National Library
of Canada

Canadian Theses Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale
du Canada

Services des thèses canadiennes

# CANADIAN THESES

# THÈSES CANADIENNES

## NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

## AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ
MICROFILMÉE TELLE QUE
NOUS L'AVONS REÇUE

NL-339 (r.86/06)

Canada

# EVALUATION OF SOME DISTRIBUTED FUNCTION ARCHITECTURES FOR ARRAY PROCESSING DATA MANIPULATION

by

John Jonas Gudaitis

B. Sc., University of Missouri at Rolla, 1975.

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

© John Jonas Gudaitis 1985

SIMON FRASER UNIVERSITY

July, 1985

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-30711-0

## APPROVAL

Name: John Gudaitis

Degree: Master of Science

Title of thesis: Evaluation of Some Distributed Function Architectures for Array Processing Data Manipulation

Examining Committee:

Chairperson: Thomas K. Poiker

Richard F. Hobson
Senior Supervisor

Brian V. Funt

Louis J. Hafer

Hassan K. Reghbati
External Examiner
Assistant Professor
Department of Computing Science
Simon Fraser University

Date Approved: 3 July 1985

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Evaluation of Some Distributed Function Architectures for Array Processing Data Manipulation

Author:

(signature)

John Jonas Gudaitis

(name)

August 12, 1985

(date)

**ABSTRACT**

Parallelism has been used extensively in supercomputer systems to improve performance but has had limited application in microcomputers. A structured architecture machine (SAM) was designed for use as a high performance engineering workstation. It has a distributed function architecture that allows modular extensibility to increase performance. SAM uses indirect high-level language execution to give good performance while providing the user friendly interface associated with interpretive systems. This thesis investigates data manipulation strategy for SAM architectures through simulated execution of array processing benchmarks. Attached slave processors provided the simplest method for optimizing system performance. Maintaining good firmware structure without performance degradation requires assistance from special hardware to support control constructs and device interfacing. Performance of benchmark execution on SAM is more than an order of magnitude better than execution of compiled C versions running on a VAX750 and a SUN workstation.

## TABLE OF CONTENTS

# LIST OF TABLES

## LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

This thesis explores some aspects of the performance of a structured architecture machine developed for the interpretation of high-level languages (HLL). It is hoped that the results will help dispel some myths associated with interpretive HLL computer systems and thus allow such systems to become more popular. This thesis will examine some strategies to improve data manipulation performance, borrowing ideas used in supercomputer systems and applying them to a microcomputer system. A brief introduction to past efforts in these areas follows. Section 1.1 reviews efforts in HLL support. The performance of instruction sets is then discussed in section 1.2. Section 1.3 discusses the use of structured architecture to design cost effective systems. Finally, an overview of the thesis is given in section 1.4.

## 1.1 High-Level Language Support

HLL support has been a subject of investigation ever since the days of the first computer, with many researchers attempting to reduce the semantic gap between high-level language concepts and the underlying computer architecture that actually supports these concepts. Reviews of this research can be found in [12]

and [72]. Chu[13] categorized computer systems architecture according to the proximity of the HLL seen by the user to the machine language actually executed.

Type 1, von Neumann, is used in most commercial computer systems. It is characterized by a low level register-oriented instruction set, generally requiring a complex compilation process to convert a HLL program into machine language code. Complex instruction sets such as that of the VAX-11[21] provide some instructions oriented toward HLLs but are too limited to have much effect on reduction of the semantic gap. This type of architecture has been criticized by Backus[6] and Chu and Abrams[16] for contributing to the so called "software crisis". They believe that the tedious design - edit - compile - load - run - debug process lowers programmer productivity. Efficiency considerations have influenced the design of HLLs to be supported by compilation. Very high level languages (VHLLs), such as APL, Lisp, Prolog, NIAL that support program development at a high level of data abstraction, are usually interpreted by software on this type of architecture. This extra layer of software causes the perceived inefficiency of language interpretation.

Type 2 architectures raise the level of the machine language by supporting interpretation of syntax-oriented intermediate code. While this reduces the semantic gap, it does not reduce the necessity for large subroutine libraries or improve the software development process. Borroughs B5500 is an

example of this architecture supporting algol.

Type 3 architectures are the indirect execution type exemplified by the Symbol system [64]. The main improvements over type 2 are a hardware translator and an improvement in proximity for the intermediate polish string language.

Type 4 is the direct execution architecture studied by Chu, Bloom, and others. There is no intermediate language so therefore no semantic gap. All user software and system programs are written in the HLL.

Dietzel and Patterson [20] have suggested a High-level language computer system as a possibility for a more productive environment for software development. Their definition concerns only the interface presented to the user, so that type one systems are included if the operating system insulates the user from the lower layers of software. This could include ROM based BASIC microcomputer systems. Although this concept can be supported with compilation, an interpretive high-level language system offers benefits especially for VHLLs.

## 1.2 Instruction Set Performance

The instruction set selected (user architecture) affects the software development process, but performance also determines user acceptance. The user architecture must allow efficient implementation. Compilation may not achieve this goal for applications where source code is changed frequently.

Flynn and Hoevel [31] have pointed out the inefficiencies of the compilation approach on conventional machines. They derived a directly executable language (DEL) that reduced the semantic gap and thereby reduced the size of the intermediate code. This should improve performance by reducing the number of instruction fetches, executes, and memory references. Their DEL featured a transformationally complete instruction set which they felt would simplify compilation. They also showed how to design an efficient system to interpret a DEL. While their research showed significant improvement over other complex instruction sets, other methods have produced better run time performance. Reduced instruction set computers (RISCs), for example, seem to give better performance, at least for some HLLs[74].

Some recently designed computer systems use statistical methods for instruction set design [36,75,68]. The most frequently used operations in a HLL are given a corresponding machine instruction. Instruction set usage obeys the 20-80 rule [97], so only a small subset of a language needs to be implemented efficiently to get good performance. Actually, the main factors in scalar block-structured HLL performance have been found to be procedure call and variable binding[57,76]. Thus the fast overlapped register banks of RISC may be largely responsible for its good performance[35].

Thurber[89] notes a lack of support for HLL data structures in computer architectures. An example of data structure support

is vector processing. One approach to providing support for arithmetic operations on arrays has been to use vectorizing compilers running on supercomputers. While this has had some success in improving performance, it forces users to write more complex low level code. Such a programming environment requires control coding at a level lower than that of the machine hardware and has been shown to provide less than full performance [22]. Language extensions are more useful but usually limited in the data manipulations allowed. It has been noted that supercomputer development has spawned novel high performance hardware but has not contributed to advances in software [19]. Hardware improvements will be more effective when they are transparent to the user. VHLLs with full support for vector processing should offer a better environment for program development and an opportunity for performance improvement through language directed design.

Chu [14,15] promotes the advantages of a direct execution computer, especially its conceptual simplicity. This architecture has been criticized for poor performance [44,57]. Hardware interpretation can improve this problem, but this architecture still suffers from redundant syntax analysis, which adds unnecessary hardware costs to the system. With enough hardware support, such a system may be competitive with other methods in run time performance. However, indirect execution can provide a user interface indistinguishable from direct execution but with lesser hardware requirements and reduced redundant

processing. Hobson[40] has extended the DEL approach to interpreted languages. He has derived a directly interpretable language (DIL) with a one to one correspondence to source operations that permits recovery of the source. Thus only one copy of a program need be maintained with obvious benefits when programs are modified.

### 1.2.1 Increasing Performance of Instruction Set Execution

While the instruction set puts limits on performance, actual performance is dependent on implementation strategies. Speed increases due to technology advances are limited so further performance enhancement must come from changes in system architecture. This section examines some methods that have been used to reduce instruction execution time.

Consider the interpretation of a typical HLL instruction as shown in figure 1-1. It consists of a sequence of primitive actions which accomplish the required task. For scalar instructions the entire sequence is repeated for each instruction. Compiled code does not require steps 2 and 3 since verification can usually be done by a compiler for strongly typed languages. For vector instructions the last three actions need to be repeated for each element of the vector. Therefore increasing performance of a scalar machine involves all phases of instruction execution, while vector machine performance may be enhanced by reducing the execution time of the loop section,

especially for long vectors.

Instruction pipelining has been used to improve performance of mainframe computer systems for scalar instruction sets. To achieve higher performance, some later phases of an instruction can be overlapped with the early phases of the next instruction. To accomplish this requires some extra hardware resources. In conventional high performance von Neumann machines, centrally controlled instruction and data manipulation units are used to allow concurrent execution of phases [7]. This requires complex control features to handle scheduling and data dependency problems [48]. Statistics on HLL instruction usage indicate that such methods may not significantly improve scalar performance, since subroutine calls predominate in performance determination. Branches and a scarcity of functional instructions further

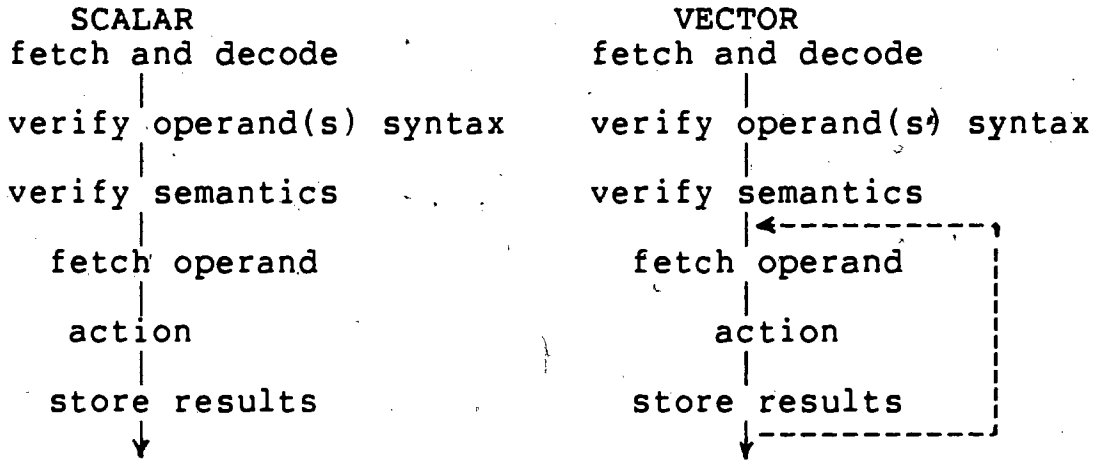


Figure 1-1. Instruction execution sequence.

reduce the performance potential of scalar overlapped computers.

Multiple processing units have been used to increase performance [24, 25, 66]. Using n processors has potential for n fold speedup. In practice, multiple homogenous processor systems suffer from scheduling overhead and memory access contention. Some of these systems exhibit breakover point behavior where addition of a processor can actually decrease system performance [88]. Memory and processor communication requires complex bus interconnections [55]. It is also difficult to express some problems in a form that can be used on such a system.

Thurber[89] suggests using a functionally distributed architecture to overcome these problems. A pipeline is an example of this architecture. A multiprocessor distributed function architecture (DFA) can achieve overlapped performance without complex central control. The Symbol computer system mentioned earlier is an example of this type of architecture designed for a single HLL. The Symbol system offered improved performance because of its modular multiprocessor architecture which allows overlapped operation of the translation and execution processes. Unlike the homogenous multiple processor architecture, this system consists of heterogenous processors each designed to perform specific functions. With appropriate functional partioning and interfaces the design of each module is independent of the others.

In a typical overlapped computer, dedicated hardware modules can be used to achieve good performance since machine

instruction execution is very regular. The greater diversity in high-level language instructions makes the control problem for overlapping more difficult. The problem is further complicated in languages allowing vector operations since vector length determines the ratios for phase execution times. A distributed function computer can compensate for timing variablity by the use of queues between processors to absorb variation in workload. Use of queues can also increase performance. This has been shown in a scalar Cray 1 type architecture[81] and a data flow machine model [52].

## 1.3 Structured Architecture

While the cost of hardware components has dropped rapidly, the design phase of system development has become increasingly costly. Structured architecture attempts to reduce the complexity of computer system design by borrowing some of the principles learned from structured programming. Modularity is just as important in computer system design as in software design. Parnas [73] suggests that system decomposition be based on module independence. DFA architectures satisfy his criteria since each module is designed for a particular class of functions. Encapsulation of related functions also can meet the VLSI constraint of minimizing chip pin count. Conventional computer systems do not distinguish between control computation and data manipulation. A resource may be shared between a control operation and a data manipulation. This can lead to

extra interaction between independent processes because of possible contention for the resource. Shared resources must also be designed to accommodate both types of processing. An example is a multiplier used both for array indexing and a user specified multiplication of some data.

The size and complexity of current microprogramming projects require new methods for firmware development. This has given rise to a new field of study - firmware engineering. User microprogramming and large system microprogramming have led to a need to upgrade microprogramming techniques. In particular, the horizontal microprogramming used in supercomputers is only appropriate for static architectures.

A structured architecture machine (SAM) [38] has been designed with a modular extensible architecture for indirect high-level language interpretation. The aim is to develop a single user workstation that will offer performance comparable to the usual execution of compiled code, with enhanced performance for array manipulation and special engineering applications. Microprogramming is used for control rather than hardware because of its greater flexibility. This is especially important for an experimental system undergoing frequent modifications. Language independent features can later be supported by hardware to improve performance. Seldom used language features can be implemented in a subset of the HLL with very little performance penalty.

The SAM project differs from Flynn and Hoevel's work in several respects. It is aimed at interpretation of DILs rather than compiled DELs. SAM uses a language directed architecture rather than microprogramming of an unbiased host microengine. Functional task partioning results in separate modules for environmental control, program management, and data manipulation. This allows parallel execution of the different stages of instruction execution.

## 1.4 Overview of Thesis

### 1.4.1 Thesis Goals

This thesis evaluates the performance of some system designs during the data manipulation phase of DIL interpretation. Structured architecture design is used to derive a system suitable for DIL interpretation. Hobson [42] has already studied improvements in the fetch-decode and verification stages of execution, introducing a hardware operand verification unit (OVU) to reduce semantic verification time. This study concentrates on improvement of the later stages of instruction execution. In extending the SAM architecture, arithmetic units are added to provide overlapped operation.

Execution time is not the only factor in measuring performance. Program development time, compile time, and debug time reduce performance, especially for programs with a low number of production runs. Interpretive systems perform better

in these respects but such measures are difficult to quantify, so this thesis measures execution time performance. Even for this aspect, the results show that a properly designed interpretive system offers performance comparable to the usual execution of compiled code.

Since the SAM project is aimed at development of a low cost high performance single user HLL system, implementation methods must be cost effective. Techniques used for performance improvement on supercomputers may not be appropriate. For example, 64 bit data buses are too costly unless this size of data is used frequently. A goal of the SAM project is to find ways to reduce microprogramming complexity. A multiprocessor system, each unit vertically microprogrammed, offers much more flexibility than a central horizontally microprogrammed system. Statistical analysis can be used to select instructions to be supported directly.

### 1.4.2 Methods

The effect of architectural changes was measured by simulating benchmark execution. Microcode interpreters were coded in microAPL [41]. An architecture support package written in APL supports hardware modeling. Introduction of new architectural features required modification to this package. A discrete event simulator was written to support similation of multiple processors. This was written in APL and has a structure almost identical to the system architecture. A top level module

handles timing of transfers to the appropriate module at the next level. These modules (PMU, DMU, APU) then simulate instruction execution in the corresponding physical modules.

During system 1 simulation, statistics were gathered on the amount of time spent in each phase of instruction execution. This information then guided task partioning for SAM extensions.

Although this study concentrated on supporting vector instructions, benchmark performance using a scalar DEL was also determined for the simplest SAM system. A comparison with vector performance revealed that it was not worthwhile pursuing methods to enhance scalar data manipulation performance.

### 1.4.3 Organization of Thesis

The results of this study are organized into 4 chapters. Chapter 2 introduces some data manipulation strategies and examines performance limitations. Two different microarchitectures are used for SAM building blocks. Chapters 3 and 4 explore implementions of these strategies using the building blocks and their influence on SAM's performance. Finally, chapter 5 summarizes the results and compares the performance of SAM to other machines.

### 1.4.4 Assumptions

Execution of an ADEL type instruction set[40] is assumed in this study for the vector HLL. No specific HLL is assumed

since syntax details are handled by a program management unit (PMU), but APL and ADEL are used as typical examples of vector oriented languages. In this study our concern is with execution of internal DIL code accessed by the data manipulation unit (DMU) after completion of verification. Only those instructions required for benchmark simulation are defined and implemented.

A numeric type with varying size for variables is assumed. The size varies as needed to maintain precision. Many HLLs base type distinctions on arbitrary historic considerations. Furthermore, actual physical implementation of the type is system dependent for the same HLL. Variables are assumed to be local to the current environment since binding of non local variables is implementation and language dependent. Local variables are accessed directly from the current data segment that was set up on entry to the current environment. The operand syllable is used as a direct index into the current data segment. APL requires a slightly different binding method and an implementation on SAM will use hardware assist in variable binding [42]. The above model was used for this study since the hardware design was not complete.

# CHAPTER 2

## SYSTEM MODELS AND EXPECTED PERFORMANCE

In this chapter, some system architectures are presented and idealized performance limits for two benchmarks are derived by considering the number of parallel resources available along with the maximum data flow available in the data paths of the system models. Real system implementations and performance will be examined in chapters 3 and 4.

Flynn and Hoevel used a nonfunctional ratio to measure instruction set inefficiency. However, their NF ratio suffers from a favoritism for complex instruction sets and is not a good measure of run time performance. Performance estimates are distorted since complex instructions, especially variable length ones require longer decode and execution times. This thesis uses a functional ratio of implementation (FRI) to measure the proportion of time spent on functional calculation. FRI is defined as the ratio of time spent in a functional calculation divided by the total time spent. A ratio of one means that there is no overhead spent in moving data to where it is acted upon. This can occur in an associative processor or in a conventional processor with full overlap. FRI can be greater than one if multiple arithmetic units are used concurrently or if a pipelined arithmetic unit is used.

Three versions of SAM are considered in this thesis, a minimal functional system and 2 others that expand DMU to increase performance.

## 2.1 Benchmarks

Two primary benchmarks are used to compare the performance of architectural modifications. Benchmark one is a simple addition or multiplication of two vectors of equal length. It is expressed as a 5 syllable DIL instruction

DLR,D,L,R,OP.

This and simple variations of it are the most frequently used arithmetic instructions in languages like APL and therefore important to system performance [8,9,18,40].

Benchmark two is matrix multiplication. Its DIL form consists of 6 syllables

DOTDLR, D, L, R, OP1, OP2.

While not used frequently by average users, its long execution time and complex data accessing make it important to performance, especially in an engineering workstation environment. We consider the standard brute force method, cf. figure 2-1(a), and also a variation used in high performance vector machines, cf. figure 2-1(b). These algorithms multiply a 1 by m matrix with a m by n matrix. Other methods such as Strassen's [39,92] are not well suited to vector or cache machines since data accesses are not sequential.

```
for i=1 to l
    for j=1 to m
        C[i,j]=0
        for k=1 to n
            C[i,j]=C[i,j] + A[i,k] * B[k,j]
```

(a)

```
C = 0
for i=1 to l
    for k=1 to n
        for j=1 to m
            C[i,j]=C[i,j] + A[i,k] * B[k,j]
```

(b)

Figure 2-1: Algorithms for Matrix Multiply.

## 2.2 System 1 Performance

System 1, cf. figure 2-2, is a minimal 3 processor system. It consists of an environment control unit (ECU) for user interaction, a program management unit (PMU) to handle instruction sequencing, and a data manipulation unit (DMU) to fetch and process data. A more detailed description of SAM is available in other publications [38].

ECU accepts user input, translates HLL source input to a linear DIL form, and initiates program execution when requested. PMU fetches DIL code from segmented memory, verifies operand syntax, and sends verified code to DMU. Some control constructs can be handled entirely within PMU. DMU takes DIL code from the PMU - DMU interface, verifies operand semantics, fetches operand data from a data segmented memory, performs specified actions on the data, and returns results to segmented memory.

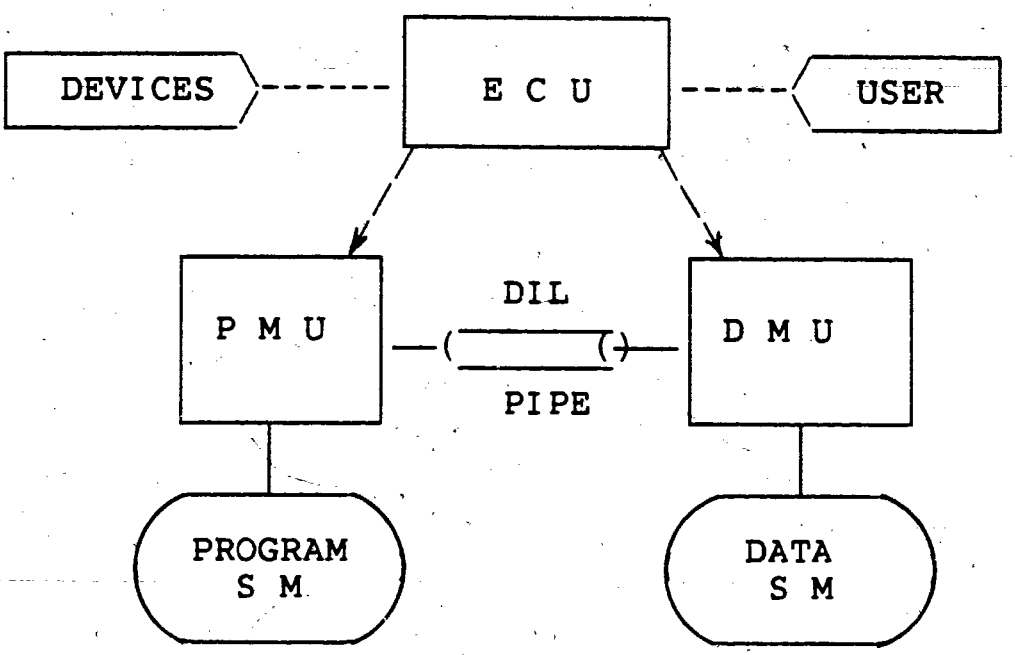


Figure 2-2: Minimal SAM system.

