

Copyright © 1990, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

**A RECONFIGURABLE MULTIPROCESSOR  
SYSTEM FOR DSP BEHAVIORAL SIMULATION**

by

Wook Koh

Memorandum No. UCB/ERL M90/53

13 June 1990

*COVER PAGE*

**A RECONFIGURABLE MULTIPROCESSOR  
SYSTEM FOR DSP BEHAVIORAL SIMULATION**

by

Wook Koh

Memorandum No. UCB/ERL M90/53

13 June 1990

**ELECTRONICS RESEARCH LABORATORY**

College of Engineering  
University of California, Berkeley  
94720

TITLE PAGE

**A RECONFIGURABLE MULTIPROCESSOR  
SYSTEM FOR DSP BEHAVIORAL SIMULATION**

by

Wook Koh

Memorandum No. UCB/ERL M90/53

13 June 1990

**ELECTRONICS RESEARCH LABORATORY**

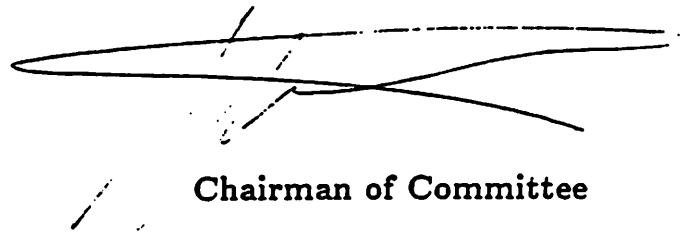
College of Engineering  
University of California, Berkeley  
94720

# A RECONFIGURABLE MULTIPROCESSOR SYSTEM FOR DSP BEHAVIORAL SIMULATION

Ph.D.

Wook Koh

Department of EECS



Chairman of Committee

## ABSTRACT

A major part of the design effort for DSP systems is devoted to the algorithmic verification and specification process. The behavioral simulation of DSP algorithms on a programmable computer will provide the flexibility to develop the algorithms and enable the short design cycle. However, the simulation often requires high computational throughput and the simulation of large amounts of data. It takes too long or is too costly to simulate on a general purpose computer.

Therefore a dedicated simulation engine called SMART has been developed and presented in this report. It is a multiprocessor architecture optimized for real-time behavioral simulation of Digital Signal Processing (DSP) systems. The first prototype, containing 10 processors, is currently operational with a peak performance of 120 MFLOPS.

The SMART system features a *Configurable Bus* and a *Bypass Unit* to trade off overall communication bandwidth and latency by taking advantage of the local communication between processors. The system performance is further improved by a *Distributed Shared Memory* system which lets the communication latency overlap

with the computation time of the processors. *Barriers, locks and events* are supported by hardware to minimize the synchronization overhead. The benchmarks have demonstrated that the SMART architecture actually achieves the targeted low communication and synchronization overhead.

In a SMART simulation environment, the designer can describe the algorithms using a high level language: C and Silage. The C programming environment, which requires the partitioning information in the program, is currently available. A high level software system, based on Silage, is under development to auto-schedule the algorithmic description onto the SMART processor array with a balanced loading and an efficient usage of the communication system. Performance of the actual SMART system was measured for typical DSP programs using floating-point operations. The measurement shows an average speedup of 76 times over SUN 3/60 and a speedup of 29 times over SUN SPARC Station 1. With extensive uses of library routines in programming, the speedup can be easily doubled over the above results. The performance is expected to increase even further when the system is upgraded from 120 MFLOPS to 200 MFLOPS.

## Acknowledgement

A large number of people have contributed in many different ways towards the successful completion of the project described in this thesis. My greatest indebtedness is to my advisor, Professor Jan Rabaey, who always had a great faith in me, and gave invaluable guidance, support and inspiration throughout the course of this project.

I am also indebted to Professor Robert W. Brodersen. Technical discussions with him always seem to provide new insights into the problems. Professor David Hodges, Professor Randy Katz and Professor David Patterson have guided me through my Master Project and taught me valuable lessons to survive and excel in graduate schools.

A project of this scope and nature would not have been possible without the collaborative efforts of Alfred Yeung and Phu Hoang. I am really grateful to have the opportunity to work with them for the last two and half years. For the CAD support, which was indispensable for this project, I am thankful to Brian Richards, Rajeev Jain and Mani Srivastava for maintaining LAGER IV. Thanks are due to Phil Schrupp for maintaining PCB software, Jonathan Min for providing an interface to the Visula program, Robert Yu for helping me to program PLDs, and Kirk Thege and Kevin Zimmerman for maintaining the computers.

This research was sponsored in part by DARPA and the AT&T Company. AT&T was always helpful for giving necessary informations and components to build our system. The MOSIS group deserves a great deal of credit for providing a high quality of VLSI and PCB foundaries. They were also very helpful and cordial in trying to resolve any problem that occurred.

I must thank to my close friends, relatives and parents – Dr. Pum J. Koh and Mrs. Hye S. R. Koh – without whose support this endeavour would not have been possible.

Finally, I dedicate this dissertation to my dearest friend, S. Rim, who gave me the courage to overcome the adversity when I was going through a very difficult time in my life.

# Contents

<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>ix</b>
<b>1 INTRODUCTION</b>	<b>1</b>
<b>2 SMART ARCHITECTURE</b>	<b>4</b>
2.1 Introduction to the SMART Architecture . . . . .	4
2.1.1 Multiprocessor Review . . . . .	4
2.1.2 Pipelining and Parallelism - A Pitch Extractor Example . . . . .	9
2.2 Memory Structure . . . . .	14
2.2.1 Cache Coherence Scheme Review . . . . .	14
2.2.2 Implementation of the Distributed Shared Memory . . . . .	21
2.3 Reconfigurable Bus . . . . .	24
2.3.1 Previous Work in Reconfigurable Bus Architectures . . . . .	24
2.3.2 Our Reconfigurable Bus . . . . .	28
2.3.3 Interprocessor Communication . . . . .	30
2.3.4 Self-Reconfigurable Bus . . . . .	33
2.4 Synchronization . . . . .	36
2.4.1 Barriers . . . . .	37
2.4.2 Locks . . . . .	39
2.4.3 Events . . . . .	39
2.5 Benchmark . . . . .	41
<b>3 SMART ARCHITECTURE IMPLEMENTATION</b>	<b>46</b>
3.1 SMART System . . . . .	46
3.2 SMART Processor Array . . . . .	50
3.3 Processing Unit . . . . .	52
3.4 Master Access Controller and Slave Access Controller . . . . .	59
3.4.1 Access Controllers Instructions . . . . .	60
3.4.2 Functional Blocks in the MAC Chip . . . . .	62
3.4.3 Functional Blocks in the SAC Chip . . . . .	67



<b>4 DISCUSSIONS ON DESIGN AND IMPLEMENTATION</b>	<b>68</b>
4.1 Architecture Design . . . . .	69
4.2 Functional Description and Simulation . . . . .	69
4.3 Board Implementation . . . . .	74
4.4 Chip Implementation . . . . .	76
4.5 Testing and System Integration . . . . .	77
<b>5 PROGRAMMING SMART</b>	<b>81</b>
5.1 Overview of Programming Environment . . . . .	81
5.2 Programming in C . . . . .	84
5.2.1 Macros for Accessing Memory . . . . .	85
5.2.2 Macros for Synchronization . . . . .	86
5.2.3 Macros for System Configuration . . . . .	87
5.2.4 A FFT Example . . . . .	89
5.3 Real-Time Input/Output . . . . .	92
5.4 Performance of the SMART System . . . . .	95
5.5 Auto Scheduling . . . . .	98
<b>6 CONCLUSIONS</b>	<b>101</b>
<b>Bibliography</b>	<b>104</b>
<b>A MAC Instructions</b>	<b>111</b>
<b>B MACRO Definitions</b>	<b>118</b>
<b>C IMAGE.C Program</b>	<b>121</b>
<b>D How to Run the SMART System</b>	<b>140</b>

# List of Figures

2.1	Pitch Extractor. . . . .	11
2.2	Bus Reconfiguration of the Pitch Extractor Algorithm. . . . .	12
2.3	Examples of Bus Reconfiguration. . . . .	13
2.4	Static Coherence Scheme. . . . .	16
2.5	Dynamic Coherence Scheme. . . . .	18
2.6	Communication Scheme in SMART Architecture. . . . .	20
2.7	Memory Address Segmentation. . . . .	22
2.8	The CHiP Architecture. . . . .	25
2.9	The MP/C Architecture. . . . .	27
2.10	SMART Memory and Reconfigurable Bus. . . . .	29
2.11	Bus Transaction for Interprocessor Communication. . . . .	31
2.12	Examples of Self Reconfigurable Bus. . . . .	35
2.13	Examples of Barrier Synchronization. . . . .	38
2.14	Examples of Lock Synchronization. . . . .	40
2.15	Examples of Event Synchronization. . . . .	42
3.1	Block Diagram of the SMART System. . . . .	47
3.2	SMART Processor Array. . . . .	49
3.3	SMART Processor Board. . . . .	51
3.4	System Clock. . . . .	53
3.5	Block Diagram of the SMART Processing Unit. . . . .	54
3.6	Internal Clock Signals. . . . .	58
3.7	Block Diagram of MAC. . . . .	63
3.8	Block Diagram of SAC. . . . .	64
3.9	Pictures of MAC and SAC - die size 1.1cm x 1.1cm . . . . .	65
4.1	Thor Modeling Example - Arb.c. . . . .	71
4.2	Simulation Example. . . . .	73
4.3	Test Configuration. . . . .	79
5.1	Programming Environment for SMART. . . . .	83
5.2	Programming Example - FFT. . . . .	90

**5.3 Real-Time Data Interface. . . . . 93**  
**5.4 Silage to SMART Compiler. . . . . 99**

# List of Tables

2.1	Benchmark Results for 16 Processors. . . . .	43
2.2	Benchmark Results for 64 Processors. . . . .	43
5.1	Performance Results for 10 Processors. . . . .	95
A.1	DSP32C Memory Configuration, Mode 7 (ROM-less version). . . . .	112
A.2	Address Mapping for Memory Bank 0 (Group 1 Instructions). . . . .	113
A.3	Address Mapping for Memory Bank 0 (Group 2 Instructions). . . . .	114
A.4	Address Mapping for Memory Bank 1 (Internal Access Only). . . . .	115

# Chapter 1

## INTRODUCTION

*The woods are lovely, dark and deep.  
But I have promises to keep,  
And miles to go before I sleep,  
And miles to go before I sleep.  
- Robert Frost*

DSP applications have become an important component in the rapidly expanding field of Application Specific Integrated Circuits (ASIC). Examples of applications in this field include digital audio, speech synthesis and recognition, telecommunication, image and video processing and robotics. Due to the high throughput and special input/output requirements, it is often necessary to design special purpose chips, which are optimized for only that particular application. Recently, a great deal of attention has been focused on the development of computer aided design environments, which may help to shorten the design time of those dedicated devices [36].

It has been noted that a major part of the design effort for DSP systems is devoted to the algorithmic verification and specification process. This process often requires high computational throughput and the simulation of large amounts of data [45] [61]. For example a computation rate of 1 GOPS (Giga Operations Per Second) or more is typical for High Definition Television (HDTV) algorithms [27]. Furthermore, to verify the behavior of the algorithms, many frames of data have to be simulated. These requirements dictate a hardware solution.

The process also includes the simulation of the noise and distortion behavior

taking into account the effect of quantization, rounding and truncation. It is only after a careful checking of all those parasitic effects that Application Specific Integrated Circuits (ASIC) can be implemented.

While techniques such as bread-boarding and fast-prototyping can fulfill the requirements, they typically exhibit long development time and offer very little programmability which is essential in optimizing the parameters of some algorithms.

The behavioral simulation of DSP algorithms on a programmable computer will provide the flexibility to develop the algorithms and enable the short design cycle. Some commercial multiprocessor computers are capable of providing high computation power but the high overhead in inter-processor communication, difficulty in mapping the algorithms to the architecture, lack of instructions for supporting DSP applications and the high cost of the machines often limit the effectiveness of these machines.

In this report, a dedicated compute-engine called SMART (an acronym for Switchable Multiprocessor Architecture supporting Real Time applications) is presented. <sup>1</sup> The machine attempts to speedup simulation of DSP algorithms by at least an order of magnitude as compared to general purpose computer architectures.

The speedup is achieved by using the following two methods. First, in order to handle the number crunching bottleneck, a high performance DSP processor with both floating point and fixed point instructions is used as the core processing unit (DSP32C from AT&T Bell Labs [23] [8]). The processor executes up to 10 million floating point (single precision IEEE) multiplications and additions per second. In addition, it provides special instruction sets to support DSP applications. This results in several times of speedup as compared to general purpose processors [67].

Second, another level of simulation speedup can be obtained by exploiting the high degree of concurrency present in most signal processing algorithms [32]. The SMART system should be able to simulate a large variety of DSP algorithms with various degrees and grains of concurrency. Pipelining and Parallelism are two methods used to achieve concurrency.

---

<sup>1</sup>Early studies on the architecture can be found in [38] and [39].

This report describes the architecture and implementation of the SMART machine, and justifies and evaluates some of the architectural features with the aid of system, software and application considerations. Chapter 2 provides an introductory section, which reviews other multiprocessor systems, suggests architecture requirements, and discusses pipelining and parallelism in a typical DSP algorithm. Then the three major aspects of architecture – the memory structure, the reconfigurable bus and the synchronization – are described. The set of software benchmarks are presented to demonstrate the effectiveness of the architecture. In chapter 3, the physical implementation of the SMART architecture is presented in a top-down manner. We first present an overview of the the SMART prototype system. We next provide the detail of implementation of the SMART processor array: the printed circuit board, the processor, the memory structure and the custom VLSI chip sets. Chapter 4 briefly discusses our experiences regarding design and implementation issues. Chapter 5 discusses the supporting software environment of the system. A table which shows the performance of the system is also presented. The last chapter includes some general discussions of the system and the project. Concluding remarks and future developments on SMART are also outlined.

## Chapter 2

# SMART ARCHITECTURE

*To see a world in a grain of sand  
And a heaven in a wild flower,  
Hold infinity in the palm of your hand  
And eternity in an hour  
- William Blake*

### 2.1 Introduction to the SMART Architecture

This section reviews contemporary multiprocessor systems, suggests architecture requirements, and discusses pipelining and parallelism in a typical DSP algorithm – a pitch extractor example.

#### 2.1.1 Multiprocessor Review

The last decade has witnessed the introduction of a wide variety of new computer architectures for parallel processing that complement and extend the major approaches to parallel computing developed in the early years [17]. The recent proliferation of parallel processing technologies has included new parallel hardware architectures such as systolic and hypercube, interconnection technologies such as multistage switching topologies, and programming paradigms such as applicative languages. The sheer diversity of the field makes it difficult for a designer to find out which architecture is right for his target applications.



Our SMART project is motivated by the demand in the signal processing area to build a simulation engine for simulating the behavior of DSP algorithms. The scope of this target system is real-time operation [42] for medium speed DSP applications (speech, telecom, audio and robotics) and a simulation speedup for high performance DSP applications (video and image processing).

In order to find out which architecture is right for our application, we first reviewed the commercially available systems. From programming experiences and studies on well-known machines such as Connection Machine [77] [75], WARP [22] [21], Sequent Balance [71], Intel iPSC [34] and NCube [55]. we have learned the following lessons and adopted them as the basic architecture requirements for the SMART system.

**Number of Processors** — According to the Amdahl's law [6], as the number of processors in a parallel computer increases, it becomes more and more difficult to use those processors efficiently.

For instance, Connection Machine has 65,536 processing elements with a peak performance of 2500 MFLOPS/2500 MIPS. The applications of the machine include simulation of VLSI circuits, picture processing and language processing [76], all of which contain large amounts of parallelism. However when the target algorithms do not have enough parallelism to make use of all those processing elements, most of the processing power is wasted. It is extremely difficult to partition typical medium speed DSP algorithms to thousands or millions of processors.

Therefore, in order to build a system with a low *cost/performance design* we are interested in a system with a relatively small number (10 - 100) of extremely powerful processors.

**Overhead of Interprocessor Communication** — A general purpose multiprocessor system often provides a flexible programming environment at the cost of a substantial amount of performance degradation in interprocessor communication and synchronization.

For example, the Intel iPSC and the NCube are two message-passing mul-

tiprocessors. Both machines are based on the hypercube architecture originally developed at Caltech [70]. The hypercube topology provides a network that is easily scalable to a large number of processors. The network is dense enough to assume each processor is connected to every other processor, while sparse enough to allow a simple implementation.

However performance measurements on the above machines show that it takes about 1msec to send one byte of data from one processor to a nearest neighbor processor.<sup>1</sup> The large communication latency is a result of interprocessor communication protocols implemented in software.

Real-time DSP applications cannot accept such run-time overhead in the interprocessor communication and synchronization. Therefore, the design of the SMART architecture places its highest priority on achieving low interprocessor communication and synchronization overhead.

**SIMD vs. MIMD** — The simulation system should be able to simulate a large variety of DSP algorithms with various degrees and grains of concurrency. The simulation system should efficiently support both pipelining and parallelism.

A SIMD machine cannot exploit pipelining and parallelism simultaneously because this requires execution of different instructions among processors. Therefore our SMART system has a MIMD (Multiple-Instruction stream, Multiple-Data stream) architecture.

**Local vs. Global Communication** — A system should support both local and global communications efficiently. For instance, the communication pattern of a FFT algorithm, when partitioned in a pipelined fashion, mainly consists of local communications [38]. An architecture such as WARP [22] [21] can support those local communications quite well.<sup>2</sup> On the other hand, the communication pattern of a Matrix Multiplication algorithm, when partitioned in a parallel fashion, mostly consists of global communications [38]. According to a benchmark result [38], a system

---

<sup>1</sup>The latency for one byte is quite large due to the initial cost to begin message transmission.

<sup>2</sup>WARP is a systolic array machine [44] [12] designed at Carnegie Mellon.

with a high speed shared bus which provides a direct path between distant processors performed better than a system with only local connections such as WARP. Therefore, we developed a system which can reconfigure the processor interconnections to support both local and global communications [Section 2.4].

**DSP Processor** — On certain DSP-related benchmarks, the performance of programmable DSP processors has consistently exceeded that of general purpose processors (with arithmetic co-processors) by several times throughout their ten year history [48] [50] [52].

Recently, there have been a lot of efforts to develop real-time simulation systems based on powerful DSP processors [46]. In order to handle the number crunching bottleneck, the SMART system uses a high performance DSP processor with both floating point and fixed point instructions as the core processing unit (DSP32C from AT&T Bell Labs [23] [8] ).

The SMART architecture can be applied to far more powerful, new DSP processors such as Texas Instruments TMS320C30 [63], Motorola DSP96002 [41] , and Intel i860 [68] [56]. <sup>3</sup>

**I/O Interface** — The general purpose systems usually do not provide good environment to interface to real-time data. A real-time system should have I/O interfaces which are easy to intergate with the peripheral devices and fast enough to keep up with the incoming sample rate.

For instance, the DSP32C processor provides a fast hardware conversion feature to convert typical 8-bit real-time data representation, which is used in audio-A/D and D/A converters, to and from the 32-bit floating-point representation, which is used for numeric processing. The SMART system also provides a standard VME bus interface and a custom FIFO interface which allow easy intergration to other peripheral systems. Especially, since a custom FIFO is directly interfaced to the core processor, it is possible to build a high speed interface to the real-time data.

---

<sup>3</sup>The Intel i860 processor is more *general* than other DSP processors.

**Multiprocessor System using DSP Processors** — There are many other real-time simulation systems using multiple DSP processors [28].

For instance, the Dolby architecture [49] is a shared memory multiprocessor system. It consists of four Motorola DSP56000's. A transaction to access the shared memory requires first requesting the bus, then reading the memory, checking a semaphore, and resetting the semaphore. Each memory transaction requires about 30 instruction cycles [49]. This relatively high cost implies that only large-grain parallelism can be supported efficiently.

D. Schwartz has developed a multiprocessor system using AT&T DSP32 processors [15]. The system has been specifically designed to support cyclo-static scheduling [74]. The goal of the scheduling is to yield an optimal solution for the following three parameters: the number of processors required, the minimum latency between iterations of the algorithm, and the minimum time between the availability of an input and the availability of the corresponding output. However, the search for an optimal solution requires an exponential run time. Furthermore, the scheduling algorithm does not allow data dependent execution in its fully specified flow graphs [13]. Therefore the system can handle only sub-set of the DSP algorithms.

The ASPEN parallel computer using the DSP32 has been developed by AT&T [62].<sup>4</sup> The system is designed for a large-vocabulary speech recognition system, and is also applicable to other signal processing problems. The processing elements are interconnected to form a complete binary tree [11]. The system is suitable for pattern recognition, and for calculation of means and variances, which is well suited for binary tree connection. However, the architecture is not suitable for other typical applications. For instance, it is quite difficult to map the FFT (Fast Fourier Transform) to the tree architecture. Furthermore, the system does not provide sufficient interprocessor communication bandwidth for the FFT algorithm.

**SMART architecture** — Based on the above observations and architecture requirements, we developed a SMART architecture. It utilizes a *reconfigurable bus* [Section 2.3] to adapt the interconnection to the irregular communication pattern of

---

<sup>4</sup>The computer using the DSP32C is described in [2].

an algorithm, a *distributed shared memory* [Section 2.2] to provide a shared memory system with low interprocessor communication overhead, and hardware support for three types of commonly used *synchronizations* [Section 2.4]. The following section discusses pipelining and parallelism in a typical DSP algorithm and gives the background on how the architecture was initially developed.

### 2.1.2 Pipelining and Parallelism - A Pitch Extractor Example

The simulation system is targeted to simulate a large variety of DSP algorithms with various degrees and grains of concurrency. Pipelining and Parallelism are two methods used to achieve concurrency [58] [61] [26]. Pipelining increases concurrency by dividing a computation into a number of steps and allowing a number of tasks to be in various stages of execution at the same time. For our target DSP applications, it is assumed that the program operates on an essentially infinite stream of input data, executing once per input sample. Therefore we can overlap execution of successive iterations to obtain our speedup. Pipelining is a common form of this general technique. On the other hand, parallel processing emphasizes the concurrent manipulation of data to solve a single problem. The SMART architecture is geared towards exploiting block level pipelining and sub-block parallelism simultaneously to obtain a speedup in simulation time.

The bus reconfiguration is used to achieve efficient communication pattern when both pipelining and parallelism are present. A DSP system usually consists of sequentially data-dependent sub-system blocks. The pipelining of sub-system blocks can increase the throughput of the system and further pipelining within a sub-system block may increase the throughput even more. But some sub-system blocks such as iterative singular value decomposition (SVD) [25] or adaptive filters have constraints such that pipelining within the block is not possible. This is the case when, for instance, the output of the current sample has to be produced before the next sample can be processed. Pipelining within the block is not possible without changing the algorithm. However, the iterative SVD has a large amount of parallelism in the

algorithm. For this case, pipelining will be pursued at the system level and parallelism will be exploited within the iterative SVD block. The following example shows how important it is to exploit both pipelining and parallelism for load balancing.

Figure 2.1 shows an implementation of a high quality pitch extraction algorithm for speech analysis and synthesis [24]. The algorithm shows the combination of pipelining and parallelism typical in DSP applications. Sub-system blocks like the hamming window operation, the FFT, the amplitude calculation operate sequentially on streams of data to extract a pitch from voice samples. The numbers beside the blocks in the figure state the percentage of the computation load for each block. If each block is mapped into a pipelined processor, the *template matching* block will be the throughput bottleneck and we will only get a throughput improvement 1.6 times over the single processor system. However the template matching block can be partitioned naturally over 8 parallel processors [Figure 2.2 (a)], which reduces the load of the bottleneck processor to one eighth. By load balancing sub-system blocks and by mapping into pipelined processors, we get a speedup of 12.1 times over a single processor using 15 processors.

Figure 2.2 (a) shows the data dependence graph for the pitch extraction after partitioning and load balancing process. A data dependence graph consists of nodes, arcs, and delay elements. A node represents operations on data. A directed arc shows the data dependence or the relation between the producer and the consumer. A delay element is a property of the arc connecting two nodes. If there is a unit delay on the arc connecting node A and B, then the  $n$ -th data consumed by node B will be the  $(n-1)$ st data produced by A. Delays can be placed on any *feed-forward cut-set* to increase the degree of pipelining without altering the computation, as long as the latency of the system is not a consideration.

A one to one mapping of this load balanced data dependence graph on a multiprocessor architecture with a customized communication pattern would result in the architecture of Figure 2.2 (b). The dedicated architecture can now be mapped onto the one dimensional SMART architecture as shown in Figure 2.2 (c). Processors P0 to P5 and P14 are working in a pipelined fashion. Processors P6 to P13 are working in parallel to each other and working in pipelined fashion with other processors.

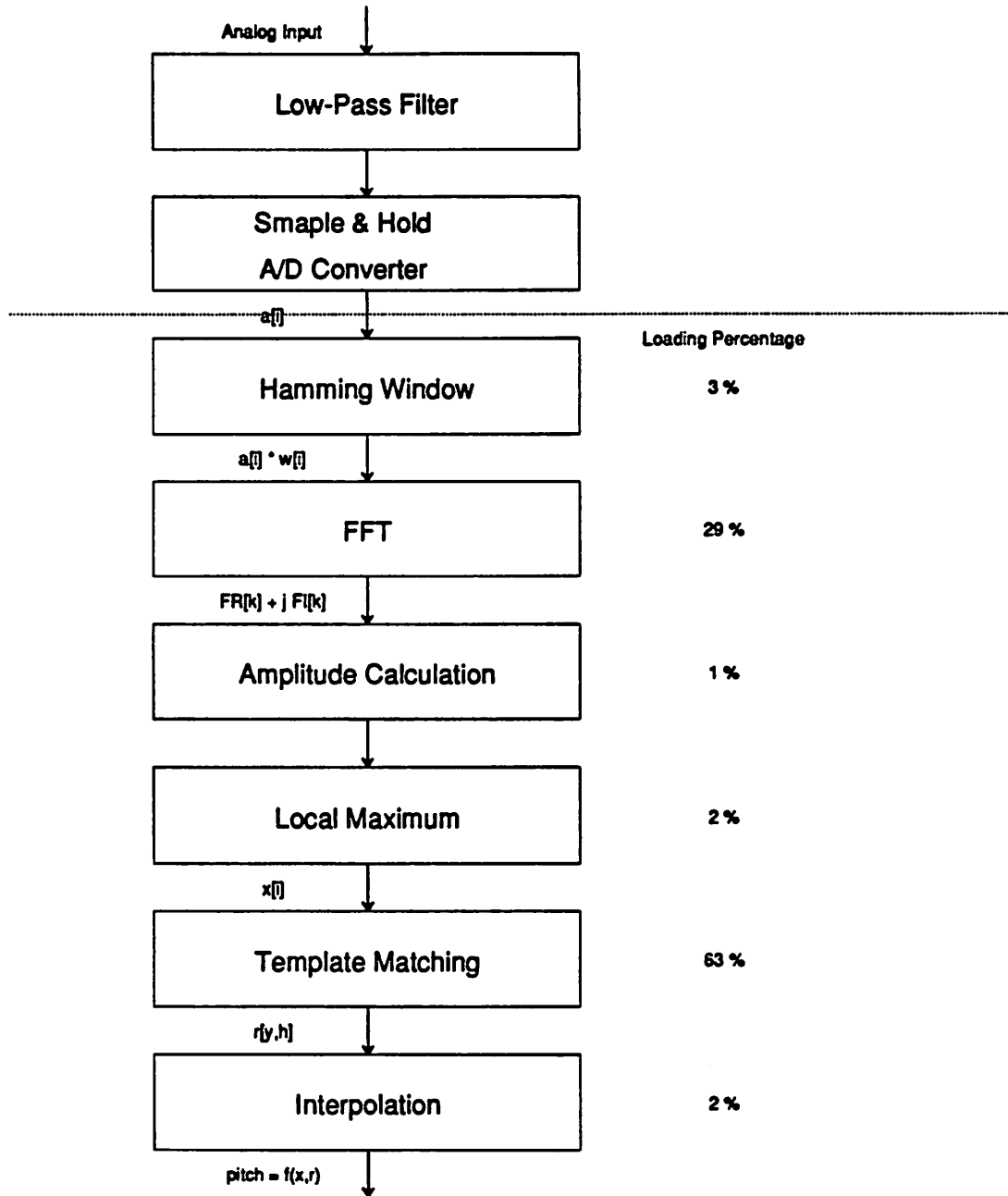


Figure 2.1: Pitch Extractor.

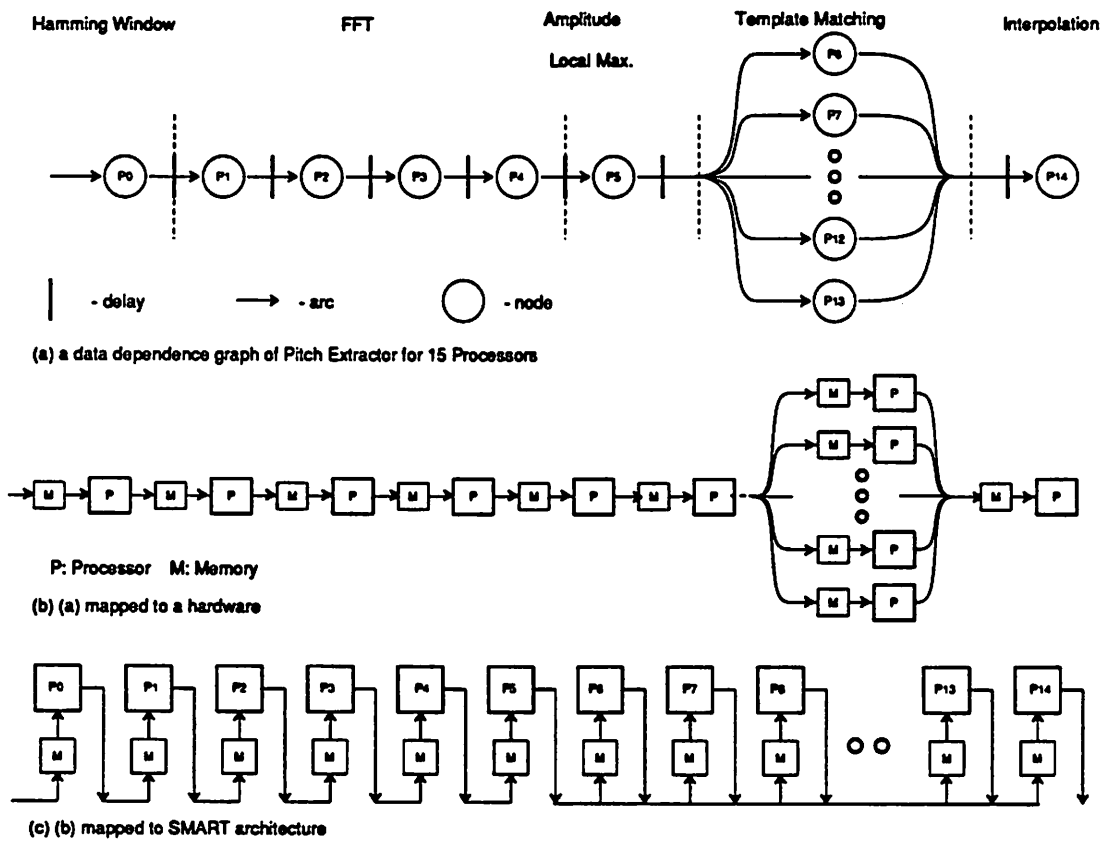
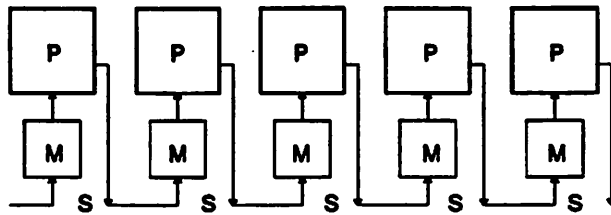
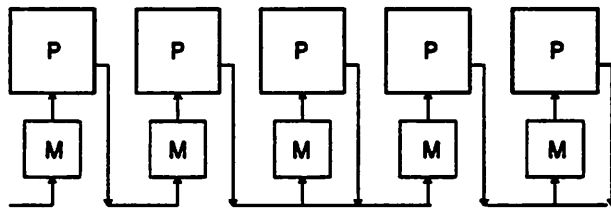


Figure 2.2: Bus Reconfiguration of the Pitch Extractor Algorithm.

