4,345

Performance & Simulation Time of Memory Models Using LLL Loop 9

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.553	0.859	0.980	0.980	0.982
	3,954	2,544	2,231	2,231	2,225
FMRF/RRF	0.489	0.765	0.986	0.994	1.001
	4,469	2,856	2,218	2,199	2,184
FMRF/1Q	0.178	0.296	0.499	0.655	0.685
	12,257	7,393	4,383	3,339	3,191
FCFS with Data Cache	0.392	0.567	0.752	0.752	0.752
	5,576	3,858	2,905	2,905	2,905
FCFS/RRF	0.197	0.347	0.617	0.941	0.994
	11,099	6,306	3,544	2,323	2,199
FCFS/WRF	0.197	0.346	0.607	0.913	0.994
	11,099	6,318	3,599	2,394	2,199
FCFS/1Q	0.176	0.288	0.464	0.605	0.638
	12,386	7,580	4,712	3,614	3,425

Performance & Simulation Time of Memory Models Using LLL Loop 10					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.597	0.889	0.986	0.991	0.995
	5,018	3,366	3,038	3,021	3,010
FMRF/RRF	0.500	0.843	0.986	0.999	1.000
	5,987	3,550	3,038	2,996	2,994
FMRF/1Q	0.260	0.479	0.717	0.718	0.735
	11,501	6,246	4,175	4,171	4,072
FCFS with Data Cache	0.335	0.707	0.810	0.811	0.811
	8,947	4,232	3,696	3,692	3,692
FCFS/RRF	0.289	0.592	0.993	0.997	1.000
	10,374	5,057	3,016	3,003	2,995
FCFS/WRF	0.157	0.318	0.600	0.950	1.000
	19,052	9,404	4,992	3,150	2,995
FCFS/1Q	0.141	0.287	0.492	0.666	0.698
	21,213	10,446	6,088	4,493	4,292

Performance & Simulation Time of Memory Models Using LLL Loop 11					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	0.998	1.002	1.002	1.002	1.003
	7,088	7,062	7,062	7,062	7,058
FMRF/RRF	0.294	0.295	0.295	0.295	0.295
	24,049	24,030	24,030	24,030	24,030
FMRF/1Q	0.294	0.295	0.295	0.295	0.295
	24,039	24,020	24,020	24,020	24,020
FCFS with Data Cache	0.646	0.820	0.972	0.972	0.973
	10,956	8,633	7,280	7,280	7,272
FCFS/RRF	0.294	0.295	0.295	0.295	0.295
	24,058	24,030	24,030	24,030	24,030
FCFS/1Q	0.294	0.295	0.295	0.295	0.295
	24,058	24,030	24,030	24,030	24,030

Performance & Simulation Time of Memory Models Using LLL Loop 12					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF with Data Cache	1.000	1.003	1.003	1.003	1.003
	7,077	7,059	7,059	7,059	7,058
FMRF/RRF	0.849	1.000	1.001	1.001	1.001
	8,340	7,077	7,069	7,069	7,069
FMRF/1Q	0.503	0.504	0.504	0.504	0.504
	14,071	14,046	14,046	14,046	14,046
FCFS with Data Cache	0.898	0.972	0.973	0.973	0.973
	7,883	7,284	7,273	7,273	7,271
FCFS/RRF	0.800	1.000	1.000	1.000	1.000
	8,841	7,079	7,079	7,079	7,079
FCFS/WRF	0.800	1.000	1.000	1.000	1.000
	8,840	7,079	7,079	7,079	7,079
FCFS/1Q	0.469	0.470	0.470	0.470	0.470
	15,077	15,055	15,055	15,055	15,055

Appendix IV

Simulation results of Lawrence Livermore Loops 4, 5, 6 and 11 in different memory models using the set of trace files where no Read-After-Write access hazards exist. The Read-After-Write hazards are removed by modifying the decoupled codes, then using the PIPE interpreter to generate the new set of trace files.