This analysis is mainly concerned with vector loop performance. For arrays of reasonable length, setup time should have minimal effect on performance. Consider the execution of the loop section of a typical diadic vector instruction.

| fetch left operand | fetch right operand | perform action | store results | loop |

t ->

Performance for one processor is determined by the sum of the times taken for each phase. Run time loop cost is

$$T_{loop} = t_{lf} + t_{rf} + t_{action} + t_{store} + t_{overhead} \quad (2\text{-}1)$$

Usually $t_{lf} = t_{rf} = t_{fetch}$ and ideally there is no overhead, so

$$T_{loop} = 2\, t_{fetch} + t_{action} + t_{store} \quad (2\text{-}2)$$

$$\text{Instruction time} = n \times T_{loop} + t_{setup}. \quad (2\text{-}3)$$

$$FRI = t_{action} / (T_{loop} + t_{setup} / N). \quad (2\text{-}4)$$

With a single processor, the only way to improve FRI and increase system performance is to speed up memory transactions.

Now consider algorithm (a) for matrix multiply in figure 2-1, henceforth called algorithm 1(a). Performance mainly depends on execution time of the inner loop. In the inner loop we need to fetch sequential elements of A, fetch elements of B in column order, multiply them and add the result to a local running sum. Thus, for system 1 with no parallelism, inner loop cost is

$$T_{innerloop} = t_{sf} + t_{nsf} + t_{mul} + t_{add} \quad (2\text{-}5)$$

where sf is sequential fetch and nsf is nonsequential fetch. and

$$FRI \simeq ( t_{mul} + t_{add} ) / T_{innerloop}. \quad (2\text{-}6)$$

This is the dominant term for cost since it must be done l m n times.

For algorithm 1b which calculates complete rows of the result, all accesses are sequential. This improves performance if sequential accesses are faster than nonsequential accesses. The cost for the inner loop becomes

$$T_{innerloop} = t_{sf} + t_{mul} + t_{sf} + t_{add} + t_{store}. \quad (2\text{-}7)$$

Note that A[i,k] is a constant that only needs to be loaded once at the beginning of the loop. Comparing algorithm a with algorithm b and assumming that a sequential store takes the same time as a sequential fetch, we find that a is faster if sequential fetch = nonsequential fetch; they are equal if

sequential fetch = 2 nonsequential fetch; and b is faster if sequential fetch < 2 nonsequential fetch.

## 2.3 System 2 Performance

In system 2, auxiliary special function units (SFU's) are added on DMU's external BUS to assist with its processing load, cf. figure 2-3. In this study only arithmetic slave units are considered. These extra arithmetic processors can be used to improve DMU performance especially for complex actions such as multiply, divide, and floating point operations. System complexity is reduced if the processors also perform integer arithmetic. Performance can be further increased by concurrent execution of DMU and slave processors. Arithmetic units can be attached as either separate SFU's or a chip set can be attached as a single SFU with a shared data buffer.

We now consider overlapped performance for this system. Performance without overlapping would be the same as in system

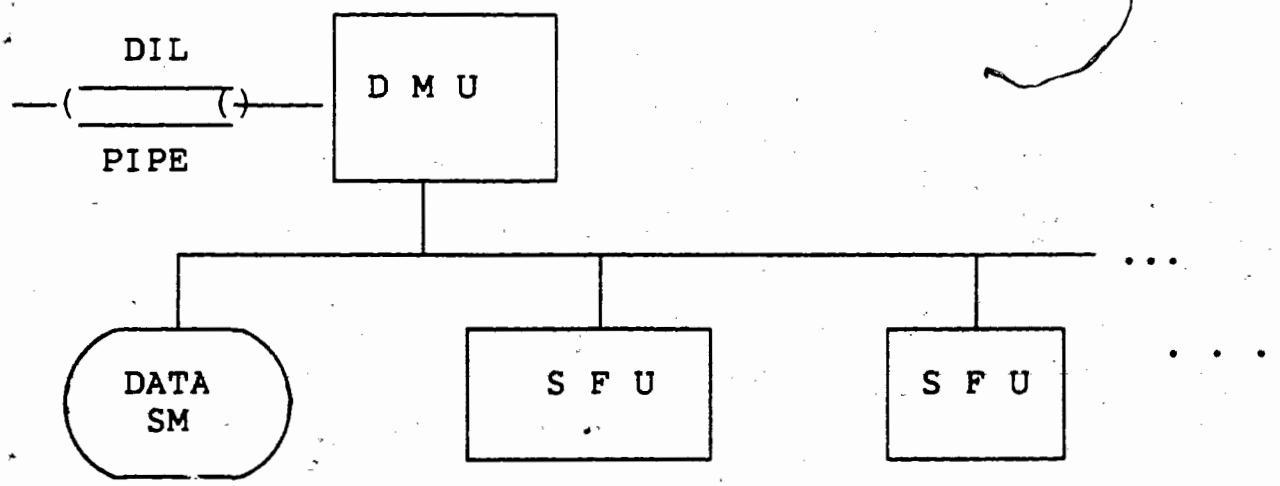


Figure 2-3: DMU with Auxillary SFU''s.

one but with decreased action time for multiply. Action time now includes time to communicate action codes and/or synchronize SFU execution.

At this point some aspects of the interface between units need to be considered since this affects the amount of overlap that can be achieved. Assume that SFUs are connected as slaves to DMU with actions initiated by DMU. The algorithm for overlapped execution, cf. figure 2-4, uses a one stage software pipeline [59] to support overlapped operation.

```
  fetch left operand
  fetch right operand
  start action
DO loop (vector length times)
  fetch left operand
  fetch right operand
  store previous result
  start action
end
store last result
```

Figure 2 - 4 : Overlapping data fetch with action.

Operands are loaded, an action initiated, and then new operands are loaded while the action takes place. When available, the result is stored and a new action started. The operand registers must be buffered or isolated so that new values can be loaded without affecting the current operation. If the output register is unbuffered the result must be stored before a new action is initiated and therefore the store and action phases cannot be overlapped. Figure 2-5(a) shows timing for the unbuffered output case and figure 2-5(b) the buffered output case.

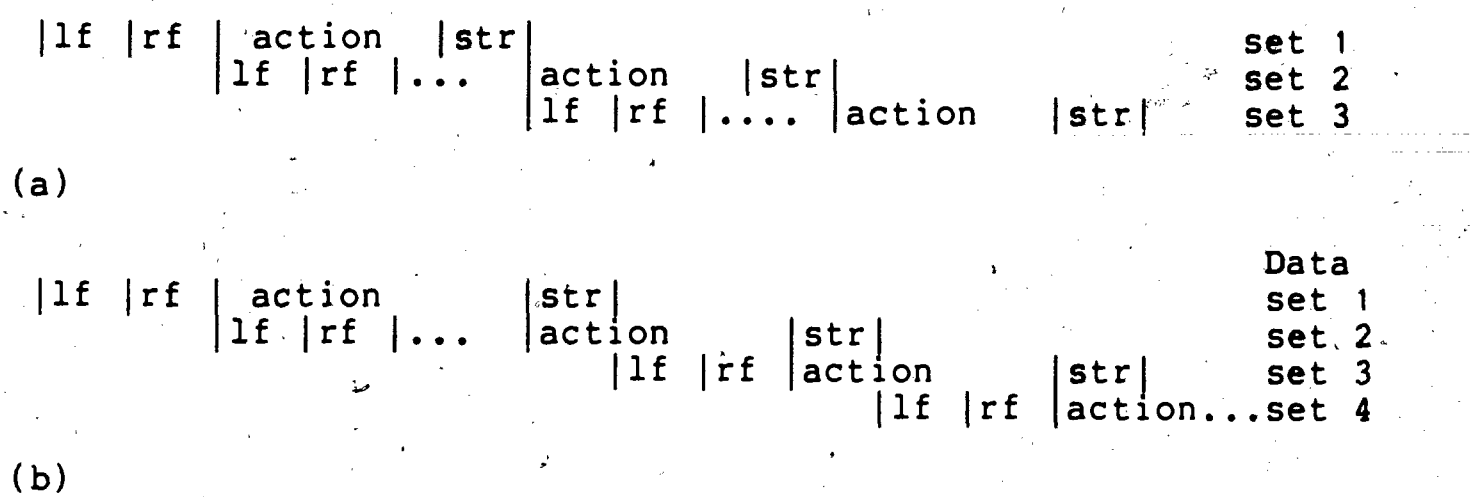


Figure 2-5: Effect of Buffering on Timing.

The loop time performance of case (a) for benchmark 1 is

$$T_{loop} = MAX\{ 2t_{fetch}, t_{action}\} + t_{store} \quad (2\text{-}8)$$

and

$$FRI = t_{action} / ( MAX\{2 t_{fetch}, t_{action}\} + t_{store}) \quad (2\text{-}9)$$

FRI cannot be 1 since the denominator above cannot be smaller than $t_{action} + t_{store}$.

For case (b),

$$T_{loop} = MAX \{ 2t_{fetch} + t_{store}, t_{action}\} \quad (2\text{-}10)$$

Here FRI approaches 1 for longer actions if $t_{action} > 2 t_{fetch} + t_{store}$.

Benchmark 1 was unaffected by the method used to attach SFU's. Benchmark 2 will be affected since results from one SFU must be sent on the bus to another if actions are performed in separate SFU's. For algorithm 1 (a), cost of the inner loop is

$$Max\{ t_{sf} + t_{nsf} + t_{bus} , t_{mul} + t_{add} \} \quad (2\text{-}11)$$

if addition and multiplication are done on separate units. $T_{bus}$ is the time required to move results from one unit to the input

of another. If a single unit is used,

$$T_{innerloop} = Max\{ t_{sf} + t_{nsf}, t_{mul} + t_{add} \}. \quad (2\text{-}12)$$

Note that the first iteration is not overlapped thus reducing performance.

For algorithm 1b with memory accesses overlapped with actions the cost is

$$Max\{ 2\, t_{sf} + t_{store} + t_{bus}, t_{mul} + t_{add}\} \quad (2\text{-}13)$$

if different SFU's are used and

$$Max\{ 2\, t_{sf} + t_{store}, t_{mul} + t_{add}\} \quad (2\text{-}14)$$

if a single SFU is used.

## 2.4 System 3 Performance

System 3 uses an independent arithmetic processing unit (APU), cf. figure 2-6, capable of chaining multiple actions. System 3 ideal performance for benchmark 1 is the same as system 2 with complete buffering.

For benchmark 2, cost of the inner loop for matrix multiply algorithm 1(a) becomes

$$T_{innerloop} =$$
$$Max\{ t_{sf} + t_{nsf}, t_{mul}, t_{add}\}. \quad (2\text{-}15)$$

In this case the first 2 iterations are not overlapped.

For algorithm 1 (b), innerloop performance is

$$Max \{ 2\, t_{sf} + t_{store}, t_{mul}, t_{add} \}. \quad (2\text{-}16)$$

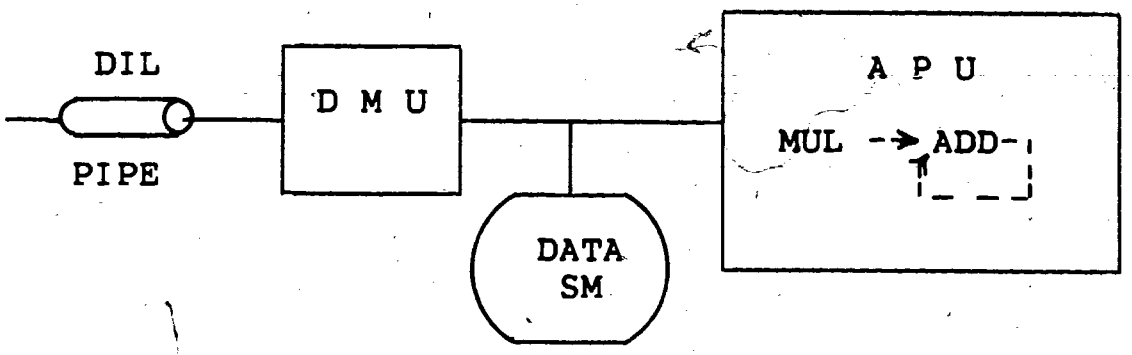


Figure 2-6: System 3 - Independent APU.

# CHAPTER 3

## SAM 0.5

The first version of SAM used a building block module based on a Micros MK16 [71], for PMU and DMU. Figure 3-1 shows a block diagram of this module. Important features are a data stack(DS), a data buffer(DB), and a segmented memory(SM) accessed through windows(WDO), or set of currently open segments. In addition a microprogram stack(MSTACK) supports a microprogram call mechanism. Figure 3-2 shows details of the MK16 internal architecture. See glossary for definitions of other terms. It is essentially a two bus single accumulator microengine. Dyadic

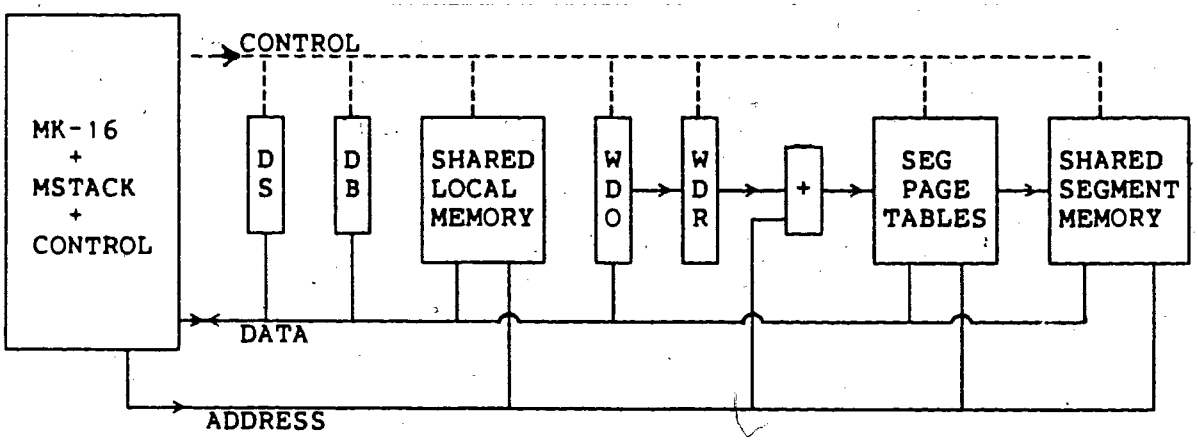


Figure 3-1: SAMjr Schematic.

microoperations take 2 or 3 cycles depending on which source and destination registers are used. MK16 is controlled by a 32 bit microword. Eight more bits were added to control external features via the control bus and to simplify microprogramming. These bits select the SFU's that will transmit or receive bus data. The least significant 4 bits select the SFU register to be used. This must be the same in both source and destination SFU's. Internal and external microoperations proceed concurrently. SAM 0.5 design and simulation was based on availability of a 4 MHz MK16 as promised by Micros Corporation. The SAMjr prototype used an LSI emulator. The clock cycle was extended to 333 nsec to accommodate external data flow.

The previous chapter discussed theoretical performance limits. We now consider implemention on a concrete system. We may not be able to achieve the theoretical performance because of system objectives such as modular structure of firmware, the use of vertical microprogramming, and reasonable cost. A real implementation must consider details such as special function unit synchronization overhead and the detection of overflow from an action. Supporting dynamic precision requires automatic recovery from output precision changes. An MK16 emulator was used to obtain concrete results, introducing architectural changes to evaluate the effect on microprogram structure and performance. Microinstructions are described using a set of mnemonics, which are descibed in a glossary at the end of this chapter. These are executable APL functions describing data flow

and side effects of an microoperation. During simulation, they are executed as a result of microprogram execution. Timing calculations were appended to microinstructions to calculate run time performance during simulation.

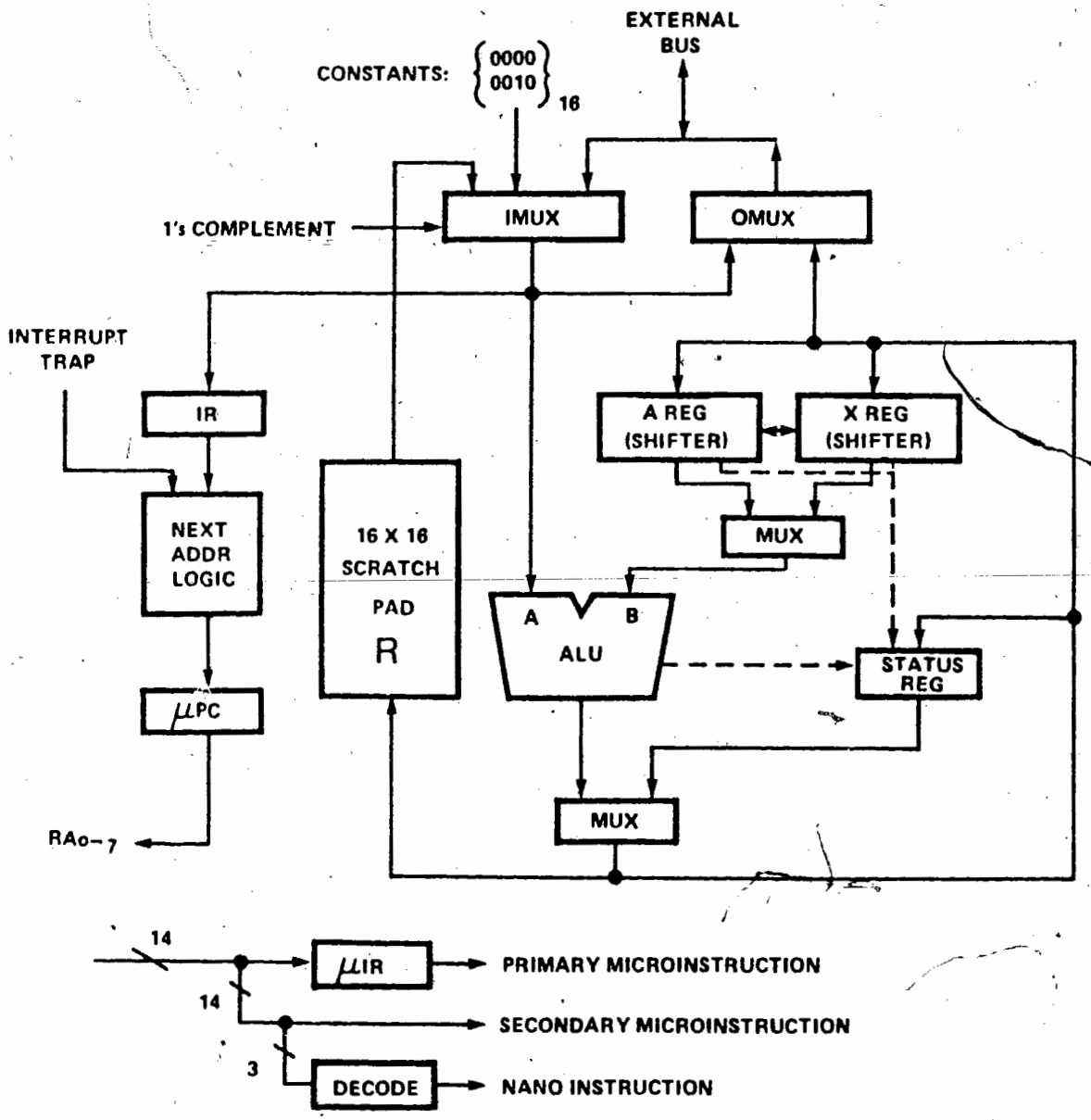


Figure 3-2: Micros MK16 Schematic.

## 3.1 System 1

### 3.1.1 Microprogram structure

One goal of this project was to examine how microprogram structure affected performance, i.e. Can good modular design coexist with high performance? Figure 3-3 shows the instruction execution hierarchy. DIL code execution proceeds from instruction fetch to format execution using a table driven EXEC, a variable procedure call mechanism. The format procedure performs verification, checks operand rank, and calls the

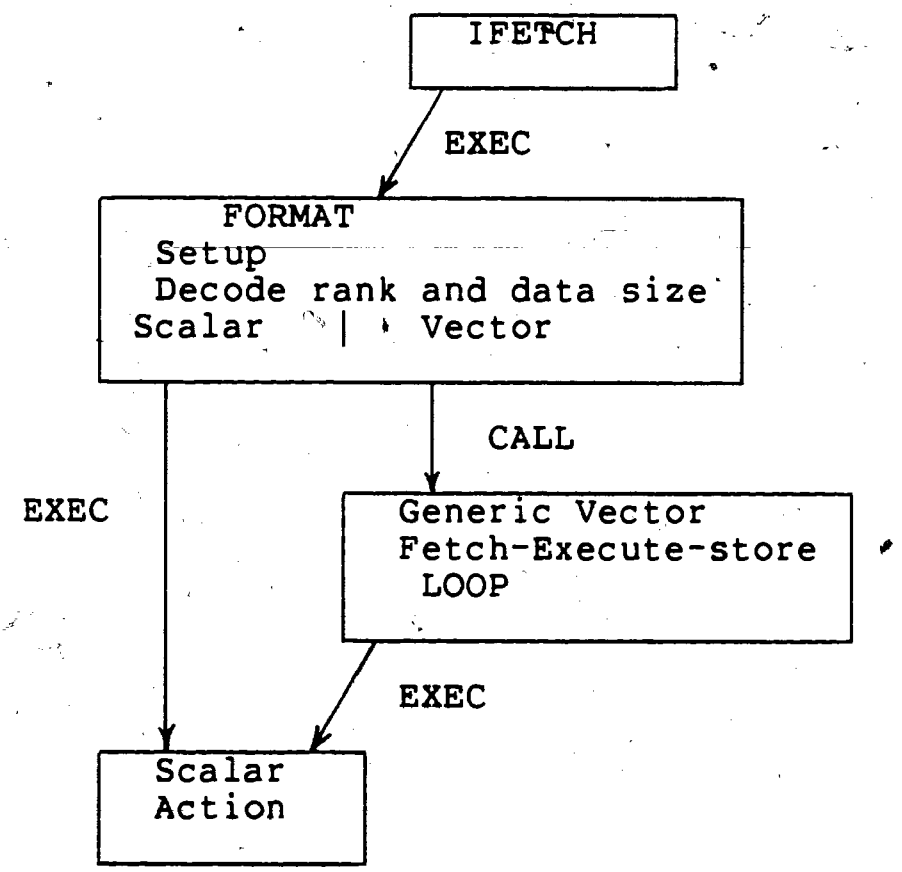


Figure 3-3: DMU instruction execution.

appropriate rank and data size specific fetch routines. For array operands, operand sylables provide an index into an array descriptor, which gives rank, size, and array data location in segmented memory. The action is EXEC'd from within the vector control microprogram. Dyadic action routines were implemented with a standardized interface, with arguments in DB[left] and DB[right] and results returned in DB[dest].