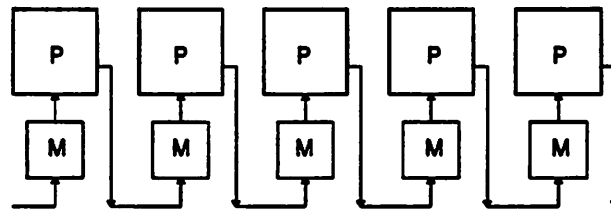




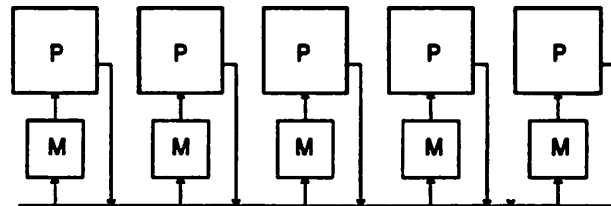
(a) SMART array



(b) an irregularly configured bus



(c) one-dimensional systolic array



(d) a single shared bus

P: Processor, M: Memory, S: Switch

Figure 2.3: Examples of Bus Reconfiguration.

Figure 2.3 shows examples of a reconfigurable bus in simplified diagrams. The SMART architecture can be statically reconfigured under software control to match irregular communication patterns. This results in a flexible architecture and increased bus bandwidth at the cost of hardware switches. The one dimensional systolic array can be realized by opening all the switches [Figure 2.3 (c)] or and the shared-bus multiprocessor system by closing all the switches [Figure 2.3 (d)]. In terms of interconnection, the one dimensional systolic array and the shared-bus multiprocessor system are special cases of SMART architecture.

The following sections deal with the three main aspects of the SMART architecture – the memory structure, the reconfigurable bus and the synchronization – which are formulated from the analysis of DSP examples. These features are implemented in a special semi-custom VLSI chip set [Chapter 3].

## 2.2 Memory Structure

The design objective of memory architecture in a multiprocessor system is to balance the processor speed with the bandwidth of the memory at a reasonable cost. In real-time system applications, a major concern in memory design is the reduction of the communication overhead between processors.

First, cache coherence schemes are discussed in terms of reducing the communication overhead. Then, an implementation of a *distributed shared memory* with a new cache coherence scheme is presented.

### 2.2.1 Cache Coherence Scheme Review

Multiprocessors can be grouped into two different sets of architectural models: a tightly coupled multiprocessor and a loosely coupled multiprocessor [14]. Tightly coupled multiprocessors communicate through a shared memory. Hence the rate at which data can communicate from one processor to another is on the order of the bandwidth of the shared memory. Complete connectivity is provided between the processors and memory. Loosely coupled multiprocessors communicate by exchanging

messages. They do not generally encounter the degree of memory conflicts experienced by tightly coupled systems. Loosely coupled systems are efficient when the interactions between processes are minimal. On the other hand tightly coupled systems can tolerate a higher degree of interactions between processes. Because of the large amount of interaction between processes in our target applications, the throughput of the hierarchical loosely coupled multiprocessor may be too low for applications that require fast response times [26]. Therefore, a tightly coupled structure is chosen for the SMART system.

The presence of caches in a tightly coupled system introduces problems of *cache coherence*. A system of caches is *coherent* if and only if a *read* performed by any processor on any data in main memory always delivers the most recent value. Since the producer and consumer processors of the data communicate through the shared data in cache or main memory, the way to solve cache coherence is closely related to interprocessor communication overhead. The overhead is the sum of the time spent by the producer in writing a shared data word plus the time spent by the consumer in reading the data. In the following paragraphs, the interprocessor communication overhead for *static coherence scheme* and *dynamic coherence scheme* is discussed. After that we present our cache design, where the main goal is to achieve the low communication overhead.

**Static Coherence Scheme** — Solutions to the cache coherence problem can be generally grouped into two methods. The first method, called *static coherence*, does not allow multiple copies of the *shared writable* data to exist in different caches at any given time.

One of the possible implementations is shown in Figure 2.4 (a). Shared data structures which are modifiable remain in shared main memory – called non-cacheable data. References to these shared data are made directly to main memory. On the other hand, *Read-only* segments of data need not be non-cacheable. Read-only segments of data which are shared by several processors may be copied into the cache – called cacheable data. The cacheability of read-only data reduces conflicts in main memory. However, when processors communicate to each other, both a producer and

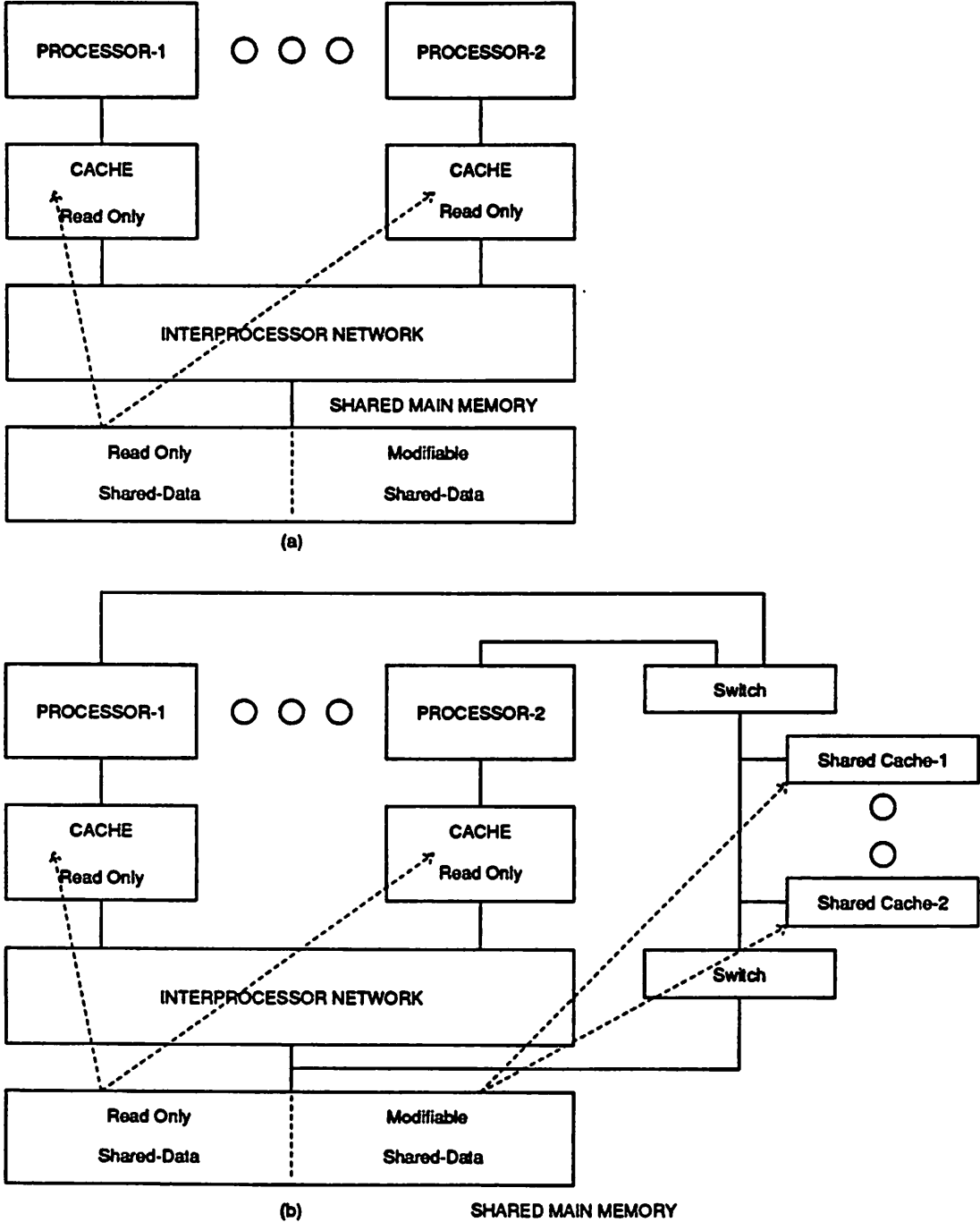


Figure 2.4: Static Coherence Scheme.

a consumer have to access the main memory, not the cache. The overhead to access the main memory is too large for a high degree of communication between processes.

The other type of implementation is to provide *shared cache* for the shared writable data as well as *private cache* for the private data. The shared cache is connected to the processors and the main memory through switching networks. Figure 2.4 (b) illustrates this shared cache concept. The shared data is accessed through a shared cache while instruction fetches and private data references are made in private caches. Data references by producers and consumers proceed at the cache speed except when conflicts occur at the shared cache or a miss occurs. The communication overhead of the this scheme depends on how well the switching networks are designed. The communication overhead is small for a rather small number of processors. However, as the number of processors increases to tens and hundreds, the potential gain in speed will be limited by the transmission delays through the switch to access the shared cache and by the number of conflicts at the cache. Therefore this cache scheme is not appropriate for our application.

**Dynamic Coherence Scheme** — The second method for solving cache coherence, called *dynamic coherence*, allows multiple copies of the same data to exist in caches, provided that all the copies are identical.

To enforce the cache consistency rule in hardware, a tag is associated with each cache block. It indicates the state of the shard data: *read-only state*, *exclusively read-write state*, *exclusively read-only state* and *invalid state*.

If the processor associated with each of the caches has not modified its copy since the data was loaded in its cache, the copy is in *read-only state* (RO). In order to modify the copy in the cache, a processor has to own the copy with *exclusive read-write* (RW) or *exclusive read-only* (EX) states. A RW state means the cache is the only one with the copy and it has been modified. Similarly, an EX state means the cache is the only one with the copy and it has *not* been modified. Therefore at any time, in order to keep cache coherence, only one processor can own a RW or an EX state of a copy. On the other hand, an *invalid* (INV) state means the copy is not valid. For instance, when a processor owns an RW state, all the other copies in other

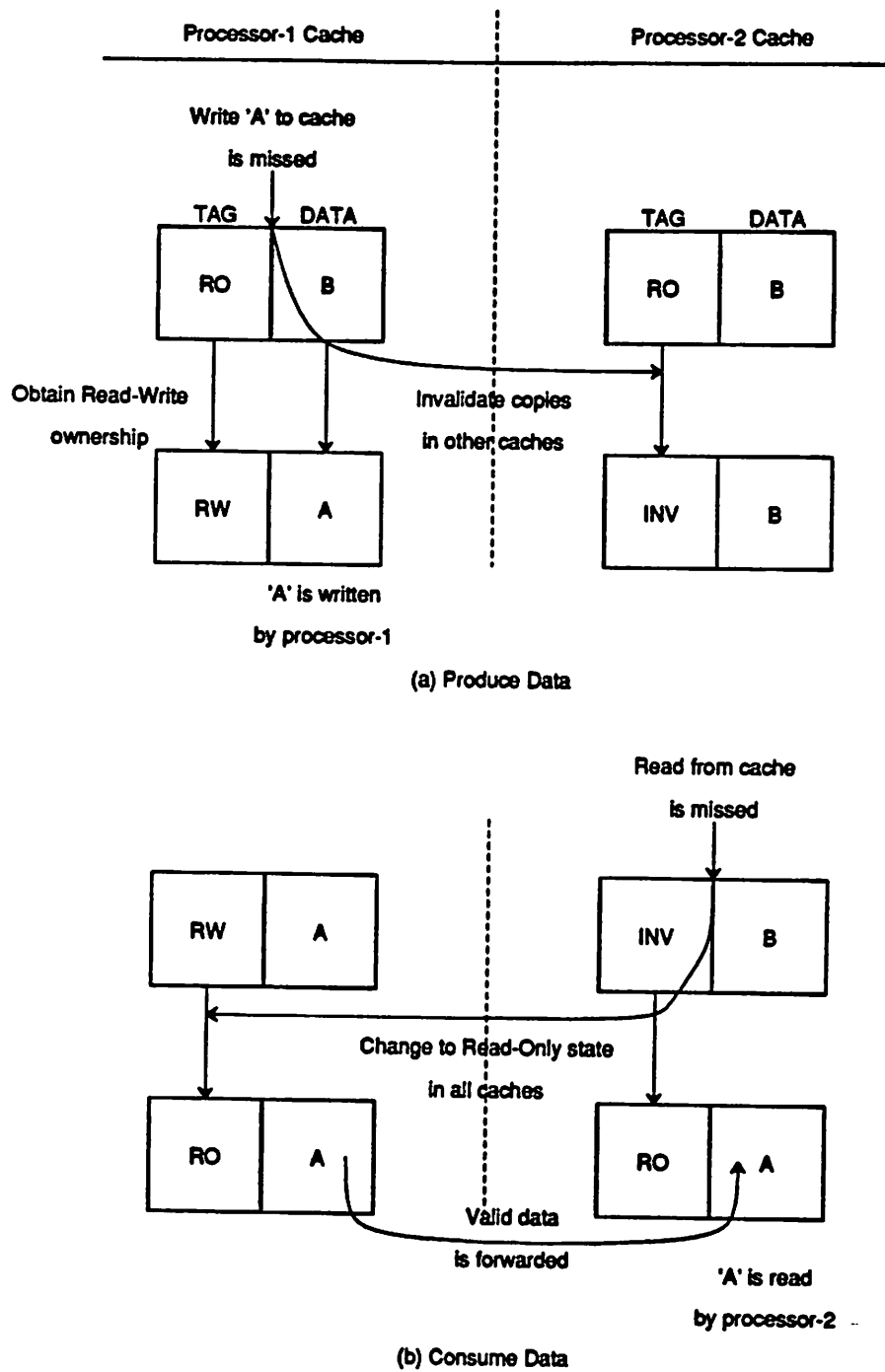


Figure 2.5: Dynamic Coherence Scheme.

processors will be in INV states. <sup>5</sup>

Figure 2.5 shows a simple example of how a producer and a consumer change states of copies during interprocessor communication. In order to modify a block copy, a producer has to wait until it owns the block copy with RW state and invalidates other copies. Then in order to read the modified copy, a consumer has to regain its RO state and wait for the missed data. Therefore both a producer and a consumer have to pay communication overhead.

This dynamic coherence scheme is more flexible than the static coherence method, but also more complex and possibly more costly. When a producer and a consumer alternately writes and reads data the communication overhead is far worse than a simple static coherence scheme.

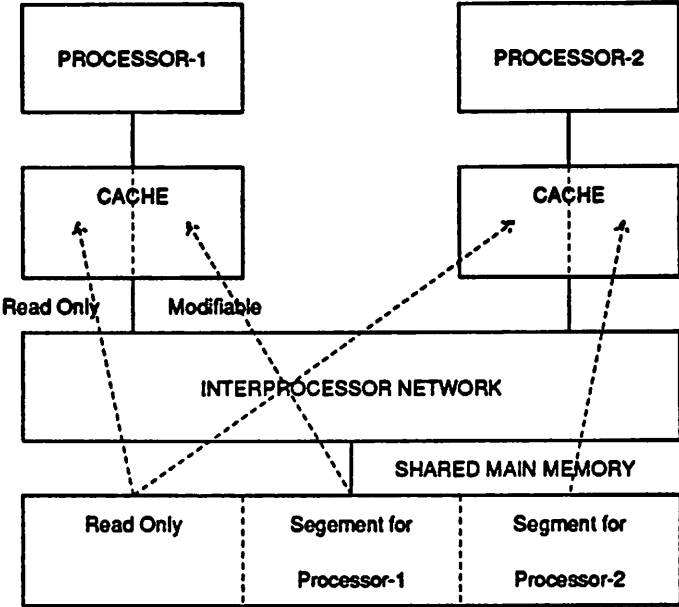
**Our Scheme** — Our cache design is based on a variation of a static coherence scheme. Figure 2.6 shows our shared cache implementation. It illustrates how to achieve cache coherence and low interprocessor communication overhead simultaneously. A read-only segment of data which is shared by several processors is cacheable by all processors. In other words, multiple copies of shared read-only data are allowed to exist in different caches at any time.

On the other hand, a shared data structure which can be modified is cacheable by only one processor. As shown in Figure 2.6 (a), shared data words in address segment-1 can be cacheable by only processor-1, and shared data words in address segment-2 can be cacheable by only processor-2. The caches are directly connected to the associated processors to reduce the access time to the cache. When a processor accesses a shared data in its own address segment, it can access the copy in its own cache – unless there is a cache miss. However, when a processor accesses a data word in other address segment, it will have to access the data in another processor's cache through the interprocessor network.

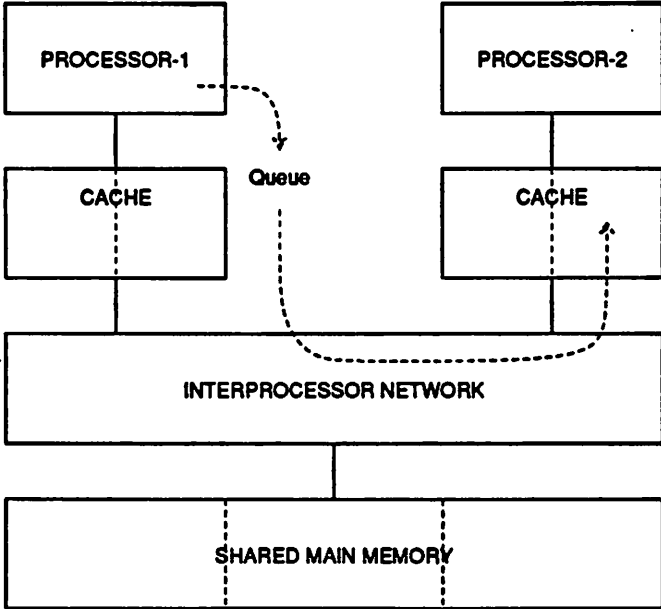
The other processor's caches, even though they are not directly connected, are still accessible by the processor through the interconnection network. The access

---

<sup>5</sup>The cache coherence scheme is a rather complicated process. There are several variations of this scheme. More detailed information on cache coherence is discussed in [40] [35] [37].



(a)



(b)

Figure 2.6: Communication Scheme in SMART Architecture.



times of *write* operations to those caches are as small as accessing its own cache, since the processor does not have to wait for the data to reach the destination; the data will be pushed into the interprocessor communication queue and will be automatically forwarded to its destination as shown in Figure 2.6 (b), regardless of hit or miss on the other side. In the case of a *read* operation to another processor's cache, however, the processor has to wait until the read data comes back; that is, it has to pay communication overhead.

Therefore in order to keep communication overhead small, the producer has to write a shared data word to the consumer's cache and the consumer has to read the data from its own cache without spending time on accessing the interconnection network. Hence both a producer and a consumer pay no communication overhead unless there is a cache miss. When there is more than one consumer, the producer may broadcast the data to all the consumers at once.

## 2.2.2 Implementation of the Distributed Shared Memory

A communication scheme in which the producer *writes* explicitly the shared data word to the consumer's cache memory was proposed in the previous section. This section describes how it is implemented in our system.

Since the SMART system is designed to support real-time applications, it is efficient to schedule processes at compile time and to use the physical address to access the memory. Hence the producer and the consumer know statically where – what physical address – to *write* the data to and where to *read* the data from.

Furthermore, after the address space of AT&T DSP32C processor (24-bit byte address) was partitioned into various functions for 64 processors as in Figure 2.7, the whole memory segment size for each processor (32KB) was small enough to be implemented by cache memory. Therefore, it was not necessary for the main memory to back up the cache miss nor to implement the extra hardware to deal with the miss. However the major ideas of the memory structure remain the same. The prototype system can be regarded as a special case of the proposed cache coherence scheme without any cache miss. In the next generation of the SMART system, since the

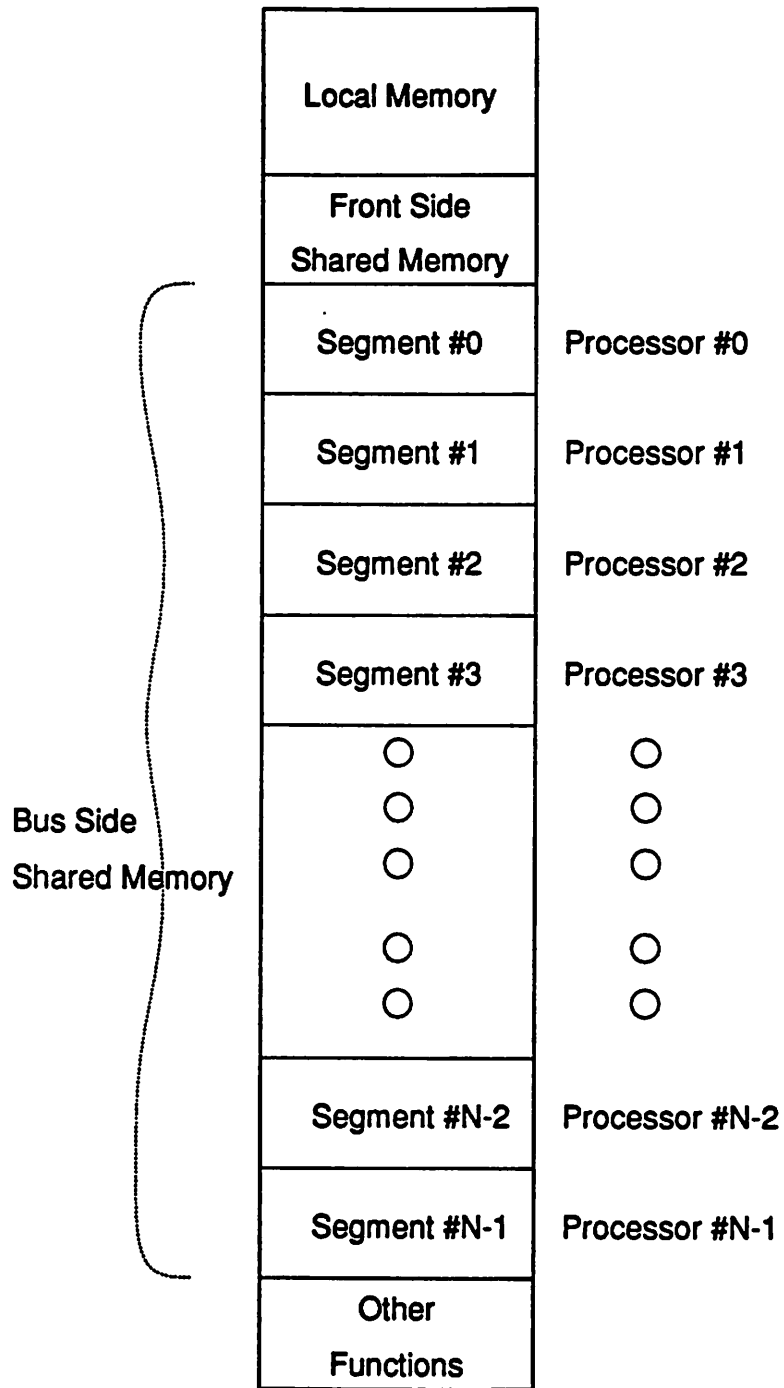


Figure 2.7: Memory Address Segmentation.

processor is expected to have 32-bit address space, it will be necessary to implement the main memory to cover the large segment address space.

In order to implement the above interprocessor communication scheme, the following address map – which is same for every processor – is provided as shown in Figure 2.7. The map consists of address spaces of *local memory*, *front side shared memory*, *bus side shared memory* and address mapped functions.<sup>6</sup>

Figure 2.10 (a) shows the configuration of those memories.<sup>7</sup> Each processor has its own *local memory* to store the private data and program which are not shared among processors. The *front side* address space is used for accessing the shared data from the front side of the cache. The *bus side* shared address space, which is assigned for the communication through the interconnection network, is partitioned at the high address bits into equal memory segments and distributed among processors in proper order [Figure 2.7]. The physical memory of each segment corresponds to the physical memory of the associated processor's front side address space. In other words data, which is written to segment *i*, can be read from the front side of the processor *i*, thereby achieving the interprocessor communication. The name *shared distributed memory* is used for this memory scheme to emphasize the feature that shared memory is segmented and distributed to enable the low interprocessor communication overhead.

Figure 2.10 (a) also shows a *bus master* unit and a *bus slave* unit. Those units enable access to the cache memories of other processors. The bus master unit provides the interface between the processor and the network. It consists of an interprocessor communication queue, a network arbitration block, and extra circuitry. The queue stores the address and the data for interprocessor read and write operations. The network arbitration block decides when to send a data word from the queue to the

---

<sup>6</sup>The address mapped functions are specific to the system. A function is invoked by accessing a predefined memory location. In the SMART system, for instance, an input and an output FIFO memory can be accessed with an address of *0xf40000*. A far more detailed address map is shown in Appendix-A. Chapter 3 shows more detailed informations on the implementation of the actual memory structure.

<sup>7</sup>Figure 2.10 (b) shows the distributed shared memory system which is integrated with a reconfigurable bus. The distributed shared memory system can be interfaced with other networks as well.

network. The extra circuitry is responsible for generating appropriate control signals which are specific to a particular network. The bus slave unit controls access to the bus side of the distributed shared memory. The bus slave unit constantly monitors network traffic and compares the destination processor ID field of the network with the local processor ID. If the result of the comparison matches, a read or write operation to the distributed shared memory is performed.

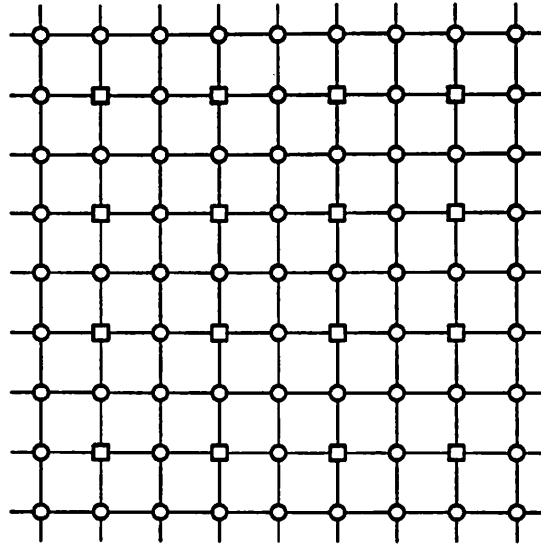
The distributed shared memory may have a main memory structure for a system with a large address space. In the case of a cache miss, due to the access through the shared address space or the front address space, the miss can be handled the same way as for the case of a single processor cache based system.

## 2.3 Reconfigurable Bus

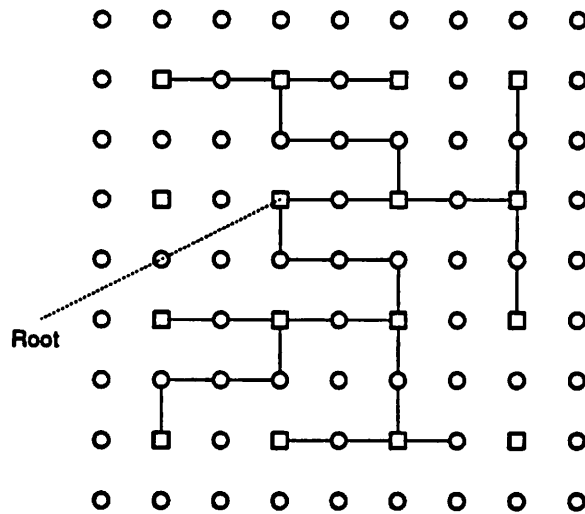
### 2.3.1 Previous Work in Reconfigurable Bus Architectures

The matching of the multiprocessor structure to an algorithm has a fundamental influence on performance and cost effectiveness [26]. For instance, the hexagonally connected mesh is used for L-U decomposition. The binary tree is used for sorting. The double rooted tree is used for searching. Furthermore when an algorithm with an irregular communication pattern is mapped onto a multiprocessor system with a regular and fixed interconnection network, it is very difficult to schedule the algorithm to achieve efficient communications.

To alleviate this problem, reconfigurable interconnection schemes were proposed by several people. Synder has examined this problem in the CHiP architecture [72] [47]. The CHiP computer consists of three components: a collection of homogeneous processing elements (PE), a switch lattice, and a controller. As shown in Figure 2.8 (a) The switch lattice is a regular structure formed from programmable switches connected by data paths. The PEs are not directly connected to each other, but rather are connected at regular intervals to the switch lattice. A configuration setting enables the switch to establish a static, circuit switching connection between two or more of its incident data paths. In Figure 2.8, for instance, a mesh intercon-



(a) The switch lattice



(b) The switch lattice configured into a binary tree

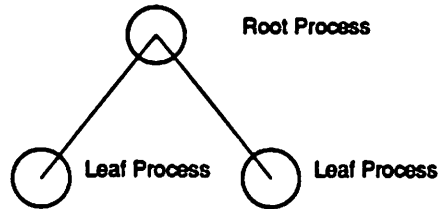
Figure 2.8: The CHiP Architecture.

nection pattern of the PEs are reconfigured to a binary tree connection. In order to host a large number of switch lattice, the CHiP architecture requires wafer-scale integration (WSI).

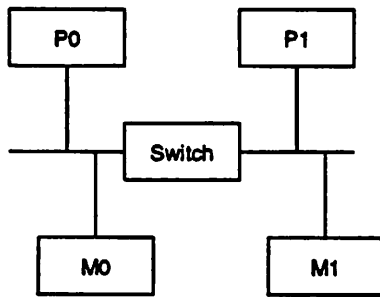
Like the SMART architecture, the interconnection network is reconfigured to enable more efficient use of the hardware. The CHiP architecture is geared to the fine grain parallelism with thousands and millions of processing units. Hence the PEs, the switch lattice, and the controller have to be very simple to host a larger region of the switch lattice in a single chip. On the other hand, the reconfigurable bus of the SMART architecture is developed for the medium and coarse grain parallelism. It has the shared memory architecture and consists of extremely powerful processors (up to one hundred processor range).

The reconfigurable bus for the coarse grain parallelism has been used in the MP/C system [59]. It has the shared memory aspect of the tightly coupled multiprocessor systems and the connection simplicity associated with the loosely-coupled multicomputer systems. A large address space is partitioned into contiguous segments of memory that can be accessed by a single processor. The partitioning is accomplished by switching the buses. To demonstrate the operation of the MP/C, consider the 3-node tree structured multicomputer in Figure 2.9 (a). A typical tree algorithm will first run on the root node, then concurrently on the two leaves, and finally on the root again. Initially, P0 is active, playing the role of the root node, and it can access M0 and M1, as in Figure 2.9 (c). Then P1 is activated and each processor does the computation at a leaf, as in Figure 2.9 (d). Finally, P1 deactivates itself, P0 resumes the root's role and regains control over the whole address space, as in Figure 2.9 (e).

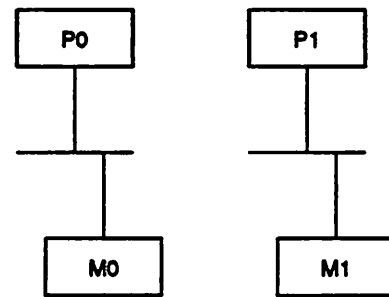
The MP/C architecture has some similarities with the SMART system, since it uses the memory segmentation and the bus switching. However, unlike the SMART system, in order to access shared data, the MP/C architecture must use the interconnection network, making the cost of accessing the shared data quite expensive. Furthermore, when the switch is open, the processors cannot access the shared memory on other buses at all. Therefore the applications are limited, and the programming effort is larger than the SMART system which has the *bypass* ability to bridge the



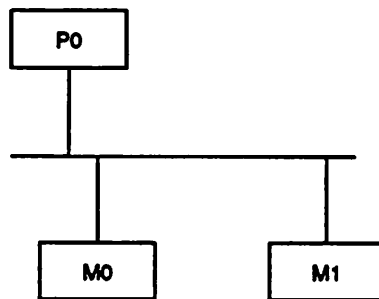
(a) Three-node binary tree



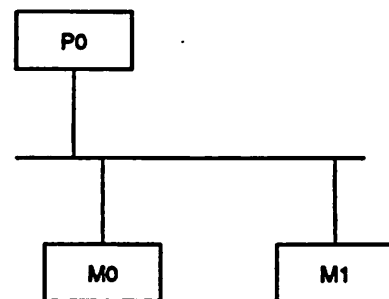
(b) MP/C implementation



(d) Leaves active



(c) Root active



(e) Finally, Root active

Figure 2.9: The MP/C Architecture.

gap between the separate buses.

### 2.3.2 Our Reconfigurable Bus

A high-speed reconfigurable bus for shared memory architecture was proposed to provide high performance and bandwidth at low cost for up to a 100 processor range. It alleviates the problems of the bus saturation typical for a single shared bus shared memory system. The basic function of the reconfigurable bus is to divide the single shared bus into several independent sections to increase usage of the bus. A ring connection is formed by connecting one end of the bus to the other end of the bus to reduce the maximal distance between any two processors in the array.

Figure 2.10 (b) shows an example of a reconfigurable bus. Processor-1 and processor-2 have switches closed and processor-3 and processor-4 have switches opened. Switches can be opened and closed dynamically under software control. When the buses are separated by open switches, they can be accessed simultaneously. For instance, the bus master unit of processor-1 may send a data word to the bus slave unit of processor-2. At the same time, the bus master unit of processor-3 can send a data word to the bus slave unit of processor-4.

When a group of processors is actively communicating with each other, we reduce the delay of communication between them by closing the bus switches in between. When they are not frequently communicating with each other, we increase the system throughput by opening the bus switches between the processors. Thus, we can obtain the efficient bus reconfiguration by trading the delay of global communication for the bus bandwidth. It is possible to dynamically reconfigure the bus under *hardware control*. It is called the self-reconfiguration bus, which is explained in Section 2.3.4.