Performance & Simulation Time of Memory Models Using LLL Loop 4

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF/RRF	0.991	0.995	0.995	0.995	0.999
	5,790	5,763	5,763	5,763	5,743
FMRF/1Q	0.991	0.995	0.995	0.995	0.999
	5,790	5,763	5,763	5,763	5,743
FCFS/RRF	0.967	0.995	0.995	0.995	0.999
	5,932	5,763	5,763	5,763	5,744
FCFS/WRF	0.967	0.995	0.995	0.995	0.999
	5,932	5,763	5,763	5,763	5,744
FCFS/1Q	0.968	0.997	0.997	0.997	0.999
	5,928	5,757	5,753	5,753	5,743

Performance & Simulation Time of Memory Models Using LLL Loop 5					
Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF/RRF	0.490	0.960	1.000	0.999	1.000
	8,450	4,310	4,139	4,143	4,139
FMRF/1Q	0.370	0.371	0.531	0.531	0.531
	11,188	11,142	7,791	7,795	7,791
FCFS/RRF	0.339	0.340	1.000	1.000	1.000
	12,226	12,173	4,141	4,141	4,141
FCFS/WRF	0.339	0.340	1.000	1.000	1.000
	12,226	12,173	4,141	4,141	4,141
FCFS/1Q	0.246	0.246	0.509	0.509	0.509
	16,848	16,795	8,126	8,126	8,126

Performance & Simulation Time of Memory Models Using LLL Loop 6

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF/RRF	0.569	0.987	0.995	1.000	1.000
	8,527	4,921	4,880	4,857	4,856
FMRF/1Q	0.543	0.547	0.708	0.710	0.710
	8,946	8,879	6,860	6,837	6,836
FCFS/RRF	0.396	0.397	0.997	1.000	1.000
	12,270	12,217	4,869	4,857	4,856
FCFS/WRF	0.395	0.397	0.997	1.000	1.000
	12,283	12,230	4,869	4,857	4,856
FCFS/1Q	0.396	0.399	0.674	0.674	0.675
	12,249	12,170	7,208	7,201	7,194

Performance & Simulation Time of Memory Models Using LLL Loop 11

Scheduling Policy	Interleaving Factors				
	4	8	16	32	64
FMRF/RRF	0.640	0.996	0.997	1.000	1.000
	5,647	3,628	3,624	3,615	3,615
FMRF/1Q	0.418	0.419	0.420	0.421	0.421
	8,649	8,618	8,598	8,590	8,590
FCFS/RRF	0.640	0.993	0.997	0.999	0.999
	5,647	3,639	3,627	3,617	3,617
FCFS/WRF	0.639	0.993	0.997	0.999	0.999
	5,661	3,639	3,628	3,619	3,619
FCFS/1Q	0.396	0.397	0.397	0.398	0.398
	9,123	9,106	9,102	9,086	9,086

Appendix V

Simulation results of different input multiplexing schemes in the memory models using two request scheduling policies, the First-Come-First-Serve/Read-Request-First scheduling policy and the Free-Module-Request-First/Read-Request-First scheduling policy. No Read-After-Write access hazards are in the trace files that are used in this simulation.

The performance of a memory model is computed by normalizing its simulation time with the simulation time of a *contention-free* memory model (described in chapter 5). In some simulation results, the performance of the FMRF/RRF memory model is greater than 1.0 (better than the *contention-free* memory model). This is because the instructions, that issue load address requests for the Execute Processor, was delivered to the Access Processor earlier in the FMRF/RRF memory model than in the *contention-free* memory model. Thus, during the first loop iteration, the Execute Processor received the operands from the FMRF/RRF memory model earlier than from the *contention-free* memory model. So, the simulation time of the FMRF/RRF memory model is shorter than that of the *contention-free* memory model.

Performance & Simulation Time of Memory Models Using LLL Loop 1						
Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.729	0.997	1.000	1.002	1.002
		4,550	3,324	3,314	3,307	3,307
	Time-Division	0.728	0.995	0.999	0.999	0.999
		4,551	3,335	3,317	3,317	3,317
	AP-First	0.729	0.997	1.000	1.002	1.002
		4,550	3,324	3,314	3,307	3,307
	EP-First	0.728	0.995	0.998	0.998	0.998
		4,550	3,331	3,323	3,323	3,323
	Round-Robin	0.728	0.997	0.999	0.999	0.999
		4,550	3,325	3,318	3,317	3,317
FCFS/RRF	No Multiplexing	0.649	0.995	0.995	0.999	0.999
		5,108	3,331	3,331	3,317	3,317
	Time-Division	0.653	0.993	0.996	0.996	0.996
		5,079	3,337	3,327	3,327	3,327
	AP-First	0.585	0.995	0.995	0.999	0.999
		5,665	3,331	3,331	3,317	3,317
	EP-First	0.626	0.994	0.998	0.998	0.998
		5,299	3,335	3,323	3,323	3,323
	Round-Robin	0.585	0.995	0.995	0.999	0.999
		5,665	3,331	3,325	3,317	3,317