An objective in implementing the algorithms was to find a microprogram structure that is not costly in execution time. For vector algorithms, loop code is the most important element in determining performance. The first efforts aimed at generality to conserve microcode space. Consider the first benchmark, a DLR vector instruction. The first attempt, cf. figure 3-4, will be used to illustrate microcoding with microAPL. This is a quite general microprogram that works for any of the defined numeric precisions and is modularized with no performance penalty with calls to Lfetch, Rfetch, and STORE routines which can be shared with other formats. Point to point data transfer is indicated by the left arrow, "<—". AR and XR refer to A reg and X reg respectively in figure 3-2. DB is the external data buffer in figure 3-1. R refers to one of the 16 registers in the scratchpad in figure 3-2.

Loop execution starts with fetches of left and right operands through CALLs to left operand fetch, LFETCH, and right operand fetch, RFETCH, subroutines. The status register, SR, is loaded with an operand size tag to allow fetch subroutines to

```
    VLOOPG
 C   DMU General vector control
 C   R[count] is loaded by calling program
    LOOP:
      CALL 'LFETCH' Δ SR <-R[LTAG]
      CALL 'RFETCH' Δ SR <-R[RTAG]
      EXEC DB[OP] Δ SR<-R[TAG]
      ->ERROR IF OVFL
      CALL 'DSTORE' Δ SR<-R[TAG]
      R[COUNT]<-SNZ DEC1 R[COUNT]
     ->LOOP IF ¬ZERO
      clear status and return
    ERROR: CALL 'RECOVER'        "Data size overflow"

   LFETCH
C  General left operand fetch
C  Fetch operand and leave in buffer DB[L0...L3]
C  SR contains left size tag on entry
    R[LINDX]<-INC2 SRW R[LINDX]                  "Memory address"
    ->0 if INT16 Δ DB[L0]<-S[WORD]       "Memory data"
    R[LINDX]<-INC2 SRW R[LINDX]
    ->0 if INT32 Δ DB[L1]<-S[WORD]

   ADD
C  Generic DMU add microprogram
    ->SHORT IF INT16 Δ AR <-DB[R0]
    ->LONG IF INT32 Δ XR <-DB[R1]
  SHORT: AR <- SAR AR PLUS DB[L0]
    ->0 IF ¬OVFL Δ DB[D0] <-AR
    ->SKIP if ¬CARRY Δ DB[D1] <-D'0'
     DB[D1] <-D'-1'
  SKIP: AR <-R[TAG]
    ->0 Δ R[TAG] <- LSHIFT AR,0
  LONG: AR <-SAR AR PLUS DB[L0]
     XR <-SAR XR PLUSC DB[L1]
    ->CONV IF OVFL Δ DB[D0] <-AR
    ->0 Δ DB[D1] <-XR
  CONV: CALL 'XOVER'                 "Convert to floating point"
```

Figure 3-4: General Vector Control Microprogram and Support Functions.

"Δ" is used to separate microoperations within a microinstruction.

determine data size. The tags were read from memory within the format routine. Within LFETCH, SRW(segment read word) sets up the segment window address and starts a memory cycle. R[LINDX]

is an address pointer used by DMU control microprograms. In the next line, S[WORD] accesses the memory data word. If LTAG size bits were set to indicate 16 bit integers, then control returns to VLOOPG. Otherwise, more data will be fetched. RFETCH is similar to LFETCH. The generic operation syllable is then EXEC'd. The action subroutine, in this example, ADD, then decodes and executes the specific action, setting condition flags when needed. In the ADD microprogram, overflow is set if the result size is larger than that of the operands. This condition is tested by the branch on OVFL in VLOOPG. If there was no overflow, the result is stored in memory by a DSTORE subroutine. This is similar to the fetch routines, except that SWW(segment write word) is used. Then a loop counter, R[count], is decremented and the loop repeated if the counter is not equal to zero. This general microprogram takes 13 cycles + action time per loop for 16 bit data and 19 cycles + action time for 32 bit data. Action time refers to the number of cycles taken by the action firmware microprogram. Setup time in the format routine required 26 cycles. An even more general microprogram combining scalar and vector versions of a DLR format was written, but its performance was unacceptably slow since loop overhead was twice that of figure 3-4.

A different approach, cf. figure 3-5, uses further run time "compilation" to a specific DLR loop for each precision. Size specific data fetch is in now incorporated into the vector control loop. This requires slightly more microstore space, but

improves performance. This technique yields an execution time of 8 cycles + action time per loop for 16 bit data and 14 cycles plus action time for 32 bit data. Thus we pay a penalty of up to ~50% for generality. Inefficiency in the general algorithm stems from the necessity in MK16 to reload SR for each precision test. Freeing the status register also allows us to move the counter incrementation and gain a cycle, although this only works if the action function does not sample the zero flag.

Another possible solution with a minor change to the microarchitecture is to dedicate fields in SR for left, right, destination tags and the arithmetic flags. This technique would result in a loop execution time of 8 cycles + action time. This method requires a change to the microarchitecture to allow any bit in the SR to be tested. This method also has a problem in that SR may not be able to hold all precision tags if many data sizes are used. A separate bit is required for each precision,

```
   VLOOP16

C  DMU vector control loop for int16 diadic action
C  R[CNT] is loaded by calling program
  LOOP:
    R[LINDX]<-INC2 SRW R[LINDX]
    DB[L0]<-S[WORD] Δ "memory fetch"
    R[RINDX]<-INC2 SRW R[RINDX]
    DB[R0]<-S[WORD]
    EXEC R[OP]
    R[DINDX]<-INC2 SWW R[DINDX]
    ->ERROR IF OVFL Δ S[WORD]<-DB[D0] Δ R[CNT]<-SNZ DEC1 R[CNT]
   ->LOOP IF ¬ZERO Δ "Continue loop if zero flag is not set"
    return
   ERROR: CALL 'RECOVER'
    return
```

Figure 3-5: Size Specific Dyadic Vector control Microprogram.

since an IF microop is used to test data size. Sequential tests are needed to decode data size, but these were combined with data fetches so they did not increase loop time. Initial loading of SR is slowed since tags must be shifted and combined with SR. This increases setup time unless OVU hardware is used for this function.

It is important at this stage to weed out delays due to minor idiosyncracies in the microarchitecture since apparent speedups from other architectural changes could simply be due to masking the effects of these idiosyncracies when each processor has less to do in an algorithm. Therefore fixed precision routines are used for further study, and the problem of code compaction is left to be solved when a complete system is implemented.

### 3.1.2 Memory streaming

It should be possible to speed up sequential memory accesses. The simulator was modified to measure the advantages of such a technique. Vector instructions should execute faster since bus address cycles are no longer needed. Memory interface hardware can be designed to provide data streaming capability. Stream buffers are provided between the data bus and memory. Data bus transfers proceed at 8 or 16 bits per cycle, while transfers from memory to stream buffers use wider data paths. The memory system can use interleaving or the new nibble mode chips to support extra bandwidth. The width required depends on

the ratio of memory cycle time to processor cycle time. Once started, a memory stream no longer requires bus address cycles. Pipelining in the memory interface hides address translation time delays. A memory stream interface was designed and incorporated into SAM 1 and could have been added to SAM 0.5 if desired. The code segment in figure 3-6 shows the use of streaming in a typical vector loop for a diadic action.

```
VLOOP16

Initialize counter
Left stream address <- zero
Right stream address <- zero
Dest stream address <- zero
LOOP:
  DB[left] <- SSN left
  DB[right] <- SSN right
  EXEC action
  inc count Δ dest SDN DB[result]
  ->loop if count ≠0
```

Figure 3-6: Dyadic Vector Loop with Memory Streaming.

Two new microoperations, SSN and SDN, support read and write streaming respectively. Memory transactions take only one cycle if streams were previously initialized. Loop time is reduced to 5 cycles + action time for 16 bit data and 8 cycles + action time for 32 bit data.

Matrix multiplication should be faster using algorithm (a) using normal memory fetching. Streaming should equalize the two algorithms, unless nonsequential fetches require more than 2 cycles, in which case algorithm (b) becomes faster.

The standard interface for dyadic action routines caused some difficulty in implementing matrix multiply. The addition routine leaves the accumulated sum in DB[D] but this register will be overwritten by the next multiply action. This means that the sum must be saved elsewhere and restored within each loop. This was done, keeping the sum on the data stack, DS. This slows performance, but not significantly, since execution time is dominated by multiply time. The problem could be resolved by defining a special accumulate action that uses different registers but this would not work for a general inner product.

For a general Matrix Multiply which accepts any size data, $T_{innerloop}$ was 24 cycles plus action time for 16 bit data and 35 cycles plus action time for 32 bit data. A size specific version, cf. figure 3-7, was run resulting in an innerloop time of 13 cycles plus action time for 16 bit data and 21 cycles for 32 bit data. Although nearly twice as fast as the general microprogram, this routine still has a lot of overhead, mostly due to moving the running sum from DB to DS and back. Supporting a combined multiply accumulate action solves the problem but only for the specific matrix multiply. This technique saves 6 cycles for 16 bit data and 10 cycles for 32 bit data. Streaming reduced loop time by one cycle for 16 bit data and 3 cycles for 32 bit data.

```
DOTA16

C  DMU int16 Dot Product control microprogram computes
C   inner prod of N by K matrix (A) with K by M matrix (B)
C   DB[OP] contains left action of dot
C   DB[OP2] contains right action
C   Initial values are loaded by format microprogram
  LOOP1:
    R[COL] <- R[M]
  LOOP2:
    AR <-NEGATE R[COL]  "R[COL] counts cols down from R[M]"
    AR <-R[M] PLUS AR
    AR <-LSHIFT AR,0        "Convert to word offset"
    R[RINDX] <-R[RA] PLUS AR
    R[COUNT] <-R[K]Δ PUSH
  INNERLOOP:
    R[LINDX]<-INC2 SRW R[LINDX]
    AR <-DEC1 R[M] Δ DB[L0]<-S[WORD]
    R[RINDX]<-INC2 SRW R[RINDX]
    AR <-LSHIFT AR,0 Δ DB[R0]<-S[WORD]
    EXEC DB[OP2] Δ R[RINDX] <-R[RINDX] PLUS AR
    ->ERROR IF OVFL Δ AR <-DB[D0]
    DB[R0] <-AR
    AR <-DS[R0]
    DB[L0] <-AR
    EXEC DB[OP]
    ->ERROR2 IF OVFL Δ AR <-DB[D0]
    R[COUNT] <-SNZ DEC1 R[COUNT]
   ->INNERLOOP IF ¬ZERO Δ DS[R0] <- AR
    R[DINDX] <-INC2 SWW R[DINDX]
    AR <-R[K] Δ POP Δ S[WORD] <-DB[D0]
    AR <-LSHIFT AR,0
    R[LINDX] <-R[LINDX] MINUS AR
    R[COL] <- SNZ DEC1 R[COL]
   -> LOOP2 IF ¬ZERO
    R[LCOUNT] <-SNZ DEC1 R[LCOUNT]
   -> LOOP1 IF ¬ZERO
    return
   ERROR: CALL 'RECOVER'          "Mul error"
   ERROR2: CALL 'RECOVER'         "Add error"
    return
```

Figure 3-7: Size Specific Dot Product Control Microprogram using Algorithm (a).

In theory, algorithm 1(b) requires only two sequential memory fetches per loop, since A[i,k] is constant within the inner loop. However the standard action interface again causes

problems. A[i,k] cannot be held in DB[L] during the loop because it will be overwritten by the C value read in for the addition part. As in algorithm 1(a) the constant may be moved to a temporary location and restored or a new accumulate action can be defined. Again the data stack was used, cf. figure 3-8.

```
    DOTB16

C  DMU int16 Dot Product control microprogram computes
C  inner prod of N by K matrix (A) with K by M matrix (B)
C  DB[OP] contains left action of dot
C  DB[OP2] contains right action
C  Initial values are loaded by format microprogram
  LOOP1:
    R[COL] <- R[K]
    R[RINDX] <-R[RA] Δ PUSH
  LOOP2:
    R[LINDX]<-INC2 SRW R[LINDX]
    R[COUNT] <-R[M] Δ DS[L0] <-S[WORD]
  INNERLOOP:
    DB[L0] <-DS[L0]
    R[RINDX]<-INC2 SRW R[RINDX]
    DB[R0]<-S[WORD]
    EXEC DB[OP2]
    ->ERROR IF OVFL Δ AR <-DB[D0]
    DB[R0] <-AR
    SRW R[LINDX]
    DB[L0] <-S[WORD]
    EXEC DB[OP]
    ->ERROR IF OVFL Δ R[COUNT] <-SNZ DEC1 R[COUNT]
    R[DINDX] <-INC2 SWW R[DINDX]
   ->INNERLOOP IF ¬ZERO Δ S[WORD] <- DB[D0]
    AR <-R[M]
    AR <-LSHIFT AR,0
    R[DINDX] <-R[DINDX] MINUS AR
    R[COL] <- SNZ DEC1 R[COL]
   -> LOOP2 IF ¬ZERO
    R[LCOUNT] <-SNZ DEC1 R[LCOUNT]
   -> LOOP1 IF ¬ZERO Δ R[DINDX] <-R[DINDX] PLUS AR
    return
   ERROR1: CALL 'RECOVER'      "Multiply overflow"
   ERROR2: CALL 'RECOVER'      "Add overflow"
    return
```

Figure 3-8: Size Specific Dot Product Control Microprogram using Algorithm (b).

This size specific version of algorithm b gives an innerloop time of 12 cycles plus action time for 16 bit data. A 32 bit version takes 21 cycles plus action time. Streaming will reduce loop time by 2 cycles for 16 bit data and 4 cycles for 32 bit data, but necessitates separate read and write destination streams.

### 3.1.3 Analysis of results

During the simulation of system one, timings of the different phases of instruction execution were determined. These results, cf. figure 3-9, give an estimate of the workload of the different processes which may later be executed by separate processors. These results point out the primary bottlenecks in system performance. For vector addition, most of the time was spent in the setup and fetch phases. The dyadic scalar addition microprogram takes 3 cycles for 16 bit data and 6 cycles for 32 bit data. For multiplication, the balance swings to the action phase because of the slowness of multiplication on MK16. The multiplication microprogram takes 32 to 48 cycles for 16 bit data and ~200 cycles for 32 bit data. A VLSI functional multiplier equalizes addition and multiplication times so both cases will approximate the results from addition in systems 2 and 3.

In order to maximize system performance, processor workloads need to be balanced. All overhead is due to fixed setup time. This can be reduced with hardware support for the

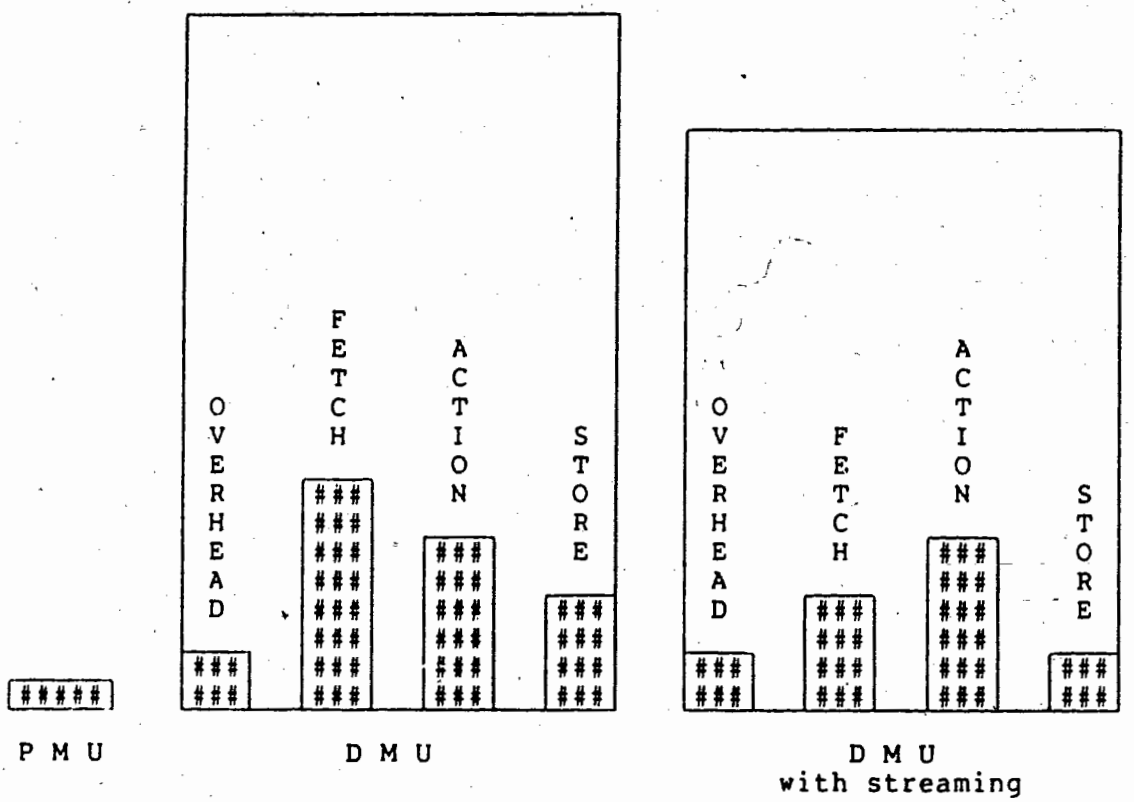


Figure 3-9:Workload Distribution of Vector Add

verification phase. A special operand verification unit (OVU) is being designed for this purpose[42]. Since memory fetch comprises the major component of DMU execution time during vector loops, memory streaming can be used to reduce fetch time. With memory streaming, fetch plus store and action phases are nearly equal for addition, so introduction of an extra processor could almost double performance if these operations are done in parallel.

While benchmark 2 execution time is dominated by multiply time, cf. figure 3-10, DMU overhead is significant even for the innerloop - 40% for the general version and 25% for the size specific microprogram. For small arrays, outer loops add even more overhead. If multiply time is reduced significantly by

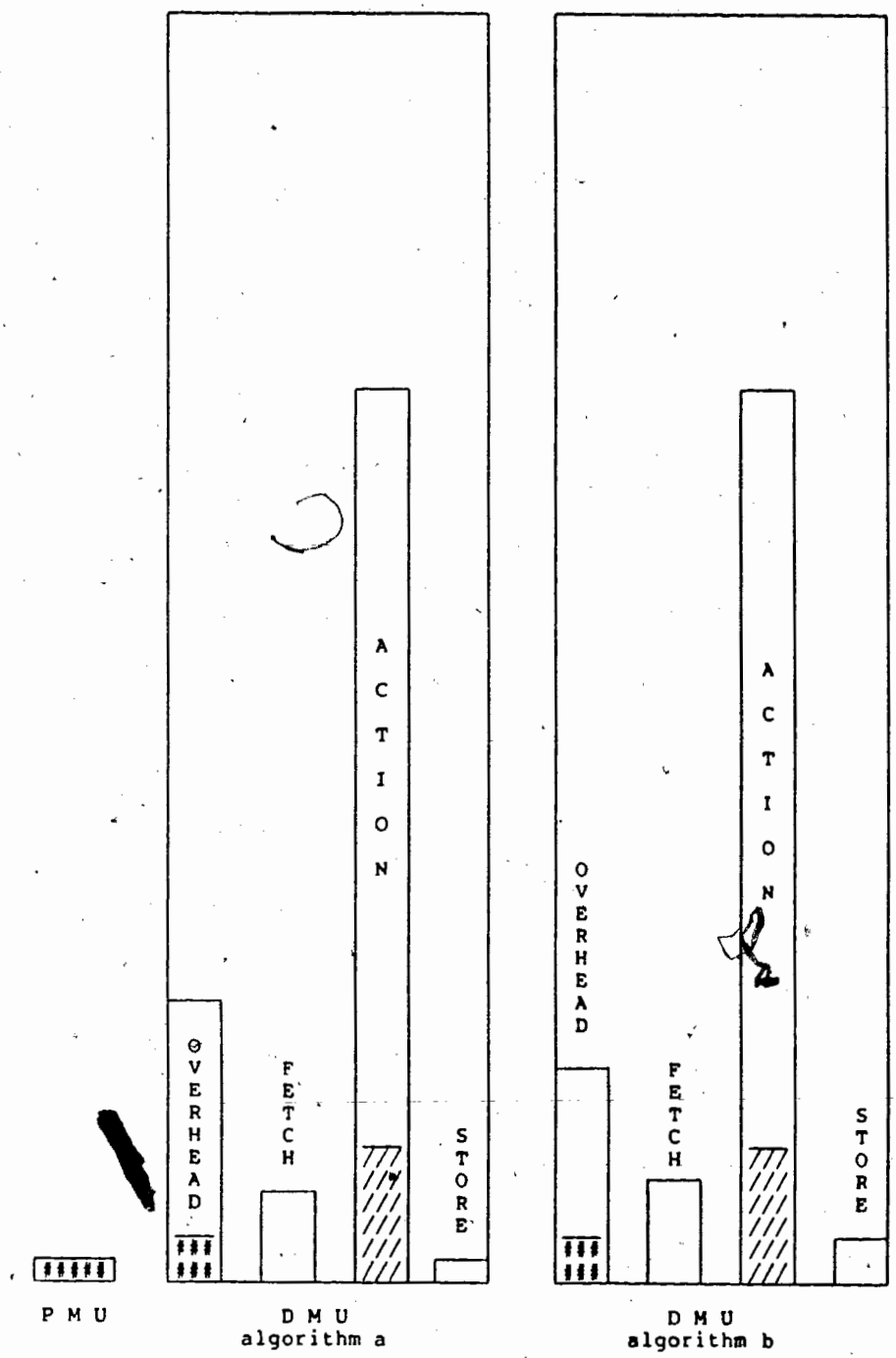


Figure 3-10:Workload Distribution of Benchmark 2

Solid area of overhead, |##|, indicates fixed setup time. Shaded area, |//|, indicates action time if multiply time = add time.

using a hardware multiplier, DMU overhead becomes a system bottleneck. An unexpected result of this study was that algorithm (b) was faster than algorithm (a) even without memory streaming. From figure 3-10, it appears that this is due to the higher overhead of algorithm (a), which requires complex address calculations in both the inner and outer loops.