A bypass unit which forwards the interprocessor communication from one bus to another is an important function of the system. As shown in Figure 2.10 (b), when the buses are disconnected, they are automatically linked by the bypass unit to support global communication. Through bypass units, the data may be sent through several buses automatically to reach the memory in other bus groups. Hence it is



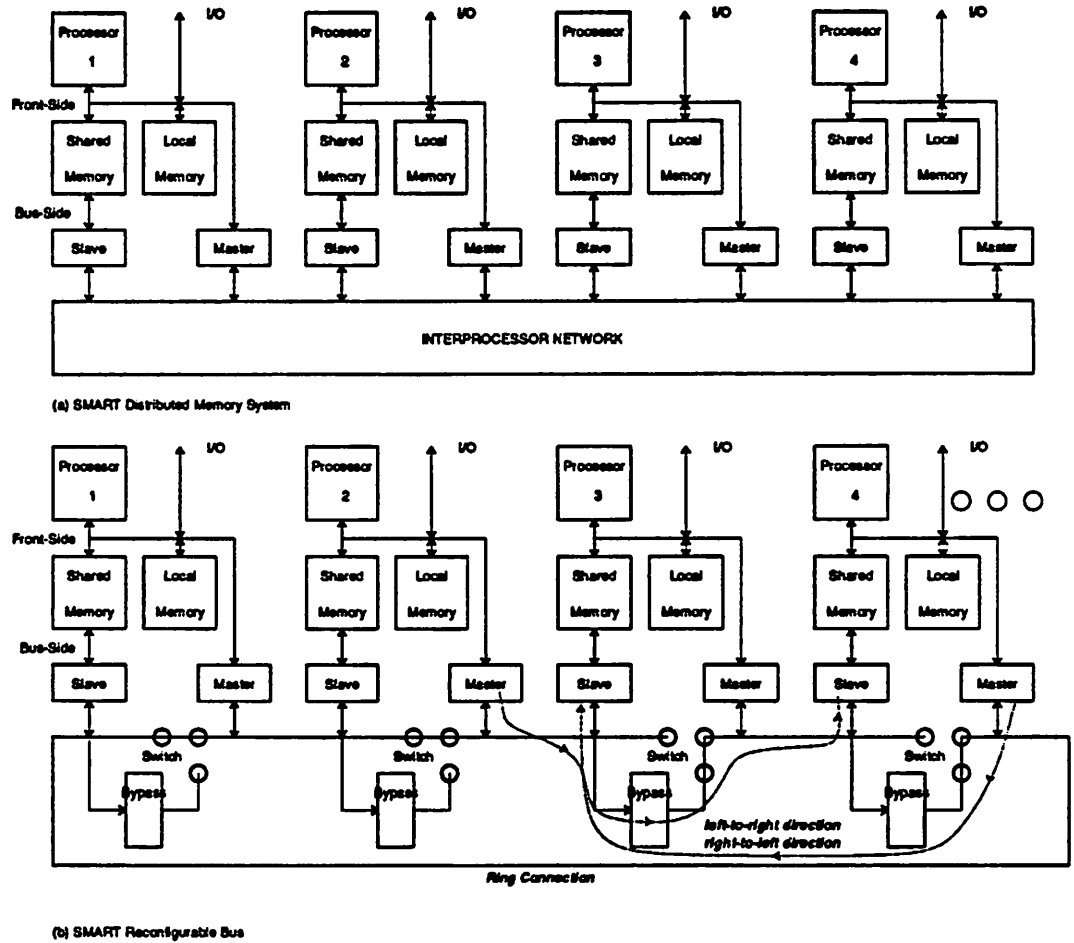


Figure 2.10: SMART Memory and Reconfigurable Bus.

possible to program independently of the bus reconfiguration. The program results can be logically the same, although the performance will be affected. Therefore the reconfigurable bus provides a simple programming environment to a programmer.

The bi-directional bypass and the self-reconfigurable bus features are important features for a large number of processors. The bi-directional bypass (both *left-to-right* and *right-to-left* directions) has the shorter average distance for the communication across the buses than the one-directional bypass (left to right direction) as shown in Figure 2.10 (b). On the other hand, the self-reconfigurable bus has switches dynamically reconfigured at every bus cycle. This allows the dynamic behavior of the bus which is hard to be achieved under *software control*. However, for a small number of processors, the performance gain is too small compared to the complexity of implementation. Those features were not implemented for the first prototype SMART system with 10 processors. Instead, the one-directional bypass and the software controlled reconfigurable bus were implemented.

### 2.3.3 Interprocessor Communication

Section 2.2, *Memory Structure*, has presented the shared distributed memory to achieve small interprocessor communication overhead. The previous section has described how the reconfigurable bus improves the performance of the single shared bus through reconfiguration, bypass and ring connection. This section shows a combined view of the shared distributed memory and the reconfigurable bus using a simple interprocessor communication example.

The Figure 2.11 shows how interprocessor communications (A-F) between the processor-1,2,3,4 are handled by the hardware in a pipelined fashion when the bus is configured as in Figure 2.10 (b). Recall that each processor has a distributed shared memory, a bus master unit, a bus slave unit and a bypass unit. Each box in Figure 2.11 represents a pipeline stage with horizontal axis showing the time unit (100ns) and the vertical axis representing the bus transactions. For example Master-2 represents a master cycle by processor-2 and Arb-3 represents an arbitration cycle by processor-3.

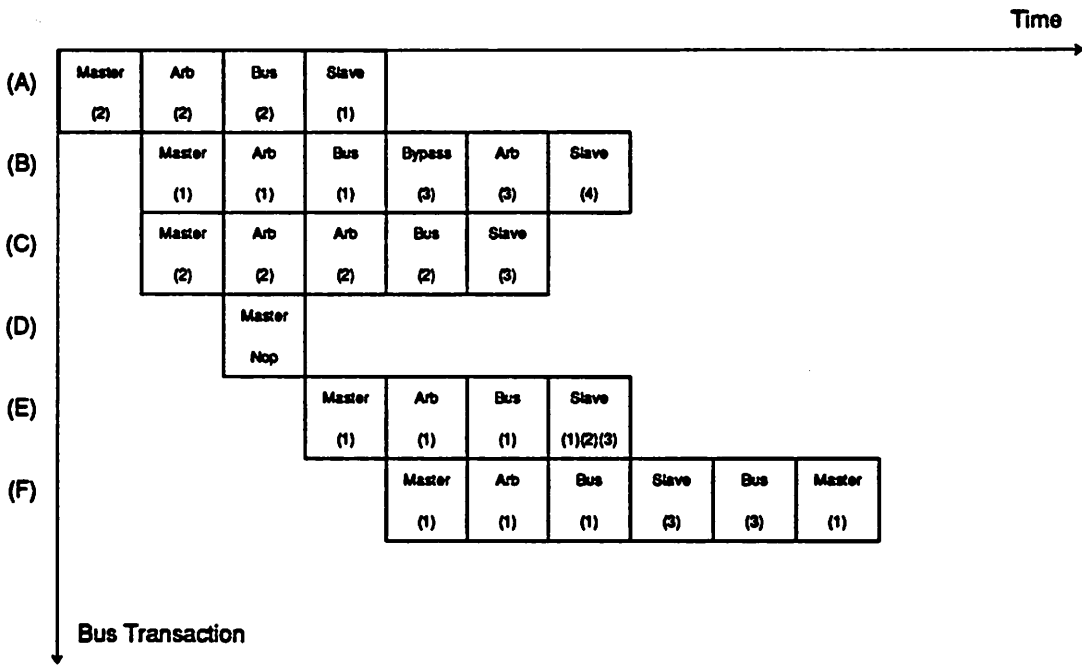


Figure 2.11: Bus Transaction for Interprocessor Communication.

Bus transaction-(A) is the write operation from processor-2 to processor-1's distributed shared memory. The communication will be later accomplished by the read operation by processor-1 from the Front Side Memory port. The first pipeline stage of the bus transaction-(A) is the *master cycle*, where the bus master unit of processor-2 receives the write request from the processor-2. At the second pipeline stage (*arbitration cycle*), the arbitration cycle selects the bus master unit or the bypass unit within the bus domain which has an access to the bus in the following cycle. A simple statically prioritized arbitration algorithm is implemented in hardware: a bypass unit has the higher priority than a master unit, and processors with a low identification number(ID) have higher priority than processors with a high ID. During the following *bus cycle*, data, address and control are placed on the reconfigurable bus by either the bus master unit or the bypass unit of processor-2. Then the bus slave unit of processor-1, which is always snooping the bus, finds out that the current bus transaction belongs to its own distributed shared memory and writes data to the distributed shared memory during the *slave cycle*. Sometime later when processor-1 needs the data sent by processor-2, it simply reads the data from the front side memory port without accessing the shared bus.

Bus transaction-(B) is another write operation from processor-1 to processor-4. However, since the switch of processor-3 is open, there is no direct path from processor-1 to processor-4. During the bus cycle, the processor-3's bypass unit, which is activated as a result of an opened switch, detects that the current transaction to processor-4 should be forwarded to the adjacent bus. Therefore the bypass unit latches the contents of the bus into the *bypass queue*, generates an arbitration request, and place the copy on the adjacent bus. The slave unit of processor-4 will update the distributed memory accordingly. If there are more than one bypass unit involved in the transaction, each bypass unit will take two extra cycles to forward the data to the next bus.

When the bus transaction-(B) was generated by processor-1, bus transaction-(C) was simultaneously initiated by processor-2. Since the bus couldn't accommodate two data at the same time, the transaction-(B) was granted the first arbitration, and the transaction-(C) was suspended for one pipeline stage. Then it tried for another

bus arbitration and won the following arbitration. After the arbitration cycle, the bus cycle and the slave cycle executes the same pipeline stages as transaction-(A).

Bus transaction-(D) is a NOP (No Operation), which means no bus master or bypass unit wants to use the bus at that time.

Bus transaction-(E) represents bus read operation from processor-1 to processor-3's distributed shared memory. The bus master cycle and arbitration cycle operates the same way as in the case of write operation. However the bus cycle consists of the following three sub-pipeline stages: At the first stage, the address and control signals are placed on the bus, which will be latched by the slave unit-3. During the following stage, it will read the data from its bus side memory port. At the last stage the data will be placed on the bus by slave unit-3 and latched by bus master unit-1. Then the data will be forwarded to processor-1, which has been waiting for the arrival of the data.

The bus transaction-(F) is a broadcast from processor-1 to processor-1, 2, 3. Since it's a broadcast, all the bus slave unit will update its memory. The broadcast can be bypassed through switches as the same way as the bypass write operation.

All the functional blocks of reconfigurable bus such as the bus master unit, the bus slave unit, the bypass unit, the switch, the arbitration and other functions are implemented as 2 VLSI chips: Master Access Controller (MAC) and Slave Access Controller (SAC). The details of those chips are described in Section 3.4.

### 2.3.4 Self-Reconfigurable Bus

The reconfigurable bus presented in previous sections was reconfigured under *software control*. Changes in the switch configuration had to be specified in the program. Sometimes it is not clear what the optimal bus reconfiguration is for a given algorithm. The optimal reconfiguration may be hard to estimate or even change frequently during the different phases of the algorithm. Therefore the architecture of the self-reconfigurable bus is developed to dynamically and automatically reconfigure the bus under *hardware control*.

The self-reconfigurable bus has a couple of major advantages over the soft-

ware reconfigurable bus. First of all, the programmer or the automatic scheduler does not have to know anything about the bus switches and their reconfigurations. Therefore it reduces some of the programming effort. Second, switches are dynamically reconfigured at every bus cycle. Therefore the network can be reconfigured to maximally utilize the bus resources.

The bus transaction consists of four cycles, as in the reconfigurable bus: a master cycle, an arbitration cycle, a bus cycle, and a slave cycle. The arbitration cycle is the main source of the difference. The arbitration is not used to decide which processor will be granted access to the bus as in the reconfigurable bus, but rather is used to find out which processor will be given which portion of the bus in the following cycle. In other words, switch reconfiguration is determined simultaneously with the arbitration. When a data word needs to be transferred through the bus switch in one bus cycle the switch will be closed, otherwise the switch will be opened.

Figure 2.12 shows three examples of self-reconfigured bus at bus cycle  $(i), (i+1), (i+2)$ . For example, at bus cycle  $(i+2)$ , processor-3 transfers data to processor-1 and processor-4 transfers data to processor-6, and switches are reconfigured accordingly.

When multiple processors concurrently request the usage of the bus, the bus is reconfigured to accommodate the maximal number of processor requests and to send the data as near as possible to the destination, as in Figure 2.12. During the arbitration cycle- $(i)$ , a bus request is made by processor-1 to send data-“a” to processor-6 which conflicts with other processors bus request. The arbitration result- $(i)$  shows that each processor is fully granted access to the bus except processor-1 which is granted access to the bus as far as data-“a” could travel without the conflict, between processor-1 and processor-2. The bypass unit in processor-2, activated when the bus switch is opened, generates a bus request at the next arbitration cycle- $(i+1)$  and propagates data-“a” to its destination at bus cycle- $(i+1)$  as if the data was generated by its own processor. The bypass operation on data-“a” is repeated once again by processor-4 until the data reaches its final destination at bus cycle- $(i+2)$ .

The above example describes how a bidirectional bypass unit, with a combined dynamic bus reconfiguration scheme, transfers data to its destination regardless

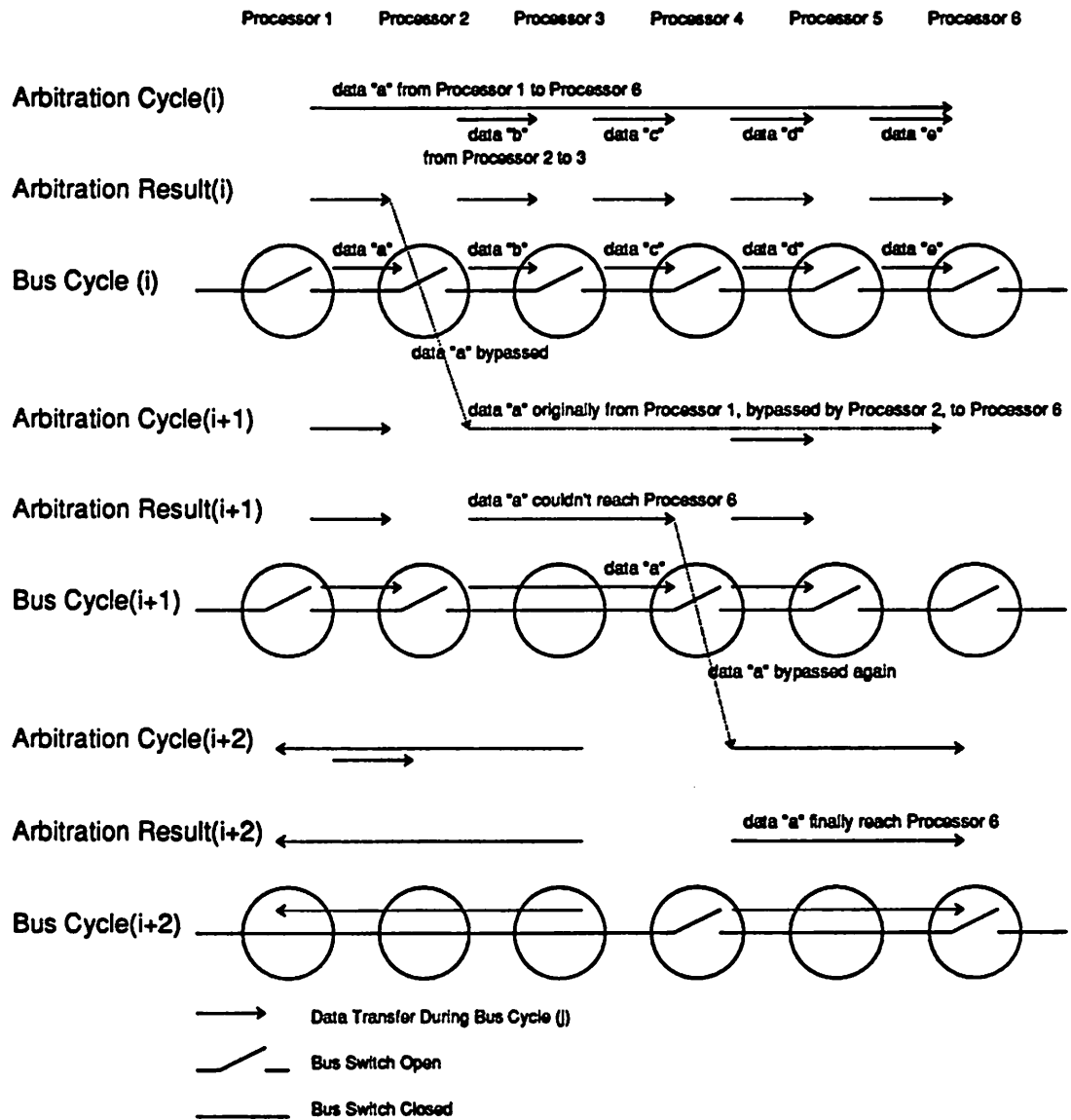


Figure 2.12: Examples of Self Reconfigurable Bus.

of switch reconfiguration and how the self-reconfigurable bus reconfigures switches at every bus cycle. A programmer does not need to partition the bus and is able to treat the interconnection network as if it is a single shared bus which supports tightly coupled shared memory.

## 2.4 Synchronization

In addition to the efficient communication provided by the distributed shared memory and the reconfigurable bus, issues like process synchronization [16], data coherence, and event ordering are important factors in the performance of a multiprocessor system.

The synchronization mechanism in the SMART system can be categorized as the following three types: barriers, locks and events. When a group of processes must be synchronized at a certain point in the program, the *barrier* is used by preventing any process from going further until every process in the group reaches the synchronization point. On the other hand, when several processes try to access a shared resource such as the shared memory, the *lock* which is a simple type of semaphore guarantees that only one process will be granted access while other processes *busy wait* until the resource is unlocked. The event is something that must happen before a process can proceed. The *event* is accomplished by *post* and *clear* operations between processes through distributed shared memory to notify the occurrence of a predefined event.

The barrier and the event are useful to ensure the order of execution, thereby preserving the data dependency between processes. For instance, the producer writes a set of data to the consumer's distributed shared memory which the consumer needs in order to execute the process. To maintain data coherence, the producer issues a barrier synchronization after writing data and the consumer issues a barrier synchronization before reading the data. After the barrier, while the consumer is accessing the data, the producer will create a whole new set of data on the other part of the memory to execute the next process, thereby any memory contention between producer and consumer is avoided. When a block of data is produced and consumed,



the barrier synchronization is issued only once for the whole block. In most of the applications we confronted, the barrier synchronization seemed to be very useful. Therefore, the SMART architecture provides special hardware support for the barrier synchronization; it takes only two instruction cycles to check if all the processors are synchronized. It is also possible to synchronize selectively for a local group of processors.

The event can be also used to ensure the data coherence by posting the validity of the data from the producer to the consumer, and by posting the acknowledgement of receiving the data from the consumer to the producer. Using events has the advantage that consumer and the producer do not have to be at the synchronization point at the same time. However, the event involves extra overhead due to handshake and software operations for posting, checking and clearing.

The following sections discuss the details of the three synchronization mechanisms. Chapter 5 shows how those synchronizations are used in real application programs.

### 2.4.1 Barriers

A barrier is a synchronization point which involves the following operations:

1. Mark the current process as present at the barrier point.
2. Wait for all the other processes in the synchronization group to arrive.
3. Proceed when all the processes are present at the barrier.

The barrier synchronization is often issued to ensure the validity of the data. The barrier synchronization is satisfied when all the processors in the barrier group have reached the synchronization point and all the data transactions in the waiting queue are transmitted. The application of barrier can also be interpreted as keeping the precedence relations between actors or processes running on different processors. It is especially useful when the computation time is data or time dependent, or very hard to estimate exactly.

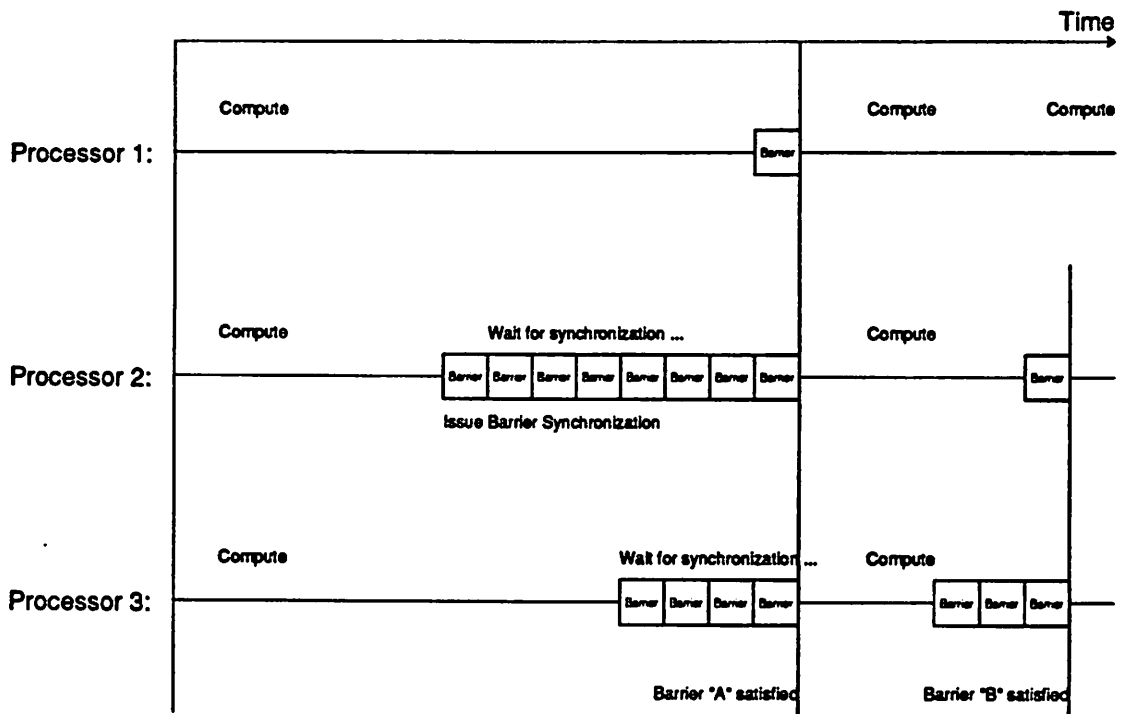


Figure 2.13: Examples of Barrier Synchronization.

Examples of Barriers are shown in Figure 2.13. *Barrier-A* point is used to synchronize processor 1, 2 and 3. Note that processor 2 and 3 are waiting for processor 1 to arrive at the barrier point, and about 2 bus cycles (200ns) after processor 1 reaches the barrier point, all the processors proceed to the next steps. *Barrier-B* point is to selectively synchronize only processor 1 and 2. There are five hardware barrier synchronizations which can be programmed selectively to satisfy the different types of barrier synchronization. In case the application needs more than those five types for each processor, the events can be used, sacrificing speed however.

## 2.4.2 Locks

A lock ensures that only one process at a time can access a shared data structure. A lock has two states: locked and unlocked. Before attempting an access to a shared data structure, a process waits until the lock associated with the data structure is unlocked, indicating that no other process is accessing the data structure. The process then sets the lock, accesses the shared data structure and frees the lock. While a process is waiting for a lock to become unlocked, it *spin-locks* in a tight loop, producing no work.

The locking mechanism provided on SMART systems perform the actions required to establish a lock as a fast single indivisible operation. The locking operation takes about 200ns (if it is not locked), and the unlocking operation takes about 100ns. Figure 2.14 shows an example of lock operations which involves three processors.

The hardware implementation of the lock is fast but primitive. Higher level semaphores can be built on top of the lock to give more versatility. Waiting mechanisms other than spin-lock can be also implemented in software on top of the hardware lock.

## 2.4.3 Events

An event is something that must happen before a task or process can proceed. The completion of a task, the arrival of needed data and the occurrence of certain conditions to be noted are examples of the events. An event is accomplished

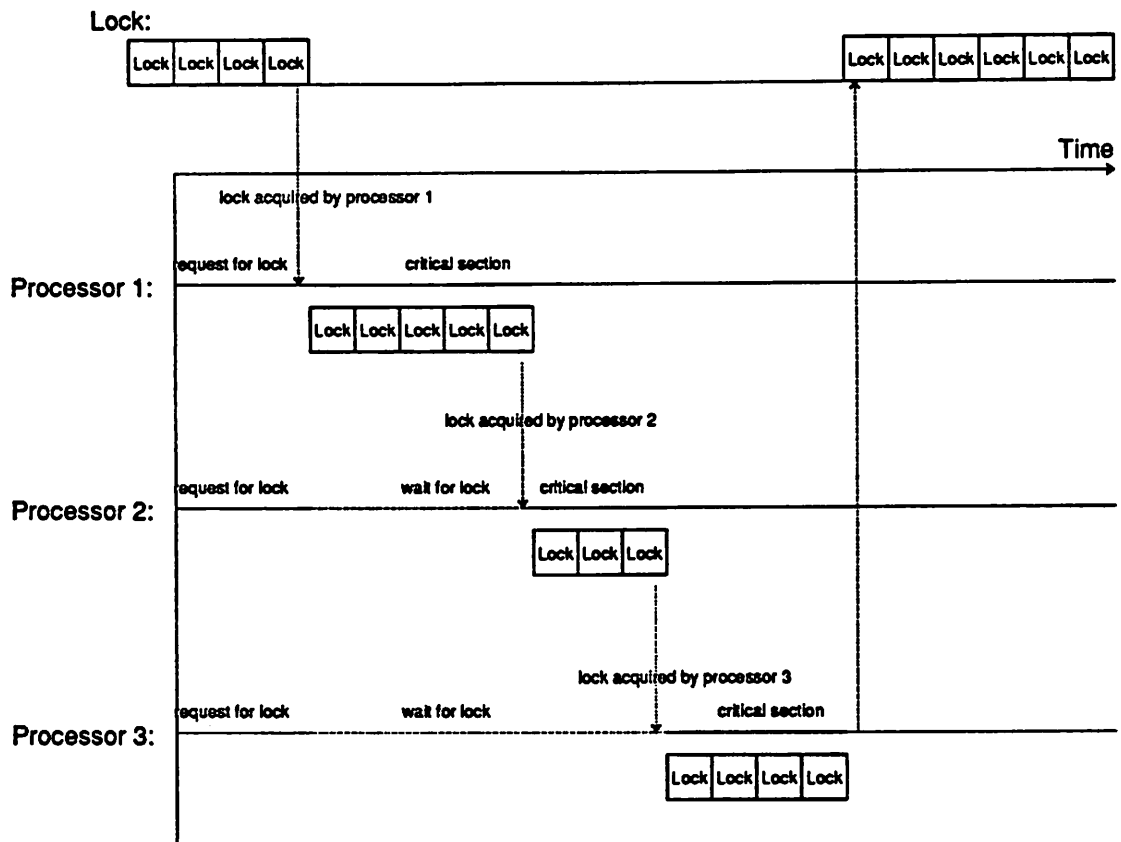


Figure 2.14: Examples of Lock Synchronization.

by handshaking between the source processor (which reports the event) and the destination processor (which accepts the event) in the following steps [Figure 2.15]:

1. The source processor checks the acknowledge flag queue to prevent the new event from overwriting the last events which are not served yet.
2. The request flag is posted on the destination processor event queue by the source processor.
3. The destination processor checks the event queue and serves the event.
4. The acknowledge flag is posted to the source processor queue by the destination processor.

No special hardware feature is required to implement the event. A small portion of the distributed memory can be allocated for the request and the acknowledge event flags. The request flag is placed in the destination processor's memory and the acknowledge flag is placed in the source processor's memory. Therefore the flags are posted through the bus side memory and the flags are checked and cleared through the front side memory. When an event flag is set, there are two different ways to initiate the checking. One way is to let the destination processor poll the flag periodically. The other way is to let the source processor, which has posted the event, initiate the checking in the destination processor. This operation involves the interrupt operation from the source to the destination processor. The latter method has the advantage in that an important event can be served right away. On the other hand, serving the interrupt in the middle of executing another program can be quite expensive.

## 2.5 Benchmark

We have studied typical DSP algorithms extensively to determine the effectiveness of the proposed architectures. The results of four DSP algorithms (256-point

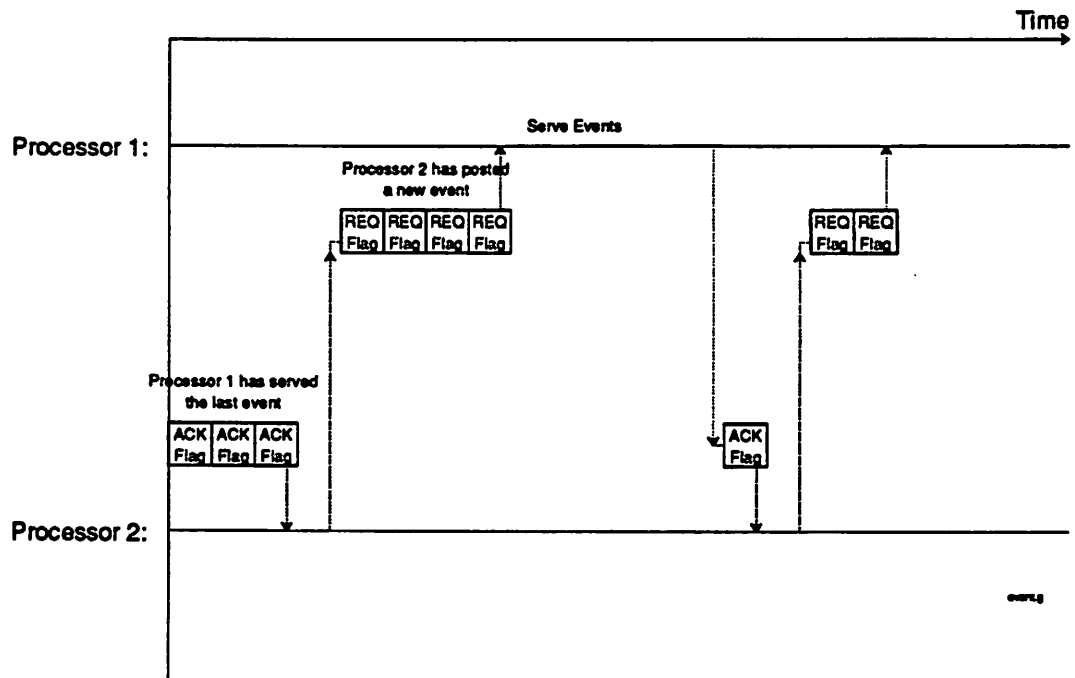


Figure 2.15: Examples of Event Synchronization.

	256-pts FFT	Echo Cancellor	Pitch Extr.	Matrix-Vector Mult.
Speedup	13.65	14.04	13.29	14.58
Communication Overhead (%)	0.64	0.13	0.03	1.11
Idle Time (%)	8.91	18.52	16.12	1.80
# of Buses	16	3	4	1
Average Bus Usage (%)	32.67	0.65	4.05	14.33
Average # of Bus Requests	1.00	1.59	1.50	8.62

Table 2.1: Benchmark Results for 16 Processors.

	256-pts FFT	Echo Cancellor	Pitch Extr.	Matrix-Vector Mult.
Speedup	39.13	41.80	53.89	34.95
Communication Overhead (%)	20.39	0.73	1.11	15.45
Idle Time (%)	16.75	28.81	13.68	17.19
# of Buses	64	10	12	1
Average Bus Usage (%)	72.7	2.96	30.90	34.17
Average # of Bus Requests	1.00	2.20	2.57	32.74

Table 2.2: Benchmark Results for 64 Processors.

FFT, Echo Cancellor [20], Pitch Extractor [24], and 64x64 Matrix-Vector Multiplication) are shown in Table 2.1 and Table 2.2. More details of the benchmark analysis can be found in [38]. The performance on the real SMART system hardware is presented in Section 5.4.

The four different algorithms were chosen to represent two major aspects of concurrent programming in the SMART architecture. The first aspect is the degree of communication. The algorithms were chosen to represent communication intensive algorithms (FFT), computation intensive algorithms (Echo Cancellation), and in between (Pitch Extraction and Matrix-Vector Multiplication) so that we can understand how the architecture performs for different types and degrees of communication. The second aspect is the type of concurrency exploited: pipelining (FFT), parallelism (Matrix-Vector Multiplication) or both (FFT, Echo Cancellation and Pitch Extraction). However, caution is required in analyzing the results using the above classification, since the degree of communication and the type of concurrency varies depending on the granularity, mapping, programming, problem size and other factors.