Performance & Simulation Time of Memory Models Using LLL Loop 2

Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.995	0.997	1.000	1.000	1.000
		7,539	7,521	7,500	7,500	7,500
	Time-Division	0.985	0.996	0.999	0.999	0.999
		7,610	7,530	7,506	7,506	7,506
	AP-First	0.995	0.997	1.000	1.000	1.000
		7,539	7,521	7,500	7,500	7,500
	EP-First	0.993	0.995	0.998	0.998	0.998
		7,553	7,534	7,513	7,513	7,513
	Round-Robin	0.994	0.996	0.999	0.999	0.999
		7,548	7,530	7,509	7,509	7,509
FCFS/RRF	No Multiplexing	0.607	0.609	0.999	0.999	0.999
		12,345	12,316	7,509	7,509	7,509
	Time-Division	0.598	0.599	0.997	0.997	0.997
		12,547	12,529	7,519	7,519	7,519
	AP-First	0.607	0.609	0.999	0.999	0.999
		12,345	12,316	7,509	7,509	7,509
	EP-First	0.607	0.609	0.998	0.998	0.998
		12,350	12,320	7,513	7,513	7,513
	Round-Robin	0.607	0.609	0.999	0.999	0.999
		12,345	12,316	7,509	7,509	7,509

Performance & Simulation Time of Memory Models Using LLL Loop 3						
Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.995	0.997	1.000	1.000	1.000
		6,117	6,105	6,086	6,085	6,085
	Time-Division	0.994	0.996	0.999	0.999	0.999
		6,123	6,111	6,093	6,093	6,093
	AP-First	0.995	0.997	1.000	1.000	1.000
		6,117	6,105	6,086	6,085	6,085
	EP-First	0.994	0.996	0.999	0.999	0.999
		6,122	6,109	6,090	6,089	6,089
	Round-Robin	0.995	0.997	1.000	1.000	1.000
		6,117	6,105	6,086	6,085	6,085
FCFS/RRF	No Multiplexing	0.502	0.503	0.999	1.000	1.000
		12,120	12,105	6,094	6,085	6,085
	Time-Division	0.502	0.502	0.997	0.999	0.999
		12,125	12,111	6,101	6,093	6,093
	AP-First	0.502	0.503	0.999	1.000	1.000
		12,120	12,105	6,094	6,085	6,085
	EP-First	0.502	0.503	0.998	0.999	0.999
		12,125	12,109	6,098	6,089	6,089
	Round-Robin	0.502	0.503	0.999	1.000	1.000
		12,120	12,105	6,094	6,085	6,085

Performance & Simulation Time of Memory Models Using LLL Loop 4						
Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.991	0.995	0.995	0.995	0.999
		5,790	5,763	5,763	5,763	5,743
	Time-Division	0.989	0.994	0.994	0.994	0.995
		5,801	5,771	5,771	5,771	5,763
	AP-First	0.992	0.997	0.997	0.997	0.999
		5,783	5,752	5,752	5,752	5,743
	EP-First	0.991	0.997	0.997	0.997	0.998
		5,788	5,754	5,754	5,754	5,747
	Round-Robin	0.992	0.997	0.997	0.997	0.999
		5,783	5,752	5,752	5,752	5,743
FCFS/RRF	No Multiplexing	0.967	0.995	0.995	0.995	0.999
		5,932	5,763	5,763	5,763	5,744
	Time-Division	0.964	0.991	0.991	0.991	0.995
		5,953	5,787	5,787	5,787	5,765
	AP-First	0.967	0.995	0.995	0.995	0.999
		5,932	5,763	5,763	5,763	5,744
	EP-First	0.966	0.994	0.994	0.994	0.998
		5,937	5,772	5,772	5,772	5,748
	Round-Robin	0.967	0.995	0.995	0.995	0.999
		5,932	5,763	5,763	5,763	5,744

Performance & Simulation Time of Memory Models Using LLL Loop 5						
Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.490	0.960	1.000	0.999	1.000
		8,450	4,310	4,139	4,143	4,139
	Time-Division	0.489	0.671	0.674	0.674	0.674
		8,459	6,167	6,141	6,141	6,141
	AP-First	0.490	0.960	1.000	1.000	1.000
		8,443	4,310	4,139	4,139	4,139
	EP-First	0.490	0.961	0.996	0.996	0.998
		8,453	4,307	4,157	4,157	4,148
	Round-Robin	0.490	0.960	0.998	0.998	1.000
		8,443	4,310	4,149	4,149	4,140
FCFS/RRF	No Multiplexing	0.339	0.340	1.000	1.000	1.000
		12,226	12,173	4,141	4,141	4,141
	Time-Division	0.338	0.340	0.674	0.674	0.674
		12,239	12,175	6,143	6,143	6,143
	AP-First	0.339	0.340	1.000	1.000	1.000
		12,226	12,173	4,141	4,141	4,141
	EP-First	0.338	0.340	0.996	0.998	0.998
		12,237	12,172	4,157	4,147	4,148
	Round-Robin	0.339	0.340	0.997	0.997	1.000
		12,226	12,173	4,151	4,151	4,142