The results presented in this section were parameterized into DMU loop time and action time. The results can be used to determine maximum cycle times for an arithmetic unit so that the system is not slowed.

## 3.2 System 2

In system 2, it is assumed that a combinational chip set is used to support actions. Support for vector action routines requires changes to the DMU instruction hierarchy. When a format program finds from operand tags that a vector action is required, it must determine if a specific vector action microprogram is available. In the SAM 0.5 simulation, a simple local memory table lookup is used. A more complete discussion is given in section 4.2. Two variations of the DMU-SFU interface are considered to see how they affect structure and performance. They are ordered according to increasing SFU control complexity.

In the simplest configuration, cf. figure 3-11 (a), limited cabability functional units are attached to the external data and control buses. To simplify the figure these are shown as a single bus. Each unit contains its own data buffer and status register. SFU's can have up to 16 registers selectable under microprogram control. Read or write of a selected register is enabled by the control bus. With this method only a simple finite state machine is needed to control each of the functional units. If each unit performs actions of varying precision, then a tag buffer (possibly part of IR) will be needed to control the unit. Otherwise a separate unit is needed for each action - precision pair. DMU needs to know which unit is being loaded, thus requiring a specific vector loop for each action,

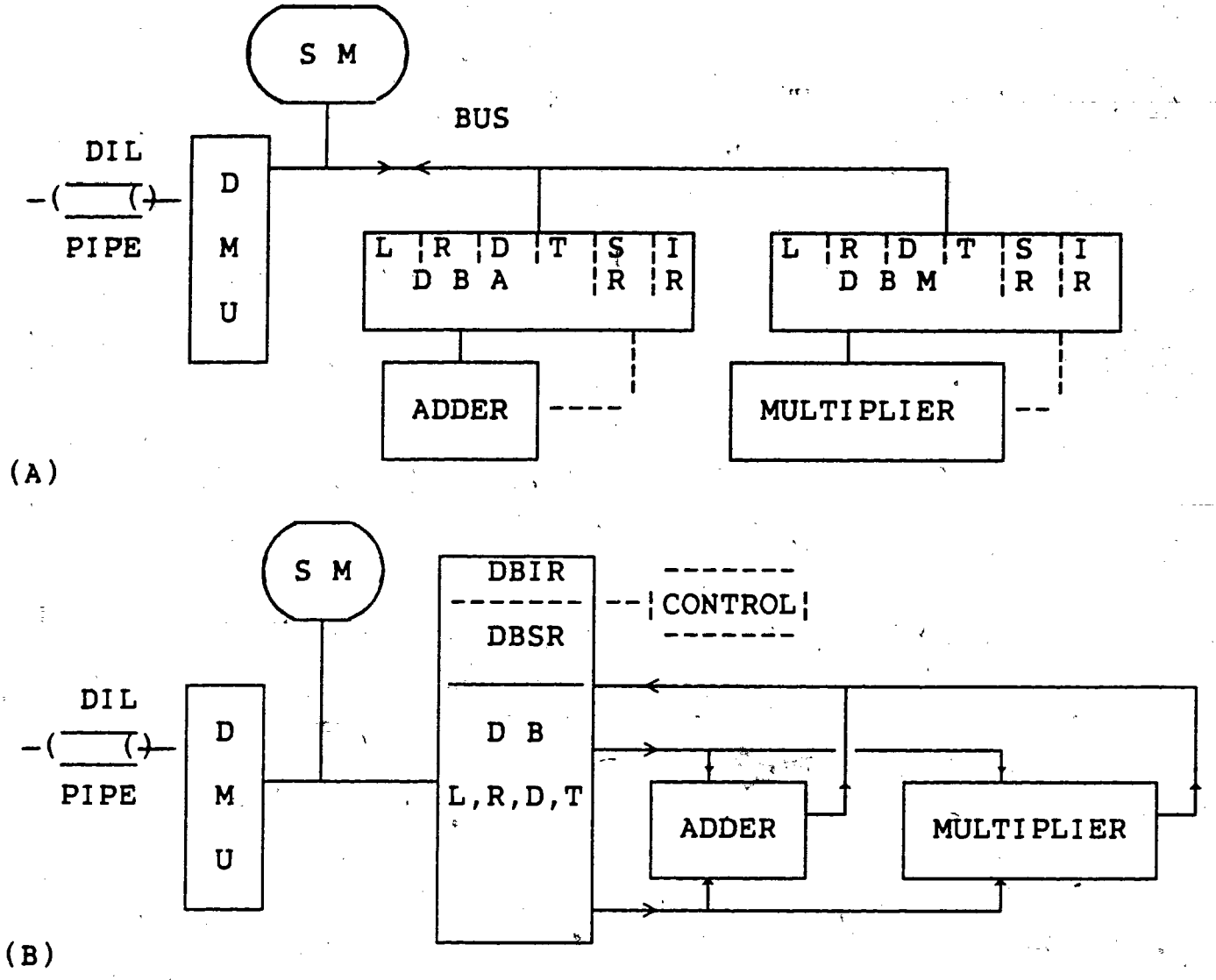


Figure 3-11: System 2 Configurations.

i.e. VADD16, VADD32, VMUL16 ...

In this configuration, DMU decodes the function code and then uses appropriate functional units to perform the required action, cf. figure 3-12. Action can be started by sending an action code to a functional unit, e.g. DBA [IR] <- 'add16'. Unfortunately, APL does not allow assignment to or subscripting of a function, so equivalent code was used in the simulation,

e.g. STARTADDER Δ DBAIR <- 'add16'.

Another method is to use a SFU control bit to indicate action

```
VMUL16

C  DMU vector control loop for int16 diadic Multiply
C  R[CNT] is loaded by calling program
  LOOP:
    R[LINDX]<-INC2 SRW R[LINDX]
    DBM[L0]<-S[WORD]                    "memory fetch"
    R[RINDX]<-INC2 SRW R[RINDX]
    DBM[R0]<-S[WORD]
    DBM[IR] <- 'mul16'
    NOP
    R[DINDX]<-INC2 SWW R[DINDX]
    SR<-DBM[SR]
    ->ERR IF OVFL Δ S[WORD]<-DBM[D0] Δ R[CNT]<-SNZ DEC1 R[CNT]
   ->LOOP IF ¬ZERO      "Continue loop if zero flag is not set"
    return
   ERR: CALL 'RECOVER'
    return
```

Figure 3-12: Size Specific Vector Multiply Microprogram.

start, e.g. DBAS [R0] <- SSN right.

This saves a bus cycle since the last data fetch can be combined with function start but may double the number of addresses needed for each functional unit if a conventional symmetric addressing convention is used. Again, equivalent code was needed in the simulation,

e.g. STARTADDER Δ DBA[R0] <- SSN right.

The interface microprograms presented so far depend on a knowledge of SFU action time. They can be made more hardware independent by replacing the NOP's with the sequence

```
WAIT: SR<-DBM [SR]
->WAIT IF BUSY
```

In the second configuration shown in figure 3-11 (b), the functional units are on a separate bus with a common data buffer through which they communicate to the DMU data bus. A control

register DBIR is loaded by DMU. This register then selects the unit to perform the next action. Only one functional unit can be active at any time unless combined actions such as an inner product step are defined and implemented in an arithmetic unit. Since all dyadic primitives of a language may not be supported in hardware, DMU must either support a vector loop for each action as in the previous case or execute the action syllable to determine if the action is supported.

### 3.2.1 System 2 performance

We now look at DMU performance assumming a fast combinational arithmetic chip set is available.

The question arises of how best to use the slave. Even without attempting to overlap operations, performance is greatly improved for some functions supported by hardware. For example, the action part of the 16 bit multiply can be reduced to 1 cycle using a combinational logic chip from an average of 16 x 2.5 cycles using DMU firmware in system 1. Without overlap using the interface structure of figure 3-11 (a), a VADD16 or VMUL16 microprogram takes 9 cycles plus action time - 1 per loop for 16 bit data and 15 cycles plus action time - 1 for 32 bit data. Memory streaming reduces these times to 6 cycles plus action time for 16 bit data and 9 cycles plus action time for 32 bit data. Performance increases are due to increased chip area that can be dedicated to specific functions.

To increase performance further, we attempt to utilize the cycles when DMU is waiting for an arithmetic unit to finish by loading the next operands, cf. figure 3-13. Performance increases are greatest when fetch-store time and function time are approximately equal. With buffered output we can also store the previous result while waiting for the action to complete. Some extra hardware is required to support overlapping. First, the inputs must be double buffered since the old inputs must be available during the action phase. Therefore another set of registers must be available to accept the new operands. This will usually be part of an arithmetic chip. Data must be moved and latched before new data arrive. Since bus data movement takes place in the first half of a clock cycle after SFU control decoding, approximately 3/4 of a cycle or ~250 nsec is available to move the data if streaming is used and 1 3/4 cycles without streaming. A simple output latch is acceptable if the output is unloaded before a new action is started. However if we wish to overlap output storage with actions a double set of output registers is needed, since new results may be latched before the old outputs are unloaded. Some arithmetic units have an output latch built into the chip. This, along with the dual port data buffer, will suffice to support full overlapping.

With full overlap, the loop time for structure (a) is reduced to 9 cycles if action < 5 for 16 bit data and 15 cycles for action <10 for 32 bit data. With streaming, the corresponding times are 6 cycles if action < 2 and 9 cycles for

```
VMUL16

C  DMU vector control loop for int16 diadic Multiply
C  R[CNT] is loaded by calling program
    DBM[L0]<-SSN left
    DBM[R0]<-SSN right
    DBM[IR] <- 'mul16'
  LOOP:
    DBM[L0]<-SSN left
    DBM[R0]<-SSN right
   WAIT: SR<-DBM[SR]
    ->WAIT IF BUSY
    DBM[IR] <- 'mul16' Δ R[CNT]<-SNZ DEC1 R[CNT]
    ->ERROR IF OVFL Δ dest SDN DBM[D0]
   ->LOOP IF ¬ZERO Δ "Continue loop if zero flag is not set"
   WAIT1: SR<-DBM[SR]
    ->WAIT1 IF BUSY
    ->ERROR IF OVFL Δ dest SDN DBM[D0]
    return
   ERROR: CALL 'RECOVER'
    return
```

Figure 3-13: Overlapped Vector Multiply Microprogram for structure (a).

action <5 for 32 bit data. If BUSY is also checked, as in figure 3-13, an extra cycle is required.

For structure (b), loop performance with individual action routines is the same as that of structure (a). If DMU EXEC's the action syllable, a general vector action microprogram can be used, in which case an extra cycle is needed. This saves microstore space and limits device specific actions to fewer action microprograms.

Matrix multiply was implemented using algorithm (a). Size specific versions were used to reduce the loop overhead found in the generic version in system 1. Performance without overlapping is similar to that on system 1, except that multiplication time is decreased. Overlapped performance will be considered next.

The use of slave action processors solved implementation problems for matrix multiply that occurred with a single processor DMU. With structure 3-11(a) there was no register contention since each unit has its own set of registers. There is still some overhead since results must be moved between units. Innerloop time was 16 cycles for 16 bit data if add and multiply time were less than 11 cycles and 24 cycles for 32 bit data if actions < 19 cycles.

Structure (b) is unable to support overlapping of multiplication with addition. Register contention is also a problem, so DS was used to hold intermediate results. Innerloop time for 16 bit data was 15 cycles if add time < 5 cycles and multiply time < 3 cycles.

Matrix multiply using algorithm (b) was also implemented using size specific microprograms. Structure (a) results in an innerloop time of 13 cycles for 16 bit data and 22 cycles for 32 bit data.

### 3.2.2 Analysis

System 2 is not much faster than system 1 for simple actions. This is due to synchronization overhead, which requires 2 or 3 cycles to initiate actions and copy SFU status to SR for testing. This is a great amount of overhead for short vector loops with simple actions and even more significant with memory streaming. A way to eliminate the overhead of testing the APU flags in (b) is to implement specific DMU control algorithms for

each action as in structure (a) and use knowledge of the APU performance to avoid testing for busy by assumming that the results are available after a fixed time. However, this links DMU algorithms to APU hardware reducing independence.

While slave processors relieved register contention problems for matrix multiply, some overhead was still incurred. The SFU addressing convention caused the overhead of data transfers in structure 10 (a) to be greater than necessary, since indirect data movement was necessary if differently numbered device registers were selected. This overhead could be reduced if inputs and outputs were addressed as separate SFUs. With structure (b), there is a register contention problem unless function control includes the ability to use any register for input or output to any functional unit. Then different registers can be used for the two functions. This would require a minimum of 4 interface registers (DB) for the benchmarks investigated in this thesis.

## 3.3 System 3

System 2 (b) can be enhanced to provide a pipelined multiply accumulate action, cf. figure 3-14. More complex control and extra data paths allow chaining of arithmetic units. DMU starts APU by loading an operation code into the APU instruction register. It can then test the APU status register to determine if the results are ready. APU should be able to perform all arithmetic primitives. Otherwise, actions need to be executed by DMU or special vector actions must be decoded during setup. For this study, only ADD, MUL, MULACC actions need be defined in pipelined and nonpipelined versions. A 2 stage pipeline is used to support overlapped vector processing.

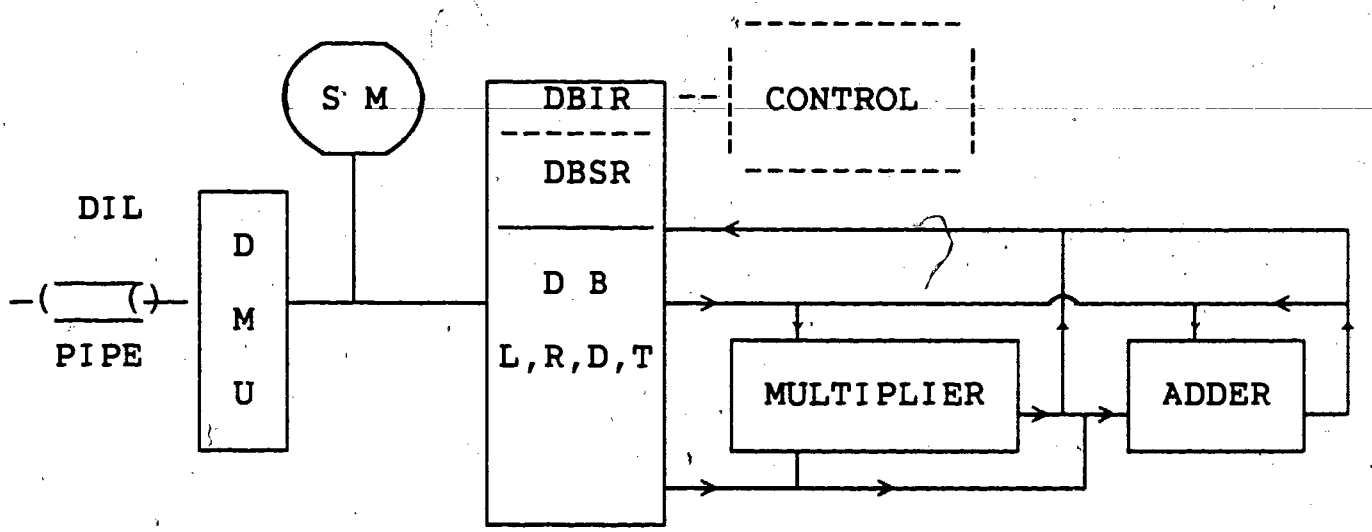


Figure 3-14: System 3 Architecture.

Performance of a general VLOOP16 microprogram, cf. figure 3-15, is 10 cycles for actions less than 2 cycles and 12 cycles for actions of 3 or 4 cycles. The destination address cycle is overlapped with APU operation. While this is adequate for short actions such as integer add (although there is little speedup over figure 3-5), there is very little overlap of DMU execution with APU action. An interesting observation of this microprogram is that performance can be optimized for short actions by adding NOP's before the statement labeled WAIT. For example, if a 16 bit addition took 2 cycles, an extra NOP before the WAIT would enable the loop to perform in 11 cycles instead of 13. Overlapping reduces loop time to 9 cycles for actions < 5. Memory streaming can reduce this to a 7 cycle loop with wait on action >4, cf. figure 3-16.

```
    VLOOP16

C  DMU vector control loop for int16 diadic Actions
C  R[CNT] is loaded by calling program
  LOOP:
    R[LINDX]<-INC2 SRW R[LINDX]
    DB[L0]<-S[WORD]
    R[RINDX]<-INC2 SRW R[RINDX]
    DB[R0]<-S[WORD]
    DB[IR] <- R[op]
    R[DINDX]<-INC2 SWW R[DINDX]
   WAIT: SR<-DB[SR]
    ->WAIT IF BUSY
    DB[IR] <- R[op] Δ R[CNT]<-SNZ DEC1 R[CNT]
    ->ERROR IF OVFL Δ S[WORD] <-DB[D0]
   ->LOOP IF ¬ZERO Δ "Continue loop if zero flag is not set"
    return
   ERROR: CALL 'RECOVER'
    return
```

Figure 3-15: General Vector action Microprogram for System 3.

Innerloop time for matrix multiply using algorithm (a) was 11 cycles for 16 bit data if add and multiply times were less than 8 cycles and 15 cycles for 32 bit data if action times are less than 12. The reduced loop time resulted from hardware support of a combined multiply-accumulate action. The running sum was held in the adder's output register to avoid the register contention encountered in system 2 (b). The extra data paths in system 3 eliminated operand movement overhead.

```
    VLOOP16

 C  DMU vector control loop for int16 diadic Actions
 C  R[CNT] is loaded by calling program
     DB[L0]<-SSN left
     DB[R0]<-SSN right
     DB[IR] <- R[op]
   LOOP:
     DB[L0]<-SSN left
     DB[R0]<-SSN right
    WAIT: SR<-DB[SR]
     ->WAIT IF BUSY
     DB[IR] <- R[op] Δ R[CNT]<-SNZ DEC1 R[CNT]
     ->ERROR IF OVFL Δ dest SDN DB[D0]
    ->LOOP IF ¬ZERO Δ "Continue loop if zero flag is not set"
    WAIT1: SR<-DB[SR]
     ->WAIT1 IF BUSY
     ->ERROR IF OVFL Δ dest SDN DB[D0]
     return
    ERROR: CALL 'RECOVER'
     return
```

Figure 3-16: Overlapped Vector action Microprogram.

## 3.4 Glossary of SAMjr Microprogramming Terms

AR is the A-register which is used as a primary accumulator in SAMjr. It may be applied to the ALU B input. It can also be shifted(rotated) left or right.

CALL is a mechanism for accessing microprogram subroutines. It must be the leftmost operation in a microstatement except for a label. The microprogram operand name must be quoted. Only internal MK16 microoperations may be used in conjunction with CALL.

CARRY is set by ALU operations which overflow or underflow if an appropriate sample request is also issued. It is also set by shift or rotate operations.

D is used to specify decimal constants.

DB is an external 16 element data buffer used to help standardize the passing of operands to microprograms.

DEC1 decrements an ALU input by one as it passes through the ALU.

DEC2 decrements an ALU input by two. It also forces the least significant bit to zero as a side effect.

DS is an external data stack for convenient and fast context storing. The top 16 values may be indexed. When DS is PUSHed or POPed, all 16 values are changed. DS is currently 1K words deep.

EXEC accomplishes microprogram address decoding. The least significant 8 bits on the data bus are used as a microprogram index.

INC1 increments its right argument by one.

INC2 increments its right argument by two and forces the least significant bit of the result to zero. This is usefull for incrementing word addresses.

INT16 is a pseudonym for carry. It is used for data size tests.

INT32 is a pseudonym for overflow.

LOCAL refers to a nonsegmented memory. It is word or byte addressable. Local memory requests require exactly 2 processor

cycles, first an address and then a data cycle.

LSHIFT operates on AR or the AR,XR pair. Shift inputs may be 0, SIGN, VSIGN, or CARRY. If SIGN is the input, a rotate is performed.

NEG is one of the SR flags. It reflects the sign of the ALU output after any sample operation.

NEGATE performs a 2's complement of the ALU-A input.

NOT performs a logical complement of any ALU input.

OVFL is one of the SR flags. It reflects arithmetic overflow after a sample operation. AR must be one of the operands.

PLUS performs diadic addition. Either AR or XR may be added to the ALU-A input.

PLUSC is a variation on PLUS that adds the CARRY flag to the sum. It is used for multiple precision arithmetic.

POP causes the DS frame to be poped at the end of the current microinstruction. A value can be accessed from DS in the same microinstruction.

PUSH selects a new DS frame providing 16 indexable locations for general purpose use.

R denotes one of the 16 scratch pad registers. A specific register is selected by subscripting, viz. R[X].

RSHIFT shifts AR or the AR, XR pair one place to the right. Shift input may be 0, SIGN, VSIGN, or CARRY.

S refers to the currently selected window into segmented memory. It is subscripted by BYTE or WORD.

SAR samples the arithmetic flags, ZERO, NEG, CARRY, and OVFL.

SDN is used to indicate a store to a segmented memory stream buffer. The left argument specifies which buffer is to be used. The right argument specifies the bus data source.

SIGN is a mnemonic that refers to the most significant bit of AR, AR[0].

SNZ samples only NEG and ZERO flags. A conditional branch on NOT CARRY may be specified in the same microinstruction.

SR is the 16 bit satus register. The least significant 8 bits are the negative, zero, overflow, and carry flags.

SRW initiates a memory fetch cycle for the currently selected segment. Its right argument specifies the segment offset address.

SSN specifies a data fetch from a segmented memory stream buffer. Its right argument specifies which stream is to be used.

SWW initiates a segmented memory store cycle. Its right argument specifies the segment offset.

VSIGN refers to SIGN EXOR (AR[0] OR AR[1]). It is used as a shifter input.

WDO is a widow register array used to store frequently referenced segments. WDO is currently limited to 8 values accessed by subscripting.

WDR is the windoe address register used to hold the currently selected segment address.

WORD functions to return an appropiate index into LOCAL or S arrays.

XR denotes the X-register which is used as a secondary accumulator. It may be applied to the ALU-B input. XR is also used in double precision shifts and rotates with AR.