The benchmark results show close to ideal speedup and very low communication overhead in all cases. *Speedup* is defined as the time taken to execute the sequential algorithm for a problem on a single processor divided by the time taken to execute the parallel algorithm for the same problem on a multiple processor. *Communication overhead* gives the extra amount of time used in communication compared to an ideal multiprocessor machine in which there is no contention or penalty in accessing any portions of the global shared memory. The low communication overhead in the test benchmark result confirms the ability of SMART to adapt to different communication patterns. *Idle time* is a measure of how well the load is partitioned among processors and has a direct effect on speedup. For each bus, the *bus usage*, which is defined as the fraction of the observation period in which the bus was used, is measured to represent the degree of interprocessor communication through the bus. The *average bus usage %* is an average of the *bus use %* over all the buses.

The benchmark programs simulated on the 16-processor system showed speedup from 13.65 to 14.58 over the single processor, as summarized in Table 2.1. During the observation period, processors have paid less than 2% of the time for



interprocessor communication overhead and much less than 1% of the time for synchronization overhead. <sup>8</sup> However, idle time of the processors was from 1.80% to 18.52%; it was especially high for the irregular programs, such as Echo Cancellor and Pitch Extractor, since load balancing is very hard to achieve for those cases.

The bus was reconfigured for each algorithm, from 1 bus to 16 buses, to obtain the best results; the FFT program, when reconfigured differently as a single-bus, showed 2.40 times of performance degradation due to the shortage of bus bandwidth.

When many processors request the bus at the same time, the communication can be temporarily congested, which can cause a bus saturation for a short period of time. The degree of congestion is measured by the number of processors requesting (arbitrating) the bus when at least one is requesting. The Matrix-Vector Multiplication benchmark shows that an average of 8.62 processors, among 16 processors, requested the bus at the same time, although the bus is used only 14.33% of the time.

The same set of benchmarks, simulated on 64 processors, showed a speedup from 34.95 to 53.89. The reconfigurable bus becomes more and more important, as more processors are used, as the programs become more irregular, and as the program becomes more communication bounded. The reconfigured bus, in many cases, has performed about a factor of 2 to 3 times better than fixed bus interconnections (a single-shared-bus and a pipelined bus).

Although the SMART architecture has successfully reduced the overhead of communication and synchronization to a low value, the idle time accounts for most of the performance degradation. More speedup improvements can be expected (by 10% - 20%) with more careful load balancing of application programs.

---

<sup>8</sup>Since the synchronization overhead is negligible, the numbers are not included in the table.

## Chapter 3

# SMART ARCHITECTURE IMPLEMENTATION

*The Eagle soars in the summit of Heaven,  
The Hunter with his dogs pursues his circuit.  
O perpetual revolution of configured stars,  
O perpetual recurrence of determined seasons,  
O world of spring and autumn, birth and dying!  
The endless cycle of idea and action,  
Endless invention, endless experiment,  
Brings knowledge of motion, but not of stillness;  
Knowledge of speech, but not of silence;  
Knowledge of words, and ignorance of the Word.  
- T. S. Elliot*

### 3.1 SMART System

The major components of the SMART system are the processor array, the Heurikon CPU Board [29] using the VxWorks real-time operating system [78], the Analog/Digital Unit, the Interface Unit, and the *host system*, as depicted in Figure 3.1. Figure 3.2 shows a view of the SMART system. A single 20-inch VME rack hosts the Heurikon CPU board, a SMART array of 10 cells, as well as associated power supplies and fans. All the components except the host station are housed inside a VME card cage.

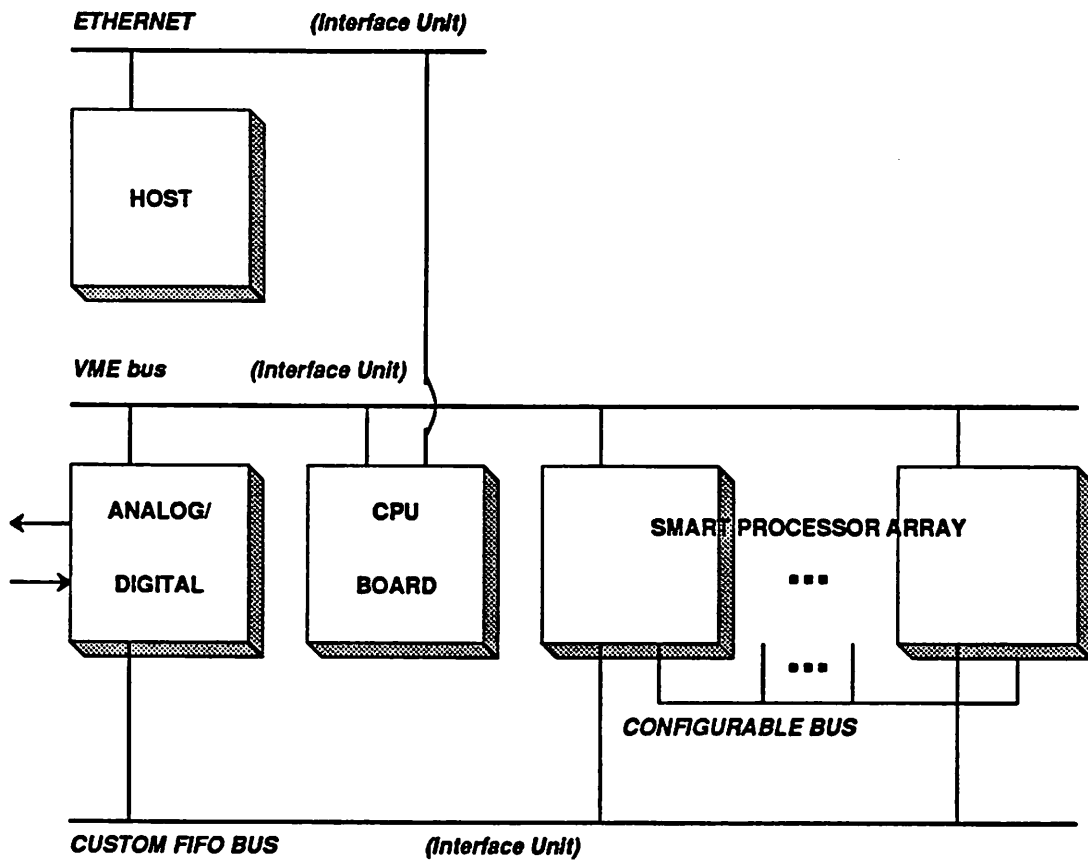


Figure 3.1: Block Diagram of the SMART System.

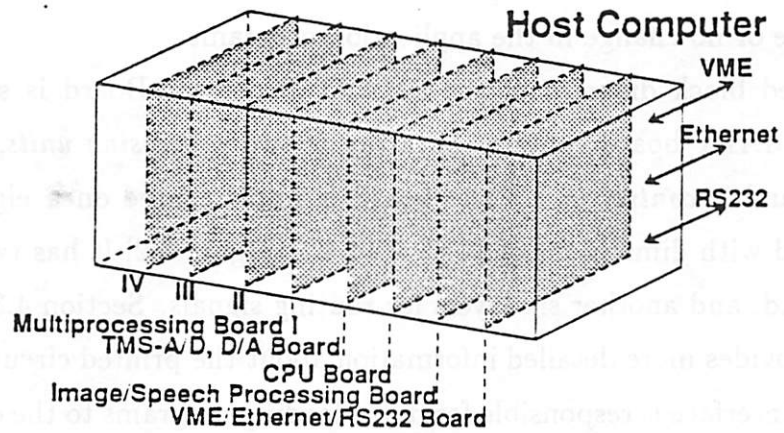
Computation is performed by the SMART MIMD processor array which consists of a set of programmable core processors (AT&T's DSP32C) with 32-bit floating-point capability (peak 20 MFLOPS) connected to each other via the reconfigurable shared bus Figure 3.2. The array is controlled by the *CPU Board*, a micro-computer running a real-time Unix-like Operating System. Data is supplied to and received from the processor array in real-time through the *Analog/Digital Unit*. The *Interface Unit*, consisting of Ethernet, VME bus, and RS232, provides efficient communication and cooperation between Units. Communication between the host computer and the CPU board is established by the Ethernet which also allows the CPU board to access data files in the file server of the host system directly. In addition to the VME bus, an optional custom bus can be implemented to speed up communication between the SMART system and the outside world.

The *host* is a SUN workstation system that functions both as a software development station and as a large auxiliary data storage unit. The workstation provides a UNIX environment for cross-compiling and developing application programs.

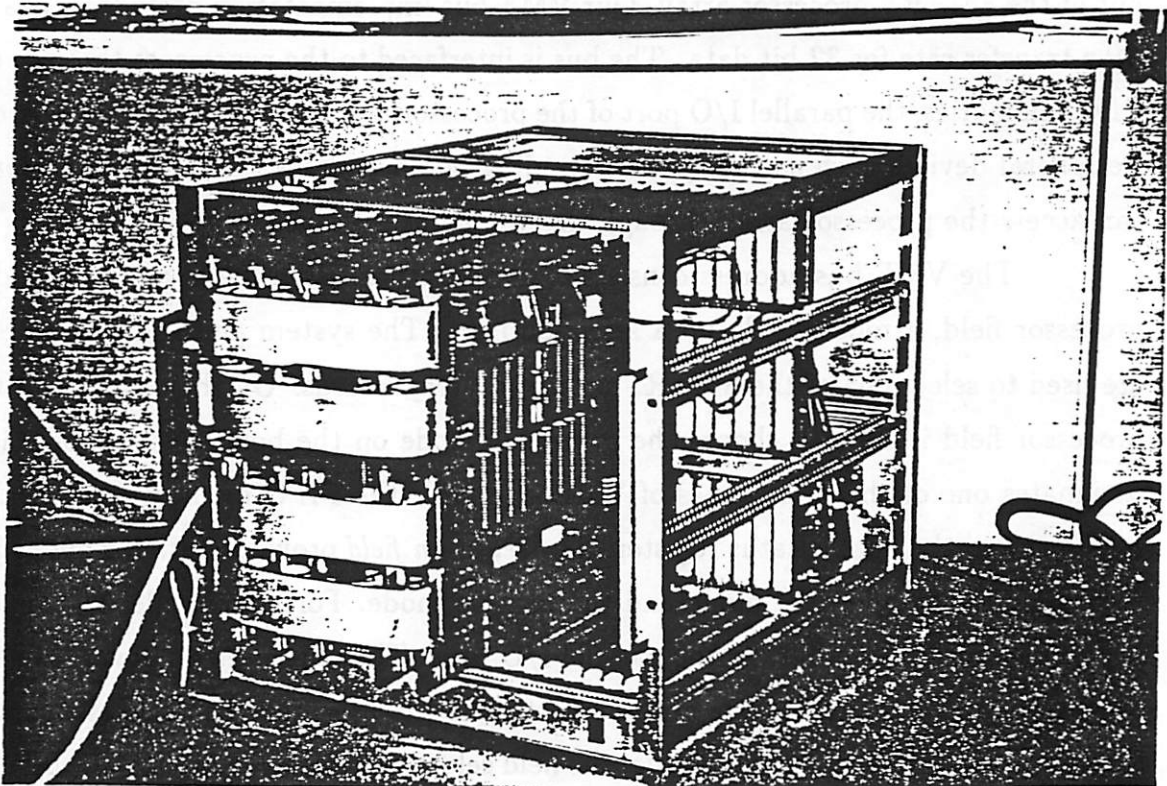
The first prototype of the 10 processor SMART system is operating reliably at 24 Mhz system clock with peak 120 MFLOPS performance. We are currently working on improving the system speed to a 40 Mhz system clock, which would yield 200 MFLOPS of peak performance.

All the main features of the architecture, such as the reconfigurable bus, the bus master unit, the bus slave unit, the distributed shared memory system, and the synchronization mechanism, are implemented as semi-custom VLSI chips. One set is required per processing node.

The following sections describe the implementation details of the SMART processor array and then focuses on a single processing node which consists of a DSP32C processor, Master and Slave Access Controllers, and memory modules. Chapter 5 presents the programming methods to utilize the described system.



(a) SMART Processor Array in VME Card Cage



(b) Photo of SMART Processor Array

Figure 3.2: SMART Processor Array.

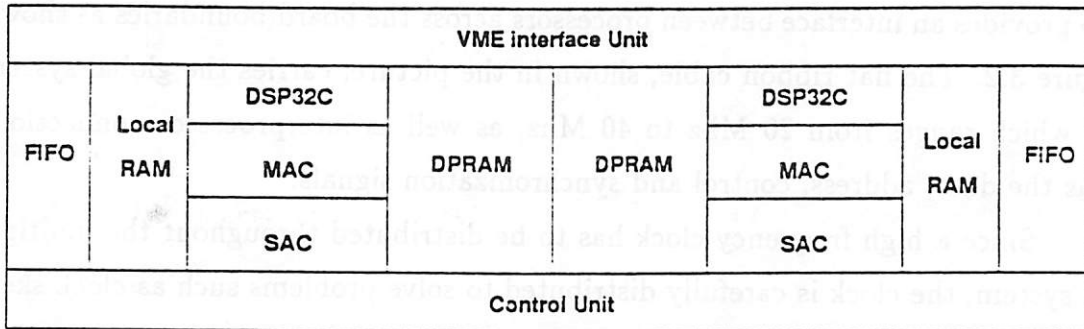
## 3.2 SMART Processor Array

The 10 processor SMART prototype consists of five printed circuit boards. All boards are *functionally* equivalent: they can be easily added, removed, or interchanged with a little or no change in the application programs.

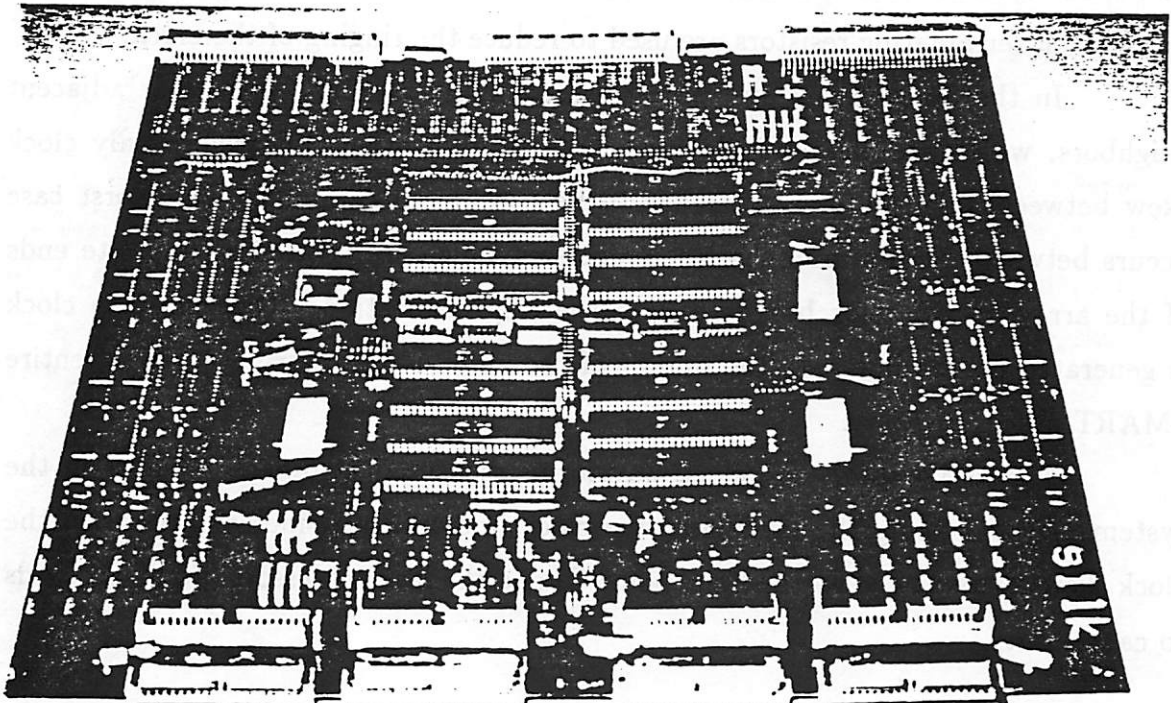
A simplified block diagram of the SMART Processor Board is shown in Figure 3.3. Each SMART board contains an array of two *processing units*, a *VME interface unit* [57] and a *control unit*. The board is implemented on a eight layer printed circuit board with dimensions of 14.6"  $\times$  14.4". [Figure 3.2] It has two layers for power and ground, and another six layers for routing signals. Section 4.3 (Board Implementation) provides more detailed information about the printed circuit board.

The VME interface is responsible for down-loading programs to the core processors, monitoring the internal status of the processors and transferring data in and out of the SMART processor array. Our VME bus implementation provides a peak 6 Mhz transfer rate for 32-bit data. The bus is interfaced to the processors through the FIFO memories, the parallel I/O port of the processor and the input/output registers. Peripheral devices such as the speech sampling board and the Heurikon CPU board can access the processor array through the VME bus interface.

The VME bus address consists of five fields: a system field, a board field, a processor field, a mode field and a selection field. The system and the board fields are used to select one of the SMART processor array boards. On the other hand the processor field is used to choose the processing node on the board. The mode field designates one of the three types of VME interface: the FIFO memory, the parallel I/O port and the board status register. The *selection field* provides detailed operation regarding the parallel I/O mode and the register mode. For the parallel I/O mode, it provides the address bits of the parallel I/O port. They are used to down-load the program and data, inspect the status registers of the processor and access memory through DMA. For the register mode, the field selects between the input, output, and *scan* registers. The input register contains the status of the processor array such as full and empty bits of the FIFO memory and the processor interrupt acknowledge signals. The output register provides signals to reset the board, interrupt the processor, and



(a) SMART Processor Printed Circuit Board



(b) Photo of SMART Processor Printed Circuit Board

Figure 3.3: SMART Processor Board.

enable the scan mode for testing. The scan register is used to scan the test vectors in and out of four VLSI custom chips on each board.

The control unit, which resides on the board side opposite to the VME interface, generates, distributes and buffers the global clock and system control signals. It also provides an interface between processors across the board boundaries as shown in Figure 3.2. The flat ribbon cable, shown in the picture, carries the global system clock, which ranges from 20 Mhz to 40 Mhz, as well as interprocessor connections such as the data, address, control and synchronization signals.

Since a high frequency clock has to be distributed throughout the multiple board system, the clock is carefully distributed to solve problems such as clock skew and transmission line effects. To shorten the buffered wire length and balance clock skews, the system clock is buffered locally on each board. To alleviate transmission line effects, terminating resistors are used to reduce the ringing of the clock.

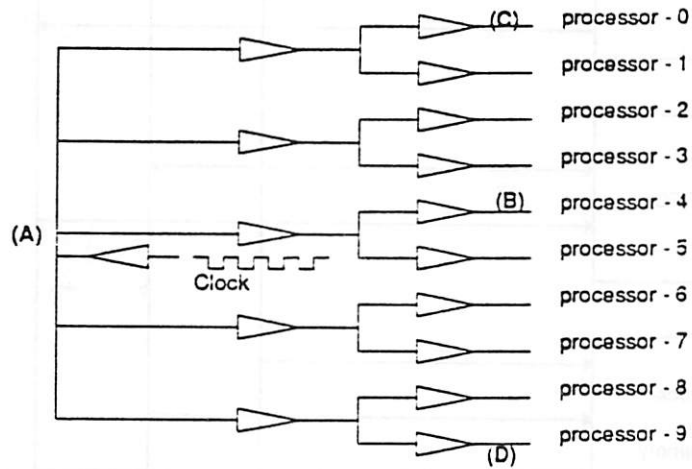
In the SMART system, processors communicate only with their adjacent neighbors, which are physically located close to each other. Therefore only clock skew between adjacent processors has to be considered carefully. The worst case occurs between the first and the last processors which are located at opposite ends of the array of processor boards due to the ring connection. Therefore the clock is generated on the board in the middle of the system and distributed to an entire SMART processor array.

Figure 3.4 shows the clock waveforms measured at different points in the system. By balancing the loading of the clock line on both sides of the array, the clock skews experienced by the first (point-C) and the last processors (point-D) tends to cancel out.

### 3.3 Processing Unit

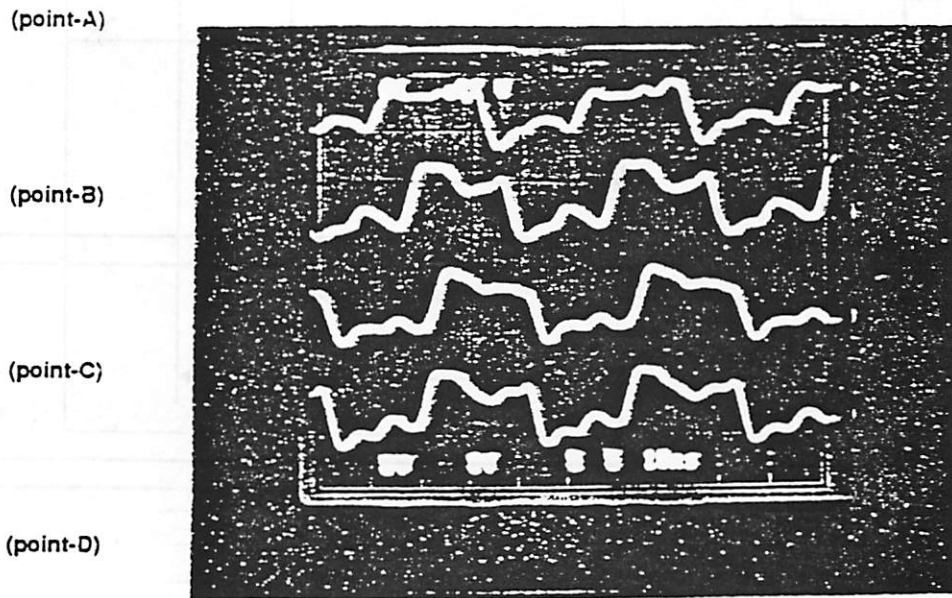
Each SMART processing unit consists of an AT&T WE DSP32C processor, an input/output FIFO, local RAM, *distributed shared memory* and an access controller chip set (consisting of a Master Access Controller (*MAC*) and a Slave Access Controller (*SAC*)) as depicted in Functional Block Diagram [Figure 3.5].





(a) Global Clock Distribution

Vertical Axis - 5 V / Unit, Horizontal Axis - 10 ns / Unit



(b) Clocks Measured at Key Points of the System

Figure 3.4: System Clock.

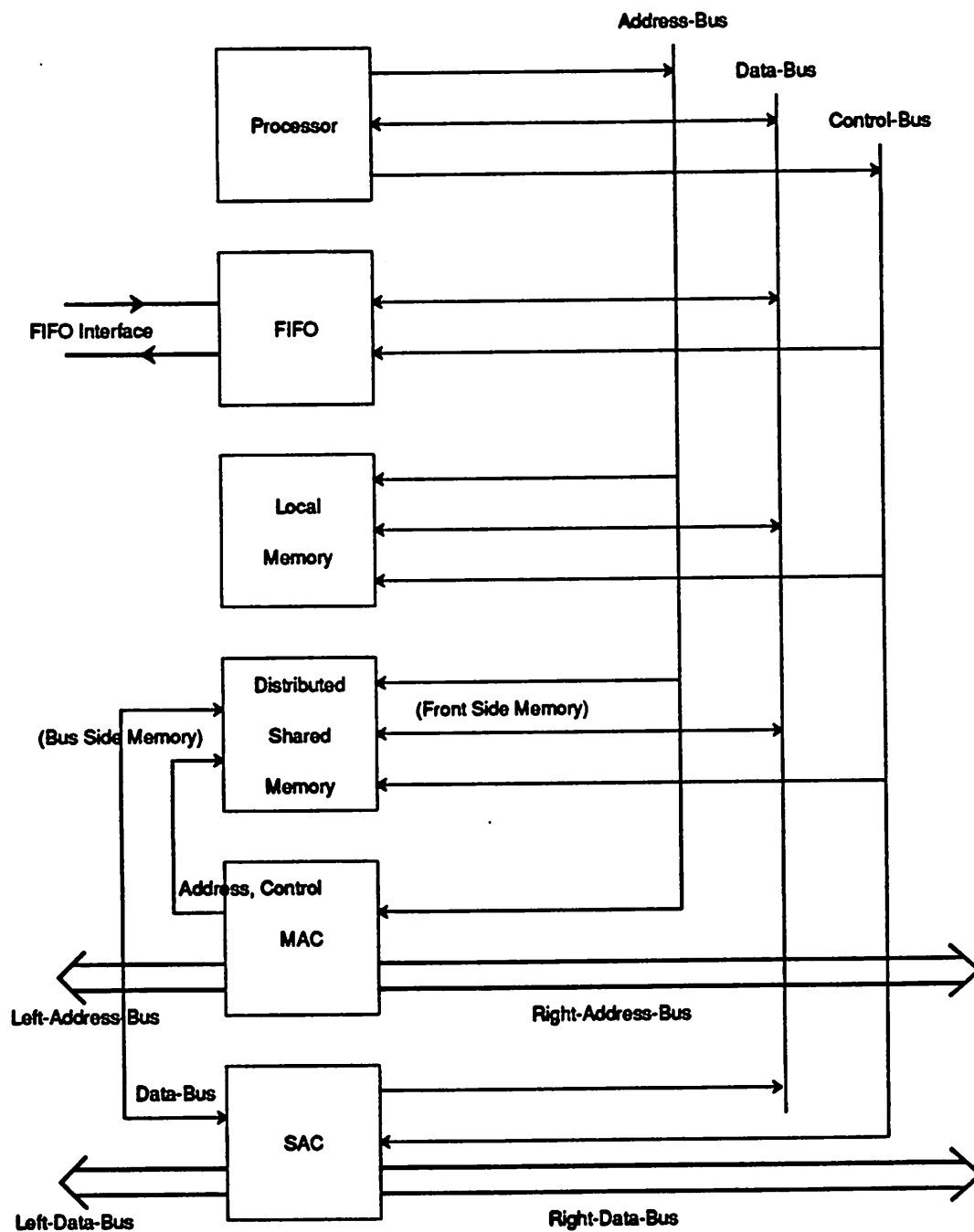


Figure 3.5: Block Diagram of the SMART Processing Unit.

**Processor Bus** — The role of the processor bus is to provide a path between the access controller (MAC and SAC chips), the memory blocks and the processor. The bus consists of an address field (26 bits), a control field (5 bits) and a data field (32 bits). Through the processor bus the processor can issue instructions to the access controller for accessing not only the local memory, the distributed shared memory, the FIFO but also the distributed shared memory of other processing units. In other words, the access controller decides how the instructions are interpreted and provides the necessary signals to access memory. The access controller is further described in Section 3.4 and the instructions are listed as a table in Appendix-A. The processor bus is restricted to a processing unit. The external interface is discussed in the following paragraph.

**External Interface** — Communication between the processor and the Heurikon system CPU board is established through the VME interface via the parallel I/O port and FIFO memory. The FIFO memory also provides an asynchronous custom interface to the outside world. <sup>1</sup>

The left and right reconfigurable buses connect adjacent processing units to form a linear reconfigurable array. They provide the only communication path between the processing units. For processing units sitting on different boards, short ribbon cables are used to implement the reconfigurable bus.

**Memory Structure** — Each processor's memory consists of a 4KB input FIFO and a 4KB output FIFO, 256KB of static local RAM, and 16KB of distributed shared memory.

Each input and output FIFO memory module consists of four  $1K \times 9$  fifo chips with a 50ns access time. One side of the FIFO is connected to the processor bus, while the other is connected to the external interface. Data, which were queued from the external interface, can only be accessed from the processor side. On the other hand, data, which were queued from the processor side, can only be accessed from the external interface. In order to avoid a FIFO overflow and an underflow, it

---

<sup>1</sup>More detailed information on the FIFO interface is given in Section 5.3.

is required to check *full* and *empty* flags before accessing it. As technology advances, the current FIFO chips may be replaced by a larger and faster FIFO chips without any hardware or software modifications.

A local static RAM module is used for storing the program and private data. It consists of eight  $64K \times 4$  static RAM chips with 25ns access time. Even though the specification of the overall access time from the processor to the local memory is 50ns, logic delays in the implementation require static RAM with a 25-30ns access time. Since the DSP32C processor requires such a high memory bandwidth, the cost of building such a memory system is rather high.

The distributed shared memory module is used for storing shared data, or for interprocessor communication. It consists of eight  $2K \times 8$  static dual-port RAM chips with a 30ns access time. To facilitate concurrent access to the distributed shared memory, one port is connected to a processor bus, and the other port is connected to access controller chips (MAC and SAC). The cost of dual-port RAM is quite expensive in terms of the price and the physical package size. If the core processor had provided a high-speed handshaking facility to take care of a case when requests were accepted from both sides at the same time, we would have been able to replace the dual-port RAM by the one-port static RAM, which would bring down the implementation cost significantly. In the selection of the core processor, the memory interface should have been more carefully evaluated.

**DSP32C Processor** — The DSP32C is a 32 bit CMOS Digital Signal Processor packaged in a standard 133-pin pin-grid-array (PGA). It offers a unique set of architectural features that include: 32 bit floating point arithmetic, 16 or 24 bit integer arithmetic, 16MB of address space, on-chip ROM and RAM, serial, parallel and external memory I/O ports all equipped with direct memory access capability (DMA), 4 40-bit accumulators and 22 general purpose registers, 2 external and 6 internal individually maskable interrupts. At its practical maximum operating clock frequency (40 Mhz), <sup>2</sup> the DSP32C executes 10 MIPS for integer operations and and

---

<sup>2</sup>Currently, AT&T provides a 50 Mhz version of the DSP32C chip. However, to run at 50 Mhz, the processor requires static RAM chips with less than 20ns access time. Therefore, in order to achieve

20 MFLOPS for floating point operations.<sup>3</sup> Each processor in the system is identified by a unique 6-bit *processor identification number (PID)* and therefore a maximum of 64 processors can be supported. The PID's are assigned in such a way that processors on the left-hand-side have smaller PID numbers than their right-hand-side neighbors.

The address space of DSP32C is divided into two banks; bank 0 and bank1, and only bank 0 is expandable off-chip. The external memory of bank 0 is divided into two sections, a low partition and a high partition. The number of wait states for each partition, where each wait state is one fourth of the instruction cycle, is independently configurable via the processor control word register. Configuring the wait states allows DSP32C to access fast memory without handshaking, and slow memory or peripheral devices with handshaking, using a synchronous ready signal. More detailed information about how the memory space is configured is provided in the following section and Appendix-A.

**Internal Clocks of MAC and SAC chips** — A 4-phase non-overlapping clocking strategy is used in SMART as shown in Figure 3.6. The function of the clock generator block is to generate these non-overlapping clock signals, guaranteeing that the skews between the falling edge of the reference clock and the falling edges of the internal clock signals are always constant, as required by the synchronous interfacing scheme used between chips. The four phase clock – phi1, phi2, phi3 and phi4 – is chosen since this matches the clocking scheme of the DSP32C processor. The processor may generate memory access in any phase. Furthermore, since each DSP32C performs one memory transaction in two phases, the a1 and a2 clocks which run at 20 MHz are used in the MAC and DSP32C interface.

The p1 and b1 clocks define a bus cycle for the switchable bus in MAC and SAC respectively. In one bus cycle, a message is transferred from one processor to all the other processors in the same bus segment. Notice that the bus cycle of SAC lags

---

high performance, the access to the static memory is usually implemented with one-wait state at 50 Mhz system clock [8] or without any wait state at 40 Mhz system clock. We have chosen the 40 Mhz system clock, since the system generally performed better without the wait state, especially when a program is stored in the external memory.

<sup>3</sup>For further information on the DSP32C processor, consult to the DSP32C manual [7]

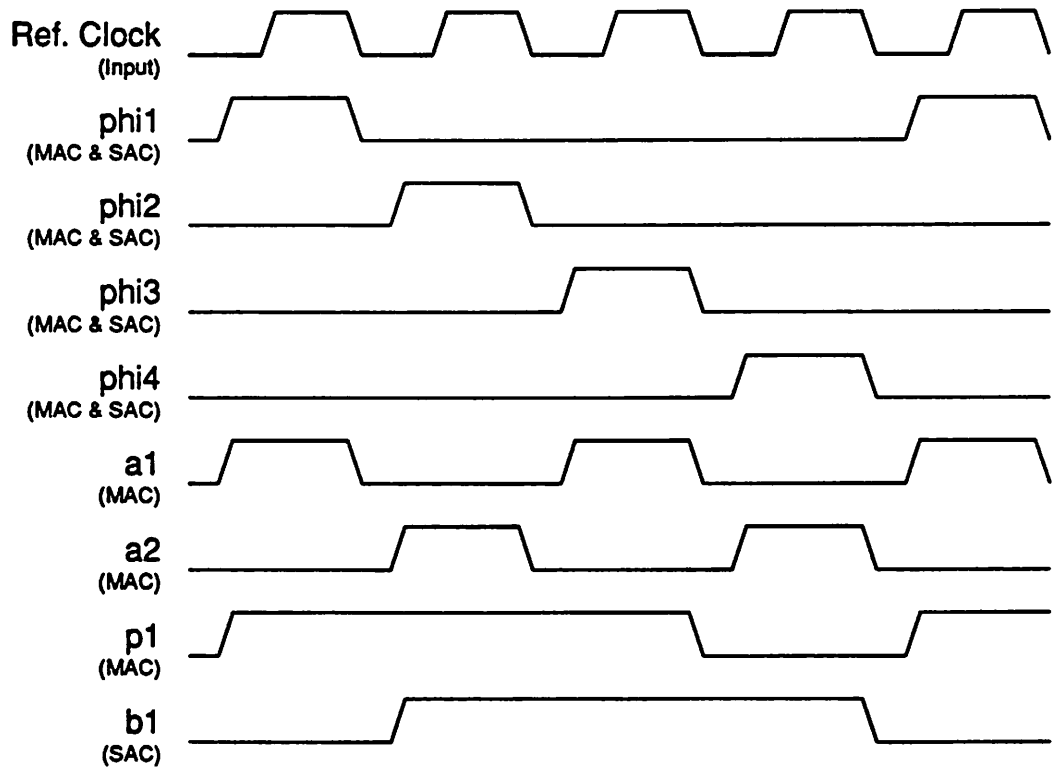


Figure 3.6: Internal Clock Signals.

that of MAC by one phase. The one phase difference accommodates the propagation delays from MAC to SAC and therefore eliminates time-critical interface problems between MAC and SAC.