Performance & Simulation Time of Memory Models Using LLL Loop 6						
Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.569	0.987	0.995	1.000	1.000
		8,527	4,921	4,880	4,857	4,856
	Time-Division	0.570	0.779	0.783	0.784	0.784
		8,524	6,235	6,201	6,197	6,191
	AP-First	0.570	0.989	0.998	1.000	1.000
		8,526	4,909	4,866	4,856	4,856
	EP-First	0.570	0.989	0.995	0.996	0.997
		8,519	4,909	4,879	4,874	4,873
	Round-Robin	0.570	0.989	0.998	1.000	1.000
		8,526	4,908	4,866	4,857	4,856
FCFS/RRF	No Multiplexing	0.396	0.397	0.997	1.000	1.000
		12,270	12,217	4,869	4,857	4,856
	Time-Division	0.395	0.397	0.783	0.784	0.784
		12,282	12,220	6,205	6,195	6,195
	AP-First	0.396	0.397	0.997	1.000	1.000
		12,270	12,217	4,869	4,856	4,856
	EP-First	0.396	0.397	0.995	0.997	0.997
		12,278	12,222	4,879	4,869	4,873
	Round-Robin	0.396	0.397	0.997	1.000	1.000
		12,270	12,217	4,869	4,857	4,856

Performance & Simulation Time of Memory Models Using LLL Loop 7

Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.624	0.987	0.998	1.004	1.004
		3,499	2,213	2,188	2,175	2,175
	Time-Division	0.623	0.846	0.853	0.860	0.860
		3,506	2,581	2,559	2,539	2,539
	AP-First	0.624	0.987	0.998	1.004	1.004
		3,499	2,213	2,188	2,175	2,175
	EP-First	0.623	0.980	0.969	0.998	0.998
		3,504	2,229	2,253	2,189	2,189
	Round-Robin	0.624	0.983	0.994	1.000	1.000
		3,499	2,222	2,198	2,185	2,185
FCFS/RRF	No Multiplexing	0.390	0.598	0.750	0.999	0.999
		5,603	3,650	2,913	2,186	2,186
	Time-Division	0.389	0.597	0.746	0.857	0.857
		5,621	3,661	2,929	2,549	2,549
	AP-First	0.390	0.598	0.750	0.999	0.999
		5,603	3,650	2,913	2,186	2,186
	EP-First	0.389	0.597	0.746	0.996	0.996
		5,608	3,656	2,926	2,192	2,192
	Round-Robin	0.390	0.598	0.750	0.999	0.999
		5,603	3,649	2,913	2,186	2,186

Performance & Simulation Time of Memory Models Using LLL Loop 8						
Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.451	0.816	0.961	0.981	0.978
		5,035	2,782	2,363	2,315	2,322
	Time-Division	0.450	0.817	0.848	0.857	0.860
		5,052	2,781	2,677	2,651	2,641
	AP-First	0.452	0.830	0.947	0.954	0.961
		5,024	2,736	2,398	2,380	2,363
	EP-First	0.450	0.810	0.909	0.973	0.984
		5,042	2,802	2,499	2,333	2,307
	Round-Robin	0.452	0.817	0.957	0.988	0.992
		5,027	2,781	2,372	2,299	2,289
FCFS/RRF	No Multiplexing	0.281	0.462	0.654	0.971	0.983
		8,068	4,916	3,470	2,340	2,310
	Time-Division	0.281	0.461	0.653	0.853	0.857
		8,078	4,929	3,476	2,663	2,651
	AP-First	0.283	0.462	0.653	0.973	0.964
		8,011	4,918	3,475	2,334	2,355
	EP-First	0.283	0.460	0.655	0.951	0.980
		8,016	4,932	3,469	2,387	2,317
	Round-Robin	0.283	0.462	0.654	0.953	0.964
		8,011	4,916	3,471	2,384	2,346