ZERO is a status flag which represents the inclusive NOR of all ALU out bits after a sample operation.

# CHAPTER 4

## SAM 1.0

SAM has been revised to correct some deficiencies of MK16. In this chapter, an implementation of SAM using a new microprocessor chip is explored. In addition, the suitability of commercial arithmetic chips is examined. The new version, SAM 1.0 uses a microprocessor chip specifically designed as a control processor, SJ16 [39], cf. figure 4-1. SJ16 incorporates a 16 bit testable SR, a case statement, a separate count register, and the ability to test external status signals in one cycle. Cycle time has been reduced by limiting the data flow possible in one cycle. Currently implemented versions run at 4 to 5 MHz. Extra parallelism is present in this chip. A 3 bus datapath allows dyadic microactions in one cycle, although there are restrictions on the destination register.

The architecture of SAMjr was also changed to reflect new requirements, cf. figure 4-2. The general purpose data buffer was removed since an increased number of internal registers allows them to be used in place of DB. DB is now a specialized set of registers in OVU and is not used in this chapter. The data bus shown in this figure corresponds to the external SJBUS in figure 4-1.

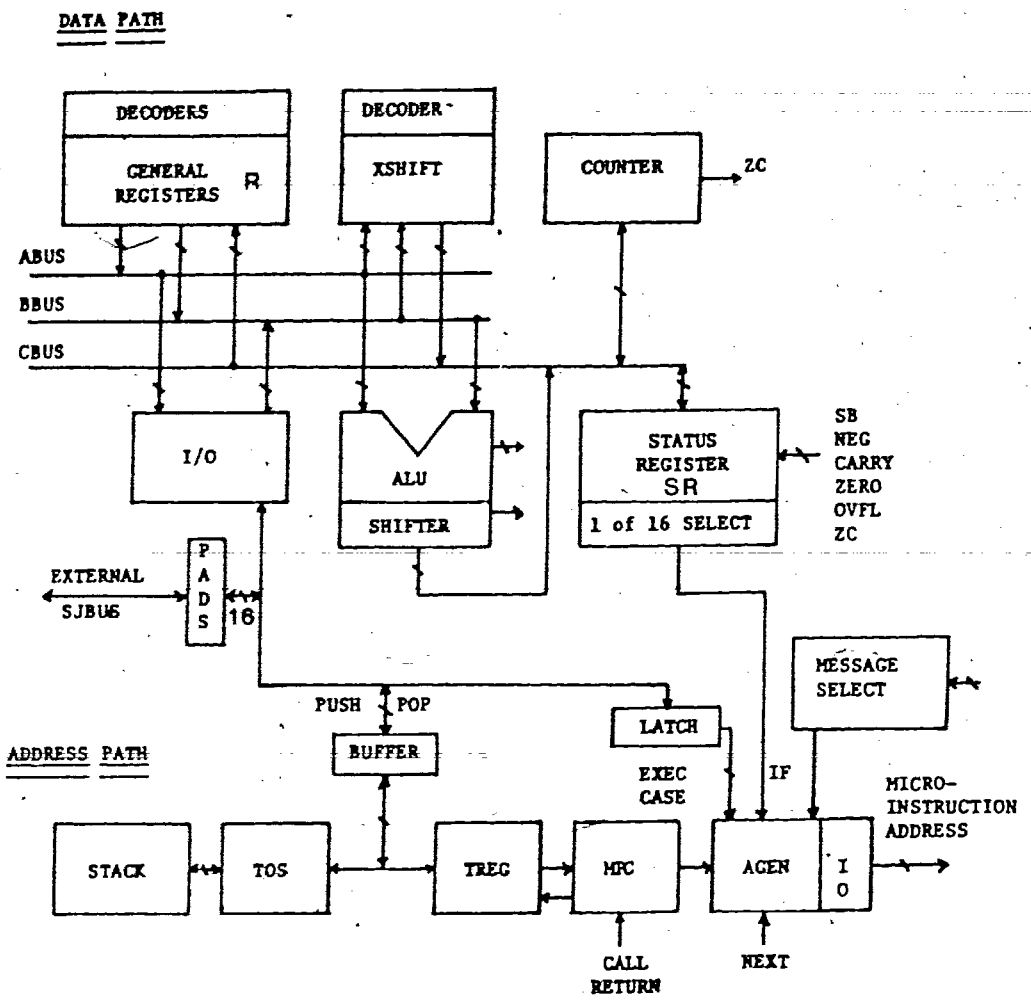


Figure 4-1: SJ16 Microarchitecture.

Memory streaming has been implemented in this system. A stream is set up by loading a segment number to select a segment and starting a memory fetch cycle with a SRB or SRW microoperation. After a one cycle delay, data can be fetched with SSN, which also initiates a new fetch cycle if the last buffer data word is read. Address translation hardware is pipelined to provide one cycle response. A write stream is similar. Data is simply written into the buffer. Writethrough is performed whenever the buffer becomes full.

A 48 bit microinstruction controls the datapath. Sixteen bits are now available to control SFU's via the control bus, 8 each for source and destination units. It is now possible to specify different registers in source and destination. Up to 3 or 4 microoperations can be specified in one microinstruction.

As in SAM 0.5, microoperations are described by a set of mnemonics. Some new terms are CASE, which performs a 16 way branch on the 4 least significant bits of SJBUS, COUNT, which tests and increments the 16 bit counter, SD (segment destination), which writes bus data into a WDO data buffer, SS (segment source), which reads a WDO data buffer onto SJBUS, and SWRITE to flush a write data buffer into memory.

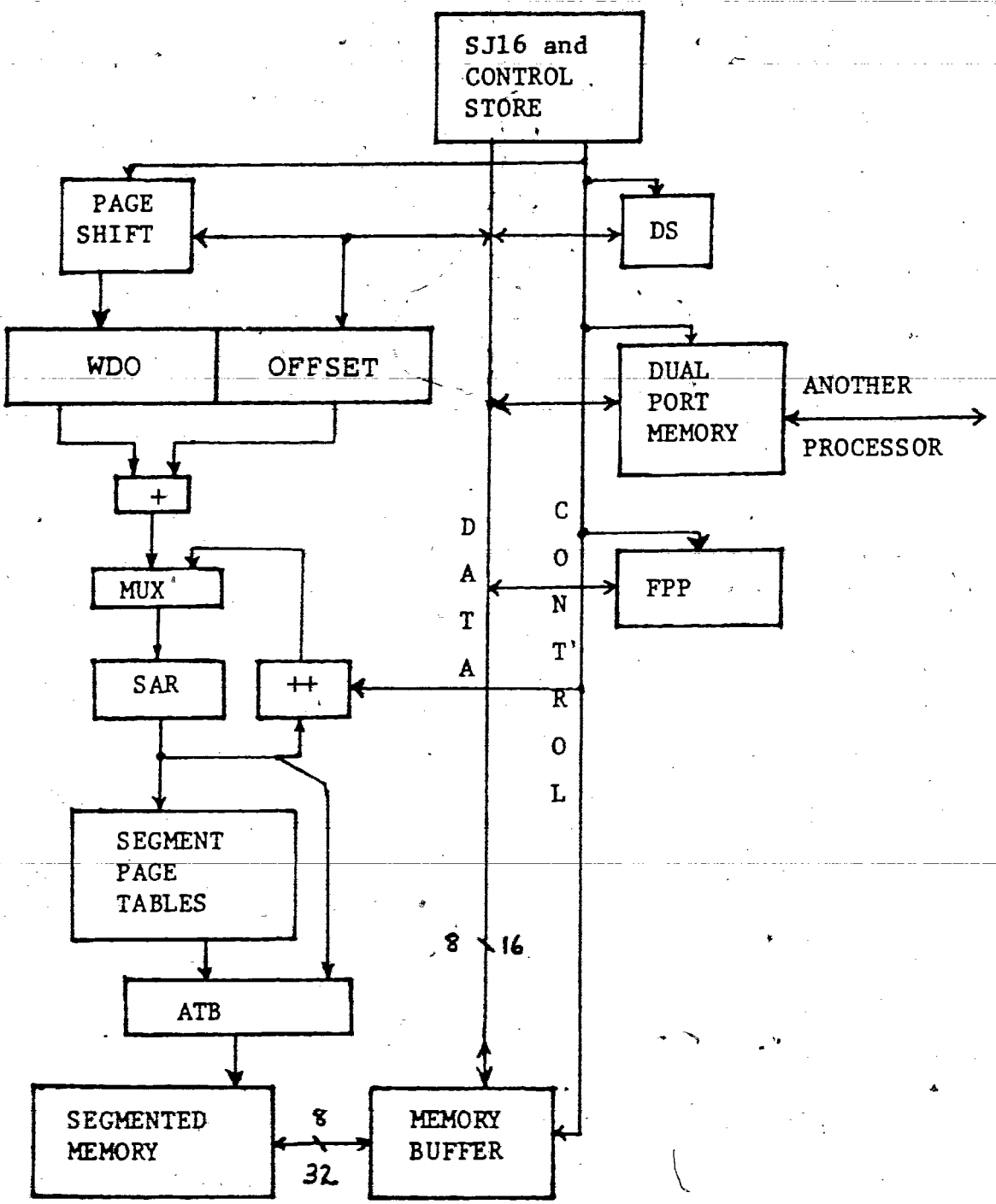


Figure 4-2: New SAMjr Architecture.

## 4.1 System 1

The microprogram fragment in figure 4-3 shows the loop section of a size specific vector DLR microprogram for a one processor SJ16 system. For comparison with SAM 0.5, memory streaming is not used. The SS microoperation initiates a memory data transfer to SJBUS without starting a new fetch cycle. Loop performance is 10 cycles plus action time for 16 bit data, while a 32 bit version takes 19 cycles + action time. The increased number of cycles is required because memory accesses take 3 cycles instead of 2 as in MK16. This can be transparent to the microprogram. If a memory fetch is not complete when the data is requested with a SS or SSN microoperation, a wait cycle is inserted. With memory streaming, only 5 cycles plus action time are required for 16 bit data and 7 cycles + action time for 32 bit data. Action time for 16 bit add is two cycles because the

```
      VLOOP16

C DMU control loop for 16 bit diadic actions
C Counter is loaded by calling function
   LOOP:
      LEFT SRW R[LINDX]
      R[LINDX]<-R[LINDX] PLUS D '2'
      R[L0]<-SS left
      RIGHT SRW R[RINDX]
      R[RINDX]<-R[RINDX] PLUS D '2'
      R[R0]<-SS right
      OPS EXEC R[ACT]
      DEST SWW R[DINDX]
      ->ERROR IF OVFL Δ R[DINDX]<-R[DINDX] PLUS D '2'
     ->LOOP IF ¬COUNT Δ DEST SD R[D0]
      ->0 Δ SWRITE DEST
ERROR: CALL 'RECOVER'
```

Figure 4-3: SJ16 Vector Control Loop for 16 bit Data.

destination register is not the same as a source register. Multiply takes approximately 16 x 2 cycles using a new mulop instruction unless a fast multiply trading space for speed is used. This can be implemented using CASE to test 4 bits at once. Memory transactions severly limit performance without memory streaming; therefore further discusion will assume its presence.

Benchmark 2 using algorithm 1(a) is set up as an outer procedure to control the outer 2 loops and a call to a vector inner product routine to handle the inner loop, cf. figure 4-4. This function is called directly if both operands are vectors. This structure is not costly since a call can be done in parallel with other microoperations and also because this inner loop is primarily responsible for overall performance. As in SAM 0.5, the standard interface caused problems. Again DS was used for temporary storage, but introduced significant overhead. The hardware counter was not used for loop counting since it is used in the multiply microprogram. A size specific vector dot product results in a loop time of 10 cycles plus action time for 16 bit data and 15 cycles plus action time for 32 bit data.

Extra support for matrix multiply can be implemented with a combined multiply accumulate action to reduce overhead. Extra decoding is also needed in the format routine to verify that the op syllables are "+" and "x". This results in a loop time of 6 cycles plus action time for 16 bit data, cf. figure 4-5, and 8 cycles plus action time for 32 bit.

```
      DOTA16

C DMU control loop for 16 bit Matrix Dot Product
C R[OP] contains first action of DOT
C R[OP2] contains second action
DEST SWW D'0'                    "Start dest stream at 0 offset"
LOOP1:
  R[RCOUNT] <-RSHIFT 0,R[M]
  LOOP2:
    LEFT SRW R[LINDX]   "Start left stream at current col of A"
    R[COUNT1] <-NOT R[K]
    R[T] <-R[M] MINUS R[T]
    R[RP]<-LSHIFT R[T],0   "R[RP] points to current col of B"
    CALL 'VDOTA16' Δ R[T] <-R[M]
    ->ERROR IF OVFL Δ dest SDN R[D0]
    R[RCOUNT] <-SF R[RCOUNT] PLUS D'-1'
   ->LOOP2 IF ¬ZERO
   R[LCOUNT] <-SF R[LCOUNT] PLUS D'-1'
  ->LOOP1 IF ¬ZERO Δ R[T] <-LSHIFT R[K],0
 return
 ERROR: CALL'RECOVER'                    "Size overflow"


      VDOTA16
C  DMU control for general 16 bit vector dot product
C  left is a simple row vector
C  right is a col vector with step specified by R[T]
C  Starting address of right in R[RP]
   LOOP:  RIGHT SRW R[RP]
      R[L0]<-COPY SSN left
      R[R0]<-COPY SSN right
      OPS EXEC R[OP2]
     -> ERROR2 IF OVFL Δ R[L0] <-COPY DS[RQ]
      R[R0] <-NOP R[D0]
      OPS EXEC R[OP]
      DS[R0] <- COPY R[D0]          "Save partial result"
     ->ERROR1 IF OVFL Δ R[COUNT1] <-SF R[COUNT1] PLUS D'-1'
    -> LOOP IF ZERO Δ R[RP] <-SF R[RP] PLUS R[T]
    return
   ERROR1: recover from action1 overflow
   ERROR2: recover from action2 overflow
```

Figure 4-4: Dot Product Microprogram.

```
      MATMUL16

C DMU control loop for special 16 bit Matrix multiply
C R OP contains combined action code
DEST SWW D'0'       "Start dest stream at 0 offset"
LOOP1:
  R[RCOUNT] <-RSHIFT 0,R[M]
  LOOP2:
   LEFT SRW R[LINDX]   "Start left stream at current col of A"
   R[COUNT1] <-NOT R[K]
   R[T] <-R[M] MINUS R[T]
   R[RP]<-LSHIFT R[T],0   "R[RP] points to current col of B"
   CALL 'VINPRD16' Δ R[T] <-R[M]
   ->ERROR IF OVFL Δ dest SDN R[D0]
   R[RCOUNT] <-SF R[RCOUNT] PLUS D'-1'
  ->LOOP2 IF ¬ZERO
  R[LCOUNT] <-SF R[LCOUNT] PLUS D'-1'
 ->LOOP1 IF ¬ZERO Δ R[T] <-LSHIFT R[K],0
return
ERROR: CALL'RECOVER'                    "Size overflow"

      VINPRD16
 C  Special DMU control for 16 bit innerproduct
 C  left is a simple row vector
 C  right is a col vector with step specified by R T
 C  Starting address of right in R RP
 LOOP:  RIGHT SRW R[RP]
    R[L0]<-COPY SSN left
    R[R0]<-COPY SSN right
    OPS EXEC R[OP]
   -> ERROR IF OVFL Δ R[COUNT1] <- R[COUNT1] PLUS D'1'
 -> LOOP IF ¬OVFL Δ R[RP] <- R[RP] PLUS R[T]
  return
  ERROR: CALL 'Recover'
```

Figure 4-5: Special Matrix Multiply Microprogram.

Algorithm b also has problems with register contention, so DS was used to hold the current A[i,k] value which is constant for the inner loop. Innerloop time is 8 cycles plus action time for 16 bit data and 13 cycles plus action for 32 bit.

## 4.2 System 2

We now look at system 2 using SJ16. A new firmware structure is required to support special function units if they do not perform all dyadic scalar actions. This was also true for SAM 0.5 but detailed discussion was deferred to this chapter since SJ16 has better decoding capabilities. When DMU performed all actions, the simple decode hierarchy as used in SAM 0.5,system 1 was adequate. Even in this case supporting specific vector actions would have improved performance. With the addition of specialized slaves, a great amount of complexity was introduced. First, in supporting vector actions, the fetch routine has to know the source of its data. Since either operand can be scalar or array type, four combinations need to be supported. This can be alleviated by changing the architecture so that scalars and vectors are accessed identically by the firmware. Second, the data transfer destination now depends on the specific action, so the fetch routine must know the action code. In SAM 0.5, an external data buffer was used as a standard interface between fetch and execute modules. This was feasible since an extended cycle time allowed external data manipulation without performance penalty. With SJ16, internal data manipulation is faster so performance is better if data is streamed directly to SJ16 registers. Also, since hardware memory streaming allows vector operand fetching in one cycle, storing in intermediate buffers causes performance degradation. If an attached processor performs all data manipulations in a

category(ie. all scalar dyadic primitives), then interfacing is simplified. However the early SAM systems will probably only contain more primitive specialized processors. To cope with this lack of symmetry, the firmware structure was changed as shown in figure 4-6. A discussion of possible decoding mechanisms is given in section 4.6. This structure necessitates a large number of DMU action subroutines. This is necessary since hardware support for scalars may be different than for vectors. For example, if a Weitek 1064/65 chip set [93] is used as a slave floating point processor, flow through mode would be used for scalar processing, while pipeline mode could be used for vectors. This structure also allows specific fetch-action routines to speed up statistically frequent actions on SJ16. Vector add loop time can be reduced by sacrificing structure and

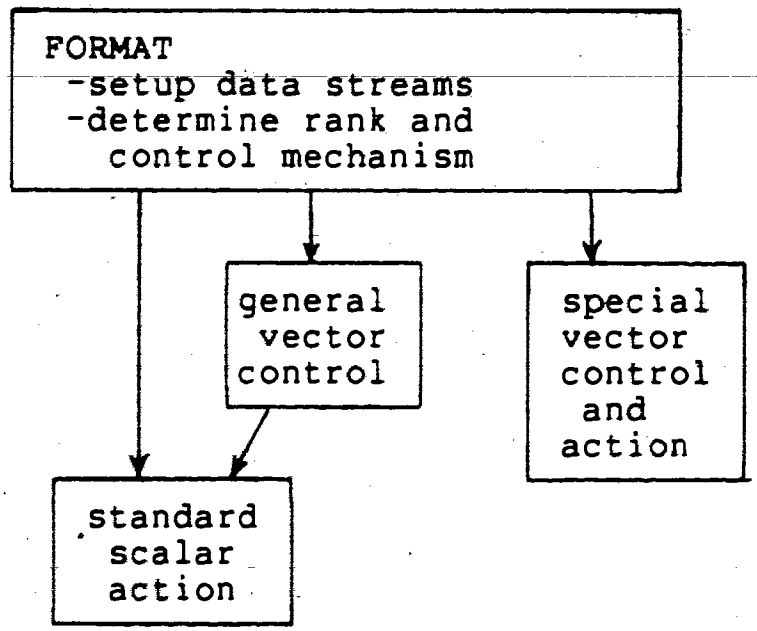


Figure 4-6:Decode Firmware Structure.

symmetry, and integrating addition with operand fetch, cf. figure 4-7. The microprogram is small and simple and thus does not require rigid structuring for easy understanding. The loop time is 3 cycles for 16 bit integers. A similar microprogram gives a loop time of 6 cycles for 32 bit integers. This is the limit for bus data movement so only increases in bus bandwidth can improve performance. The cost is the requirement for executable routines for vector versions of each dyadic action for each precision. This could have been done in system 1, but a one processor system may not have the microcode space resources to allow this.

```
      VVDADD16

   C DMU control loop for special vector 16 bit add
   C Counter is loaded by calling function
      LOOP:
         R[D0]<-SSN left
          ->DONE IF COUNT Δ R[D0]<-R[D0] PLUS SSN right
         -> LOOP IF ¬OVFL Δ dest SDN R[D0]
         CALL 'Recover'
      DONE: dest SDN R[D0]
         return
```

Figure 4-7: Special Vector Integer Add Microprogram.

### 4.2.1 SFU interfacing

In SAM 0.5, SFU interfacing sometimes resulted in 2 to 3 cycle overhead. In SAM 1, memory streaming results in short vector loops, so overhead must be reduced. A one bit message bus is used to determine SFU status. Only one condition can be tested with this bus, but SFU status contains at least 2 conditions, overflow and busy. Testing for busy can be eliminated by delaying the system clock if results are not ready when requested. This is already done for memory delays. All other SFU conditions can be combined into a single message that can usually be tested in combination with another microop.

A four bit test bus was also considered for SFU status determination but is unnecessary if the above techniques are used. A new TCASE microop would be necessary and microprogram length would be greatly increased. There is also a problem in determining the source for the test bus.

We now consider each configuration in detail. With a slave attached as per structure (a) a straight forward implementation of VADD16 or VMUL16 without overlap takes 6 cycles plus action time while VADD32 or VMUL32 takes 9 cycles plus action time. The extra cycle over the one processor case is needed to bring in the external status. The arrangement of microoperations allows no parallelism since the conditional branches are adjacent. Moving the loop check reduces these times by a cycle.

With simple overlap, loop performance is 5 cycles if action ≤ 2 for 16 bit operands and 7 cycles if action ≤ 4 for 32 bit operands. The greater resources of SJ16 allow us to attempt further increases in loop performance. The larger SJBUS control space permits multiple addresses for functional units that allow loading the last operand and starting an action in one cycle. Use of these techniques and a rearrangement of microinstructions reduces the loop time of the unoverlapped case to 3 cycles plus action time giving a FRI of 1/4 for a typical one cycle add. $T_{loop}$ for 32 bit data is 6 cycles plus action time.

Loop cycle time for the overlapped case is reduced to 3 cycles if action ≤ 2 for 16 bit data and 6 cycles if action ≤ 4 for 32 bit data. FRI ranges from 1/3 to 2/3 depending on SFU action time. With a simple output stage the start signal must be buffered until the output is unloaded. With double buffering, full overlap can be achieved. $T_{loop}$ is 3 cycles if action ≤ 3 for 16 bit data and 6 cycles if action ≤ 6 for 32 bit data giving FRI=1/3 to 1. The performance is the same for both the simple output and the double buffered output cases for short actions.