Two clock-generation methods were implemented to provide the internal clocks. The first method is to derive the clock signals from the reference clock using a digital phase-locked loop (DPLL) [60]. The DPLL block was implemented for each MAC and SAC chips. A simple clock generator circuit could have been used but the skew between the reference clock and the internal clocks would vary from chip to chip due to process variations and differences in operating temperature. On the other hand, the DPLL clock could provide a virtual zero-delay clock driver. In order to save design efforts, a design used in the Berkeley SPUR Project [31] was taken with some modifications to suit our needs.

The second method is to generate the clocks externally using combinational logic gates. It was provided, in case the first method did not work. Since the system did not run reliably with the DPLL clocks, the second method is being used in the current system. It is believed that the large current switching noise on the board is causing the instability for DPLL clocks. The debugging effort has been concentrated on solving the clock skew problems due to the change in clock-generation method.

### 3.4 Master Access Controller and Slave Access Controller

All the functions needed to integrate the individual processors and memory modules to form the SMART multiprocessor system are implemented into two 208-pin VLSI Access Controllers: <sup>4</sup> the Master Access Controller (MAC) and the Slave Access Controller (SAC). As its name implies, MAC is the brain of the Access Control, and handles address and control signals, while SAC is working as a companion or a slave to MAC, and handles data signals.

---

<sup>4</sup>More information regarding circuit and layout of those chips is discussed in A. Yeung's Report [79].

To maximize the performance, the SMART architecture permits each processor to operate at or near full speed with a minimum of interference from the other processors. Although the DSP32C processors were not specifically designed for a multiprocessor environment, the support circuitry, implemented as a MAC and SAC chip set, isolates the processor from the complications of a multiprocessor environment.

MAC and SAC direct the traffics between processors through the reconfigurable bus, handle synchronization, generate control signals to the access input/output FIFO, the local memory as well as distributed memory for interprocessor communication.

The functionality and the implementation of MAC and SAC will be discussed in this section. A description of the instruction set of the access controllers will first be explained, followed by the explanation of the functional blocks in the MAC and SAC chips.

### 3.4.1 Access Controllers Instructions

The DSP32C processor passes instructions to MAC and SAC chips in a memory mapped fashion. MAC decodes the address according to the SMART processor bus address map [Appendix-A] and carries out the necessary steps to complete one of the following four types of instructions:

- System Configuration
- Local Memory Access
- Interprocessor Memory Access
- Interprocessor Synchronization

Those instructions can be divided into two groups based on their handshake interface between MAC and DSP32C. Group 1 instructions include instructions whose execution should be completed within a pre-determined constant period of time and therefore no handshaking is required. On the other hand, Group 2 instructions such as barrier and lock instructions usually take a variable amount of time to execute.



In this case, DSP32C must busy-wait until MAC asserts the system ready signal (SRDYN). Details of Group 1 and Group 2 instructions are shown in Appendix-A. The following paragraphs describe the functionality of the four classes memory mapped instructions.

**System Configuration Instructions** — System Configuration Instructions manipulate the configuration registers in the MAC. Most of these registers are set up during the initial system setup and do not change much during the run time. Their functions include:

- Configuration for reconfigurable bus and synchronization bus.
- Enabling and disabling the bypass and the synchronization signals,
- Releasing the lock.
- Setting the processor identification for interprocessor communication

**Local Memory Access Instructions** — The local memory access instructions involve accessing the following memory modules through the processor bus: local memory, input/output FIFO, and distributed shared memory. Those modules were discussed in Section 3.3.

**Interprocessor Memory Access Instructions** — The memory access through the bus slave unit is used when a processor needs to communicate with another processor or to access data in a distant shared memory.

In the case of an interprocessor write operation, the write request is queued for the communication through the reconfigurable bus in the bus master unit. If queue is not full, the acknowledge signal (SRDYN) is asserted. Otherwise the acknowledgement is delayed until some space is freed in the queue. In the case of an interprocessor read operation, the read request is queued as in the write request, but the acknowledgement is delayed until the data is returned.

A broadcast operation is served in the same way as a write operation by the bus master unit. The request is broadcasted to a group of processors specified by the

instruction. Once a data word is broadcasted on the reconfigurable bus, all bus slave units in the group update their own memory.

**Interprocessor Synchronization Instructions** — The barrier synchronization and the lock synchronization instructions can be issued through memory mapped instructions. The synchronization instructions belong to Group 2 except the unlock instruction which does not require any handshaking between the access controller and the processor. The barrier synchronization is custom-designed in hardware for high speed applications. To synchronize a set of processors at a certain point in the program, every processor in the set is required to execute a synchronization instruction. Then the acknowledge signal, which causes the completion of the synchronization, will be issued by MAC.

Controlled access to a shared resource can be implemented with locks and unlocks. In the case of a lock request, the lock is acquired and the acknowledge signal (SRDYN) is asserted by MAC, if the lock has not been acquired by anybody. Otherwise the acknowledgement is delayed until the lock is freed. The lock is released by an unlock operation. This does not need any acknowledgement signal.

A simple programming example using synchronization instructions is shown in Section 5.2.

### 3.4.2 Functional Blocks in the MAC Chip

This section describes the functional block diagram of MAC [Figure 3.7] and its functions. The block diagram includes a master control unit, a bus master unit, a bus slave unit, a bypass unit, a switch block for the reconfigurable bus, an arbiter for the arbitration, logic for the synchronization, and an internal clock generator.

**Master Control Unit** — The DSP32C issues instructions to MAC by generating an external memory address. The Master Control Unit decodes the instructions and controls the majority of the operations in MAC.

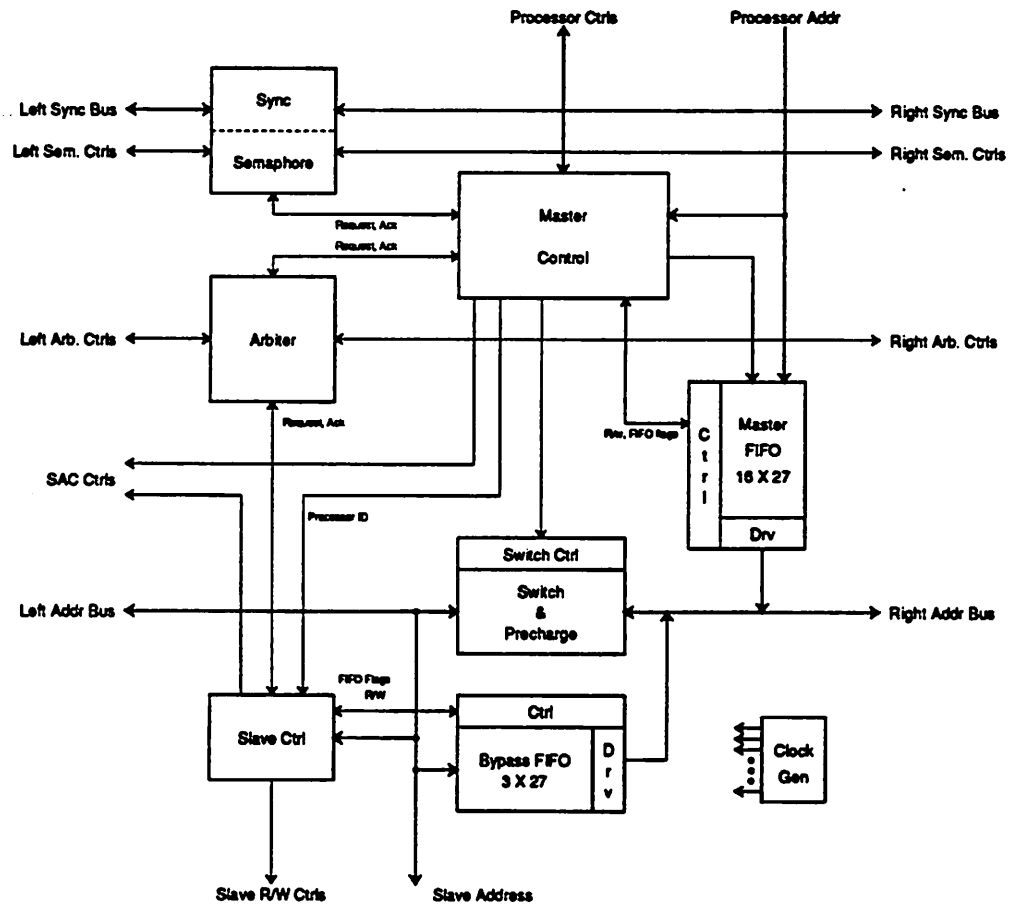


Figure 3.7: Block Diagram of MAC.

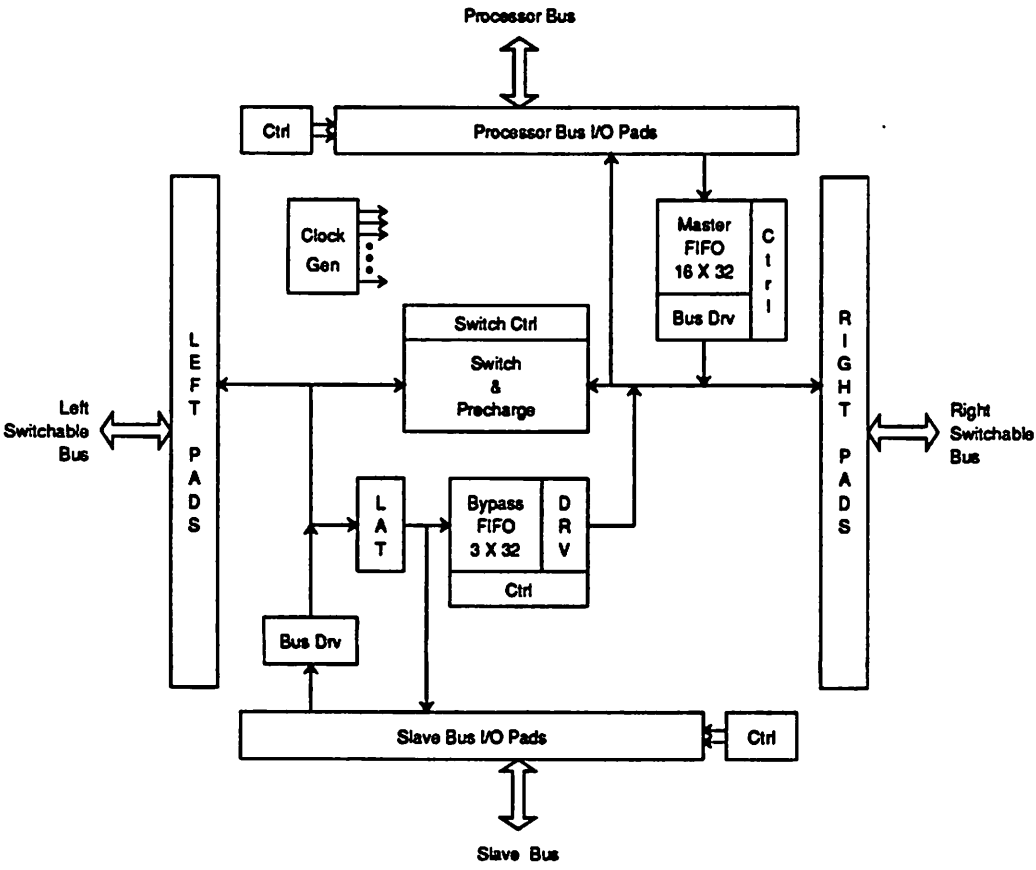
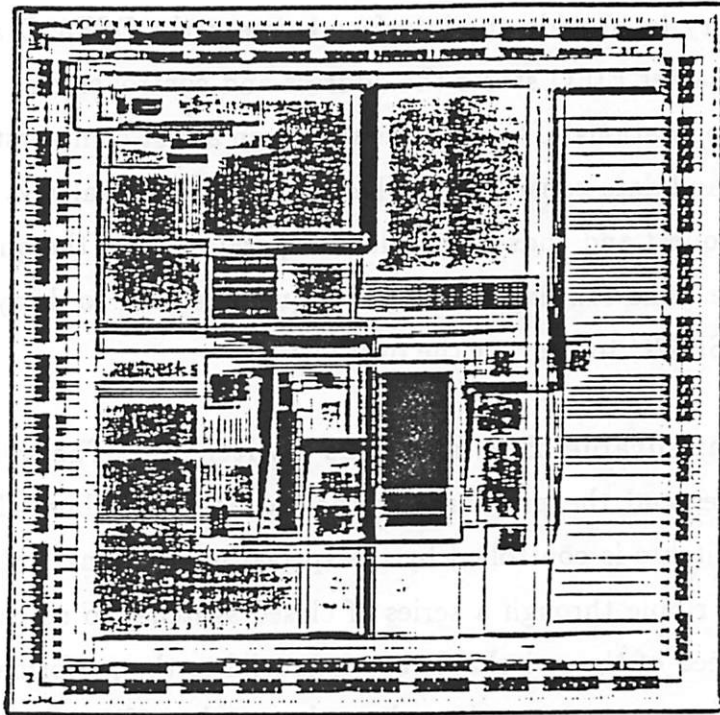


Figure 3.8: Block Diagram of SAC.

MAC Chip



SAC Chip

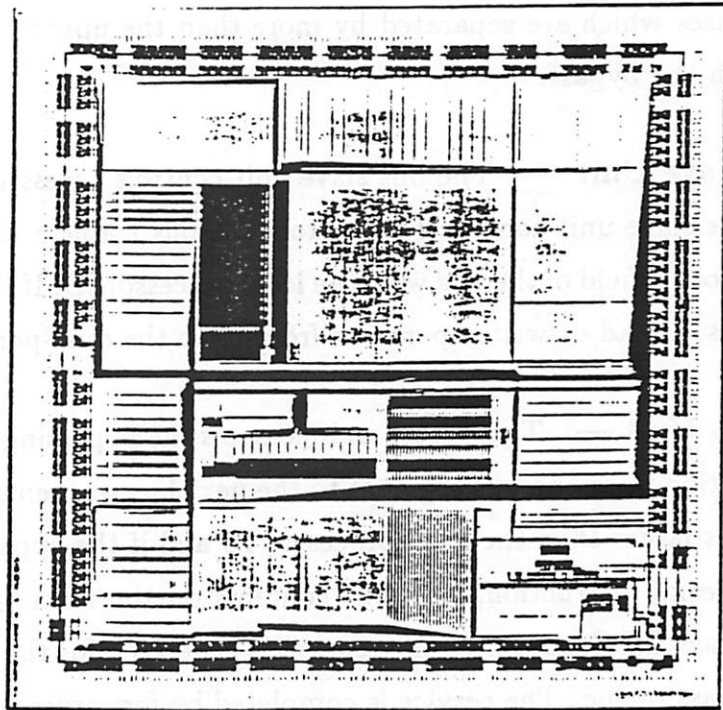


Figure 3.9: Pictures of MAC and SAC - die size 1.1cm x 1.1cm

**Bus Master Unit** — The bus master unit consists of the master FIFO and extra circuitry to provide the interface between the DSP32C and the reconfigurable bus. The master FIFO stores the address and control signals for interprocessor read and write operations. The queue is served on a first-come-first-serve basis to preserve the order and data consistency. The serving operation always starts with a reconfigurable bus request and ends with a delete operation from the queue upon completion of the service. The full flag of the FIFO queue is used to delay the acknowledge signal to the DSP32C to prevent the overflow.

**Reconfigurable Bus and Bus Switch** — The switch block implements the switches and the precharge of the reconfigurable bus. The opening and closing of the switches is controlled by the special instructions. In one bus cycle, a data word has to ripple through a series of closed switches to reach its destination. Therefore the speed of the switchable bus is crucial to the performance of the system, because this imposes an upper bound on the number of processors in the same shared-bus. The buses which are separated by more than the upper bound should communicate through the bypass.

**Bus Slave Unit** — The bus slave unit controls access to the the dual-port RAM. The bus slave unit constantly monitors the bus traffic and compares the destination processor ID field of the bus with the local processor ID. If the result of the comparison matches, a read or write operation from or to the dual-port RAM is performed.

**Bypass Unit** — The bypass unit controls the bypassing of the interprocessor communication from one bus segment to the next bus segment. If the destination processor ID is larger than the local processor ID and if the bypass unit has been activated by a special instruction, then the contents on the reconfigurable bus is copied into the bypass FIFO queue. A bus request is generated for the next bus segment to serve the bypass queue. The service is completed by forwarding the contents of the bypass queue to the next bus segment.

**Arbiter** — To avoid collisions on the bus, a MAC chip performs arbitration with all the other MAC chips on the same shared bus before any MAC can send out the data. The prioritized bus arbitration scheme is used for its simplicity of implementation. For instance, a bypass unit has the higher priority than a master unit, and processors with a low identification number (ID) has higher priority than processors with a high ID. To obtain higher throughput on the bus, the arbitration is pipelined with the data transmission. The bus pipeline was described in Section 2.3.3.

**Sync (barrier) and Semaphore (lock)** — The barrier and lock synchronizations are implemented in the *Sync* and *Semaphore* block. The left and the right synchronization buses are directly connected to the right and left buses of the two adjacent processors to form an interprocessor synchronization bus. There are five barrier synchronization bits in the bus and each bit can be configured and programmed to control a specific group of processors for the barrier synchronization. A global barrier synchronization, which does not have switch, is also provided to synchronize all the processors in the system.

### 3.4.3 Functional Blocks in the SAC Chip

The Slave Access Controller (SAC) is a slave to the Master Access Controller (MAC). It responds to the control signals from the MAC chip and carries out the requested actions exclusively on the data, transferring it between its four 32-bit buses. The functional block diagram of SAC is shown in Figure 3.8. It includes a bus master unit, a bypass unit, a switch block for the reconfigurable bus and a clock generator.

The primary function of the master and bypass unit is to provide a temporary storage for the data section of a request. Placing and removing requests from the queue is entirely controlled by MAC. The switch block and the clock generator performs the same function as in the MAC chip.

## Chapter 4

# DISCUSSIONS ON DESIGN AND IMPLEMENTATION

*The biggest change of a design comes at the last stage of its implementation.*

*- Murphy's Engineering Law*

The design of the SMART system has involved architecture design, functional organization, logic design and various levels of physical implementation. The implementation encompasses VLSI design, printed circuit board design, software development, and system integration of the hardware and the software.

With the help of the recent developments in the CAD technology and the design disciplines, the SMART system was designed and implemented in a two and half year period by two graduate students. The layout and testing of the two VLSI custom chips – MAC and SAC – were performed by Alfred Yeung [79].

This chapter concentrates on the *integrated* use of CAD programs, the functional description and the disciplines in various levels of implementation. The design and implementation steps are presented in the following order: architecture design, functional description, board implementation, chip implementation, testing and system integration.



## 4.1 Architecture Design

The SMART system is designed to meet functional specifications as well as price and performance goals, set by the requirements to achieve real-time behavioral simulation. The main design goals are as follows:

- High Computational Throughput
- Low Interprocessor Communication and Synchronization Overhead
- High I/O Bandwidth
- Fast Switchable Bus
- Modular Design in Increasing Number of Processors
- Short Design Cycle

Once a set of functional requirements of the architecture was established, the architecture was optimized and defined in functional description language. The functional description and simulation in designing a system was a major design discipline.

## 4.2 Functional Description and Simulation

Whenever a hardware design is developed, its functionality needs to be verified. One way to verify the functional operation of the design is to build a breadboard, a prototype or an actual hardware, drive the test vectors and monitor the results. However this is very expensive in terms of time and cost. As the target system becomes more complex, it is very hard to locate the error. Therefore the functions of the architecture and design needs to be verified before the prototype is built. This significantly reduces debug cycle time and cost.

Functional simulation is one of the most effective methods to verify the functions of the architecture and the design. In terms of hardware debugging, the functional simulation can separate the errors in the logic from the errors due to the electrical problems. It is also easy to prepare input stimuli for functional simulation

and to debug and modify the design. Moreover, components that are not available at the time of design can be simulated by providing an abstract model, such as for the Master Access Control and Slave Access Control custom chips. In terms of hardware design, the functional description helps the designer to think in a top-down manner. Hardware can be described and simulated at the behavioral level, register transfer level and gate level. Hence the functional description can be gradually developed from the behavioral level to the gate level. This allows the designer to concentrate on the current level of the design.

The SMART Processor Array has been described and simulated using the THOR functional simulation system [5]. The description consists of the models of circuit elements and their interconnections. Models were written in a language called CHDL, which is based on the C programming language with added features for modeling. The models describe the behavior of a primitive functional element such as the AT&T DSP32C processor, memory elements, and sub-blocks within the access controller chips.

Figure 4.1 shows a model *Arb.c* which handles bus arbitration, and is a sub-block of the MAC chip. The model consists of the interface section, the initialization section, and the behavioral description section. The behavioral description section is the main body of the model describing the algorithms or designs. It allows all standard C statements. In addition, data formatting and conversion constructs are provided. It allows a programmer to check data values more rigorously. For instance,  $arbL = !arbLN$  in the example can be expressed as  $fnv(arbL, 0, 0, arbLN, 0, 0)$  using the THOR constructs. When *arbLN* has the logic values of 0, 1, 2, 3 (ZERO, ONE, UNDEF, FLOAT), the *arbL* will be assigned to 1, 0, 2, 2, respectively.

The model is executed whenever an input or biput changes. For instance, when clock *phi4* changes, the model is executed and *byPassRequest.c4* and *masterRequest.c4* signals is latched into the s[1] and s[2] registers. The output arbRN is be re-evaluated as a result. More informations on modeling are described in [5].

The interconnections were described by a component oriented language called CSL. It allows a hierarchical description where a group of elements can be defined and treated as an entity called a sub-network. For example the descriptions

## Arb.c

```

MODEL (Arb)
{
  IN_LIST
    /* From Clock */
    SIG(phi1);
    SIG(phi2);
    SIG(phi3);
    SIG(phi4);
    SIG(a1);
    SIG(a2);
    SIG(p1);
    /* From MAC_L */
    SIG(arbLN);
    /* From CenuC:r */
    SIG(switchState_c4);
    /* From ByPass or Master */
    SIG(byPassRequest_c4);
    SIG(masterRequest_c4);
    /* From busR */
    SIG(wrnR);
  ENDLIST;

  OUT_LIST
    /* To ByPass (also SAC:SByPass) and Master */
    SIG(byPassGrant_c3);
    SIG(masterGrant_c3);
    /* To MAC_R */
    SIG(arbRN);
  ENDLIST;

  ST_LIST
    SIG(arbL);
    SIG(arbR);
    GRP(s,5);
  ENDLIST;

  arbL = !arbLN;

  if(phi4){
    s[1] = byPassRequest_c4;
    s[2] = masterRequest_c4;
  }
  s[0] = arbL || !switchState_c4

  arbR = s[0] && !(s[1] || s[2]);

  if(phi3){
    byPassGrant_c3 = (byPassRequest_c4 == 1) && s[0] && wrnR ;
    masterGrant_c3 = (masterRequest_c4 == 1) && s[0] && wrnR
                    && !byPassRequest_c4;
  }
  arbRN = !arbR;
}

```

Figure 4.1: Thor Modeling Example - Arb.c.

of the Master Access Controller and Slave Access Controller Chips consist of networks of functional sub-block elements. The description of the whole system consists of multiple processing node sub-networks which again consist of the DSP32C processor, Memory, and the MAC and SAC sub-networks.

Once models were described and interconnected, the simulation was achieved by exercising the input stimuli and monitoring the results. Figure 4.2 shows a simulation timing diagram for a three processor system. In this particular example, all bus switches are opened, and the bypass operations are enabled. The vertical axis represents signal or bus names and horizontal axis shows the timing step. Each timing step resolution corresponds to 1/16 of instruction cycle time (6.25ns).

At time 506, processor 1 drives *I-addrP* with a value of 0x187001,<sup>1</sup> *I-msnP* with 0x0, *I-mwnP* with an active low, and *I-cycleinP* with an active low. This results in a write operation to processor 3. The signals are decoded and queued into the bus master unit of the Master Access Controller (MAC) chip. *I-srdynP* is driven low at time 510 to acknowledge processor 1's request. Since the reconfigurable bus *I2II-bus* is busy bypassing data which are coming from *III2I-bus* the request is serviced at time 610. The high address bits on *I2II-bus* has been changed to an appropriate form for bus transmission (0x3070010). Due to the limited screen size, this output only shows the signals of processor 1.

For our purpose it was sufficient to model only the parallel port and access operations of the DSP32C. This processor model has allowed us to write high level command program to exercise the various functional blocks of the system.

The functional simulation environment, however, has its limitations. Even though the delay of the signal can be modeled up to certain level, it is hard to include the timing information in the model, which is necessary to detect problems such as critical paths, race conditions, clock skew, and setup/hold times. Whether the timing model should be integrated with the functional simulation is a controversial issue. However the present CAD environment lacks a support to solve the above mentioned problems.

---

<sup>1</sup>0x represents a hexadecimal notation.

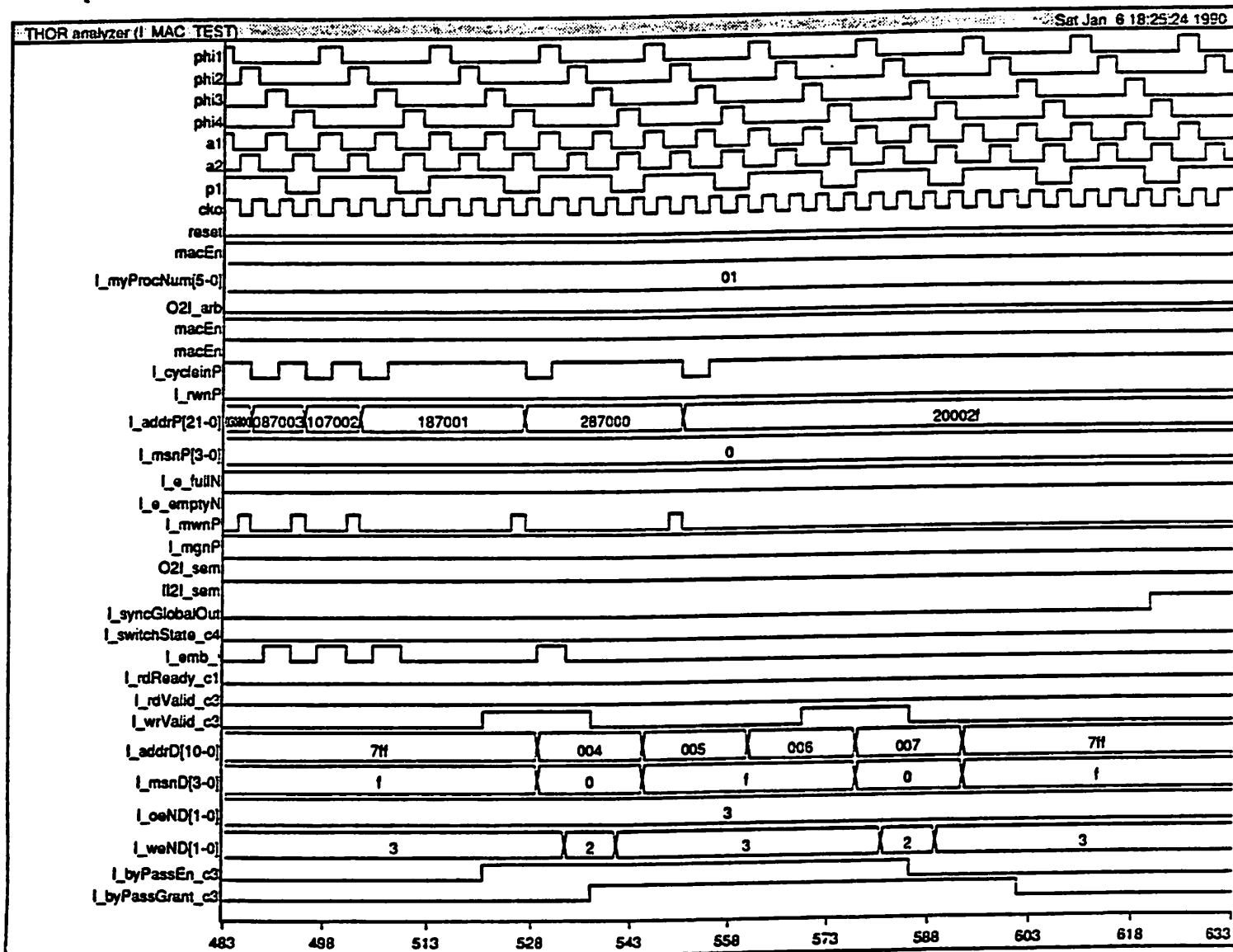


Figure 4.2: Simulation Example.

The functional description was used as the specification for custom chips and printed circuit boards. Therefore the description was placed at the center of the design and used throughout the whole design process.

The functional simulation vectors were utilized to test the physical layout of the custom chip. Signal values in the model were used as stimuli for an event-driven switch level simulator called *irsim* [19] to verify the physical layout.

### 4.3 Board Implementation

The PCB design task is basically to select, place and interconnect components such as processors, memory chips, passive elements (resistors and capacitors), connectors and jumpers, on a printed circuit board. Electrical components have a wide variety of specifications ranging from memory chips with different speed and size to connectors with various shapes and pin-out. The designers has to explore various design alternatives using different electrical components with design constraints, such as speed, board size, and power consumption.

The board implementation process consists of the schematic design phase (circuit diagram), simulations, and the printed circuit board implementation. The schematic design describes what physical components are used and how they are connected. Its principal use is to capture the design data for a printed circuit board implementation and to document for future debugging and enhancement. The schematic design can be described by a textual description as well as a graphical entry. The textual description called SDL (Structural Description Language), which is part of a LAGER silicon compilation system [19] [9], has been used to design both the board and the chips. The architecture is hierarchically described with library cells at the lowest level of a design tree. The placement information on the printed circuit board, such as position and rotation angle, were also hierarchically annotated. The Design Manager, which is also a part of the LAGER system, has been used to take the SDL description and the parameter values to form a complete hierarchical description of the design and to direct the CAD tools to generate the output. The output is an intermediate netlist file for a printed circuit board design tool.

The *Visula PCB Design environment* was used to implement its PCB design. It performed the placement, routing and produces physical mask films starting from the netlist information of the schematic design [65]. Components were placed either from the placement information specified in SDL or by using the auto-placement tool of *Visula*. Physical routing was done 100% by the auto-routing routine after several tries of re-placement and re-routing. Analysis of the routing result shows that good placement is essential for successful routing.

The printed circuit board has dimensions of 14.6"  $\times$  14.4", uses 8 layers (2 layers for power and ground), and has 8 mil trace and 8 mil spacing design rule. Each contains 665 parts, 2801 connections (excluding power and ground connection), 7775.7 inches of total routing length, and 8584 holes (vias and pads). Each auto-routing run took 11 hours, followed by 15 hours of *smoothing*, *mitring* and *fattening* steps on SUN3/60 with 100 MB of local disk space.

Critical signals on the printed circuit board, such as global clocks and memory strobe signals, were modeled as transmission lines [53]. They were simulated by SPICE using the lossless line model. The goal of the simulation was to find out the potential problems on the printed circuit board. The simulation was also used to find out the optimal termination resistors. The simulation was useful in avoiding some of the potential transmission line problems. However, the measurement showed that sometimes the transmission line behaved quite differently from the simulation results. One of the main reason was the modeling problem. It is difficult to simulate a signal when it passes through several layers, vias and packages which have different line impedances and delays. Furthermore, the coupling effects between signals often contributes a significant amount of noise to the signal. There have been some efforts to improve the transmission line simulation environment [10] [18].

Other electrical problems, such as driving capability, noise, and cross talk of signals, were checked mostly by hand analysis. In analyzing the electrical properties of the board, we also had problems in modeling and extracting parameters to simulate accurately.