Performance & Simulation Time of Memory Models Using LLL Loop 9						
Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.489	0.765	0.986	0.994	1.001
		4,469	2,856	2,218	2,199	2,184
	Time-Division	0.489	0.716	0.729	0.730	0.733
		4,472	3,052	2,999	2,993	2,981
	AP-First	0.489	0.765	0.987	0.994	1.001
		4,469	2,856	2,215	2,199	2,184
	EP-First	0.490	0.765	0.981	0.988	0.995
		4,464	2,856	2,229	2,212	2,197
	Round-Robin	0.489	0.765	0.982	0.990	0.996
		4,469	2,856	2,225	2,209	2,194
FCFS/RRF	No Multiplexing	0.197	0.347	0.617	0.941	0.994
		11,099	6,306	3,544	2,323	2,199
	Time-Division	0.197	0.347	0.618	0.728	0.731
		11,091	6,307	3,539	3,003	2,991
	AP-First	0.197	0.347	0.619	0.931	0.994
		11,099	6,306	3,532	2,347	2,199
	EP-First	0.197	0.347	0.619	0.926	0.992
		11,088	6,304	3,534	2,361	2,204
	Round-Robin	0.197	0.347	0.619	0.929	0.994
		11,099	6,306	3,532	2,354	2,199

Performance & Simulation Time of Memory Models Using LLL Loop 10

Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.500	0.843	0.986	0.999	1.000
		5,987	3,550	3,038	2,996	2,994
	Time-Division	0.487	0.702	0.713	0.717	0.717
		6,153	4,267	4,201	4,178	4,174
	AP-First	0.448	0.669	0.650	0.637	0.635
		6,691	4,476	4,605	4,701	4,718
	EP-First	0.499	0.914	0.929	0.937	0.934
		5,994	3,276	3,223	3,196	3,207
	Round-Robin	0.481	0.917	0.934	0.941	0.939
		6,219	3,265	3,205	3,189	3,187
FCFS/RRF	No Multiplexing	0.289	0.592	0.993	0.997	1.000
		10,374	5,057	3,016	3,003	2,995
	Time-Division	0.288	0.591	0.709	0.715	0.716
		10,379	5,062	4,221	4,190	4,179
	AP-First	0.289	0.592	0.491	0.688	0.688
		10,374	5,057	6,098	4,354	4,353
	EP-First	0.288	0.591	0.923	0.934	0.934
		10,379	5,066	3,243	3,204	3,207
	Round-Robin	0.289	0.592	0.929	0.937	0.939
		10,374	5,057	3,224	3,196	3,188

Performance & Simulation Time of Memory Models Using LLL Loop 11

Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.640	0.996	0.997	1.000	1.000
		5,647	3,628	3,624	3,615	3,615
	Time-Division	0.640	0.877	0.878	0.878	0.878
		5,650	4,122	4,119	4,117	4,117
	AP-First	0.641	0.996	0.997	1.000	1.000
		5,643	3,628	3,625	3,615	3,615
	EP-First	0.640	0.998	0.999	0.999	0.999
		5,647	3,624	3,620	3,620	3,620
	Round-Robin	0.641	0.996	0.997	1.000	1.000
		5,641	3,628	3,625	3,615	3,615
FCFS/RRF	No Multiplexing	0.640	0.993	0.997	0.999	0.999
		5,647	3,639	3,627	3,617	3,617
	Time-Division	0.639	0.877	0.877	0.878	0.878
		5,656	4,124	4,121	4,117	4,117
	AP-First	0.640	0.996	0.997	1.000	1.000
		5,647	3,628	3,625	3,615	3,615
	EP-First	0.639	0.997	0.998	0.999	0.999
		5,653	3,625	3,622	3,620	3,620
	Round-Robin	0.640	0.993	0.997	0.999	0.999
		5,647	3,639	3,627	3,617	3,617