Size specific versions of matrix multiply algorithm (a) were implemented. For structure (a), innerloop time was 7 cycles for 16 bit data if action time ≤ 5 and 11 cycles for 32 bit data if action ≤ 8. FRI ranges from .2 to 1.4 depending on the speed of SFUs.

Algorithm (b) has been greatly simplified through the use of additional arithmetic units. With structure (a), the constant term A{I,K} can be kept in the multipliers input buffer. Also, the counter is now available for loop counting. Innerloop time with simple output buffering for 16 bit data is 4 cycles if action time ≤ 1 cycles and 6 cycles for 32 bit data if action time ≤ 2 cycles. With double output buffering, complete overlap is possible.

### 4.3 Potential Arithmetic Processors

Now that some performance limits have been found for SAM, the suitability of some commercial processors can be examined with respect to required arithmetic unit performance.

Texas instruments markets the TMS 32010 with a Harvard architecture and a high performance data path[86]. It offers a two cycle 16 bit integer multiply or pipelined multiply accumulate. Floating point operations can be provided by software. Eightysix cycles are required for 32 bit floating point addition and 43 cycles for 32 bit multiplication. Unfortunately, the instruction set and chip I-O interface limit performance. Input and output require 2 cycles because of pin sharing with instruction fetch. Testing for overflow requires 2 cycles, further limiting performance.
Decoding of the action is quite slow using normal call on contents of accumulator. Seven cycles are required to call and

return from a subroutine specified by external data. This can be reduced with an external IR used to select the current TMS action program. This requires 4 cycles, still much slower than SAM's one cycle EXEC, and unacceptable for fast scalar actions. This chip may be more useful as an independent APU. It can be programmed to execute vector operations, thus minimizing the action decode overhead. TI has introduced a new product, TMS 32020 [87], which reduces some of these problems. It should be especially useful for matrix multiply algorithm (b), since it can have a large data memory.

Hewlett Packard has designed a CMOS chip set capable of pipelined 3 cycle 32 bit floating point operations and 6 cycle 64 bit operations [37]. It also allows integer operations. It is used in the HP A700 computer but unfortunately not offered for sale separately. Using this hardware, a 4 MHz SAM system would yield full performance with equal bus and action times.

Weitek markets very high performance chips for floating point operations, WTL 1064/65 and 1164/65 [93,94]. They have a pipeline mode that allows vector proceessing at up to 5 or 10 MFlops, well above current SJBUS capability. They have a flow through mode that is well suited to DMU fetch capability.

Other coprocessor chips are available with reduced performance, i.e. Intel 8087, National 16081, Motorola 68881. These allow much better performance than SJ16 microprograms for floating point arithmetic, but are often designed to interface with specific processors. For example, Intel's 8087 duplicates

some of the master functions. It is wired in parallel with the master, duplicating bus interface and decode circuitry. Both processors detect an 8087 instruction. The master then allows the 8087 to control the bus and fetch operands. Although the 8087 has functionally separate bus and arithmetic control, fetch and actions do not proceed concurrently. since bus control is busy following the progress of the master. This chip offers floating point add time of 14 microseconds or 70 cycles at 5 MHz and multiply times of 19 microseconds for single precision and 27 microseconds for double. This is much faster than SAM microcode but the interface causes ineffecient usage of the chips processing power. Fried [32] reports only 15% efficiency using this chip in an IBM PC system for a simple floating point add. The rest of the time is spent loading operands, storing results, and synchronizing the processors. Of these, the National 16081 floating point unit is the fastest and most flexible.

### 4.3.1 Multiple APU Algorithms

Multiple arithmetic chips can be used to increase performance and relieve deficiencies in chip design. Although some commercial arithmetic chips are available that are fast and permit pipelining, many others are slow or unable to receive new operands during an action. For chips with performance equal to data fetch time but which do not permit loading new operands during an action, alternation between two such chips allows

overlapped operation of fetch with action, cf. figure 4 - 8. The vector add control microprogram gives a result every 3 cycles if the chip can perform the action within 3 cycles.

```
    VADD16

    DBA1[L0]<-SSN left
    ->done IF count Δ startadder1 Δ DBA1[R0]<-SSN right
Loop: DBA2[L0]<-SSN left
    ->done2 IF count Δ startadder2 Δ DBA2[R0]<-SSN right
    ->error1 if adder1error Δ dest SDN DBA1[D0]
    DBA1[L0]<-SSN left
    ->done IF count Δ startadder1 Δ DBA1[R0]<-SSN right
    ->Loop if ¬adder2error Δ dest SDN DBA2[D0].
    check adder2 status and recover from error
error1: check adder1 status and recover from error
Done: store remaining results and exit
Done2: store results and exit
```

Figure 4-8: Multiple fast chip control.

With slower chips, multiple units can be used to keep up to bus transfer rate, cf. figure 4-9. This microprogram for vector addition using three slow adders gives a 2 fold speedup. If add time is 6 cycles, then 2 other adders can be serviced while waiting for results. In general, we can fill in the time waiting for an action to complete with servicing of other units. With enough chips, action throughput can be made equal to bus transfer rate. For a given action delay, $T_{del}$, 1+CEILING{ $T_{del}$ / (3 x data size)} units are needed.

```
      VADD16

      DBA1[L0]<-SSN left
      ->done IF count Δ startadder1 Δ DBA1[R0]<-SSN right
       DBA2[L0]<-SSN left
      ->done2 IF count Δ startadder2 Δ DBA2[R0]<-SSN right
       DBA3[L0]<-SSN left
      ->done3 IF count Δ startadder3 Δ DBA3[R0]<-SSN right
Loop: ->error1 if adder1error Δ dest SDN DBA1[D0]
      DBA1[L0]<-SSN left
      ->done IF count Δ startadder1 Δ DBA1[R0]<-SSN right
      ->error2 if adder2error Δ dest SDN DBA2[D0]
       DBA2[L0]<-SSN left
      ->done2 IF count Δ startadder2 Δ DBA2[R0]<-SSN right
      ->Loop if ¬adder3error Δ dest SDN DBA3[D0]
      check adder3 status and recover from error
error1: check adder1 status and recover from error
error2: check adder2 status and recover from error
Done: store remaining results and exit
Done2: store results and exit
Done3: store results and exit
```

Figure 4-9: Multiple slow chip control.

## 4.4 System 3

System 3 is illustrated using a Weitek chip set, cf. figure 4-10. The chip set is connected to a two port interface register buffer (ABUF). The chips are connected in parallel on 2 input buses and an output bus. APU control enables inputs/outputs in the chips. Bus control is not shown. It is connected to the control bus and responds when APU is selected as a source or destination. It enables input/output from ABUF registers and produces the start signal. Action codes are loaded into APUIR and operands are loaded into ABUF[L,R]. A start signal, generated by the bus control interface, initiates actions. APU control then completes the action without further DMU

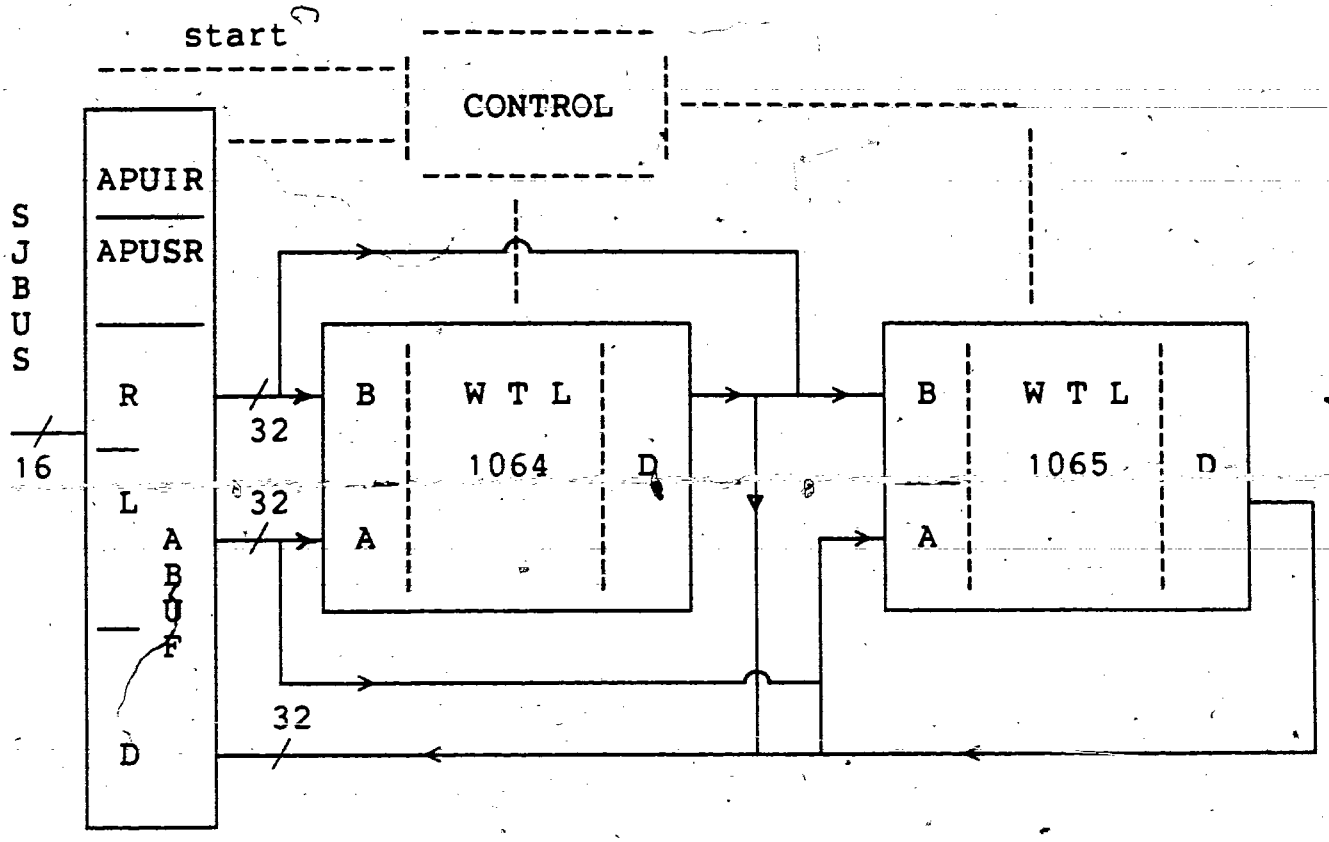


Figure 4-10:APU Weitek Chip Set Control.

intervention.

For the benchmarks in this study, APU needs only a limited instruction set. ADD, MUL, MULACC, ADDOUT, MULOUT, CONVERT instructions are defined for each size.

Timing for DMU and APU is shown in figure 4-11 for vector inner product. APU runs at twice the clock rate of SAMjr. DMU loads the last operand and initiates APU start. APU then decodes the action transfers the operands and starts the required action. In this example, operation time can last up to 4 1/2 cycles or 900 nanoseconds, much longer than the 360 nanoseconds required for a WTL1064 to complete a 32 bit multiply. "Latch"

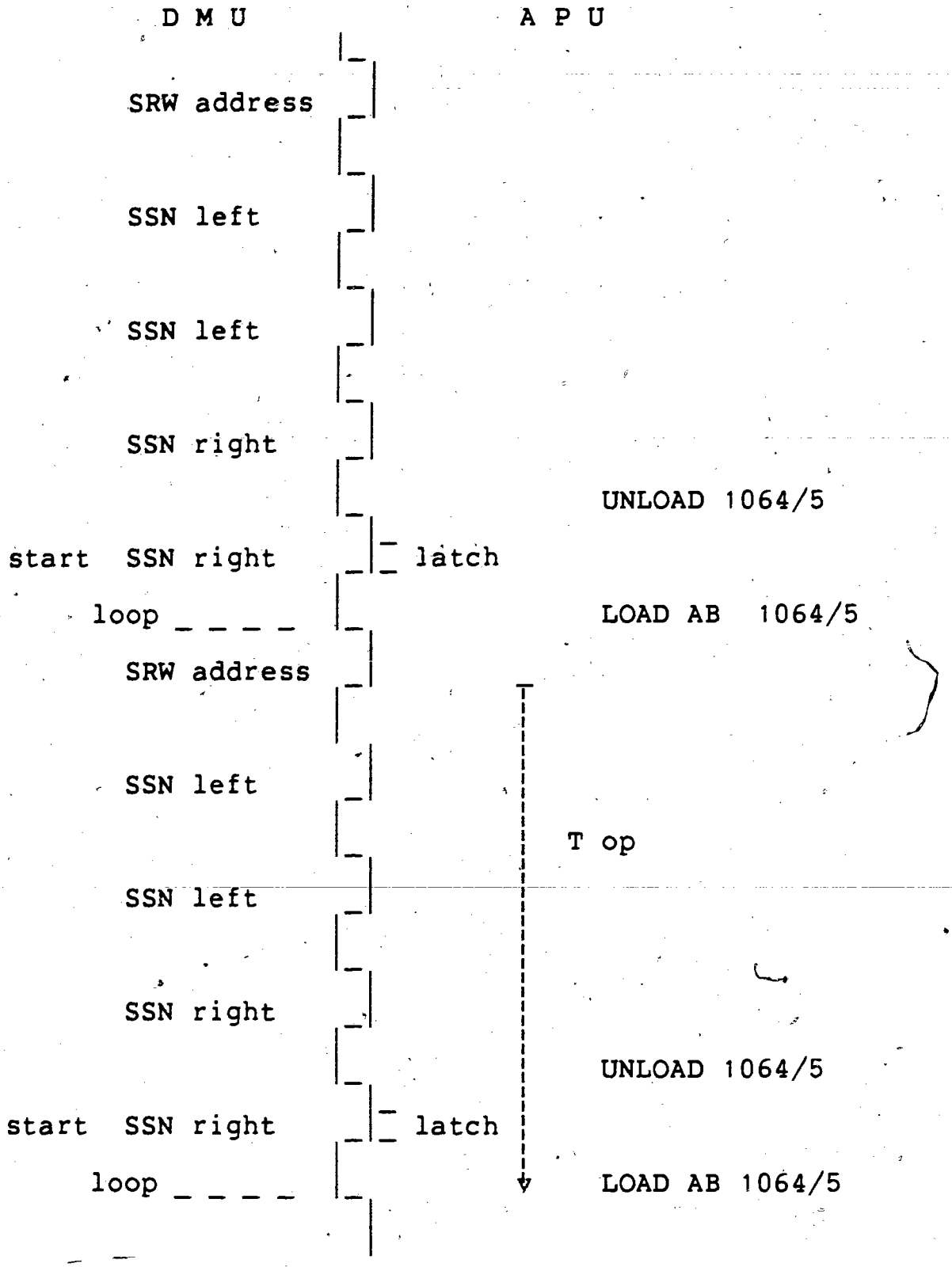


Figure 4-11: DMU and APU timing diagram for vector inner product

shows where the bus input is latched into ABUF. The load and

unload commands contol internal gating in the WTL 1064/5 chips. The result is accumulated in the WTL1065 output register and stored when control returns to the calling routine.

With the interface described above, innerloop time is 5 cycles for single precision floating point data. A similar integer unit gives a innerloop time of 3 cycles if actions complete in 3 cycles for 16 bit data. Innerloop time for 32 bit data was 5 cycles if action ≤ 5. Benchmark 1 times will be the same as system 2.

A simple extension to system 3 enables hiding of APU hardware complexity. Replacement of the input-output data buffer with simple fifos allows the case instruction to be used to decouple DMU and APU algorithms, cf. figure 4-12. DMU vector action code is simplified. Control complexity has not been

```
VLOOP16

    ->'COND' CASE APUSR
COND0000: 'Load new operands and store result"
     APUIN<-SSN left
     APUIN<-SSN right
    ->DONE IF COUNT Δ dest SDN APUOUT
    ->'COND' CASE APUSR
COND0001: "Load new operands"
     APUIN<-SSN left
     APUIN<-SSN right Δ inc R[diffcnt]
    ->DONE IF COUNT
    ->'COND' CASE APUSR
COND0010: "store result"
    ->DONE IF COUNT Δ dest SDN APUOUT Δ dec R[diffcnt]
    ->'COND' CASE APUSR
COND0011: "wait one cycle"
    ->'COND' CASE APUSR
COND01xx: "Determine error and recover"
    DONE:store diffcnt results and exit
```

Figure 4-12: DMU control code for 16 bit vector dyadic actions using case

eleminated, but transferring it to APU creates a better functional system distribution and improves module independence. DMU delivers all data to the input-q and stores results from the output-q when available. The case instruction examines APU status to determine the appropriate process. Only input full, output empty, and error(overflow) need to be tested. APU architecture is now transparent to DMU code. If allowed by the hardware, pipelined operation proceeds without using normal hardware specific software pipelining. Because queue length and data latency are unknown, DMU must keep track of the number of data sets in process. This is accomplished by keeping track of the difference between fetches and stores in R[DIFFCNT]. Upon loop completion this register contains the number of results that still need to be stored. Loop time for a 16 bit vector loop is 4 cycles if action time is $\leq 4$. This could be reduced to 3 cycles if a test bus and a TCASE microop were implemented. For 32 bit data, vector loop time is 7 cycles if action time $\leq 7$.

CASE can also be used for vector innerproduct control, resulting in a 4 cycle loop for 16 bit data and 7 cycles for 32 bit data. However, control can be simplified since only a single process needs to be handled. If FIFO full and APU errors are combined into one message, the fetch loop can be controlled with an IF, reducing loop time by a cycle. This assumes that APU is executing a combined multiply accumulate action and holding results until the end of the loop.

Further performance gains are possible if APU contains internal data memory. If the memory is logically designed as FIFO, and results are automatically stored there, store cycles can be saved in vector control loops for those formats that leave results on the stack,

i.e. SLR, SLS, SSR

, and fetch cycles will be saved in formats that reuse results,

i.e. DLS, DSR, SLS, SSR, RSR, LLS, SSS.

SLS is the most frequently used ADEL format, so this technique should improve system performance. DMU code for matrix multiply algorithm (b) is simplified since only one process needs to be controlled at a time. A simple IF test for FIFO FULL can be used in place of CASE for the memory fetch loop, and overflow detection can be deferred until the end of the vector action. Segmented memory allocation for results can also be deferred until after this check. This saves an extra allocation when overflow does occur. A one cycle innerloop for matrix multiply algorithm (b) can be devised if APU uses a pipelined multiplier and adder. The data memory must be large enough to hold a complete row vector of 64 bit precision, or if a smaller memory is used, a method of automatic overflow into segmented memory is needed.

With this independence, APU architecture can be modified transparently to DMU algorithms, i.e. queue length or number of functional units can be changed or multiple SFU's can be added to match bus performance. Figure 4-13 shows how this arrangement

extends the system hierarchy.

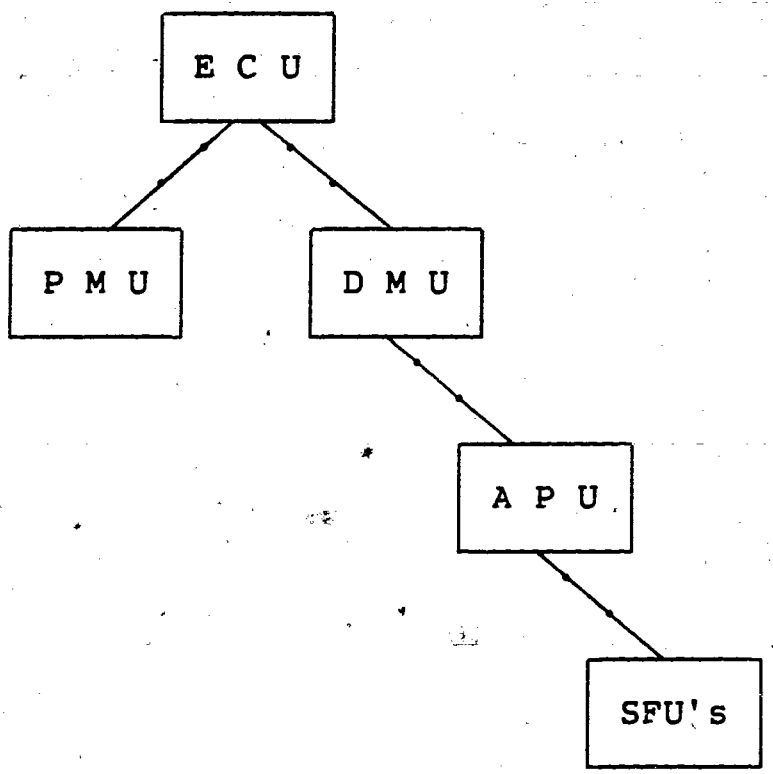


Figure 4-13: SAM system hierarchy.

## 4.5 Summary of Performance

Table 4-1 summarizes SAM 1 performance. System 2 with suitable slave processors gives optimum performance for benchmark 1, double that of system 1 even for simple actions. System 2 results assume use of integer arithmetic units that can keep up to bus data movement and allow full overlap. System 3 performance is best for benchmark 2 because of its action pipelining. Benchmark 1 performance is broken into a fixed setup time and a per element time. For benchmark 2, only the total time is given for a 2 by 4 with a 4 by 2 matrix.

TABLE 4 - 1: SAM 1 Performance (cycles)

| | | SAM 1-1 | SAM 1-2 | SAM 1-3 |
|---|---|---|---|---|
| VADD | int16 | 40 + 6N | 42 + 3N | 42 + 3N |
| | int32 | 42 + 11N | 44 + 6N | 44 + 6N |
| VMUL | int16 | 40 + 40N | 42 + 3N | 42 + 3N |
| | int32 | 42 + 200N | 44 + 6N | 44 + 6N |
| MATMUL (A) | int16 | 863 | 191 | 168 |
| | int32 | 2265 | 303 | 210 |
| | special int16 | 784 | | |
| | flt32 | | | 215 *w |
| MATMUL (B) | int16 | 829 | 171 | |
| | int32 | 2188 | 236 | |

*w Weitek chip set was used.