As the state of the art in printed circuit board advances, boards grow larger and denser, while their electrical properties become increasingly critical. If board

designers can analyze how routing affects the electrical characteristics, they can save a significant amount of time and effort. Furthermore, they can evaluate the requirements, conditions, and constraints being imposed upon the routing process in order to arrive to a practical set of compromises. There are growing efforts to incorporate modeling capabilities into PCB environments [54].

The functionality of the board was simulated using THOR by extracting netlists informations from SDL and by using the library models developed during the functional description stage. While the purpose of the functional simulation for the architecture was to check whether the architectural design was valid, the goal of the functional simulation of the board was to check whether the original functional description is implemented with the right components and connected correctly. Simulation inputs were driven from the board-system interface such as a VME bus. Most of the functional descriptions and test input vectors, developed before for the architecture, were reused to simulate the board.

## 4.4 Chip Implementation

The large number of inputs and outputs of the access controller required a partitioning into smaller sub-systems for VLSI implementation. After carefully considering design issues such as die size, pin-out, package, testability, and minimization of interface signals and critical paths between sub-systems after partition, the access controller was partitioned and implemented as two chips: Master Access Controller (MAC) and Slave Access Controller (SAC), each of which has 208 pins. MAC is the master of access controller and SAC, being the slave, only responses to the control signals generated by MAC.

The two VLSI chip set implementation is strictly based on the functional description of the architecture. The layout of the chip was achieved by using a cell-based modular design and the LAGER silicon compilation system [19]. Physical layout was implemented manually on the leafcell level if the required cell was not available in the LAGER cell-library. The chosen cells were then assembled and connected to each other to form blocks. LAGER provided a number of assembling layout generators



for blocks with different layout styles such as *Timberwolfe* [69] [3] for standard cell design, *dpp* [73] for datapath compilation, and *TimLager* [73] for memory blocks and pads. The blocks were then hierarchically placed and routed by *Flint* [51] which is an interactive floor-planner and channel router.

Simulation took a very significant portion of the design cycle. Fast and efficient CAD support were required for a successful design. The physical layout was simulated by *irsim* [19], an interactive event-driven logic-level simulator for MOS transistor circuits. Using the linear simulation model, *irsim* was also used as a timing verifier. Inputs to *irsim* containing netlist, and resistive and capacitive loading information were directly extracted from the layout. SPICE simulation was used for more accurate timing check on critical paths such as a reconfigurable bus, and a memory and processor interface. Some of the important electrical characteristics such as power and gnd signal noise were simulated with SPICE. The noise was due to the change of the current flowing through the inductive power and ground pads. Given the inductance of the packaging, the noise were reduced by assigning more Power and Gnd pins and by optimizing the output pad sizes [43]. About 40 out of 208 pins were used for Power and Gnd pins which resulted in 0.7 Volts of worst-case noise in SPICE simulation.

The complete functional description, which was used to drive physical design of the chips, not only allowed for the verification of the architecture in advance but also provided a cross-checking facility between the functional description and the physical layout. This cross-checking facility is available in the THOR functional simulator so that signal values in the functional model can be used as stimuli for *irsim* and the responses is compared with the corresponding signals in the functional simulation.

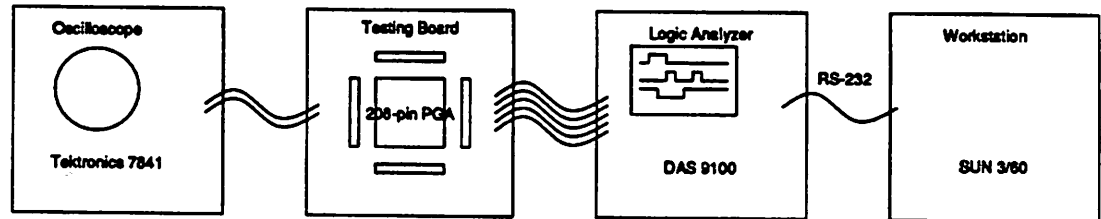
## 4.5 Testing and System Integration

Tests for a VLSI chip and a printed circuit board are developed in two phases [66]. In the first phase, known as design verification, test vectors are generated to verify logical correctness and timing behavior of the circuit through simulation. These test are carefully designed to exercise various functions of the circuit. For any

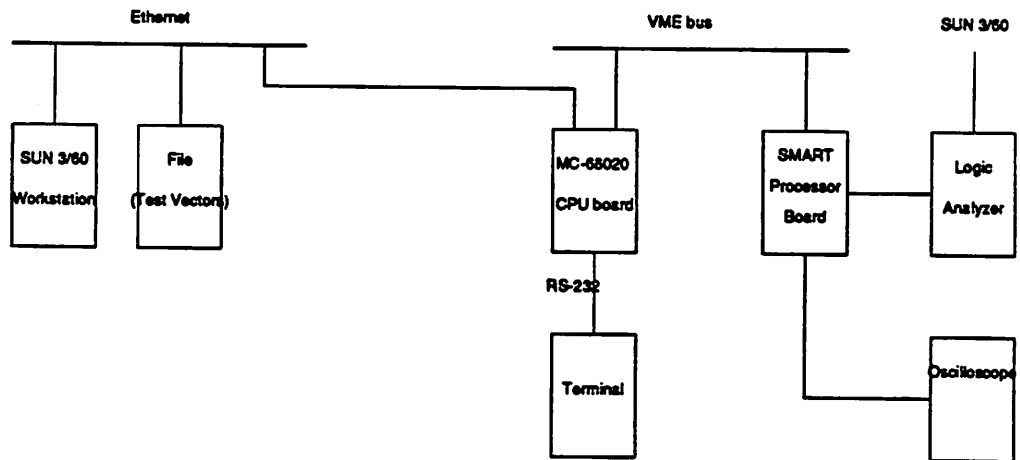
reasonably large circuit it would be impossible to use all possible input sequences. As a practical compromise, a subset of inputs, considered to be critical by the designer, is used for verification. The second phase of test generation consists of generation of manufacturing tests. Verification tests check for the right logic element to be connected in the right way. They do not necessarily check for specific types of defects produced during fabrication. Ideally, manufacturing tests must cover all faults that can possibly occur during fabrication. Since we were building the prototype system, it was the first phase of test generation that was our main interest. Diagnostic test programs were developed to perform verification testing. The testing strategy for two semi-custom VLSI chips (MAC and SAC) and SMART processor board is discussed in the following paragraphs.

Figure 4.3 (a) shows the the test setup for two VLSI chip testing. The conventional input/output test approach, driving and observing the external pins of the chip, was used. Test vectors were generated by the simulation environment which was used to verify the chip before fabrication. Programs were written to automate the process of extracting the test vectors from the simulation patterns, down-loading the vectors to the tester, and up-loading and verifying the acquired results from the tester after testing is performed. Since it is very difficult for the tester to access our 208 pin package at the same time, a boundary scan register chain was implemented at the pads to allow for testing the chips with access to only a few pins.

After the semi-custom chips were verified, the SMART processor board was tested in an environment as shown in Figure 4.3 (b). Diagnostic programs were developed in a high level language C to exercise various parts of the board functions from the host workstation. Then, cross compiled diagnostic programs were downloaded and run on the CPU board and the SMART processor array. The test results and error reports were collected from the SMART processor array to the CPU board. In case the input/output behavior – such as a logical value, timing or waveform – of the chips has to be checked, testing equipments such as an oscilloscope, a logic analyzer and on-board test logics were used to drive input or to observe the output. The oscilloscope was used to accurately measure the timing and the waveforms of the critical signals such as clock, memory interface, interprocessor bus and strobe



(a) Chip Testing Configuration



(b) Board Testing Configuration

Figure 4.3: Test Configuration.

signals. The logic analyzer was useful to generate and acquire hundreds of sequences of signal values which cannot be accessed from the CPU board. The following on-board test logics were provided to give an extra dimension to the test by temporarily reconfiguring the board: input registers to acquire the values of the signals, output registers and the strobe signals to drive the signals, and snap-shot register to record the values of the signals when a certain signal changes its value.

## Chapter 5

# PROGRAMMING SMART

*A programmer is a person who passes as an exacting expert on the basis of being able to turn out, after innumerable punching, an infinite series of incomprehensible answers calculated with micrometric precisions from vague assumptions based on debatable figures taken from inconclusive documents and carried out on instruments of problematical accuracy by persons of dubious reliability and questionable mentality for the avowed purpose of annoying and confounding a hopelessly defenseless department that was unfortunate enough to ask for the information in the first place.*

*- IEEE Grid news-magazine*

### 5.1 Overview of Programming Environment

A multiprocessor simulation environment is only useful when high level support tools for algorithm specification, processor partitioning, and code compilation are available.

We have chosen C and Silage [30] as our source languages. Being an applicative language, Silage is ideally suited for the description of algorithms with inherent concurrency. It does not require explicit expression of concurrency in the program. A high level software system is under development to auto-schedule the Silage description on the SMART processor array taking a load balancing and an efficient usage of the communication system into account. Section 5.5, *Auto Scheduling*, provides more detailed information about this system. An alternative approach is to program

the SMART machine starting from the C programming language. This approach, however, requires manual partitioning of the program. Section 5.2 describes how programs can be written in C.

Figure 5.1 provides an overview of programming environment for the SMART system. Application programs are developed and cross-compiled in the host computer due to its superior user interface and availability of related software routines. The DSP32C C-compiler developed from AT&T generates object code which is executable on the DSP32C software simulator. Another program takes the object code produced by the DSP32C C-compiler and converts it to a format which is appropriate for down-loading.

The Heurikon CPU board can directly access files shared with the host computer via NFS (Network File System) and has RPC (Remote Procedure Call) facilities to cooperate with host processes. VxWorks routines have been developed to provide low-level support such as initializing the SMART system, down-loading a program to the SMART processor array, and transferring data between a UNIX file and the SMART system. Appendix-D provides commands for compilation and down-loading, and a script file example, which includes a set of procedures for initialization, running and collecting data.

The CPU board also features a run-time software library that allows a programmer to read and write any part of the memory using DMA without disturbing a currently running program. Interrupts, which are controlled through the VME registers, are provided to interrupt the DSP32C processors. The processor, when interrupted, executes interrupt routines pointed by interrupt vector registers [8]. When the interrupt signal is kept being asserted, the DSP32C processor is interrupted at every instruction cycle. This feature can be used to single step in a parallel debugger program. The parallel debugger program, however, has not been developed yet. Therefore the currently used debugging methods utilize the external output FIFO to print out critical data at the various points in the program and uses the DMA operation to access memory. A programmer may write his own routines to satisfy his particular run-time interaction requirement [78].

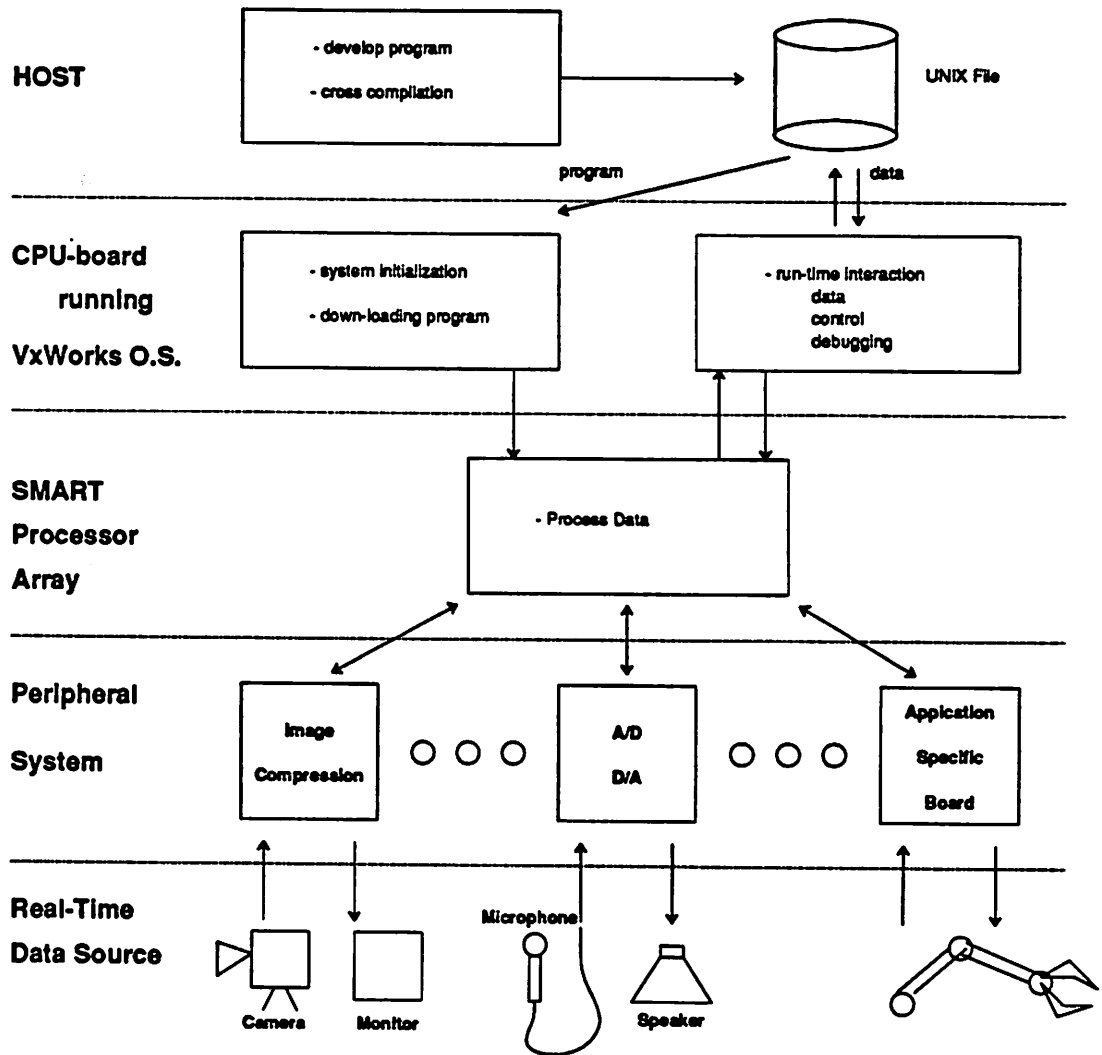


Figure 5.1: Programming Environment for SMART.

## 5.2 Programming in C

This section deals with programming algorithms in the C language. The simple, linear, reconfigurable structure of the SMART system makes it easy to exploit concurrency.

Since each processor executes its own program, a programmer can write different program for each processor, or combine them to one or two programs. When programs are combined, the *pid* (processor ID) is tested to execute a processor dependent part of the code. The 1024-point complex FFT program shown further in this section, is an example where the 10 processors are programmed with an identical code, even though the actual execution of the code may be different from one processor to another. A single combined program is generally easier to write and faster to compile than ten programs are. The run-time overhead for a combined program is usually negligible.

The image processing program in Appendix-C, called *image.c*, shows another example of how a program can be written for 10 processors. First, the initialization function *Init()* loads in a processor ID, a process number, and source and destination addresses from the input FIFO.<sup>1</sup> Then the main routine selects a function, such as minimum filter, median filter, laplacian and sobel, from library routines, based on the process number. Each function acquires data from the memory pointed to by the source address and generates data to the memory pointed to by the destination address. The above described programming style is useful when a group of applications uses a library of routines.

Macros play an important role in programming the SMART system. The macros, defined in header files such as *smart.h* and *sl.h*, hide the implementation details of the physical address mapping. The macros are divided into the following three groups: memory access macros, synchronization macros and system configuration macros. Macros defined by the SMART system software have *Sm\_* as the prefix. The data type is specified as the suffix of the macro name such as *\_fl*, *\_int*, *\_sint*, or

---

<sup>1</sup>In the case of an interprocessor communication, the source address usually points to its own distributed shared memory, and the destination address points to the destination processor's distributed shared memory.



*\_char* which stands for a float (32-bit) , an integer (24-bit), a short integer (16-bit), or a character (8-bit) data type. All macros are listed in Appendix-B. The following paragraphs illustrate the key macros which are frequently used in programming.

### 5.2.1 Macros for Accessing Memory

Macros are provided for each memory module such as the local memory, front memory, bus memory and FIFO. A macro for accessing the local memory treats the data as an array. The argument of the macro provides the index into the array. For example,

$$Sm\_lm\_fl(8) = Sm\_lm\_fl(2) + Sm\_lm\_fl(3);$$

will read two floating point numbers from the local memory, add them, and put the result back to the local memory.

On the other hand, the front memory and the bus memory require more than one argument. The front memory takes a bank select number and an index address as its arguments.

$$Sm\_fm\_fl(bank\_select, index\_address);$$

There are two banks in each distributed shared memory so that producer and consumer processors can access different banks of memory without interfering with each other. The bank select number specifies the memory bank being accessed.

The bus memory macro is used for interprocessor communication. It requires four arguments.

$$Sm\_bm\_fl(mode, pid, bank\_select, index\_address);$$

The first argument *mode* specifies one of the five different modes of access: regular read-write (L-RW), regular broadcast (L-BC), circular write (L-CW), short write (S-W) and short broadcast (S-BC). A *Regular* transaction requires an acknowledge signal from MAC to DSP32C, but a *short* transaction does not require an acknowledge signal. The acknowledge signal is used to delay the transaction when the queue in the bus master gets full so that a programmer does not have to worry about

overflowing the queue. For a *short* transaction, the interval between transactions should be controlled by software to prevent an overflow.<sup>2</sup> The second argument *pid* selects the distributed memory associated with the target processor. The third and the fourth arguments specify a bank and an index address, respectively.

The *fifo* macro does not require any argument. When it is used in a left hand side of an assignment, for example,

$$Sm\_fifo\_sint() = tmp;$$

a value of a variable *tmp* on the right hand side is *pushed* to the output FIFO. On the other hand when it is used in a right hand side of an assignment, for example,

$$tmp = Sm\_fifo\_sint();$$

a data word is *popped* from the input FIFO, and assigned to a variable *tmp*.

## 5.2.2 Macros for Synchronization

The synchronization mechanism in the SMART system can be categorized as the following three types: barriers, locks and events. The following paragraphs illustrate macros for barriers and locks.

A barrier synchronization can be issued by

$$Sm\_syncV(lines);$$

It takes *lines* as an argument whose value is interpreted as the status of 6-bit synchronization lines. For example, *Sm\_syncV(1)* requires a *global* synchronization, and *Sm\_syncV(2)*, *Sm\_syncV(4)* and *Sm\_syncV(8)* require a *local synchronization* for line 1, 2 and 3 respectively. On the other hand *Sm\_syncV(2 + 4 + 8)* will require a synchronization from all line 1, 2 and 3.

A lock can be acquired by

$$Sm\_semP();$$


---

<sup>2</sup>Section 3.4 and Appendix-A provide detailed information about read-write and broadcast operations.

If it is being locked by other processor, the processor, which requested the lock, will go into a busy-wait state until it acquires the lock. A lock can be released by

```
Sm_semV();
```

operations. It does not have any effect if the lock was not currently owned.

### 5.2.3 Macros for System Configuration

The bus can be opened and closed at any time by software. Usually the reconfiguration macros are preceded by the barrier synchronization with other processors to make sure the bus is not reconfigured in the middle of a data transaction.

```
Sm_syncV(lines);
```

```
Sm_openSwitch();
```

or

```
Sm_syncV(lines);
```

```
Sm_closeSwitch();
```

The barrier lines, except the global barrier line, can be reconfigured in the same way as the data bus.

```
Sm_wrSyncSwitch(lines);
```

It takes *lines* as an argument, which represents synchronization line reconfigurations. A bit with a value of *one* sets a switch to the *closed* state and a value of *zero* sets a switch the *opened* state. For example, *Sm\_wrSyncSwitch(0x3f)* will close all the switches, and *Sm\_syncV(0xa)* will close switches for line 1 and line 3. Since a global synchronization does not have any switch, a least significant bit, which corresponds to a global synchronization, has no effect. It is also possible to enable or disable each barrier line.

```
Sm_wrSyncDefault(lines);
```

A bit with a value of *one* sets a barrier to the *enabled* state and a value of *zero* sets a barrier to the *disabled* state.

A memory bank in a distributed shared memory can be selected using a *swap* state which is stored in the MAC chip. The swap state is often used when processors are working in a pipelined fashion. They operate on an essentially infinite stream of input data, executing once for every input. A processor reads from one bank of memory while the other processor writes to the other bank of memory. At the end of the iteration, the memory bank is *swapped* to exchange two memory banks.

The swap state is *Exclusive-ORed* with a *bank select* argument in the macro to access the distributed shared memory (`Sm_bm_fl(mode, pid, bank_select, index_address)`). For example a swap state *one* and a bank select mode *zero* will select bank one. The swap state can be cleared, set, and toggled by the following macros.

```
Sm_clrSwap();
```

```
Sm_setSwap();
```

```
Sm_toggleSwap();
```

A bypass operation can be enabled and disabled. When the bypass is disabled, no bypass operation is allowed between two adjacent disconnected buses. On the other hand, when the switch of the data bus is closed, the bypass is automatically disabled. Since there is only one data bus, it does not require any argument.

```
Sm_disableBypass();
```

```
Sm_enableBypass();
```

The processor number, called *pid*, needs to be defined by the software. The *pid* can be defined within the program or it can be loaded from the FIFO. The following code is frequently used at the beginning of the program. It first load in the *pid* from the input FIFO and write *pid* as the processor number.

```
pid = Sm_fifo_int();
```

```
Sm_wrProcNum(pid);
```

The programming environments were presented based on our current macro definitions. There are a lot more other ways to program than what has been shown here. As more programs are developed, the programming environments will be expanded and the higher level macros will be provided. The following section provides a FFT programming example using macros.

### 5.2.4 A FFT Example

Figure 5.2 shows a 1024-point complex FFT program.<sup>3</sup> Each of the ten processors works on one tenth of the total computation load in a pipelined fashion. First, the program is initialized. The *pid* (processor ID = 0..9), *maxpid* (maximal processor = 10), and *tasksize* (number of points = 1024) are loaded from the FIFO. Other informations, such as initializing variables, can also be loaded from the FIFO.

```
pid = Sm_fifo_int();
maxpid = Sm_fifo_int();
tasksize = Sm_fifo_int();
```

Once initial informations are loaded from the FIFOs, the SMART system is configured for the FFT algorithm. This is achieved by setting status registers in access controllers. First, *pid* is written to the access controllers.

```
Sm_wrProcNum(pid);
```

Since the interprocessor communication is limited to the adjacent processor, the data bus switch is opened and the bypass is disabled.

```
Sm_openSwitch();
Sm_disableBypass();
```

The global synchronization is enabled to synchronize all the processors.

```
Sm_wrSyncDefault(1);
```

---

<sup>3</sup>The header file of FFT is included in Appendix-B

```

#include "smart.h"
#define IN_OFFSET 2
main()
{
    register int i, j, pid, tasksize;
    register float *a,*b, *x,*y,*W, bwr, bwi;
    int maxpid;

    /* initialize */
    pid = Sm_fifo_int();
    maxpid = Sm_fifo_int();
    tasksize = Sm_fifo_int();
    Sm_openSwitch(); /* open all switches */
    Sm_disableBypass();
    Sm_wrProcNum(pid);

    /* global sync is enabled for processors used. */
    Sm_wrSyncDefault(001)

    /* Download twiddle from fifo to local memory */
    for (i=0; i<tasksize; i++) Sm_lm(i) = Sm_fifo_fl();

    /* For proc(0), download input data from fifo to front memory */
    /* For other procs, just initialize to some dummy values. */
    for (i=0; i<2*tasksize; i++) {
        If(pid == 0){ Sm_fm(BSEL_SWAP, i) = Sm_fifo_fl(); }
        Sm_fm(BSEL_SWAPB, i) = Sm_fm(BSEL_SWAP, i);
    }
    Sm_syncV(001); /* global sync */
    for (j=0; j<maxpid; j++) {
        /* set up i/o and twiddle pointers */
        a = Sm_fm_addr(BSEL_SWAP, 0);
        b = a+IN_OFFSET;
        x = Sm_bm_addr(L_RW, (pid+1)%MAX_PROC, BSEL_SWAPB, 0) ;
        y = x + tasksize;
        W = Sm_lm_addr(0);
        for(i=0; i< (tasksize>>1); i++) {
            bwr = (*b++ * *W++) - (*b-- * *W);
            bwi = (*b++ * *W--) + (*b++ * *W++);
            *x++ = bwr + *a++;
            *x++ = bwi + *a--;
            *y++ = *a++ - bwr;
            *y++ = *a-- - bwi;
            /* update pointers */
            W++;
            a = b;
            b += IN_OFFSET;
        }
        Sm_syncV(001);
        Sm_toggleSwap();
    }
    /* For the last proc, upload output data from front memory to fifo */
    If (pid==(maxpid%MAX_PROC)) {
        for (i=0; i<2*tasksize; i++) Sm_fifo_fl() = Sm_fm(BSEL_SWAP, i);
    }
}

```

Figure 5.2: Programming Example - FFT.

After the system initialization stages, the coefficients used in the FFT (called twiddle factors) are preloaded from the FIFO.

$$Sm\_Im(i) = Sm\_fifo\_fl();$$

For processor-0, the input data words are down-loaded from the FIFO to the front memory. After all the initialization steps, we synchronize every processors and start computation.

$$Sm\_syncV(1);$$

The main body of the program starts after global synchronization. It is similar to a C-program for a general purpose computer. The base addresses to point data and coefficients are assigned before the actual computation. Macros are provided to calculate those addresses.

$$a = Sm\_fm\_addr(BSEL\_SWAP, 0);$$

$$W = Sm\_Im\_addr(0);$$

$$x = Sm\_bm\_addr(L\_RW, (pid + 1)\%MAX\_PROC, BSEL\_SWAPB, 0);$$

The BSEL\\_SWAP, L\\_RW, MAX\\_PROC are all constants defined in the header file. For instance, float pointer *a* is initialized to the base of the front memory, while the memory bank is selected by BSEL\\_SWAP. The float pointer *W* is initialized to the base address of the local memory where the FFT coefficients were down-loaded. The float pointer *x* is used to write the result to the consumer processor's bus side memory (*Sm\\_bm\\_addr()*). The first argument L\\_RW selects the read-write mode through the bus. The second argument calculates the next processor number which can be obtained by adding the current processor number by one. %MAX\\_PROC is used for the processor-9 to write the result back to the processor-0. The memory bank is selected by BSEL\\_SWAP and the base index is 0.

The inter-loop actually performs the FFT operation using the pointers initialized just before entering the loop. At the end of each outer-loop iteration, the processors are globally synchronized and the memory banks are swapped. Therefore

after the global synchronization, every processor will start with a set of data which were produced by the adjacent processor.

*Sm\_syncV(1);*

*Sm\_toggleSwap();*

When the computation is over, the result is up-loaded to the FIFO by processor-9.

*Sm\_fifo\_fl() = Sm\_fm(BSEL\_SWAP, i);*

### 5.3 Real-Time Input/Output

The SMART system can include a variety of peripheral devices which transfer real-time data via a VME bus or via a custom FIFO bus [Figure 5.3]. Real-time behavioral simulation will often require the interfacing of a peripheral device to the SMART system. This section gives an overview of those I/O interfaces, and discusses the bandwidth of the system.

Our VME bus implementation provides a peak 6 Mhz transfer rate for 32-bit data (24 MB/sec). However, when the transfer is handled by a software program, the actual transfer rate may be reduced by an order of magnitude due to software overhead like instructions for creating a loop and calculating next data addresses within the loop. Therefore a system which needs to transfer in the order of 6 Mhz rate requires either a special hardware unit or an interface to a custom FIFO bus.

A speech processing simulation environment can be set up using the VME bus interface. For a sample rate of 8 KHz and an 8-bit representation, the VME bus is more than sufficient to handle the real-time data. An A/D & D/A board, interfaced to the VME bus, acquires the input speech from the microphone and generates the output speech to the speaker.

A custom FIFO bus consists of five 32-bit ports: one port for each board. The custom FIFO bus can transfer data at a 20 Mhz rate to and from the input and output FIFO units as shown in Figure 5.3. However the rate is limited by the



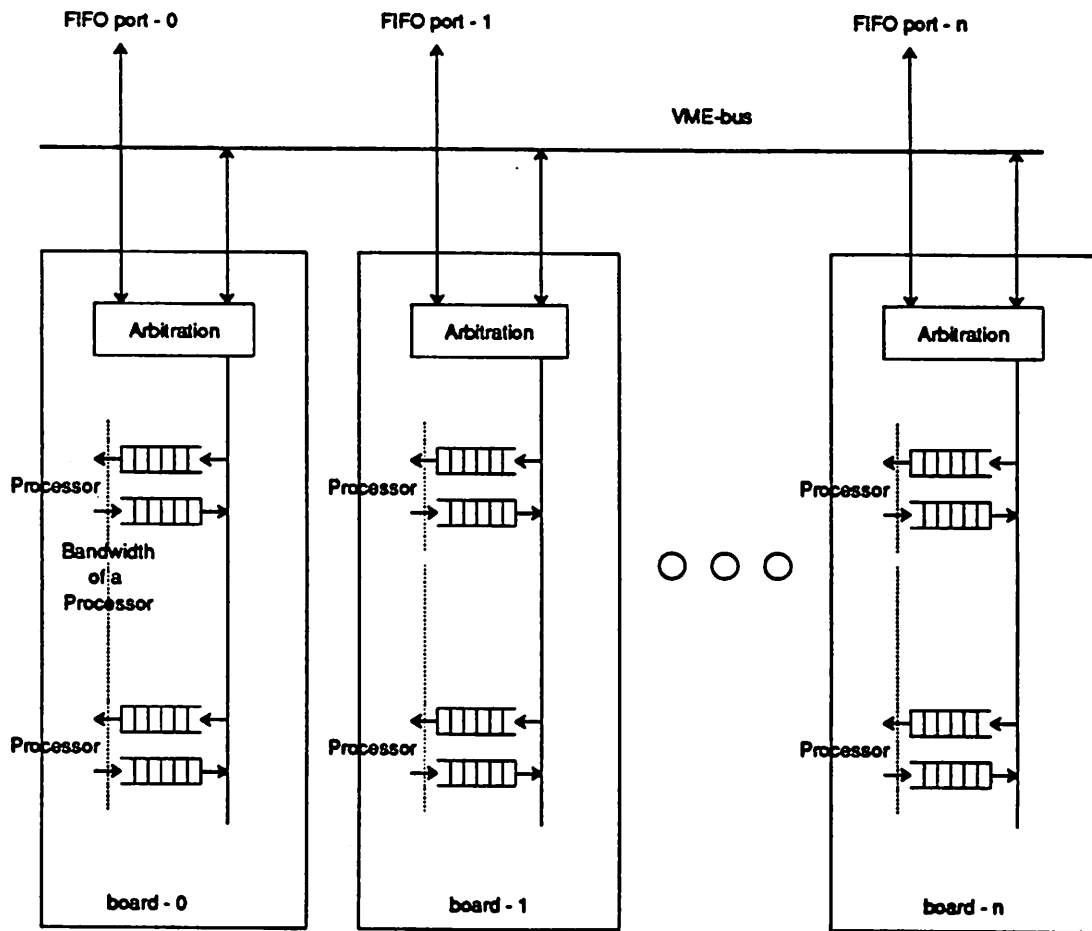


Figure 5.3: Real-Time Data Interface.

DSP32C which can transfer the data at a peak 3.3 Mhz (13.2 MB/sec) [33]. <sup>4</sup> The bandwidth of the FIFO port may or may not be large enough depending on the size of image or video applications. For example, an image processing example which requires the following bandwidth, can not be handled in real-time.

$$\begin{aligned}
 \text{Total Bandwidth Requirements} &= \\
 &512 \times 512(\text{pixels}) \\
 &\quad \times 30(\text{frames/sec}) \\
 &\quad \times 3(\text{bytes/pixel}) \\
 &\quad \times 2(\text{input, output}) \\
 &= 45\text{MB/sec}
 \end{aligned}$$

The I/O requirements of the video and image processing system can be 15 MB/sec (monochrome 512 x 512 image), 54 MB/sec (NTSC CCTR-601) or more (HDTV).

The basic I/O problem lies in the limited bandwidth of DSP32C. Therefore, if applications require I/O bandwidth to be less than 13.2 MB/sec, the SMART system can process them in real time. Otherwise, the system I/O bandwidth is not sufficient for real-time processing. However, the system can provide non-real-time simulation speedup.

The system was designed especially for real-time medium speed DSP applications such as the speech processing, whose I/O requirement was discussed earlier in this section. The system provides sufficient bandwidth for those applications.

As the semiconductor technology advances, it will be possible to build a multiprocessor system to handle the bandwidth and the processing requirements of real-time image and video processing applications. To incorporate the future technology advancement and to improve the real-time I/O bandwidth, a heterogeneous SMART system which consists of a variety of processors is proposed in Chapter 6.

---

<sup>4</sup>The peak rate was calculated based on the following assumption. The DSP32C requires SRDYN signal to access FIFO. Therefore, it takes one and a half instruction cycle to read the data from the FIFO and another one and a half instruction cycle to transfer the data to a distributed shared memory.