Performance & Simulation Time of Memory Models Using LLL Loop 12

Scheduling Policy	Multiplexing Scheme	Interleaving Factors				
		4	8	16	32	64
FMRF/RRF	No Multiplexing	0.849	1.000	1.001	1.001	1.001
		8,340	7,077	7,069	7,069	7,069
	Time-Division	0.848	0.999	1.000	1.000	1.000
		8,345	7,082	7,075	7,075	7,075
	AP-First	0.849	1.000	1.001	1.001	1.001
		8,340	7,077	7,069	7,069	7,069
	EP-First	0.848	0.999	0.999	0.999	0.999
		8,347	7,083	7,083	7,083	7,083
	Round-Robin	0.849	1.000	1.000	1.000	1.000
		8,343	7,079	7,079	7,079	7,079
FCFS/RRF	No Multiplexing	0.800	1.000	1.000	1.000	1.000
		8,841	7,079	7,079	7,079	7,079
	Time-Division	0.778	0.999	0.999	0.999	0.999
		9,095	7,085	7,085	7,085	7,085
	AP-First	0.800	1.000	1.000	1.000	1.000
		8,841	7,079	7,079	7,079	7,079
	EP-First	0.800	0.999	0.999	0.999	0.999
		8,845	7,083	7,083	7,083	7,083
	Round-Robin	0.800	1.000	1.000	1.000	1.000
		8,841	7,079	7,079	7,079	7,079

Simulation Results of Simultaneous Arrivals at the 4-Way Interleaved FMRF/RRF Memory Model

LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	4,550	287	6.3%
LLL2	7,539	1	0.0%
LLL3	6,117	1	0.0%
LLL4	5,790	1	0.0%
LLL5	8,450	420	5.0%
LLL6	8,527	263	3.1%
LLL7	3,499	50	1.4%
LLL8	5,035	176	3.5%
LLL9	4,469	3	0.0%
LLL10	5,987	343	5.7%
LLL11	5,647	263	4.7%
LLL12	8,340	947	11.4%

Simulation Result of Simultaneous Arrivals at the 4-Way Interleaved FCFS/RRF Memory Model

LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	5,108	22	0.4%
LLL2	12,345	1	0.0%
LLL3	12,120	1	0.0%
LLL4	5,932	1	0.0%
LLL5	12,226	333	2.7%
LLL6	12,270	330	2.7%
LLL7	5,603	4	0.0%
LLL8	8,068	11	0.1%
LLL9	11,099	3	0.0%
LLL10	10,374	103	1.0%
LLL11	5,647	500	8.9%
LLL12	8,841	3	0.0%

Simulation Results of Simultaneous Arrivals at the 8-Way Interleaved FMRF/RRF Memory Model			
LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	3,324	398	12.0%
LLL2	7,521	1	0.0%
LLL3	6,105	1	0.0%
LLL4	5,763	1	0.0%
LLL5	4,310	443	10.3%
LLL6	4,921	660	13.4%
LLL7	2,213	126	5.7%
LLL8	2,782	180	6.5%
LLL9	2,856	63	2.2%
LLL10	3,550	719	20.3%
LLL11	3,628	995	27.4%
LLL12	7,077	996	14.1%

Simulation Result of Simultaneous Arrivals at the 8-Way Interleaved FCFS/RRF Memory Model			
LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	3,331	5	0.2%
LLL2	12,316	1	0.0%
LLL3	12,105	1	0.0%
LLL4	5,763	1	0.0%
LLL5	12,173	334	2.7%
LLL6	12,217	331	2.7%
LLL7	3,650	114	3.1%
LLL8	4,916	80	1.6%
LLL9	6,306	98	1.6%
LLL10	5,057	411	8.1%
LLL11	3,639	995	27.3%
LLL12	7,079	996	14.1%

Simulation Results of Simultaneous Arrivals at the 16-Way Interleaved FMRF/RRF Memory Model

LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	3,314	3	0.0%
LLL2	7,500	1	0.0%
LLL3	6,086	1	0.0%
LLL4	5,763	1	0.0%
LLL5	4,139	668	16.1%
LLL6	4,880	661	13.5%
LLL7	2,188	132	6.0%
LLL8	2,363	192	8.1%
LLL9	2,218	160	7.2%
LLL10	3,038	1,918	63.1%
LLL11	3,624	994	27.4%
LLL12	7,069	996	14.1%

Simulation Result of Simultaneous Arrivals at the 16-Way Interleaved FCFS/RRF Memory Model

LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	3,331	5	0.2%
LLL2	7,509	1	0.0%
LLL3	6,094	1	0.0%
LLL4	5,763	1	0.0%
LLL5	4,141	668	16.1%
LLL6	4,869	662	13.6%
LLL7	2,913	113	3.9%
LLL8	3,470	81	2.3%
LLL9	3,544	85	2.4%
LLL10	3,016	1,915	63.5%
LLL11	3,627	994	27.4%
LLL12	7,079	996	14.1%

Simulation Results of Simultaneous Arrivals at the 32-Way Interleaved FMRF/RRF Memory Model

LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	3,307	3	0.0%
LLL2	7,500	1	0.0%
LLL3	6,085	1	0.0%
LLL4	5,763	1	0.0%
LLL5	4,143	668	16.1%
LLL6	4,857	662	13.6%
LLL7	2,175	137	6.3%
LLL8	2,315	205	8.9%
LLL9	2,199	37	1.7%
LLL10	2,996	1,919	64.1%
LLL11	3,615	994	27.5%
LLL12	7,069	996	14.1%

Simulation Result of Simultaneous Arrivals at the 32-Way Interleaved FCFS/RRF Memory Model

LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	3,317	4	0.1%
LLL2	7,509	1	0.0%
LLL3	6,085	1	0.0%
LLL4	5,763	1	0.0%
LLL5	4,141	668	16.1%
LLL6	4,857	662	13.6%
LLL7	2,186	134	6.1%
LLL8	2,340	245	10.5%
LLL9	2,323	69	3.0%
LLL10	3,003	1,919	63.9%
LLL11	3,617	994	27.5%
LLL12	7,079	996	14.1%

Simulation Results of Simultaneous Arrivals at the 64-Way Interleaved FMRF/RRF Memory Model

LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	3,307	3	0.0%
LLL2	7,500	1	0.0%
LLL3	6,085	1	0.0%
LLL4	5,743	2	0.0%
LLL5	4,139	668	16.1%
LLL6	4,856	665	13.7%
LLL7	2,175	137	6.3%
LLL8	2,322	187	8.1%
LLL9	2,184	67	3.1%
LLL10	2,994	1,926	64.3%
LLL11	3,615	995	27.5%
LLL12	7,069	996	14.1%

Simulation Result of Simultaneous Arrivals at the 64-Way Interleaved FCFS/RRF Memory Model

LLL Loop	Simulation Time	Number of Occurrences	Percentage of Simulation Time
LLL1	3,317	3	0.1%
LLL2	7,509	1	0.0%
LLL3	6,085	1	0.0%
LLL4	5,744	2	0.0%
LLL5	4,141	668	16.1%
LLL6	4,856	665	13.7%
LLL7	2,186	134	6.1%
LLL8	2,310	204	8.8%
LLL9	2,199	70	3.2%
LLL10	2,995	1,926	64.3%
LLL11	3,617	994	27.5%
LLL12	7,079	996	14.1%

Appendix VI

Simulation results of different data cache sizes on each Module Controller of an 8-way interleaved memory system are presented here. There are Read-After-Write access hazards in the simulations (loops 4, 5, 6, and 11).

Data cache sizes from 1 to 256 words per Module Controller are investigated. Half of the loops (4, 5, 7, 8, 9 and 10) indicate that the performance of cache size 1 is better than the performance of cache size 2 or 4. These results are due to the cache fetch and replacement algorithms used in the simulations. A prefetch is discarded if the address of prefetch request is mapped into a *Pending* entry. When the cache size is 1, all the addresses of memory requests are mapped into the same cache entry (since there is only one entry), thus most of the prefetches are discarded. When the cache sizes are 2 or 4, the first few prefetches are often mapped to separated entries and memory read operations are issued to prefetch the memory words into the data cache. The subsequent read requests will then be mapped into the entries where the prefetched operands are just stored. Therefore, there are numerous prefetching operations that use memory cycles, but the prefetched operands are replaced before they are used. However, as the data cache size increases, the prefetched operands are more likely to remain in the data cache until called for. Thus, the performance increases as the cache size increases.

Performance & Simulation Time

of 8-Way Interleaved Memory Model with Data Cache

Data Cache Words per Module	Lawrence Livermore Loop			
	LLL1	LLL2	LLL3	LLL4
1	0.479	0.942	0.906	0.767
	6,916	7,963	6,717	8,843
2	0.943	0.997	1.000	0.595
	3,514	7,518	6,084	11,399
4	0.996	1.000	1.000	0.608
	3,327	7,493	6,084	11,158
8	1.002	1.000	1.000	0.633
	3,309	7,493	6,084	10,710
16	1.002	1.000	1.000	0.656
	3,309	7,493	6,084	10,340
32	1.002	1.000	1.000	0.670
	3,309	7,493	6,084	10,127
64	1.002	1.000	1.000	0.703
	3,309	7,493	6,084	9,642
128	1.002	1.001	1.000	0.858
	3,309	7,493	6,084	7,902
256	1.002	1.001	1.000	0.861
	3,309	7,493	6,084	7,875