## 4.6 Space - Time Tradeoffs

We have seen that execution time performance is improved by supporting specific action routines in DMU. There are however some implementation costs associated with this support. Problems stem from the extra decoding necessary to find the specific action routine and the amount of microstore needed for the extra action microprograms. We now discuss some decoding strategies and the cost associated with each. Two mechanisms for handling this were examined. One method is to introduce a class category into the decode hierarchy. Since data type and size is unknown by PMU, class information must be generated by the format routine or OVU from operand tag information and the action byte. Action op code can be dynamically modified by appending a field (which could be wasteful of encoding space) or by table lookup. The format routine could do the table lookup instead, using IF or CASE but this would increase setup time.

Class can specify the device which performs an action. This saves microstore space if a device can perform multiple actions. A variation is to let class distinguish between standard and special actions. This permits a different action EXEC table to be used for special actions. The extra decode time decreases performance slightly for scalars and short vectors.

Another alternative is to defer the operand fetch until the action routine and let the action routine fetch operands since the destination will then be known.

The problem with these strategies is that a large number of action routines will be needed. Generic actions may be decoded into versions for each of the various argument type combinations and data sizes. i.e. 4 combinations of scalar and vectors and 4 data sizes. This is 2 bits for each field, leaving 4 bits for action specification. So, using a single EXEC for decoding, only 16 generic actions can be supported if simple symmetric fields are used. Languages such as APL have more than 16 defined primitives but less frequently used primitives can be composed from the 16 most common. APL actions are unsymmetric in that not all data types are valid for all actions. The 8 bit action code could be more fully utilized to support more actions. More than 16 primitives can be directly supported by using different EXEC tables after the action class is determined.

So far problems with mixed data sizes have not been considered. For 16 bit actions, both operands are 16 bit, so existing microprograms will work. For 32 bit actions, one operand could be 16 bits. This can be tested with the IF microoperation if SR is preloaded with left and right size tags. This complicates the microprogram, but does not degrade performance. Alternatively, mixed actions can be supported with a cost of extra decode time and extra control store. Performance may be increased for some actions. ie. A 16 x 32 bit multiply should be approximately twice as fast as a 32 x 32 bit multiply on system 1.

## 4.7 Glossary of SAMjr Microprogramming Terms

CALL is a mechanism for accessing microprogram subroutines. CALL is monadic and must therefore be the leftmost operation in a microstatement, except for a label.

CARRY is one of the SJ16 status flags. It is set by ALU operations which propagate beyond bit 0 if flag sampling is enabled (SF).

COUNT causes the COUNTER to be incremented and the zero count flag to be sampled. COUNT returns the current ZC flag value for conditional branching. Conditional branching is not mandatory.

COUNTER is a SJ16 hardware COUNTER which acts like a special register. It may be loaded with any value. COUNTER may be incremented in parallel with any other operations in SJ16 or SAMjr. Since the state of the zero count, ZC, flag may be uncertain, it is recommended that the counter be counted once after being loaded and prior to entering a count loop. This prevents a spurious loop exit.

D is a monadic function which converts a decimal string into internal form. It is intended for use with microinstruction literals.

DEC (decrement) requires the implicit use of a register which normally contains -1 called R[DEC]. Therefore R[A] may be decremented and restored in R[A] but a BBUS source may only be decremented and stored in the T-register, R[T]. Another way to decrement is to add or subtract a literal from the ABUS value.

DS, DSPOP, DSPUSH control the data stack which may be used for convenient and fast context storing. The top 8 values of DS may be indexed like R. When DS is pushed (DSPUSH) or poped (DSPOP), all 8 values are changed.

EXEC is used to decode microprogram addresses. The least significant 8 bits on the data bus are used as part of the entry point address. The exec name table name (left argument of EXEC) is used to help make up the rest of the entry point address. Exec name tables can be entered into the microcode data base for use by the linker.

INC (Increment) assumes the availability of a register which contains 1, R[INC]. Increment may also be accomplished by adding or subtracting a literal from the ABUS source. See DEC.

LOCAL applies to nonsegmented memory references. Local memory is word addressable only. Local memory requests are initiated by one of the memory qualifiers: LR, LW ( with L underscored). Local memory requires exactly 2 processor cycles to complete. Read is an address cycle, then a data cycle. Write is a data cycle, then an address cycle. Local memory is not required for a SAMjr configuration but dual port memory has a local memory interface.

LSHIFT moves ALU output left one bit replacing the least significant bit with shift bit (SB) or NOT SB.

MESSAGE is a flag which shows the inclusive OR of all other messages or interrupts. It can be tested to avoid systematically testing all other messages.

MINUS, ex. ALUA MINUS ALUB, performs ordinary subtraction.

MINUSC is subtract with borrow from the previous operation.

NEG is a SR flag. NEG reflects ALU sign output after any sample operation.

NOT is a monadic 1's complement function which may be applied to any ALU BBUS input.

OVFL is the Arithmetic overflow flag which may be set after a sample operation.

PLUS performs Diadic addition.

PLUSC is ALUA+ALUB+CARRY.

R refers to the general purpose register array. Registers are denoted by a subscripted reference to the R array, viz. R[X]. R contains 25 general registers, 4 restricted access registers, and 3 special purpose registers.

REG is a compiler pseudooperation which serves as a variable declaration. No function is performed by REG in an ASP. Any variable defined in a REG statement may be declared as local in the function header so as to avoid proliferating global names.

RSHIFT like LSHIFT, applies to the ALU output. Shift bit (SB) replaces the most significant bit of the result which is shifted right by one bit.

SAR refers to the segment address registers which hold Window (segment) address offsets. SAR is a 4 by 32 bit array. See WDO.

SB, the shift bit is updated whenever a 1 bit ALU shift function is executed. SB is useful for multiple precision shifting and for complex arithmetic functions like multiply and divide. SB is kept in the status register for optional testing.

SD, SS, SRB, SRW, SWB, SWW, SSN, SDN, SWRITE are segment memory control microoperations. Segmented memory references are initiated by one of the qualifiers: SRB, SRW, SWB, SWW (segment-read-byte, segment-write-byte, etc.), placed to the left of the address value in a microAPL expression. Memory data cycles use one of: SD, SS, SDN, or SSN (segment-destination, segment-destination-next, etc.).

SF (sample arithmetic flags) can be used anytime that new values for ZERO, NEG, CARRY, and OVFL flags are required. New values are not available for testing until the following cycle. The flags are defined as follows:
NEG contains ALU result bit 0 (the sign bit).
ZERO indicates that all ALU result bits were 0.
CARRY indicates that an arithmetic carry or borrow occured in bit position 0.
OVFL indicates that a 2's complement overflow occured.

SR is the SJ16 satus register. State information can be manually loaded into the status register or implicitly loaded by sampling flags. Sample flags (SF) causes CARRY, OVFL, NEG, and ZERO to be updated for testing in the next microinstruction. COUNT causes the zero count flag (ZC) to be sampled. Shift operations update the shift bit (SB).

WDO refers to the WDO register array used to store frequently referenced segment addresses (window values). WDO is limited to 4 values accessed by subscripting.

XOR, ex. ALUA EXCLUSIVE-OR ALUB, is a standard dyadic logic function.

ZC (zerocount) is set when COUNTER is incremented to zero. In tight loop action the counter normally exits with the value 1.

ZERO is a status flag which represents the inclusive NOR of all ALU-OUT bits after any sample operation. It is set if all bits are zero.

# CHAPTER 5

## SUMMARY AND CONCLUSIONS

This chapter summarizes results and compares performance of the architectures examined in chapters 2, 3, and 4. First, some results on scalar DEL processing are presented and compared to vector processing. Next, performance of SAM 0.5 is compared to that of SAM 1. Then, SAM performance is compared to that of a popular workstation and two minicomputers. Although only one of these supports vector processing, the others represent the chief market competition for a sytem such as SAM. Next, the impact of the techniques used in this thesis is discussed. Finally, the thesis conclusions are presented and future research directions are indicated.

### 5.1 Scalar vs Vector Processing

Benchmarks were coded as scalar algorithms and simulated SAM 0.5 results were obtained for a FORTRAN type DEL. For 16 bit data using a scalar algorithm, vector add requires 20 + 110N cycles and vector multiply takes approximately 20 + 163N cycles. Matrix multiply for 16 bit data required 6634 DMU cycles and 1401 PMU cycles. PMU time is just for instruction fetch and decode. Verification was not modeled. Performance of scalar processing is 5 to 10 times slower than that of vector

processing. Software complexity and memory requirements are also higher for scalar processing. Streaming further benefits vector processing, increasing the performance ratio of vector to scalar processing. With additional processors, vector processing is still faster. Although extra processors can help speedup some types of scalar processing, it is not easy to make use of techniques such as overlapped operation.

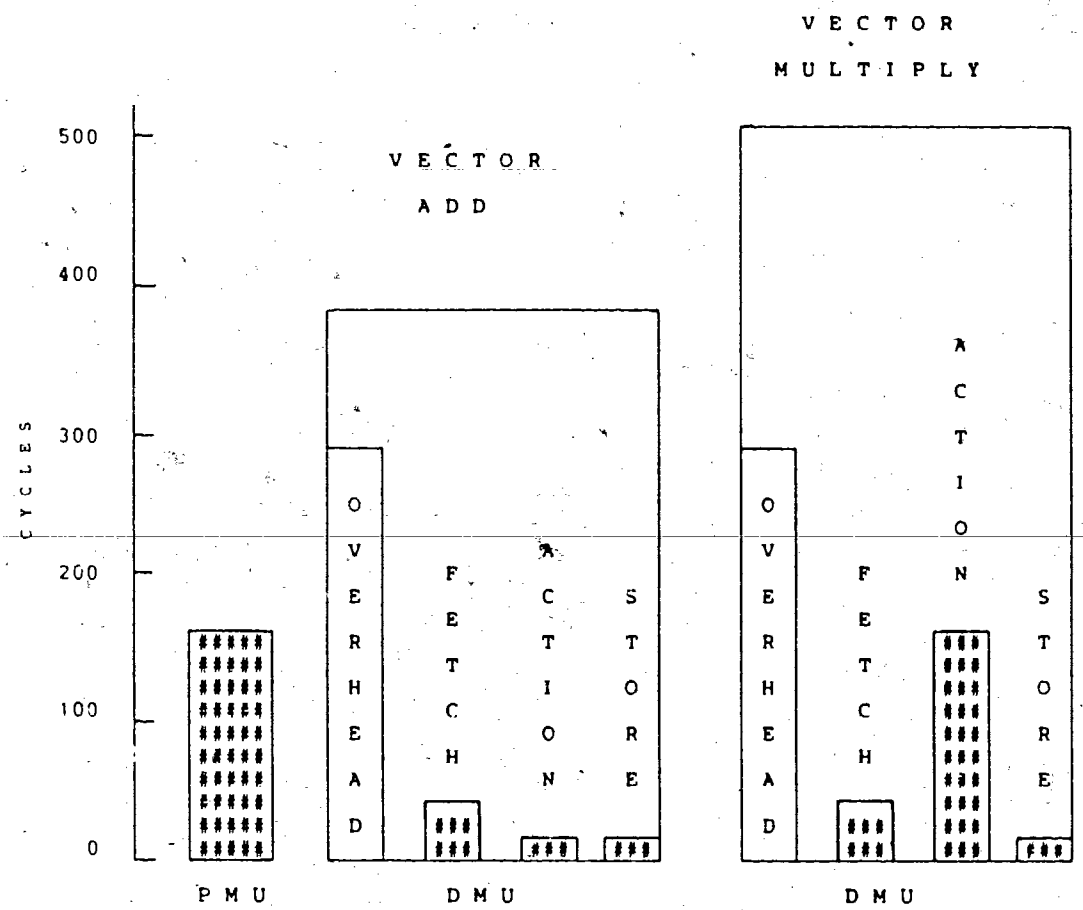


Figure 5-1: Functional Distribution of Benchmark 1 using Scalar Processing.

From figure 5-1, it can be seen that the functional distribution

of the scalar version shows little promise for significant speedup with simple methods. Multiply time will be reduced with a combinational multiplier chip, but further improvements are limited. Even if data manipulation time is reduced, PMU time will become a bottleneck.

## 5.2 Comparison of SAM .5 and SAM 1.0.

Figure 5-2 compares benchmark performance on some SAM 0.5

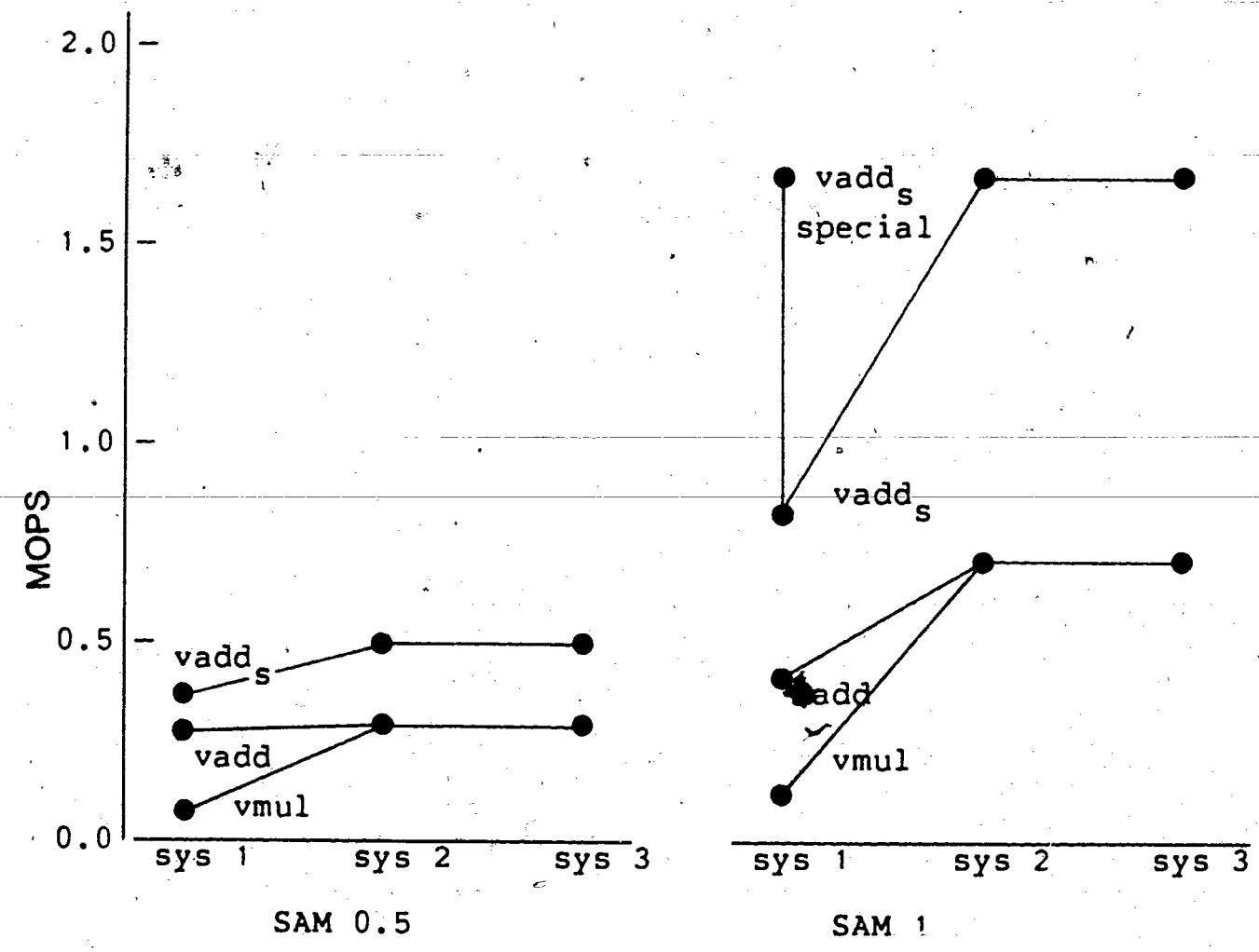


Figure 5-2 : Comparison of SAM 0.5 and SAM 1.0

and SAM 1.0 systems using size specific 16 bit integer microprograms. The best results from each system are used for comparison. System 2 data assumes that arithmetic units completed within the time required to fetch new data. This is reasonable since many commercial chips are available that can meet this requirement. Performance is expressed as millions of operations per second (MOPS). One can see from the slopes of the lines going from system 1 to system 2, that SAM 1 takes better advantage of slave capabilities. It is also faster due to decreased cycle time.

## 5.3 Comparison with other systems

Benchmarks were also run on some other computer systems to compare performance. In addition, timings for some benchmarks were obtained from manufacturers' literature [37]. Table 5-1 shows the performance of SAM running at 5MHz compared with that of some popular super mini computers and workstations. SAM outperforms much more costly systems by one to two orders of magnitude. To be fair, VAX and SUN systems include memory management and memory protection features not supported on SAM. On the other hand, the scalar C benchmarks do not include verification of vector operand compatibility or checks for data size overflow. Thus, if there is a chance that integer benchmark results will overflow, the result should be declared as float, in which case the compiler will include extra conversion code. Matrix multiply results were for 2 by 4 and 4 by 2 arrays.

TABLE 5 - 1: Benchmark Performance Comparison. (usec)

| | | SAM microcode | VAX compiled C | SUN compiled C | HP microcode |
|---|---|---|---|---|---|
| Vector Add | int16 | 9+.75N | | | |
| | int32 | 9+1.5N | 6+19.5N | | 14.3+3.5N |
| Vector innerproduct | | | | | |
| | int16 | 9+.8N | | | |
| | int32 | 9+1N | | | |
| | flt32 | 9+1N | 60+28N | 50+62N | 18.5+5.75N |
| | flt64 | 9+1.4N | 60+35N | 50+137N | 18.5+6.5N |
| MATmul | | | | | |
| | int16 | | | | |
| | int32 | 42 | 640 | 1070 | |
| | flt32 | 43 | 620 | 1230 | |
| | flt64 | 55 | 790 | 2430 | |

SAM results are for system 3.
SAM floating point results use WTL 1064/65 chips.

Relative performance of SAM will increase for larger arrays since fixed startup time will be less significant. If SJBUS were increased to 32 bits, SAM performance for 64 bit data would approximate that given for 32 bit data. This would provide a fairer comparison with VAX. Using the familiar floating point operations per second (FLOPS), performance ratings using 32 bit vector innerproduct are 2, .07, .03, .35 FLOPS for SAM, VAX, SUN, and HP respectively.

While SAM's good performance is partly due to the state-of-the-art Weitek chips, a system configuration that permits full performance is also important. This configuration of SAM has closely matched DMU - APU requirements resulting in

almost optimal use of resoures. The Weitek chips are not used in pipeline mode since SJBUS cannot deliver data fast enough. Comparison of FRI, cf. table 5-2, gives an indication of how well the systems make use of functional units. Using the HP fpp hardware, SAM loop time is 2 1/2 to 4 times faster than the HP A700 computer. FRI = 1 with this hardware, a threefold improvement in resource usage. Streaming, pipelining, and an improved SFU interface account for this speedup. Matrix multiply performance could be improved fivefold to a one cycle innerloop using algorithm 2, local memory to hold intermediate results, and a 32 bit SJBUS capacity. This is a topic for further study since the average use statistics cannot justify this extra cost. However it shows a high performance possibility for special purpose systems requiring a high percentage of such processing.

TABLE 5 - 2: FRI for Selected Computer Systems

| | SAM | VAX | SUN | HP |
|---|---|---|---|---|
| Vector Add int32 | .33 | | | .21 |
| Vector innerproduct flt32 | 1.4 | .5 | .1 | .3 |
| MATmul flt32 | .7 | .35 | .08 | |

## 5.4 Memory Streaming

Memory streaming turned out to be an important technique for improving performance of vector processing. It improves performance due to decreased memory cycles and also enhances system performance by equalizing the processor workload. In system one without streaming, data fetch required two cycles which tied up both register and bus resources. SJ16 address translation overhead can be avoided but the address cycle cannot usually be combined with another microoperation. Streaming frees up the bus address cycle and the translation time of virtual memory systems.

An on chip cache could perform almost as well but would need a larger data bus or block move capability to maintain performance. Cache systems require much more hardware support and increase minor cycle time due to address comparison overhead. Streaming can perform better than a cache system for vector algorithms since its performance comes from implicit firmware knowledge of data addressing and not from assumptions based on statistical sampling.

Supercomputers use memory streams but in a more limited way. Data streams are usually of a limited length (ie. 64 items on Cray 1). Performance depends on a high degree of memory interleaving with corresponding high bus bandwidth requirements and is affected by data location. Streaming has not been used in microcomputers and could replace instruction buffering with simpler hardware. Its use would speed up access to scalar data

that are larger than bus width if data are in memory.

Streaming can be implemented to reduce some microprogramming complexity. No checks for memory ready are needed if a memory controller inserts clock wait states until data is ready. The microprogrammer no longer has to worry about memory timing - ie. does not need to insert NOPs or wait loops. Microprograms are shorter and more readable. Matrix multiply is simplified since address pointers are not incremented during data fetch and therefore do not need to be reset to initial values when reentering loops.

## 5.5 Separate Arithmetic Processors

While the use of slave arithmetic units or arithmetic processing units will not speed up the execution of simple actions, their use greatly improves performance of those complex actions supported by hardware. While statistically not as frequent as some simple actions, these actions are important because of their long functional times. The extra chip area allows hardware support. Multiplication, division, and floating point actions can be supported with available chips. Dynamic frequency data can be used to select those actions that should be supported by slaves. Using separate arithmetic units makes it possible to overlap fetch-store with execute operations.

It was shown in chapters 3 and 4 that using slaves can eliminate the register contention problem for benchmark 2. Simple control as in figure 3-11 (a) led to increased overhead

due to extra bus cycles for DMU control of slaves. The additional control hardware and data paths of system 3 reduced bus traffic and improved performance for multiple actions. The extra paths allowed definition of special multiply-accummulate actions that increased performance by reducing overhead from data movement. This solves only the problems specific to matrix multiply but will not work for a general dot product unless APU control supports all possible actions. Of course, this special action must be decoded at some step in program translation or execution. On SAM, there is a choice as to where this is done. Buffering of input and output allows overlapped operation. This is most effective for longer actions.

Pipelined functional units offer optimum performance although SJBUS may not be able to supply operands fast enough to keep up to some available high performance units. Equivalent performance on vectors can be obtained with multiple lower performance units, while overcoming commercial chip design shortcomings. In this case, internal double buffering is not needed so available scalar oriented units can be used.