## 5.4 Performance of the SMART System

Since the SMART machine became first operational in Feb. 1990, we have programmed several DSP algorithms and commonly used routines, and have measured the performance of the system.

Task	Time(ms)	Speedup Over one DSP32C	Speedup Over SPARC-1	Speedup Over SUN 3/60
1024 point Complex FFT	33	8.8	24	121
Echo Cancellation	0.2	7.0	15	55
Pitch Extraction	2.1	8.6	52	70
Matrix Multiplication	.25	8.4	25	59
Average for FP programs	-	8.2	29	76
Sobel filtering	2150	9.1	9.5	15
3x3 maximum filtering	1500	9.0	4.1	12
laplacian filtering	1380	8.7	14	15
invert filtering	110	9.1	6.4	13
Average for INT programs	-	9.0	8.5	14

Table 5.1: Performance Results for 10 Processors.

Application area that most strongly guided the development of SMART was real-time simulation of a medium speed DSP applications such as speech, telecommunication, audio and robotics.

Table 5.1 reports the performance of the SMART prototype system for a variety of algorithms. The table includes all system overhead except for the initial program memory loading. It is assumed that the input is a stream of data. The

performance is measured from the point the input data arrives to the point the data is completely processed.

We compare the SMART performance with a SUN 3/60 and a SUN SPARC-1. Both of them have floating-point co-processors. These computers are widely used and the relative performance comparison to other computers can be easily obtained. Since the SUN 3/60 and SPARC-1 computers do not have a real-time I/O interface, we have assumed that the real-time data is in the main memory when it is first accessed, which will give us the maximum speed those computers can have.

The programs are measured in two groups. The first four programs in the table – FFT, Echo-Cancellation, Pitch-Extraction, and Matrix(100x100)-Vector(100x1)-Multiplication – represent the typical medium speed DSP applications using floating point operations. Their benchmark performances, simulated on the system software model, were shown in Section 2.5.<sup>5</sup> The last four programs in the table – Sobel-Filter, Maximum-Filter, Laplacian, and Inversion – represent the commonly used image processing routine<sup>6</sup> using only integer operations. For DSP programs using floating-point operations, the measurement shows the average speedups of 8.2 times over a single DSP32C processor, 29 times over SUN SPARC-1, and 76 times over SUN 3/60, respectively. On the other hand, for DSP programs using only integer operations, the measurement shows the average speedups of 9.0 times over a single DSP32C processor, 8.5 times over SUN SPARC-1, and 14 times over SUN 3/60, respectively.

The result shows larger numbers of speedups for programs using floating point operations than for programs using integer operations. This is because the DSP32C, even though it also provides integer operations, is a floating point oriented chip. For instance, it can execute a floating point multiplication and an addition in one instruction cycle, while it can only execute one integer operation per one instruction cycle.

The speedup over SUN 3/60 and SPARC-1 depends heavily on the performance of the DSP32C processor itself. From the architectural point of view, it is more interesting to examine the speedup over a single processor. Overall, the SMART sys-

---

<sup>5</sup>Note that the benchmark performances were measured for 16 and 64 processors.

<sup>6</sup>monochrome 512 x 512 image.

tem with 10 processors has shown a speedup ranging from 7.0 to 9.1 times over a single DSP32C processor.

In highly regular programs with a large amount of computation, we typically observed nearly the peak performance of the machine. As an example, for a FFT algorithm, the system provides 8.8 times of speedup over a single processor. On the other hand, in irregular programs like echo cancellation, we observed a smaller speedup of 7.0 times due to load unbalancing. It is a very hard problem to correctly estimate and balance the load. We hope the auto-scheduling system, which is being developed [Section 5.5], will alleviate this problem.

All the programs shown in the table were programmed in C. The SUN 3/60 and the SPARC-1 provide very efficient C-compiler. However, the DSP32C C-compiler could not generate efficient machine code due to the following problems:

- The DSP32C is a highly pipelined processor (6 stages). It is quite difficult for a compiler to generate efficient code for a processor with many pipeline stages. This is due to the data dependency between the sequence of instructions.
- The special hardware for DSP applications, such as a bit reversal addressing for FFT, can boost up the performance. However, it is very difficult for the C-compiler to recognize the pattern in the source code and utilize that special feature.

Extensive use of library routines and assembly macros can alleviate those problems. For instance, the data dependency at the instruction level can often be avoided if programmed in assembly language. Furthermore the special hardware instructions can be directly addressed to improve the performance. On the other hand, AT&T is developing a C-compiler for the DSP32C which will do a better job in optimizing the code. Hence, the optimization in programming environment will give further speedups over the performance given in the table.

Research groups that are developing *speech coding algorithms* [1] for low-power portable communication devices and *grammar processing in a speech recognition system* [64] are planning to test their algorithms on the SMART system. As

more complex application programs are developed, they will give better indications of the performance.

## 5.5 Auto Scheduling

The goal of the auto scheduling is to provide an environment where users can express DSP algorithms textually or graphically, and have a compiler partition and translate the program into sections of assembly code to be executed on the multiprocessor system [67]<sup>7</sup> [Figure 5.4].

The Silage To SMART (S2S) Compiler, which implements the partitioning algorithm consists of 4 tasks:

- Silage to Flowgraph (S2F) Translation
- Flowgraph Partitioning
- Flowgraph to C (F2C) Translation
- C to DSP32C Code Compilation

The Flowgraph Partitioning represents the most challenging aspect of the compiler. We are given a program represented by a hierarchical flowgraph, and constraints on the number of processors and amount of memory available. Assuming that the program will be applied to an infinite stream of input samples, the goal is to find a mapping of the program onto SMART, as well as a bus configuration, to obtain maximum throughput, taking into account communications and the constraints above.

An efficient software environment to balance processor loads is being developed. This algorithm uses estimates of the computation time and communication time for the individual atomic tasks to optimally partition the program onto the processors. First, pipelining at the block level will be performed. Then, within the sub-block, parallelism or further pipelining can be used to exploit the finer grain

---

<sup>7</sup>The auto scheduling algorithm and its implementation is being studied by P. Hoang.

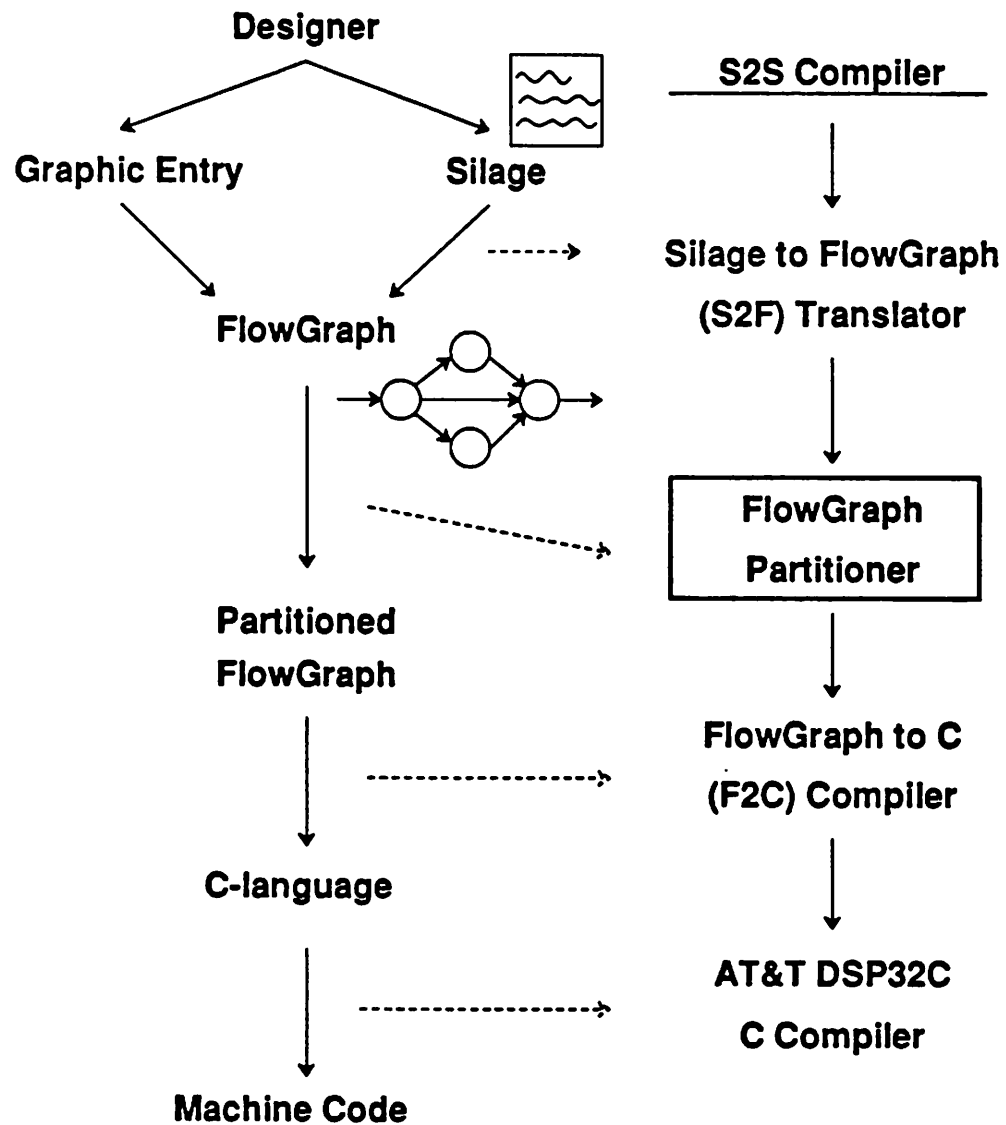


Figure 5.4: Silage to SMART Compiler.

concurrency. Initial results from the Flowgraph Partitioning are very good, although more intelligent heuristics are being investigated to make the algorithm more robust.

The S2F Translator generates a flowgraph that is hierarchical in that a node may represent an instance of a subgraph. C is chosen as an intermediate language because the DSP32C has a C compiler available, allowing us to concentrate on the partitioning strategy. The S2F Translation is near completion. The F2C Translation has not been implemented, and the last task will be done using AT&T's C Compiler for DSP32C.



## Chapter 6

# CONCLUSIONS

*Take yesterday's worries and sort them all out  
And you'll wonder whatever you worried about.  
Look back, at the cares that once furrowed your brow,  
I fancy you'll smile at most of them now.  
They seemed terrible then, but they really were not,  
For once out of the woods, all the fears are forgot.  
- Author Unknown*

SMART is a multiprocessor architecture optimized for real-time behavioral simulation of Digital Signal Processing systems. The first prototype, currently in operation, contains 10 processors (peak 200 MFLOPS).<sup>1</sup>

The SMART system features a *Configurable Bus* and a *Bypass Unit* to trade off overall communication bandwidth and latency by taking advantage of the local communication between processors. The system performance is further improved by a *Distributed Shared Memory* system which let the communication latency overlap with the computation time of the processors. *Barriers, locks and events*, which are three types of commonly used synchronization mechanisms, are supported by hardware to minimize the synchronization overhead. The software benchmark analysis have demonstrated that the SMART architecture has achieved the low communication and synchronization overhead.

---

<sup>1</sup>The prototype version is currently working at lower speed. The peak performance is 120 MFLOPS.

In a SMART simulation environment, the designer can describe the algorithms using a high level language: C and Silage. The C programming environment, which requires a manual partitioning of the program, is currently available. A high level software system, based on Silage, is under development to auto-schedule the algorithmic description to the SMART processor array with a balanced loading and an efficient usage of the communication system. Performance of the actual SMART system was measured for typical DSP programs using floating-point operations. The measurement shows an average speedup of 76 times over SUN 3/60 with MC 68881 co-processor and a speedup of 29 times over SUN SPARC Station 1 with a floating-point co-processor.

Designing the SMART system has involved many topics, such as architectural design, functional organization, logic design and various levels of physical implementation. The implementation encompasses VLSI design, printed circuit board design, software development, and system integration of the hardware and the software. With the help of the recent developments in the CAD technology and the design methodologies, the SMART system has been designed and implemented in a two and half year period by two graduate students.

A functional description was used to develop architectural concepts, to produce an architectural specification and to document the system for maintenance purposes. The whole SMART processor array was modeled and simulated before it was physically implemented. The design methodology reduced the efforts to debug the physical system dramatically.

In retrospect, there were several important design decisions which had an impact on the performance and the design time. Perhaps the most aggressive design decision we have made was to utilize the dynamic circuit design concept on the printed circuit board level. It not only provided the high speed but also saved the number of pins which were very expensive resources in our VLSI chip design. Subsequent testing of the design proved that it was unreliable in the presence of large noise. Therefore we had to switch to a static design.

Another important design decision was the selection of a core processor. We chose the DSP32C which was the most advanced DSP microprocessor at that

time. As the technology advances, however, the performance has been excelled by more powerful processors, such as the Texas Instruments TMS320C30, the Motorola DSP96002, and the Intel i860.

As a future research topic, we propose a method to integrate a variety of processors and ASIC chips, to the existing SMART system – a heterogeneous simulation environment – with a minimum design effort. That is, replace the processor and reprogram the programmable-chips, while using the same printed circuit board.

Since every processor provides a memory interface which does not vary a lot among processors, it is possible to design an access controller chip set which can be used for various types of processors. Once the processor is interfaced to the access controller, it can communicate, synchronize, and access memory through the access controller. On the other hand, a plug-in daughter board may be used to integrate a new processor whose pin-out does not match the socket implemented on the printed circuit board.

The I/O bandwidth of the system can also benefit from the heterogeneous environment. As discussed in Section 5.3, the I/O rate was basically limited by the DSP32C processor itself. Therefore I/O processors or application specific processors can be employed to handle high bandwidth real-time I/O. Once the data is brought into the system, the configurable bus can handle the high bandwidth communication.

Several other interesting ideas and alternatives can be suggested to improve the performance of the SMART system. Alternative designs in internal clock generation and distributed share memory were discussed in Chapter 3. Asynchronous bus design, and its impact on system extensibility and speed, needs to be studied. On the other hand, a self-reconfigurable bus, which was described in Section 2.3.4, can improve the performance and relieve programmers from configuring the buses. A hierarchical *distributed memory system* can be employed to simulate memory-hungry algorithms such as a speech recognition system [4].

# Bibliography

- [1] B. S. Atal M. Shroeder. Code excited linear prediction (celp): High quality speech at very low rates. In *Internaltional Conference on ASSP*, March 1985. pp. 937-940.
- [2] R. R. Shively A. L. Gorin. A reconfigurable fault-tolerant systolic signal processor. In *ICASSP'89*, 1989. pp. 2397-2400.
- [3] C. Sechen A. Sangiovanni-Vincentelli. The timberwolf placement and routing package. *IEEE Journal of Solid State Circuits*, April 1985. Vol. SC-20, No. 2.
- [4] J. Rabaey T. Stoelzle D. Chen S. Narayanaswamy R. W. Brodersen H. Murveit A. Santos. A large vocabulary real time continuous speech recognition system. In *IEEE Workshop on VLSI Signal Processing, III*. IEEE, 1988.
- [5] R. Alverson, T. Blank, K. Choi, A. Salz, L. Soule, and T. Rokicki. THOR user's manual. Technical Report CSL-TR-88-348 and 349, Stanford University, January 1988.
- [6] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, April 1967. pp. 483-485.
- [7] AT&T. *WE DSP32C Digital Signal Processor, Information Manual*, December 1988.
- [8] AT&T. *WE DSP32C Digital Signal Processor, Advance Data Sheet*, May 1989.

- [9] W. Baker, J. Burns, S. Chow, D. Harrison, M. Igusa, C. Kring, T. Laidig, B. Lin, P. Moore, J. Reed, R. Rudell, C. Sechen, R. Segal, R. Spickelmier, A. Wang, A. R. Newton, and A. Sangiovanni-Vincentelli. OCT tools distribution 2.0. Technical report, University of California at Berkeley, Electronics Research Lab, November 1987.
- [10] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison Wesley, 1990.
- [11] B. Browning et al. *The Tree Machine: A highly Concurrent Computing Environment*. PhD thesis, Caltech Institute of Technology, January 1980.
- [12] H. T. Kung C. E. Leiserson. Systolic arrays for vlsi. Technical Report CS-79-103, Carnegie-Mellon University, April 1979.
- [13] T. P. Barnwell III D. A. Schwartz. Optimal implementation of flow graphs on synchronous multiprocessors. In *Asilomar Conference on Circuits and Systems*, November 1983.
- [14] L. N. Bhuyan Q. Yang D. P. Agrawal. Performance of multiprocessor interconnection networks. *IEEE Computer*, February 1989.
- [15] S. J. A. Mcgrath T. P. Barnwell D. Schwartz. A we-dsp32 based, low-cost, high-performance, synchronous multiprocessor for cyclo-static implementations. In *ICASSP'87*, 1987. pp. 1899-1902.
- [16] A. Dinning. A survey of synchronization methods for parallel computers. *IEEE Computer*, July 1989. Vol. 22, No. 7, pp. 66-77.
- [17] R. Duncan. A survey of parallel computer architectures. *IEEE Computer*, February 1990. pp. 5-16.
- [18] R. K. Tummala E. J. Rymazewski. *Microelectronics Packaging Handbook*. Van Nostrand Reinhold, 1989.

- [19] Electronic Research Laboratory, University of California, Berkeley. *LagerIV Silicon Assembly System Manual*, Distribution 2.0 edition, 1989.
- [20] B. Marc et al. Ontwikkeling van een digitale echocanceller chip voor toepassing in een handenvrij telefoontoestel. Technical report, Katholieke Universiteit Leuven, 1987.
- [21] M. Annaratone et al. Warp architecture: From prototype to production. *Proceedings of the 1987 National Computer Conference*, June 1987. Chicago, Illinois.
- [22] M. Annaratone et al. Warp architecture and implementation. *Conference Proceedings of the 13th Annual International Symposium on Computer Architecture*, November 1988. Tokyo, Japan.
- [23] M. L. Fuccio et al. The dsp32c: At&t's second-generation floating point digital signal processor. *IEEE Micro*, December 1988. Vol. 8, No. 6.
- [24] R. J. Sluyter et al. A novel method for pitch extraction from speech and a hardware model applicable to vocoder systems. In *International Conference on Acoustics, Speech and signal Processing*, pages pp. 45-48. IEEE, 1980.
- [25] W. H. Press et al. *Numerical Recipes in C*. Cambridge University Press, 1988.
- [26] K. Hwang F. A. Briggs. *Computer Architecture and Paralle Processing*. McGraw-Hill, 1988.
- [27] K. A. Frenkel. Hdtv and the computer industry. *Communications of the ACM*, November 1989.
- [28] J. Gaudiot. Data-driven multicomputers in digital signal processing. *Proceedings of the IEEE*, September 1987. Vol. 75, No. 9, pp. 1220-1233.
- [29] Heurikon Corporation. *Heurikon HK68/V20 User's Manual*, may 1988.
- [30] P. Hilfinger. Silage, a high level language and silicon compiler for digital signal processing. *Proceedings IEEE CICC Conference*, May 1985. pp. 213-216.

- [31] M. D. Hill. Spur: A vlsi multiprocessor workstation. Technical report, University of California, Berkeley, November 1986. Memorandum No. UCB/CSD M86/273.
- [32] H. J. Hindin. Parallel processing promises faster program execution. *Computer Design*, August 1985.
- [33] Integrated Device Technology. *High Performance CMOS Data Book Supplement*, 1989. pp. S6:14-26.
- [34] Intel Corporation. *iPSC Program Development Guide*, November 1986.
- [35] J. Archibald J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, November 1986. Vol. 4, No. 4.
- [36] R. W. Brodersen J. M. Rabaey, S. P. Pope. An integrated automated layout generation system for dsp circuits. *IEEE Transactions on Computer-Aided Design*, July 1985. Vol. Cad-4, No. 3.
- [37] C. V. Ravishankar J. R. Goodman. Cache implementation for multiple microprocessors. *Proceedings Comcon Spring'83*, February 1983. pp. 346-350.
- [38] W. Koh A. Yeung P. Hoang J. Rabaey. A multiprocessor system for dsp behavioral simulation. In *IEEE Workshop on VLSI Signal Processing, III*. IEEE, 1988.
- [39] W. Koh A. Yeung P. Hoang J. Rabaey. A configurable multiprocessor system for dsp behavioral simulation. In *Proceedings of International Symposium on Circuits and Systems*. IEEE, 1989.
- [40] M.S. Papamarcos J.H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *Proc. Eleventh International Symposium on Computer Architecture*, June 1984. pp. 348-359.
- [41] G. R. L. Sohie K. L. Kloker. A digital signal processor with ieee floating-point arithmetic. *IEEE Micro*, December 1988. Vol. 8, No. 6.

- [42] J. A. Stankovic K. Ramamritham. *Hard Real-Time Systems*. IEEE Computer Society Press, 1988.
- [43] J. Keller. Power and ground requirements for a high-speed 32 bit computer chip set. Technical report, University of California, Berkeley, August 1985.
- [44] H. T. Kung. Why systolic architectures. *Computer*, January 1982. pp. 37-46.
- [45] H. T. Kung. *VLSI Array Processors*. Prentice Hall, 1988. pp. 5.
- [46] S. A. Dyer L. R. Morris. Floating-point digital signal processing chips, a new era for dsp systems design? *IEEE Micro*, December 1988. Vol. 8, No. 6.
- [47] L. Snyder. Overview of the chip computer. In *VLSI 81*. Academic Press, 1981.
- [48] E. A. Lee. Programmable dsp architectures: Part i. *IEEE ASSP Magazine*, October 1988.
- [49] E. A. Lee. Architectures for statically scheduled dataflow. Technical report, University of California, Berkeley, December 1989. Memorandum No. UCB/ERL M89/129.
- [50] E. A. Lee. Programmable dsp architectures: Part ii. *IEEE ASSP Magazine*, January 1989.
- [51] S. Lee. Automatic floorplanning techniques for macrocell-based layouts. Master's thesis, University of California, Berkeley, 1989.
- [52] M. Leonard. Digital signal processors. *Electronics Design*, November 1988.
- [53] Motorola. *MECL System Design Handbook*, Rev 1 edition, 1988.
- [54] D. Murphy. Balancing act. *High Performance Systems*, October 1989. pp. 76-83.
- [55] NCUBE Corporation. *NCUBE handbook*, 1986. version 1.1.
- [56] T. S. Perry. Million-transistor microchip. *IEEE Spectrum*, April 1989.



- [57] PRINTEX. *The VMEbus Specification*, Rev C.1 edition, October 1985.
- [58] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987.
- [59] B. W. Arden R. Ginosar. Mp/c: A multiprocessor architecture/ computer architecture. *IEEE Transactions on Computer*, May 1982. Vol. C-31, No. 5.
- [60] D. K. Jeong G. Borriello D. A. Hodges R. H. Katz. Design of pll-based clock generation circuits. *Journal of Solid-State Circuits*, April 1987.
- [61] L. H. Jamieson D. B. Gannon R. J. Douglass. *The Characteristics of Parallel Algorithms*. The MIT Press, 1987.
- [62] A. L. Gorin R. R. Shively. The aspen parallel computer, speech recognition and parallel dynamic programming. In *ICASSP'87*, 1987. pp. 976-979.
- [63] P. Papamichalis R. Simar, Jr. The tms320c30 floating-point digital signal processor. *IEEE Micro*, December 1988. Vol. 8, No. 6.
- [64] D. C. Chen R. Yu J. Rabaey R. W. Brodersen. A vlsi grammar processing subsystem for a real time large vocabulary continuous speech recognition system. In *CCIC'90*, May 1990.
- [65] Racal-Redac, Inc. *Visula, User's Guide to PCB Design*, May 1988. Vol. I & II.
- [66] H. K. Reghbati. *Tutorial: VLSI Testing & Validation Techniques*. IEEE, 1985.
- [67] E. A. Lee S. Ha. Scheduling strategies for multiprocessor real-time dsp. *GLOBECOM*, November 1989.
- [68] L. Kohn S. W. Fu. A 1 000 000 transistor microprocessor. *1989 International Solid-State Circuits Conference Digest of Technical Papers*, February 1989. pp. 54-55.
- [69] C. M. Sechen. *Placement and Global Routing of Integrated Circuits Using Simulated Annealing*. PhD thesis, University of California, Berkeley, December 1987.

- [70] C. L. Seitz. The cosmic cube. *Communications of the ACM*, January 1985. Vol. 28, No. 1, pp. 22-33.
- [71] Sequent Computer Systems, Inc. *Guide to Parallel Programming on Sequent Computer Systems*, 1986.
- [72] L. Snyder. Introduction to the configurable, highly parallel computer. *IEEE Computer*, January 1982.
- [73] M. B. Srivastava. Automatic generation of cmos data paths in lager framework. Master's thesis, University of California, Berkeley, 1987.
- [74] D. Schwartz T. P. Barnwell. Cyclo-static multiprocessor scheduling for the optimal realization of shift-invariant flow graphs. In *ICASSP'85*, March 1985.
- [75] Thinking Machines Corporation. *Introduction to Data Level Parallelism*, April 1986.
- [76] Thinking Machines Corporation. *Background Information*, 1987.
- [77] Thinking Machines Corporation. *Connection Machine Model CM-2 Technical Summary*, April 1987.
- [78] Wind River Systems, Inc. *VzWorks, Target-Specific Documentation, Heurikon HK68/V2F*, 1988. Vol. I & II.
- [79] A. K. W. Yeung. Vlsi implementation of a configurable multiprocessor system for dsp behavioral simulation. Master's thesis, University of California, Berkeley, December 1989.

# Appendix A

## MAC Instructions

The DSP32C issues instructions to MAC by performing external memory accesses. MAC figures out the instruction by decoding the address bus according to an address map.

In the DSP32C, the entire address space is divided into 2 banks: bank 0 and bank 1. In one instruction, concurrent access to two different memory banks is allowed. In SMART, the Mode 7 Memory Configuration (ROM-less version) is chosen (Table A.1).

As explained in Section 3.4, MAC instructions can be broadly categorized into two groups. Instructions in Group 1 are all mapped to external memory A and instructions in Group 2 to external memory B. Address mapping for the entire address space is shown in Table A.2, Table A.3 and Table A.4.  $\text{Adr}[12-0]$  is the external word address bus and  $\text{Msn}[3-0]$  is the byte select of the DSP32C. By asserting one or more of the byte select bits, the DSP32C can selectively access one or more bytes of the 32-bit word. All instructions in Group 1 are located in Table A.2 and instructions in Group 2 in Table A.3. Table A.4 shows address mapping for memory bank 1. All accesses to this memory bank are internal to the DSP32C.

Memory Bank	Byte Address	Memory Assignment
BANK 0	0x000000 0x0007FF	RAM0
	0x000800 0x5FFFFFFF	External Memory A
	0x600000 0xFFDFFF	External Memory B
BANK 1	0xFFE000 0xFFE7FF	RAM2
	0xFFE800 0xFFF7FF	RESERVED
	0xFFF800 0xFFFFFF	RAM1

Table A.1: DSP32C Memory Configuration, Mode 7 (ROM-less version).

A description of Group 1 instructions (Without Acknowledge) is given below.

1. External Local Memory Read/Write (*lmemRd/lmemWr*):

MAC responds by asserting chip enable signals for the local memory (LMEM). *Adr*[12-0] and *Msn*[3-0] specify the address at which the data are to be read or written.

2. Synchronization Switch Write (*wrSyncSwitch*):

The lower order 5 bits of DSP32C address bus are written into the *syncSwitch* configuration registers which controls the switches on the synchronization bus.

3. Synchronization Disable Write (*wrSyncDisable*):

The lower order 6 bits of DSP32C address bus are written into the *syncDisable* configuration registers which enable or disable the corresponding synchronization patterns.

4. Semaphore Unlock (*semV*):

MAC releases the semaphore.

5. Switch1 Write (*wrSwitch*):

DSP32C Word Address - Byte Select	
	A M M M M
	d s s s s
	r n n n n
	2 2 1 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0-3 2 1 0
	1 0 9 8 7 6 5 4 3 2 1 0
0x000000 Start	0 0-0 0 0 0
On-chip Ram (2KB)	
0x0007FF End	0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1-1 1 1 1
0x000800 Start	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0-0 0 0 0
External Local Memory (256KB)	
0x0407FF End	0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1-1 1 1 1
0x040800 Start	0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0-0 0 0 0
Reserved for Future Expansion of Local Memory (254KB)	
0x07FFFF End	0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1-1 1 1 1
0x080000 Start	0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0-0 0 0 0
Special Instructions (I)	
wrSyncSwitch	0 0 0 0 1 0 0 x x x x x x x x x x u u u u u-x x x x
wrSyncDefault	0 0 0 0 1 0 1 0 x x x x x x x x x x u u u u u u-x x x x
semV	0 0 0 0 1 0 1 1 0 0 x x x x x x x x x x x-x x x x x
wrSwitch	0 0 0 0 1 0 1 1 0 1 u x x x x x x x x x x-x x x x x
wrSwap	0 0 0 0 1 0 1 1 1 0 u x x x x x x x x x x-x x x x x
wrSwapToggle	0 0 0 0 1 0 1 1 1 1 x x x x x x x x x x x-x x x x x
wrByPassEn	0 0 0 0 1 1 0 0 0 0 u x x x x x x x x x x-x x x x x
wrSwitch2	0 0 0 0 1 1 0 0 0 1 u x x x x x x x x x x-x x x x x
wrProcNum	0 0 0 0 1 1 0 0 1 0 x x x x x x u u u u u u-x x x x
decode	0 0 0 0 1 1 0 0 1 1 x x x x x x x x x x x-x x x x x
0x0FFFFFF End	0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1-1 1 1 1
0x100000 Start	0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0-0 0 0 0
Front-side Memory Access (Read or Write)	
	0 0 0 1 x x x x x s s A A A A A A A A A A A-A A A A A
0x1FFFFFF End	0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1-1 1 1 1
0x200000 Start	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0-0 0 0 0
Bus-side Memory Broadcast (Write Only)	
	0 0 1 p p p p p p s s A A A A A A A A A A A-A A A A A
0x3FFFFFF End	0 0 1-1 1 1 1
0x400000 Start	0 1 0-0 0 0 0
Bus-side Memory Access (Write Only)	
	0 1 0 p p p p p p s s A A A A A A A A A A A-A A A A A
0x5FFFFFF End	0 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1-1 1 1 1
LEGENDS: - refer the following page	

Table A.2: Address Mapping for Memory Bank 0 (Group 1 Instructions).

DSP32C Word Address - Byte Select	
	A M M M M
	d s s s s
	r n n n n
	2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0-3 2 1 0
	1 0 9 8 7 6 5 4 3 2 1 0
0x600000 Start	0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0-0 0 0 0
BusSide Memory on Bus Side with Wait	
	0 1 1 p p p p p p s s A A A A A A A A A A A-A A A A
0x7FFFFFF End	0 1-1 1 1 1
0x800000 Start	1 0-0 0 0 0
Special Instructions (II)	
synchronization	1 0 0 0 x x x 0 0 0 0 0 0 0 0 0 0 u u u u u u-x x x x
semP	1 0 0 0 x x x 1 x x x x x x x x x x x-x x x x
0x9FFFFFF End	1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1-1 1 1 1
0xA00000 Start	1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0-0 0 0 0
Broadcast with Wait	
	1 0 1 p p p p p p s s A A A A A A A A A A A-A A A A
0xBFFFFFF End	1 0 1-1 1 1 1
0xC00000 Start	1 1 0-0 0 0 0
Circular ByPass with Wait	
	1 1 0 p p p p p p s s A A A A A A A A A A A-A A A A
0xDFFFFFF End	1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1-1 1 1 1
0xE00000 Start	1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0-0 0 0 0
Fifo Read/Write with Wait	
0xFFDFFF End	1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1-1 1 1 1
<b>LEGENDS:</b>	
x: Don't Care	
u: User Programmed Bits	
s: Dual-port Memory Bank Select Bits	
A: Byte Address of Dual-port Memory	
p: Processor Id of Destination Processor	

Table A.3: Address Mapping for Memory Bank 0 (Group 2 Instructions).

DSP32C Word Address - Byte Select	
	A M M M M
	d s s s s
	r n n n n
	2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0-3 2 1 0
	1 0 9 8 7 6 5 4 3 2 1 0
0xFFE000 Start	1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0-0 0 0 0
On-chip Ram (2KB)	
0xFFE7FF End	1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1-1 1 1 1
0xFFE800 Start	1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 0-0 0 0 0
Reserved (4KB)	
0xFFF7FF End	1 1 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1-1 1 1 1
0xFFF800 Start	1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0-0 0 0 0
On-chip Ram (2KB)	
0xFFFFF End	1 1-1 1 1 1

Table A.4: Address Mapping for Memory Bank 1 (Internal Access Only).

Bit 11 of the DSP32C address bus is written into the *switchState* configuration register which controls the switches on the reconfigurable bus.

6. Swap State Write (wrSwap):

Bit 11 of the DSP32C address bus is written into the *swapState* configuration register which controls the bank select of DPRAM access.