Performance & Simulation Time

of 8-Way Interleaved Memory Model with Data Cache

Data Cache Words per Module	Lawrence Livermore Loop			
	LLL5	LLL6	LLL7	LLL8
1	0.772	0.464	0.746	0.615
	6,219	11,785	2,928	3,694
2	0.632	0.999	0.428	0.384
	7,600	5,477	5,097	5,912
4	0.810	0.999	0.986	0.323
	5,929	5,477	2,216	7,039
8	0.810	0.999	0.986	0.453
	5,929	5,477	2,216	5,015
16	0.810	0.999	0.986	0.508
	5,929	5,477	2,216	4,471
32	0.810	0.999	0.989	0.878
	5,929	5,477	2,209	2,586
64	0.810	0.999	0.989	0.905
	5,929	5,477	2,209	2,509
128	0.810	0.999	0.989	0.949
	5,929	5,472	2,209	2,394
256	0.810	0.999	0.989	0.949
	5,929	5,472	2,209	2,394

Performance & Simulation Time

of 8-Way Interleaved Memory Model with Data Cache

Data Cache Words per Module	Lawrence Livermore Loop			
	LLL9	LLL10	LLL11	LLL12
1	0.620	0.679	0.989	1.001
	3,527	4,410	7,154	7,069
2	0.256	0.361	1.000	1.003
	8,547	8,283	7,072	7,059
4	0.244	0.370	1.001	1.003
	8,975	8,096	7,067	7,059
8	0.285	0.536	1.001	1.003
	7,668	5,581	7,067	7,059
16	0.649	0.839	1.001	1.003
	3,371	3,570	7,067	7,059
32	0.771	0.883	1.001	1.003
	2,836	3,392	7,067	7,059
64	0.838	0.886	1.001	1.003
	2,608	3,381	7,067	7,059
128	0.859	0.889	1.002	1.003
	2,544	3,366	7,062	7,059
256	0.859	0.889	1.002	1.003
	2,545	3,366	7,062	7,059

Appendix VII

Simulation results for the comparison of the FCFS/RRF scheduling policy and the AP-First and EP-First scheduling policies are presented here. The comparison indicates that memory access conflicts and access hazards are the main factors which can prevent the requests from being serviced according to the priority order.

In LLL loops 4, 5, and 6, access hazards force the requests to be serviced according to the arrival order, so that the performance for the FCFS/RRF scheduling policy and the AP-First (or EP-First) scheduling policy are almost identical. In the simulation of LLL loop 2, the performance of FCFS/RRF policy is worse than the AP-First (or EP-First) scheduling policy when the interleaving factor is 4, and it is as good as the AP-First (or EP-First) scheduling policy when the interleaving factor is infinite.

In a memory model that uses AP-First or EP-First scheduling policies, there are two read request queues: the load data request queue and the alternative load data request queue; there is only one read request queue in the memory model using FCFS/RRF scheduling policy. The probability of having memory conflicts for both requests at the heads of read queues in the AP-First or EP-First scheduling policies are smaller than the probability of having a memory conflict for the request at the head of read queue in the FCFS/RRF scheduling policy. Thus, the probability of selecting a non-conflicting request for service is higher in the AP-First or EP-First scheduling policies than in the FCFS/RRF scheduling policy.

This is the reason that the FCFS/RRF scheduling policy is worse than the AP-First or EP-First scheduling policies when the interleaving factor is 4.

Total Simulation Time of

Priority Scheduling Policy & Read-Request-First Policy

Interleaving Factor	Scheduling Policy	Lawrence Livermore Laboratory Loops			
		LLL2	LLL4	LLL5	LLL6
4	AP-First	37,465	50,459	51,921	59,877
	EP-First	36,165	50,459	51,921	59,877
	FCFS/RRF	39,034	50,464	51,919	59,888
Infinity	AP-First	20,502	36,035	33,188	37,155
	EP-First	20,525	36,058	33,202	37,171
	FCFS/RRF	20,554	36,035	33,187	37,208

Performance of

Priority Scheduling Policy & Read-Request-First Policy

Interleaving Factor	Scheduling Policy	Lawrence Livermore Laboratory Loops			
		LLL2	LLL4	LLL5	LLL6
4	AP-First	0.489	0.593	0.524	0.426
	EP-First	0.506	0.593	0.524	0.426
	FCFS/RRF	0.469	0.593	0.524	0.426
Infinity	AP-First	0.893	0.830	0.820	0.687
	EP-First	0.892	0.829	0.819	0.687
	FCFS/RRF	0.891	0.830	0.820	0.686