### 5.6 Firmware structure

Various firmware structuring schemes were used. Current quantitative measures of software complexity are crude, and not readily applied to microprogramming. Some important factors for good design are modularity and module independence. With structured firmware, all hardware specific code is contained in

one small module. Making changes is relatively easy, requiring only a new module and a modification to an EXEC table. In standard horizontally microprogrammed systems, simple hardware changes necessitate recompilation of much of the system firmware. It was found that hardware support can help maintain good firmware structure along with good performance. Without hardware support, module independence causes reduced performance increasing execution and maintenance costs. Full support of all actions allows good functional partioning of control, thus simplifying DMU algorithms. Otherwise, more complex firmware structure is needed.

It is useful to incorporate generality into the algorithms to decrease microstore requirements. Generic algorithms work for many cases, thus reducing the number of microprograms needed, but were found to have a large execution cost unless special hardware control methods were used. Better performance was achieved by specific modules for each action and data size. It is open to question whether one complex general module is better than several simpler modules.

In system 2, lack of symetry introduced microprogram complexity. Testing for special cases caused performance degradation. Overhead due to the DMU-APU interface and synchronization method became more critical as the workload became equal in the two units. System 3 extensions can correct these problems with extra hardware, making SFU internal structure transparent to DMU firmware. This allows good

performance while reducing firmware complexity.

## 5.7 Dynamic size data

In this study, dynamically varying data sizes were supported. We now examine the benefits and costs of this scheme. Use of dynamic size results in large savings in data memory requirements and execution time. Some interpretive systems use a fixed size for data to accommodate the largest data. From statistical studies we can approximate the extra performance cost of such a system. The well known Gibson mix [26], based on IBM 7090 instruction frequencies, uses an approximately equal mix of fixed and floating point arithmetic instructions. However, it may be biased in two ways. Control integers may increase the use of integer arithmetic. It is also biased toward floating point since some languages convert to float if an expression has any floats. Also, in conventional typed languages, a user must use floating point type if there is any chance of a variable's contents becoming larger than can be held in an integer. IBM 360 studies [49] show that integer arithmetic occurs with twice the frequency of floating point. Even using the Gibson mix ratios, DMU fetch and store cost using a fixed size will be double that of a dynamic size system. Unless special floating point hardware is used, action cost will be even greater.

The cost of supporting dynamic size is increased complexity in the interpreter to detect overflow and decode generic

actions. This results in a need for many size and action specific microprograms resulting in an increased microstore requirement to maintain high performance. This study did not examine the cost of overflow recovery as no data on overflow frequency were available. Available statistics indicate that overflow should be infrequent. Hennessy et.al. [36] report that 95% of constants are less than 255. It is also reported in [18] that 75 to 90 % of constants had absolute values < 8. If variable values are also usually small integers, then the results of data manipulations should still be integers. Actual results have to wait until a complete system is running to give statistical data and memory management cost.

## 5.8 Conclusions

The main motivation for this study was to provide some performance estimates for SAM. The results of this thesis show that a SAM architecture is effective for array data manipulation. Use of simple slave arithmetic units gave performance that was ultimately limited by bus data movement requirements for benchmark 1 and improved benchmark 2 performance by a factor of 4 to 7. System 3, with a more complex APU can maintain performance for benchmark 1 and improve benchmark 2 performance by 50 %.

Most commercial arithmetic units were found to have design drawbacks for use in an interpretive HLLCS. Addition of extra control logic and use of multiple units can overcome these

drawbacks.

SAM outperformed some scalar competitors by one to two orders of magnitude. It also outperforms some systems with vector support by a factor of 2 to 5, while providing the extra functions required for dynamically typed interpretive systems.

The results of this study are slightly optimistic since some details such as memory allocation and overflow recovery were not modeled. Final evaluation must await implementation of a complete system to derive needed statistical data.

This thesis explored trade-offs in microprogram structure. Generality in action routines was costly in terms of run time but saved microstore space. A general interface was also costly to run time performance, especially when separate action procesors were added. Size and action specific routines were needed to give maximum performance. This necessitates a very large number of small routines, especially for a language such as APL. Although SAM has a large microstore space, a system implementor may have to make compromises between these approaches. Statistical information can be used to select the optimized instruction set to be directly supported. A structured microprogram development approach need not be detrimental to run time performance on SAM if supported by the microarchitecture. Structured microprogramming tools were found useful for microprogram development in an evolving environment. Thus, higher level microprogramming oriented architectures analagous to language-directed architectures may offer benefits to firmware

engineering.

### 5.8.1 Future Research

Some ideas for increasing performance were not pursued due to time limitations. They may yield good results if researched further. A short discussion of these ideas follows.

There are many possible choices for an internal executable form. In fact, PMU and DMU do not need to execute the same DEL/DIL form. Indeed it may be beneficial for DMU to execute a form that has been split into a fetch-action and a store phase. This reduces the number of microprograms needed since many formats differ only in the destination specification

e.g. DLR SLR RLR all become SLR.

Results are stored in a separate segment and then pointers are adjusted. The 3 operator syllable DEL form was important for scalars to reduce redundant variable binding and minimize the size of the intermediate code. However, in SAM, PMU performs variable binding and can pass pointers to DMU. DMU benefits from reduced complexity and a reduced number of table entries for formats which may allow space for more special actions. All fetch-action formats leave results on a stack. The store format then moves results to the destination or merely changes the destination descriptor. If an APU subsystem with local memories is used results can be left in the local memory which can be considered top of stack. It may be possible to start interpreting the next format while the last action completes

perhaps allowing overlapped processing of scalar DELs.

In this study, the largest data item determined the size tag of an array. Other methods similar to those used for sparse arrays could decrease memory requirements, decrease bus traffic, and lower the cost of overflow recovery. If only a few elements of an array are of a larger size, a bit vector can be used to identify these. A modification to the APU interface was suggested using a simple FIFO as a data buffer. This can be used to hide APU hardware from DMU firmware thus enhancing module independence and reducing DMU firmware complexity.

The general algorithms could be simplified through hardware support of tagged data. In system 2 and 3, SFUs could be given the size tags of their operands. Then, for mixed data sizes, they could sign extend the smaller. DMU would only need to transfer data from memory to SFUs. This could be even simplier if the memory interface was aware of size tags. Then DMU need only initiate bus transfers which could proceed with no further intervention from DMU. Reduction in data transfers could be realized if the memory interface and SFUs recognized data which were merely sign extended beyond the low order word. Then only the low order word need be transferred and sign extended at the bus destination.

The SAM architecture could be extended with a separate store processor. Although memory contention and data dependency problems will hinder implementation, such a system could improve benchmark 1 performance by up to 50% over system 2 or 3 and

enable pipelined scalar processing. A better method of modeling memory contention is needed for simulation of this system.

This thesis has implications for the design of an indirect execution language. If maximum performance is required, EXEC table and microstore requirements may put restrictions on the number of primitive actions that can be directly supported. APL is one of the most difficult high level languages to support because it has a large number of primitives. However, many of these are seldom used and may be supported as special fuctions written in ADIL. Another iteration is needed in the ADIL design to determine which formats and primitives are needed. Some unused format table space could then be used for direct support of actions.

## BIBLIOGRAPHY


1. Agrawala, Ashok K., and Tomlinson G. Rauscher, Foundations of Microprogramming, Academic Press, Inc., New York, 1976.

2. Aho, A. V., and J. D. Ullman, Principles of Compiler Design, Addison-Wesley, Reading, Mass., 1977.

3. Alexander, Peter, "Array Processor Design Concepts", Computer Design, (December 1981), pp. 163-172.

4. Allan, Stephen J, and Oldehoeft, Arthur E., "A Flow Analysis Procedure for the Translation of High-Level Languages to a Data Flow Language", IEEE Transactions on Computers C-29, 9 (Sep. 1980), pp. 826-831.

5. Andrews, Michael, Principles of Firmware Engineering in Microprogram Control, Computer Science Press, Inc., Potomac, Maryland, 1980.

6. Backus, John, "Can Programming be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs", CACM 21, 8 (August 1978), pp. 613 - 641.

7. Baer, Jean Loupe, Computer Systems Architecture, Computer Science Press, Inc., Potomac, Maryland, 1980

8. Bingham, H.W., "Content Analysis of APL Defined Functions", APL 1975 Conference Proceedings (Pisa, Italy, June 1975), ACM, New York, 1975, pp. 60-66.

9. Bingham, H.W. and K.T. Carvin, "Dynamic Usage of APL Primitive Functions", APL 1976 Conference Proceedings (Ottawa, Canada, Sept. 1976), ACM, New York, 1976, pp. 83-86.

10. Bloom, Howard M., "Conceptual Design of a Direct High-Level Language Processor", in High-level Language Computer Architecture, Y. Chu (ed.), Academic Press, Inc., New York, 1975. pp. 187-242.

11. Burkle, H.J., A. Frick, and C. Schlier, "High Level Language Oriented Hardware and the Post - von Neumann Era", Proceedings of 5th Annual Symposium on Computer Architecture, 1978, pp. 60-65.

12. Carlson, Carl R., "A Survey of High-Level Language Computer Architecture", in High-level Language Computer Architecture, Y. Chu (ed.), Academic Press, Inc., New York, 1975. pp. 31-62.

13. Chu, Yaohan, "Concepts of High-level Language Computer Architecture", in High-level Language Computer Architecture, Y. Chu (ed.), Academic Press, Inc., New York, 1975.

14. Chu, Yaohan, "Architecture of a Hardware Data Interpreter", IEEE Transactions on Computers, C-28, 2 (Feb. 1979) pp. 101-109.

15. Chu, Yaohan, "Direct Execution Computer Design", Computer Science Technical Report, University of Maryland, TR 931,(August 1980).

16. Chu, Yaohan, and Marc Abrams, "Low-level and High-level Computer Architectures", TR 1101, (September 1981). , University of Maryland,

17. Cohen, Jacques, "Computer Assisted Microanalysis of Programs", CACM 25, 10 (October 1982), pp. 724 - 733.

18. Cook, R. and N Donde, "An Experiment to Improve Operand Addressing", Symposium on Architectural Support for Programming Languages and Operating Systems, ACM 1982 , pp. 87-91.

19. Dennis, J.B., etal., "Research Directions in Computer Architecture", in Research Directions in Software Technology, ed. Peter Wagner, pp. 514-556.

20. Dietzel, D.R., and D.A. Patterson, "Retrospective on High-Level Language Computer Architecture", Proceedings Seventh Annual International Symposium on Computer Architecture, May 6-8 , 1980, pp. 97-104.

21. Digital Equipment Corp., VAX-11 Architecture Handbook Digital Equipment Corporation, Maynard, Ma. 1979.

22. Dongarra, Jack J., and S.Eisenstat, "Squeezing the Most Out of an Algorithm in Cray Fortran", ACM Transactions on Mathematical Software, 10, 3 (September 1984), pp. 221-230.

23. Doran, R. W., Computer Architecture: A Structured Approach, Academic Press, Inc., New York, 1979.

24. Enslow, Philip H. (ed.), Multiprocessors and Parallel Processing, John Wiley & Sons, New York, 1974.

25. Evans, David J.(ed.), Parallel Processing Systems, Cambridge University Press, Cambridge, 1982.

26. Fairlough, Dennis, "A Unique Microprocessor Instruction Set", IEEE Micro 2,2 (May 1982), pp. 8-18.

27. Feilmeier, M. (ed.), Parallel Computers-Parallel Mathematics, North-Holland Publishing Co., Amsterdam, 1977.

28. Flynn, M. J., "The Interpretive Interface: Resources and Program Representation in Computer Organization", in High Speed Computer and Algorithm Organization, Academic Press, Inc., New York, 1977, pp. 41-69.

29. Flynn, M. J., "Directions and Issues in Architecture and Language", IEEE Computer 13, 10 (Oct 1980), pp. 5-22.

30. Flynn, M.J., N.R. Harris, and D.P. McCarthy, Microcomputer System Design, Lecture Notes in Computer Science 126, Springer-Verlag, New York, 1982.

31. Flynn, Michael J., and Lee W. Hoevel, "Execution Architecture: The DELtran Experiment", IEEE Transactions on Computers C-32, 2 (February 1983), pp. 156 - 174.

32. Fried, Stephen S., "Evaluating 8087 Performance on the IBM PC", Byte 9,9 , pp. 197 - 208.

33. Hammer, Michael and Gregory Ruth, "Automating the Software Development Process", in Research Directions in Software Technology, ed. Peter Wagner,

34. Hayes, John P., Computer Architecture and Organization, McGraw-Hill Book Co., New York, 1978.

35. Heath, J.L., "Re-evaluation of the RISC I", Sigarch 12 - 1 (March 1984), pp. 3 - 10.

36. Hennessy, J. etal., "Hardware/Software Tradeoffs for Increased Performance", Symposium on Architectural Support for Programming Languages and Operating Systems, ACM 1982, pp. 2 - 11.

37. Hewlett Packard, HP1000 A700 Processor Design and Common Specification.

38. Hobson, R. F., "Structured Machine Design: An Ongoing Experiment", Proceedings of 8th Annual Symposium on Computer Architecture, (May 1981), , pp. 37-55.

39. Hobson, R.F., "A Synopsis of the SJ16 Controller Chip", SFU (1981).

40. Hobson, R.F., "A Directly Executable Encoding for APL", ACM Transactions on Programming Languages and Systems, 6, 3, (July 1984), pp. 314-332.

41. Hobson, R.F., P. Hannon, and J. Thornburg, "Microprogramming with APL Syntax", Proceedings of the 14th Annual Microprogramming Conference, (October 1981), pp. 131-139.

42. Hobson, R.F., John Gudaitis, and Jonathon Thornburg, "A New Machine Model for High-Level Language Interpretation", SFU, (1985).

43. Hoevel, L. W., "Ideal Directly executed Languages: An Analytical Argument for emulation" IEEE Transactions on Computers 23, 8 (Aug. 1974), pp. 759-767.

44. Hoevel, L. W., "DELTRAN: A Case Study in a FORTRAN DEL", presented to International Workshop On High-Level Language Computer Architecture, Fort Laurerdale, (May 1980).

45. Hoffmann, Werner, "Implementation and Evaluation of Vertical Algorithms on a Microprogrammable Computer", in Parallel Computers-Parallel Mathematics, M. Feilmeier (ed), North-Holland Publishing Co., pp. 79-82.

46. Horowitz, Ellis and Sartaj Sahni, Fundamentals of Computer Algorithms, Computer Science Press, Inc., Potomac, Maryland, 1978.

47. Hwang, Kai, Shun-Piao Su, and Lionel M. Ni, "Vector Computer Architecture and Processing Techniques" in Advances in Computers, Vol. 20, Marshall C. Yovits (ed.) , pp. 115-197.

48. Ibbett, Roland N., The Architecture of High Performance Computers, The MacMillan Press, Ltd, London, 1982.

49. Iliffe, J. K., "Microsystems Support for High Level Languages", International Workshop on High-level Language Computer Architecture (1980) Fort Lauderdale, Florida.

50. Iliffe, J.K., Advanced Computer Design, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

51. Ishikawa, Chiaki, Ken Sakamura, and Mamora Maekawa Adaptation and Personalization of VLSI-Based Computer Architecture", Micro 14 ,(1981) , pp. 51 - 61.

52. Jaffe, "The Use of Queues in the Parallel Data Flow Evaluation of IF-THEN-WHILE Programs", MIT/LCS/TM-104, Massachusetts Institute of Technology, 1978.

53. Johnson, J. B., "The Contour Model of Block Structured Processes", Sigplan Notices 6, 2 (February 1971), pp. 55 - 82.

54. Kartashev, S.I., and S.P. Kartashev, "A Multicomputer System with Dynamic Architecture", IEEE Transactions on Computers, C-28, 10, (October 1979), pp. 704-720.

55. Kartashev, Svetlana P., and Steven I. Kartashev (Ed.), Designing and Programming Modern Computers and Systems Vol. 1., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

56. Kavi, Krishna, etal., "HLL Architectures: Pitfalls and Predilections", Proceedings of 9th Annual Symposium on Computer Architecture, (1982), pp. 18-23.

57. Klingman, Edwin E., Microprocessor Systems Design, Vol. II., Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1982.

58. Kogge, Peter M., "The Microprogramming of Pipelined Processors", Proceedings of 4th Annual Symposium on Computer Architecture, 1977, pp. 63-69.

59. Kogge, Peter M., The Architecture of Pipelined Computers, McGraw-Hill Book Co., New York, 1981.

60. Kuck, David J., "Parallel Processing of Ordinary Programs", in High Speed Computer and Algorithm Organization, pp. 119-179.

61. Kuck, David J., "The Burroughs Scientific Processor (BSP)", IEEE Trans. on Computers C-31, 5 (May 1982) , pp. 363-376.

62. Kung, H.T., "Why Systolic Architectures", IEEE Computer, (January 1982), pp. 37-46.

63. Laliotis, T.A., "Architecture of the SYMBOL Computer System", in Advances in Computer Architecture ed. Y. Chu, pp. 110 - 187.

64. Lawson, Harold W., etal., Large Scale Integration: Technology, Applications, and Impacts, North-Holland Publishing Co., Amsterdam, 1979.

65. Lord, Norman W., etal., Advanced Computers, Ann Arbor Science Publishers, Ann Arbor, Michigan, 1983.

66. Lorin, Harold, Parallelism in Hardware and Software: Real and Apparent Concurrency, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

67. Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architectures", CACM 20, 3 (March 1977), pp. 143-153.

68. McDaniel, G., "An Analysis of a Mesa Instruction Set Using Dynamic Instruction Frequencies", Symposium on Architectural Support for Programming Languages and Operating Systems, ACM 1982, pp. 167 - 176.

69. Maejima, H., etal., "The VLSI Control Structure of a CMOS Microcomputer", IEEE Micro , (December 83), pp. 9-16.

70. Mead, Carver and Lynn Conway, Introduction to VLSI Systems, Addison-Wesley Publishing Co., Reading, Mass., 1980.

71. Micros Systems Corp., MK-16 Computer System Manual, 1978.

72. Myers, Glenford J., Advances in Computer Architecture (2nd ed.), John Wiley & Sons, New York, 1982.

73. Parnas, David L., "On the Criteria to be used in Decomposing Systems into Modules", CACM 15, 12 (December 1972), pp. 1053-1058.

74. Patterson, David A., and Carlo H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer", Proceedings of 8th Annual Symposium on Computer Architecture, (May 1981), , pp. 443-458.

75. Patterson, David A., and Carlo H. Sequin, "A VLSI RISC", IEEE Computer 15, 9 (September 1982), pp. 8-21.

76. De Prycker, Martin, "A Performance Comparison of Three Contemporary 16-bit Microprocessors", IEEE Micro (April 1983), pp. 26-.

77. Robinet, Bernard J., "Architectural Design of an APL Processor", in High-level Language Computer Architecture, Y. Chu (ed.), Academic Press, Inc., New York, 1975. , pp. 243-268.

78. Saal, Harry J., and Zvi Weiss, "Some Properties of APL Programs", APL 1975 Conference Proceedings (Pisa, Italy, June 1975), pp. 292 - 297.

79. Saunders, Steven E., "Compiling Customized Executable Representations and Interpreters", CMU-CS-79-127, June 1979, Computer Science Department, Carnegie-Melon University.

80. Singhania, A.K., and Berra, P.B., "Associative Processor Application to Change Detection", in Parallel Computers-Parallel Mathematics, M. Feilmeier (ed), North-Holland Publishing Co. , pp. 247-256.

81. Smith, James E., "Decoupled Access/Execute Computer Architectures", Proceedings of 9th Annual Symposium on Computer Architecture, (1982), pp. 112-119.

82. Stevenson, David, "Parallel Computers in the 1980s", in Tools for Improved Computing in the 80's, ed. Paul A. Willis, Seventeenth Annual Technical Symposium, National Bureau of Standards.

83. Taki, k., Kaneda, Y., and Maekawa, S., "The Experimental Lisp Machine", IJCAI-79 Proceedings of the Sixth International Joint Conference on Artificial Intelligence (1979), pp 865-867.

84. Tanenbaum, A. S., "Implications of Structured Programming for Machine Architecture", CACM 21, 3 (March 1978), pp 237-246.

85. Tartar, John, "Multiprocessor Hardware: An Architectural Overview", Proceedings of 1980 ACM Annual Conference, pp. 518-526.

86. Texas Instruments, Inc., "Floating-Point Arithmetic with the TMS 32010", 1984.

87. Texas Instruments, Inc., "TMS 32020 Digital Signal Processor, Product Description", 1985.

88. Thurber, Kenneth J., Large Scale Computer Architecture - Parallel and Associative Processors, Haydon Book Company, Inc., 1976.

89. Thurber, Kenneth J. and Peter C. Patton, Data Structures and Computer Architecture, Lexington Books, D.C. Heath and Co., Lexington, Mass., 1977.

90. Tokoro. Mario, and Takashi Takizuka, "On the Semantic Structure of Information", Proceedings of 9th Annual Symposium on Computer Architecture, (1982), pp. 211-217.

91. Treleaven, Philip C., "VLSI Processor Architectures", IEEE Computer (June 1982) pp. 33 - 45.

92. Wallach, Y. "Alternating Sequential - Parallel Processing", Lecture Notes in Computer Science 127, Springer-Verlag.

93. Weitek Corp., WTL 1064/1065 High Speed 64bit IEEE Floating Point Multiply/ALU , Weitek Corporation, Sunnyvale, Ca. 1984.

94. Weitek Corp., WTL 1164/1165 High Speed 64bit IEEE Floating Point Multiply/ALU , Weitek Corporation, Sunnyvale, Ca. 1985.

95. Weitzman, Cay, Distributed Micro/Minicomputer Systems, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1980.

96. Wichmann, Heide, "Algorithms for Vertical Processing", in Parallel Computers-Parallel Mathematics, M. Feilmeier (ed), North-Holland Publishing Co., pp. 75-77.

97. Wiecek, C.A., "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution", Symposium on Architectural Support for Programming Languages and Operating Systems, ACM 1982, pp. 177 - 184.

98. Wiedmann, Clark, "A Performance Comparison between an APL Interpreter and Compiler", APL 83 Conference Proceedings, pp. 211-217.

99. Williams, Rhon, "A Multiprocessing System for the Direct Execution of LISP", Fourth Workshop on Computer Architecture for Non-numeric Processing ,Sigmod vol. X, no. 1, (August 1978), pp 35-41.