7. Swap State Toggle (toggleSwap):

Toggle the *swapState* configuration register.

8. Bypass Enable Write (wrBypassEn):

Bit 11 of the DSP32C address bus is written into the *bypassEn* configuration register which enables bypass operation.

9. Switch2 Write (wrSwitch2):

Bit 11 of the DSP32C address bus is written into the *switchState2* configuration register.

10. PID Write (wrPID):

The lower order 6 bits of DSP32C address bus are written into the *pidReg* configuration registers which contains the Processor Identification Number of local processor.

11. Interrupt Generate (*genIntr*):

The single-phase positive pulse is generated at the *interrupt* output of MAC. This signal can be used to generate an interrupt to the VME interface, thus providing the DSP32C the ability to interrupt the VME bus master.

12. Processor-port DPRAM Read/Write (*ppmemRd/ppmemWr*):

MAC responds by asserting chip enable signals for the processor-port of the dual-ported RAM (DPRAM).

*Adr*[12-11] together with the *swapState* specify the memory bank to be accessed. *Adr*[10-0] and *Msn*[3-0] specify the address at which the data are to be read or written.

13. Slave-port DPRAM Broadcast Write (*spmemBcWr*):

A write request is broadcasted to the slave-ports of groups of processors specified by the *pid* of the destination processor given in *Adr*[18-13]. *Adr*[12-11] together with the *swapState* specify the memory bank to be accessed. *Adr*[10-0] and *Msn*[3-0] specify the address at which the data are to be read or written.

14. Slave-port DPRAM Write (*spmemWr*):

A write request is made to the slave-port of a destination processor whose *pid* is given by *Adr*[18-13]. *Adr*[12-11] together with the *swapState* specify the memory bank to be accessed. *Adr*[10-0] and *Msn*[3-0] specify the address at which the data are to be read or written.

Following is a description of Group 2 instructions (With Acknowledge):

1. Slave-port DPRAM Slow Read/Write (*spmemSWr/spmemSRd*):

Same as *spmemWr* except that MAC generates an acknowledge and both read and write requests are allowed.



**2. Synchronization (syncV):**

Any *one's* in the lower order 6 bits of DSP32C address bus activate the corresponding synchronization patterns. MAC generates an acknowledge when every member of the processor set defined by the activated synchronization pattern is *synchronized*.

**3. Semaphore Request (semP):**

MAC makes a request to acquire the semaphore. After the semaphore is obtained, the semaphore is automatically *locked* and an acknowledge is generated.

**4. Slave-port DPRAM Broadcast Slow Write (spmemBcSWr):**

Same as spmemBcWr except that MAC generates an acknowledge.

**5. Slave-port DPRAM Circular Bypass Slow Write (spmemCBcSWr):**

Same as spmemBcSWr except that circular bypass is enabled.

**6. External FIFO Read/Write (effRd/effWr):**

A read(write) request is made to the external input(output) FIFO. An acknowledge is generated upon the completion of the access. The *empty(full) flag* of the FIFO is checked prior to the access to prevent underflow(overflow) of the FIFO.

## Appendix B

### MACRO Definitions

In this chapter, the macros which were described in Section 5.2 are *defined* in *smart.h* header file.

New macros can be defined, too. Those macros are used in *image.c* program and are listed in *smart.h* [Appendix-C].

```
/*      smart.h      */
#define BSEL_0      0x000000
#define BSEL_1      0x002000
#define BSEL_SWAP   0x004000
#define BSEL_SWAPB  0x006000

/* Bus-side Memory Access Mode */
#define S_BC        0x200000
#define S_W         0x400000
#define L_RW        0x600000
#define L_BC        0xa00000
#define L_CW        0xc00000

#define TRUE      1
#define FALSE     0

/* Max number of processors in the system */
#define MAX_PROC  10
```

## APPENDIX B. MACRO DEFINITIONS

119

```

/* Max number of processors allowed to be on one shared bus */
#define MAX_PROC_GROUP 2

/* 64KB (not counting on-chip ram) */
/* TM_SIZE is in WORDS */
#define TM_SIZE 0x004000
/* 192 KB */
/* LM_SIZE is in WORDS */
#define LM_SIZE 0x00c000

/* 2K Words or 8KB per bank */
/*
#define DP_SIZE 0x800
*/
#define DP_SIZE 0x800

/* 1K Words or 4KB */
#define FIFO_SIZE 0x400

/* Max number of local sync bits */
#define MAX_LSYNC 5
#define LSYNC_MASK 0x0000003f
#define GSYNC_MASK 0x00000001

/*****
*
*          MACRO definitions (Version 1.0)
*
*          *****/

#define Sm_wrSyncSwitch(val) (*(int *) (0x080000 + 4*(val)) = 0)
/* val = 0..31 */
#define Sm_wrSyncDefault(val) (*(int *) (0x0a0000 + 4*(val)) = 0)
/* val = 0..63 */
#define Sm_semV() (*(int *) 0x0b0000 = 0)
#define Sm_openSwitch() (*(int *) 0x0b4000 = 0)
#define Sm_closeSwitch() (*(int *) 0x0b8000 = 0)

```

```
#define Sm_clrSwap() (*(int *)0x0b8000 = 0)
#define Sm_setSwap() (*(int *)0x0ba000 = 0)
#define Sm_toggleSwap() (*(int *)0x0bc000 = 0)
#define Sm_disableBypass() (*(int *)0x0c0000 = 0)
#define Sm_enableBypass() (*(int *)0x0c2000 = 0)
#define Sm_openSwitch2() (*(int *)0x0c4000 = 0)
#define Sm_closeSwitch2() (*(int *)0x0c6000 = 0)
#define Sm_wrProcNum(val) (*(int *) (0x0c8000 + 4*(val)) = 0)
/* val = 0..63 */
#define Sm_decode() (*(int *)0x0cc000 = 0)
#define Sm_syncV(val) (*(int *) (0x800000 + 4*(val)) = 0)
/* val = 0..63 */
/* global sync is the LSB bit */
#define Sm_semP() (*(int *)0x810000 = 0)
#define Sm_fifo_fl() (*(float *)0xf40000)
#define Sm_fifo_int() (*(int *)0xf40000)

/* Memory */
#define Sm_tm(addr) (*(int *) (0x000800 + 4*(addr)))
#define Sm_lm(addr) (*(float *) (0x000800 + 4*(TM_SIZE+(addr))))
#define Sm_fm(bsel, addr) (*(float *) (0x100000 + (bsel) + (4*(addr))))
#define Sm_bm(mode, pid, bsel, addr) \
    (*(float *) (mode + (pid << 15) + (bsel) + (4*(addr))))

#define Sm_lm_addr(addr) (float *) (0x000800 + 4*(TM_SIZE+(addr)))
#define Sm_fm_addr(bsel, addr) (float *) (0x100000 + (bsel) + (4*(addr)))
#define Sm_bm_addr(mode, pid, bsel, addr) \
    (float *) (mode + (pid << 15) + (bsel) + (4*(addr)))
```

# Appendix C

## IMAGE.C Program

```
"smart.h"
```

```
/* smart.h */
```

```
#define BSEL_0    0x000000  
#define BSEL_1    0x002000  
#define BSEL_SWAP 0x004000  
#define BSEL_SWAPB 0x006000
```

```
/* Bus-side Memory Access Mode */
```

```
#define S_BC      0x200000  
#define S_W       0x400000  
#define L_RW      0x600000  
#define L_BC      0xa00000  
#define L_CW      0xc00000
```

```
#define TRUE      1  
#define FALSE     0
```

```
#define INT_TYPE   0  
#define FLOAT_TYPE 1  
#define HEX_TYPE   2
```

```
/* Max number of processors in the system */
```

```
#define MAX_PROC 10
```

```
/* Max number of processors allowed to be on one shared bus */
#define MAX_PROC_GROUP 5

/* 64KB (not counting on-chip ram) */
/* TM_SIZE is in WORDS */
#define TM_SIZE 0x004000
/* 192 KB */
/* LM_SIZE is in WORDS */
#define LM_SIZE 0x00c000

/* 2K Words or 8KB per bank */
/*
#define DP_SIZE 0x800
*/
#define DP_SIZE 0x800

/* 1K Words or 4KB */
#define FIFO_SIZE 0x400

/* Max number of local sync bits */
#define MAX_LSYNC 5
#define LSYNC_MASK 0x0000003f
#define GSYNC_MASK 0x00000001

#define Sm_wrSyncSwitch(val) (*(int*)(0x080000 + 4*(val)) = 0)
/* val = 0..31 */
#define Sm_wrSyncDefault(val) (*(int*)(0x0a0000 + 4*(val)) = 0)
/* val = 0..63 */
#define Sm_semV() (*(int*)0x0b0000 = 0)
#define Sm_openSwitch() (*(int*)0x0b4000 = 0)
#define Sm_closeSwitch() (*(int*)0x0b8000 = 0)
#define Sm_clrSwap() (*(int*)0x0b8000 = 0)
#define Sm_setSwap() (*(int*)0x0ba000 = 0)
#define Sm_toggleSwap() (*(int*)0x0bc000 = 0)
#define Sm_disableBypass() (*(int*)0x0c0000 = 0)
#define Sm_enableBypass() (*(int*)0x0c2000 = 0)
```

```
#define Sm_openSwitch2() (*(int *)0x0c4000 = 0)
#define Sm_closeSwitch2() (*(int *)0x0c6000 = 0)
#define Sm_wrProcNum(val) (*(int *)(0x0c8000 + 4*(val)) = 0)
/* val = 0..63 */
#define Sm_decode() (*(int *)0x0cc000 = 0)
#define Sm_syncV(val) (*(int *)(0x800000 + 4*(val)) = 0)
/* val = 0..63 */
/* global sync is the LSB bit */
#define Sm_semP() (*(int *)0x810000 = 0)
#define Sm_fifo_int() (*(int *)0xf40000)
#define Sm_fifo_fl() (*(float *)0xf40000)

/* Memory */
#define Sm_tm(addr) (*(int *)(0x000800 + 4*(addr)))
#define Sm_lm(addr) (*(float *)(0x000800 + 4*(TM_SIZE+(addr))))
#define Sm_fm(bsel, addr) (*(float *)(0x100000 + (bsel) + (4*(addr))))
#define Sm_bm(mode, pid, bsel, addr) \
    (*(float *)(mode + (pid << 15) + (bsel) + (4*(addr))))

#define Sm_lm_addr(addr) (float *)(0x000800 + 4*(TM_SIZE+(addr)))
#define Sm_fm_addr(bsel, addr) (float *)(0x100000 + (bsel) + (4*(addr)))
#define Sm_bm_addr(mode, pid, bsel, addr) \
    (float *)(mode + (pid << 15) + (bsel) + (4*(addr)))
```

“sl.h”

```
#define sl_swapB(i) ((short int)(((i)<<8) & 0xff00 | (((i)>>8) & 0x00ff) ))
#define MASK 0xff9fff
#define ADD 0x002000
#define sl_inc_mask(i)((unsigned char *) (((int) i + 1) & MASK))
#define sl_add_mask(i,a)((unsigned char *) (((int) i + (int) a) & MASK))
#define sl_sub_mask(i,a)((unsigned char *) (((int) i + ADD - (int) a) & MASK))
#define sl_i2c(int_short,char_l,char_h){ \
    char_l = (unsigned char) ((int_short) & 0xff); \
    char_h = (unsigned char) (((int_short) >> 8) & 0xff); }
#define sl_c2i(i){ (int) (i) }
#define Sm_fifo_sint() (*(short int *) (0xf40000))
#define Sm_fm_sint(bsel, addr) (*(short int *) (0x100000 + (bsel) + (2*(addr))))
#define Sm_bm_sint(mode, pid, bsel, addr) \
    (*(short int *) (mode + (pid << 15) + (bsel) + (2*(addr))))

int pid, process;
int wbegin, wend;
int isrc, idst;
```



"image.h"

```

struct rasterfile {
    short int    ras_magic_h;    /* magic number */
    short int    ras_magic_l;    /* magic number */
    short int    ras_width_h;    /* width (pixels) of image */
    short int    ras_width_l;    /* width (pixels) of image */
    short int    ras_height_h;   /* height (pixels) of image */
    short int    ras_height_l;   /* height (pixels) of image */
    short int    ras_depth_h;    /* depth (1, 8, or 24 bits) of pixel */
    short int    ras_depth_l;    /* depth (1, 8, or 24 bits) of pixel */
    short int    ras_length_h;   /* length (bytes) of image */
    short int    ras_length_l;   /* length (bytes) of image */
    short int    ras_type_h;     /* type of file; see RT_* below */
    short int    ras_type_l;     /* type of file; see RT_* below */
    short int    ras_mapttype_h; /* type of colormap; see RMT_* below */
    short int    ras_mapttype_l; /* type of colormap; see RMT_* below */
    short int    ras_maplength_h; /* length (bytes) of following map */
    short int    ras_maplength_l; /* length (bytes) of following map */
    /* color map follows for ras_maplength bytes, followed by image */
};

#define RAS_MAGIC_H    0x59a6
#define RAS_MAGIC_L    0x6a95

    /* Sun supported ras_type's */
#define RT_OLD        0    /* Raw pixrect image in 68000 byte order */
#define RT_STANDARD    1    /* Raw pixrect image in 68000 byte order */
#define RT_BYTE_ENCODED 2    /* Run-length compression of bytes */
#define RT_EXPERIMENTAL 0xffff /* Reserved for testing */

    /* Sun registered ras_mapttype's */
#define RMT_RAW        2

    /* Sun supported ras_mapttype's */
#define RMT_NONE        0    /* ras_maplength is expected to be 1 */
#define RMT_EQUAL_RGB    1    /* red[ras_maplength/3],green[],blue[] */

```

```
#define WHITE 255
#define BLACK 0
#define PI 3.141592654
#define TAP 12

static int reversed, LUTinversed;
static int do_med5, do_hist, do_skel, do_greymod, do_qroj, do_bloat;
do_proj, do_thresh, do_recog, draw_wind, do_rev, do_lheq;
draw_col, do_sobel, do_inverse, do_detangle, do_detpos;
rprt_thr, auto_thr, do_rats, do_roberts, do_sel_range;
do_laplace, do_lapel, do_min, do_max, do_nlheq;

static unsigned char *source, *destination, *tmpstor;
static unsigned char *src, *dst;

static int src_width, src_height, src_length, dst_length,
orig_src_width, orig_src_height,
dst_width, dst_height;
static int l_wind, r_wind, t_wind, b_wind,
window_chg, black_val, white_val;
static int thresh, scndthr, bloat_iter, lapel_thr, lo_thr, hi_thr;
static float angle;

char *src_name, *dst_name, *hist_name, *proj_name;
unsigned char LUT[256];

struct rasterfile src_header;
char *src_fp, *dst_fp, *hist_fp, *proj_fp;
```

```
"image.c"
```

```
#include "smart.h"
```

```
#include "sl.h"
```

```
#include "image.h"
```

```
Init()
```

```
{
```

```
    pid = Sm_fifo_int();
```

```
    Sm_fifo_int() = pid;
```

```
    (Sm_fifo_sint() == 0) ? Sm_openSwitch() : Sm_closeSwitch();
```

```
    /* initial system configuration */
```

```
    Sm_wrSyncSwitch(0x00);
```

```
    Sm_wrSyncDefault(0xff);
```

```
    Sm_enableBypass();
```

```
    Sm_wrProcNum(pid);
```

```
    /* process selection */
```

```
    /* source and destination address */
```

```
    /* set number of wait states at the beginning and the end */
```

```
    process = Sm_fifo_int();
```

```
    isrc = Sm_fifo_int();
```

```
    idst = Sm_fifo_int();
```

```
    wbegin = Sm_fifo_int();
```

```
    wend = Sm_fifo_int();
```

```
}
```

```
InitVar()
```

```
{
```

```
    int i,j, k;
```

```
    i = 0;
```

```
    src_header.ras_magic_h = Sm_fm_sint(1,i++);
```

```
    src_header.ras_magic_l = Sm_fm_sint(1,i++);
```

```
    src_header.ras_width_h = Sm_fm_sint(1,i++);
```

```
    src_header.ras_width_l = Sm_fm_sint(1,i++);
```

```

src_header.ras_height_h = Sm_fm_sint(1,i++);
src_header.ras_height_l = Sm_fm_sint(1,i++);
src_header.ras_depth_h = Sm_fm_sint(1,i++);
src_header.ras_depth_l = Sm_fm_sint(1,i++);
src_header.ras_length_h = Sm_fm_sint(1,i++);
src_header.ras_length_l = Sm_fm_sint(1,i++);
src_header.ras_type_h = Sm_fm_sint(1,i++);
src_header.ras_type_l = Sm_fm_sint(1,i++);
src_header.ras_maptype_h = Sm_fm_sint(1,i++);
src_header.ras_maptype_l = Sm_fm_sint(1,i++);
src_header.ras_maplength_h = Sm_fm_sint(1,i++);
src_header.ras_maplength_l = Sm_fm_sint(1,i++);
for(i=0;i<128 ;i++){
    sl_i2c(Sm_fm_sint(1,i+16),LUT[2*i+1],LUT[2*i]);
}
dst_width = src_width = src_header.ras_width_l;
dst_height = src_height =src_header.ras_height_l;
dst_length = src_length = src_width * src_height;
}
EndHeader()
{
    int i,j,k;
    for(i=0;i<(16+128*3);i++){
        Sm_fifo_sint() = Sm_fm_sint(1,i);
    }
}
InitHeader()
{
    /* read src header and broadcast them to other processors */
    /* bank 1, address 0 to 16+256 */
    int i,j,k;
    for(i=0;i<16 ;i++){
        Sm_bm_sint(S_BC,MAX_PROC-1,1,i) = Sm_fifo_sint();
    }
    for(i=0;i<128 ;i++){
        Sm_bm_sint(S_BC,MAX_PROC-1,1,i+16) = Sm_fifo_sint();
    }
}

```

```

    for(i=0;i<128 ;i++){
        Sm_bm_sint(S_BC,MAX_PROC-1,1,i+16+128) = Sm_fifo_sint();
    }
    for(i=0;i<128 ;i++){
        Sm_bm_sint(S_BC,MAX_PROC-1,1,i+16+128+128) = Sm_fifo_sint();
    }
}
Move()
{
    short int tmp ;
    int i=0,j=0;
    unsigned char * src_ptr;
    unsigned char * dst_ptr;

    src_ptr = ( unsigned char *) isrc;
    dst_ptr = ( unsigned char *) idst;
    for(i=0;i < src_height; i++){
        for(j=0;j < src_width; j++){
            *dst_ptr = *src_ptr;
            src_ptr = sl_inc_mask(src_ptr);
            dst_ptr = sl_inc_mask(dst_ptr);
        }
        Sm_syncV(1);
    }
}

MoveOut()
{
    short int tmp ;
    int i=0,j=0;
    unsigned char * ptr;

    ptr = ( unsigned char *) isrc;
    for(i=0;i < src_height; i++){
        for(j=0;j < src_width; j=j+2){
            tmp = ( unsigned int) (*ptr & 0xff);

```

```

    ptr = sl_inc_mask(ptr);
    tmp = (tmp << 8) | (unsigned int) (*ptr & 0xff);
    ptr = sl_inc_mask(ptr);
    Sm_fifo_sint() = tmp;
}
Sm_syncV(1);
}
}

```

```

MoveIn()

```

```

{
    short int tmp ;
    int i=0,j=0;
    unsigned char * ptr;

    ptr = (unsigned char *) idst;
    for(i=0;i < src_height; i++){
        for(j=0;j < src_width; j=j+2){
            tmp = Sm_fifo_sint();
            *ptr = (unsigned char) ((tmp >> 8) & 0xff);
            ptr = sl_inc_mask(ptr);
            *ptr = (unsigned char) (tmp & 0xff);
            ptr = sl_inc_mask(ptr);
        }
        Sm_syncV(1);
    }
}
}

```

```

main()

```

```

{
    /* fifo data will be coming in 16 bits at a time */

    struct rasterfile src_header, dst_header;
    int i,j,k;

```

```
Init();
if(process == 0){
    InitHeader();
    Sm_syncV(1);
    InitVar();
} else if(process == 1){
    Sm_syncV(1);
    InitVar();
    EndHeader();
} else {
    Sm_syncV(1);
    InitVar();
}
while(wbegin--> Sm_syncV(1);
switch(process){
    case 0:
        MoveIn();
        break;
    case 1:
        MoveOut();
        break;
    case 2:
        Move();
        break;
        /*
    case 3:
        LinHistEQ();
        break;
    case 4:
        NonLinHistEQ();
        break;
    case 5:
        GreyMod();
        break;
        */
    case 6:
```

```
    GreyInvert();
    break;
case 7:
    Min();
    break;
case 8:
    Max();
    break;
    /*
case 9:
    Median5();
    break;
    */
case 10:
    Sobel();
    break;
    /*
case 11:
    Roberts();
    break;
case 12:
    Laplace();
    break;
case 13:
    Lapel();
    break;
case 14:
    Histo();
    break;
case 15:
    SelectRange();
    break;
case 16:
    Thresh();
    break;
case 17:
    Rats();
```



```

        break;
    case 18:
        Bloat();
        break;
    case 19:
        Skeleton();
        break;
        /*
    default:
        Sm_fifo_int() = 0x39;
        Move();
        break;
    }
    while(wend--) Sm_syncV(1);
}

GreyInvert()
{
    short int tmp;
    int i=0,j=0;
    unsigned char *src_ptr;
    unsigned char *dst_ptr;

    src_ptr = ( unsigned char *) isrc;
    dst_ptr = ( unsigned char *) idst;

    for(i=0;i < src_height; i++){
        for(j=0;j < src_width; j++){
            *dst_ptr = 255 - *src_ptr;
            src_ptr = sl_inc_mask(src_ptr);
            dst_ptr = sl_inc_mask(dst_ptr);
        }
        Sm_syncV(1);
    }
}

Min()
{

```

```

unsigned char *src_ptr, *dst_ptr;
int i,j;

src_ptr = ( unsigned char *) isrc;
dst_ptr = ( unsigned char *) idst;

for(i=0;i < src_height; i++){
  for(j=0;j < src_width; j++){
    *dst_ptr = Min3x3(src_ptr);
    src_ptr = sl_inc_mask(src_ptr);
    dst_ptr = sl_inc_mask(dst_ptr);
  }
  Sm_syncV(1);
}
}

```

```

Max()
{
  unsigned char *src_ptr, *dst_ptr;
  int i,j;

  src_ptr = ( unsigned char *) isrc;
  dst_ptr = ( unsigned char *) idst;

  for(i=0;i < src_height; i++){
    for(j=0;j < src_width; j++){
      *dst_ptr = Max3x3(src_ptr);
      src_ptr = sl_inc_mask(src_ptr);
      dst_ptr = sl_inc_mask(dst_ptr);
    }
    Sm_syncV(1);
  }
}
}

```

```

Min3x3(in_ptr)

```

```
register unsigned char *in_ptr;
{
    register int i;
    register int res;
    register int p[9];

    p[8] = *sl_sub_mask(in_ptr , src_width + 1);
    p[1] = *sl_sub_mask(in_ptr , src_width );
    p[2] = *sl_sub_mask(in_ptr , src_width - 1);
    p[7] = *sl_sub_mask(in_ptr , 1);
    p[0] = *(in_ptr);
    p[3] = *sl_add_mask(in_ptr , 1);
    p[6] = *sl_add_mask(in_ptr , src_width - 1);
    p[5] = *sl_add_mask(in_ptr , src_width );
    p[4] = *sl_add_mask(in_ptr , src_width + 1);

    i = 9;
    res = 256;
    while(i--){
        res = (p[i] < res) ? p[i] : res;
    }
    return(res);
}
```

Max3x3(in\_ptr)

```
register unsigned char *in_ptr;
{
    register int i;
    register int res;
    register int p[9];

    p[8] = *sl_sub_mask(in_ptr , src_width + 1);
    p[1] = *sl_sub_mask(in_ptr , src_width );
    p[2] = *sl_sub_mask(in_ptr , src_width - 1);
    p[7] = *sl_sub_mask(in_ptr , 1);
    p[0] = *(in_ptr);
    p[3] = *sl_add_mask(in_ptr , 1);
```

```
p[6] = *sl_add_mask(in_ptr , src_width - 1);
p[5] = *sl_add_mask(in_ptr , src_width );
p[4] = *sl_add_mask(in_ptr , src_width + 1);

i = 9;
res = 0;
while(i--){
    res = (p[i] > res) ? p[i] : res;
}
return(res);
}

Sobel()
{
    short int tmp;
    unsigned char *src_ptr, *dst_ptr;
    int i=0,j=0;

    int kernx[3][3], kerny[3][3];

    kernx[0][0] = -1;
    kernx[0][1] = 0;
    kernx[0][2] = 1;
    kernx[1][0] = -2;
    kernx[1][1] = 0;
    kernx[1][2] = 2;
    kernx[2][0] = -1;
    kernx[2][1] = 0;
    kernx[2][2] = 1;
    kerny[0][0] = -1;
    kerny[1][0] = 0;
    kerny[2][0] = 1;
    kerny[0][1] = -2;
    kerny[1][1] = 0;
    kerny[2][1] = 2;
    kerny[0][2] = -1;
    kerny[1][2] = 0;
```

```

kerny[2][2] = 1;

src_ptr = ( unsigned char *) isrc;
dst_ptr = ( unsigned char *) idst;
for(i=0;i < src_height; i++){
    for(j=0;j < src_width; j++){
        Sob3x3(kernx,src_ptr, dst_ptr);
        src_ptr = sl_inc_mask(src_ptr);
        dst_ptr = sl_inc_mask(dst_ptr);
    }
    Sm_syncV(1);
}
}
Sob3x3(kernx,in_ptr,out_ptr)
int kernx[3][3];
unsigned char *in_ptr, *out_ptr;
{
    register int xres, res;
    register int p[10];

    p[9] = *sl_sub_mask(in_ptr , src_width + 1);
    p[2] = *sl_sub_mask(in_ptr , src_width );
    p[3] = *sl_sub_mask(in_ptr , src_width - 1);
    p[8] = *sl_sub_mask(in_ptr , 1);
    p[1] = *(in_ptr);
    p[4] = *sl_add_mask(in_ptr , 1);
    p[7] = *sl_add_mask(in_ptr , src_width - 1);
    p[6] = *sl_add_mask(in_ptr , src_width );
    p[5] = *sl_add_mask(in_ptr , src_width + 1);

    xres = p[9] * kernx[0][0];
    xres += p[2] * kernx[0][1];
    xres += p[3] * kernx[0][2];
    xres += p[8] * kernx[1][0];
    xres += p[1] * kernx[1][1];
    xres += p[4] * kernx[1][2];
    xres += p[7] * kernx[2][0];

```

```

xres += p[6] * kernx[2][1];
xres += p[5] * kernx[2][2];

res = abs(xres);

xres = p[9] * kernx[0][0];
xres += p[2] * kernx[1][0];
xres += p[3] * kernx[2][0];
xres += p[8] * kernx[0][1];
xres += p[1] * kernx[1][1];
xres += p[4] * kernx[2][1];
xres += p[7] * kernx[0][2];
xres += p[6] * kernx[1][2];
xres += p[5] * kernx[2][2];

res += abs(xres);
*out_ptr = (res < 256) ? res : 255;

```

```

}

```

```

Conv3x3(kern,in_ptr)
register int kern[3][3];
register unsigned char *in_ptr;
{
    register int res;
    register int p[10];

    p[9] = *(in_ptr - src_width - 1);
    p[2] = *(in_ptr - src_width );
    p[3] = *(in_ptr - src_width + 1);
    p[8] = *(in_ptr - 1);
    p[1] = *(in_ptr);
    p[4] = *(in_ptr + 1);
    p[7] = *(in_ptr + src_width - 1);
    p[6] = *(in_ptr + src_width );
    p[5] = *(in_ptr + src_width + 1);

```

```
    res = p[9] * kern[0][0];
    res += p[2] * kern[0][1];
    res += p[3] * kern[0][2];
    res += p[8] * kern[1][0];
    res += p[1] * kern[1][1];
    res += p[4] * kern[1][2];
    res += p[7] * kern[2][0];
    res += p[6] * kern[2][1];
    res += p[5] * kern[2][2];

    return(res);
}

abs(a)
int a;
{
    return((a > 0) ? a : -a);
}
```

## Appendix D

# How to Run the SMART System

```
#####
#                               INITIALIZATION                               #
#####
# Set environment variables and path
#
sun3> set path = (~dsp/processors/dsp32new/dsp32sl/bin $path)
sun3> setenv DSP3LIB ~dsp/processors/dsp32new/dsp32sl/lib
sun3> setenv DSP32SL ~dsp/processors/dsp32new/dsp32sl

# Bring {\bf tmap} and {\startup.o} files in current directory.
#
sun3> ln -s ~wookkoh/vw/tmap
sun3> ln -s ~wookkoh/vw/startup.o

# make library routines accessible by VxWorks:
#                               vw.o vxUtil.o vxtest.o
sun3> ln -s ~wookkoh/vw/vw.o
sun3> ln -s ~wookkoh/vw/vxUtil.o
sun3> ln -s ~wookkoh/vw/vxtest.o

#####
```



```

#                               COMPIATION                               #
#####
# Write image.c program and compile it for dsp32c.
#
sun3> d3optcc -o image -Q -m tmap -s startup.o image.c
sun3> ^wookkoh/bin/d3image image >! image.i

# Compile VxWorks library routines, if necessary.
# For example,
#
sun3> cc -c -O -I/usr/tools/vw/h vw.c

#####
#                               DOWN-LOAD SCRIPT : LOAD file           #
#####
# ^wookkoh/vw/image/LOAD
#
# First, the VxWorks library routines are loaded (ld < ).
# Then, the SMART system initialization routine is executed
#                               (init()).
# Finally, the image.i (program) is downloaded (download()).
#
ld < vw.o
ld < vxUtil.o
ld < vxtest.o
init()
download("image.i")
# The init() and download() routines are described in
#                               ^wookkoh/vw/vw.c.

#####
#                               RUNNING SCRIPT : SRUN file             #
#####

```

```

# ~wookkoh/vw/image/SRUN
#
run()
InOut16("proc.in", "lenna2.ras", "out.ras", 0x1, 0x1000000)
# First, the informations described in proc.in file are pushed
# into the SMART input FIFOs.
# Then, the input data words are acquired from lenna2.ras file
# and are down-loaded to the input FIFO 16 bits at a time.
# At the same time, the output data words are up-load from the
# output FIFO, and are generated to the out.ras file.
# The run(), InOut16() routines are described in ~wookkoh/vw/vw.c.

```

```

#####
#           INITIAL DOWN-LOAD : proc.in file           #
#####
# ~wookkoh/vw/image/proc.in
#
# The InOut16() routine initially down-load the following
# informations to the FIFOs.
# The file consists of ten groups of informations:
# one group for each processor.
# Each group consists of pid (processor ID), switch (1- closed,
# 0- opened), process (the process number selects one of the
# functions such as Move(), Sobel(), Grey(), etc.), src (source
# address of the data), dst (destination address of the data),
# wbegin & wend (programming parameters for global sync.).
# The initial down-loading can be used to initialize variables
# in the program.
#
0 pid
1 switch
0 process
100000 src
608000 dst

```

0 wbegin  
30 wend

1 pid  
0 switch  
6 process  
100000 src  
610000 dst  
4 wbegin  
30 wend

2 pid  
0 switch  
7 process  
100000 src  
618000 dst  
8 wbegin  
30 wend

3 pid  
0 switch  
8 process  
100000 src  
620000 dst  
c wbegin  
30 wend

4 pid  
0 switch  
6 process  
100000 src  
628000 dst  
10 wbegin  
30 wend

5 pid  
0 switch

7 process  
100000 src  
630000 dst  
14 wbegin  
30 wend

6 pid  
0 switch  
8 process  
100000 src  
638000 dst  
18 wbegin  
30 wend

7 pid  
0 switch  
2 process  
100000 src  
640000 dst  
1c wbegin  
30 wend

8 pid  
0 switch  
a process  
100000 src  
648000 dst  
20 wbegin  
30 wend

9 pid  
0 switch  
1 process  
100000 src  
600000 dst  
24 wbegin  
30 wend

```
#####
#                               RLOGIN                               #
#####
#
#
sun3> rlogin vw3
# Now you are in the SMART system, running VxWorks real-time
# O.S. Refer to the VxWorks Manual for shell commands and
# detailed informations.

# change directory to where your files are.
# For example,
-> cd "zabriskie:/home/zab2/wookkoh/vw/image"
# Note that -> prompt means the commands are being interpreted
# by the VxWorks shell.

#####
#                               RUNNING                               #
#####
#
-> < LOAD
-> < SRUH
-> logout
sun3>
# The script files can be concatenated into one file.

#####
#                               DISPLAY THE OUTPUT: out.ras        #
#####
# The output is a monochrome 512x512 raster image.
```

```
#  
sun3> ~wookkoh/bin/dirppix out.ras  
sun3> rm out.ras
```