

**A SHARED MEMORY MODULE FOR AN ASYNCHRONOUS  
ARRAY OF SIMPLE PROCESSORS**

By

MICHAEL JOSEPH MEEUWSEN  
B.S. (Oregon State University) June, 2003

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

---

Chair, Dr. Bevan M. Baas

---

Member, Dr. Rajeevan Amirtharajah

---

Member, Dr. Venkatesh Akella

Committee in charge  
2005



© Copyright by Michael Joseph Meeuwsen 2005  
All Rights Reserved



## Abstract

The design of an asynchronously shared memory module for the AsAP platform is presented. AsAP consists of a 2-dimensional array of processing elements with limited memory resources. The memory module expands the storage capacity available to AsAP processors, enabling the mapping of applications with large working sets. The memory module described shares an 8 K-word SRAM among four processors, but can support a 64 K-word SRAM with no additional changes. The memory module is independently clocked, supports hardware address generation, mutual exclusion, and multiple addressing modes. Simultaneous access by different processors is arbitrated using a least-recently-served priority scheme. A standard cell implementation of the memory module cycles at 555 MHz and occupies  $1.2 \text{ mm}^2$  in  $0.18 \mu\text{m}$  CMOS.

# Acknowledgments

I would like to thank all of the individuals who made this work possible. First, I thank Prof. Bevan Baas for his mentoring, advising, and funding during my stay at Davis. I also thank Intel Corporation and the University of California for their generous support of VCL.

I would also like to thank Ryan Apperson, Mike Lai, and Omar Sattari, for teaching me the ropes of VCL, and graduate school. Thank you for all the good times during my first year at UCD. I also owe you all a debt of gratitude for the L<sup>A</sup>T<sub>E</sub>X template you left for all of us to use. I would also like to thank Zhiyi Yu for his contributions to the AsAP project, and his helpful advice during the completion of this work, and his work on the back-end flow to complete P&R for the buffered memory design.

Thanks to Prof. Amirtharajah and Prof. Akella for their time and dedication while serving on my thesis committee, and for their teaching and mentorship at UCD.

I also thank those individuals who have helped me along the way to this milestone. I thank David Stewart, Jay Gilbert, and Barton Brown for their professional and personal advice and mentorship during the past five years. Your perspectives have given me insights that many are not so fortunate to have. I would also like to thank Prof. Lewis and Prof. Oklobdžija at UCD, and Prof. Tenca and Roger Traylor at OSU for their inspirational teaching.

I would like to thank my mom and dad for twenty four years of love and support. They have always encouraged me in my academic pursuits. I'd like to thank all of my old friends from OSU and high school for the laughs, the beer, and the free counseling. I'd also like to acknowledge the new friends I've made here in Davis. You all made my time at UCD much more palatable. And finally, I'd like to thank Cris for her patience during my nineteen months in California.

No person exists in isolation. The actions that we take always involve someone else. It is the actions of these people that have made this achievement possible. Thank you to you all.

# Contents

|                                                        |            |
|--------------------------------------------------------|------------|
| <b>Abstract</b>                                        | <b>iii</b> |
| <b>Acknowledgments</b>                                 | <b>iv</b>  |
| <b>List of Figures</b>                                 | <b>vii</b> |
| <b>List of Tables</b>                                  | <b>ix</b>  |
| <b>1 Introduction</b>                                  | <b>1</b>   |
| 1.1 Project Goals . . . . .                            | 1          |
| 1.2 Overview . . . . .                                 | 2          |
| <b>2 Background</b>                                    | <b>3</b>   |
| 2.1 The Processor/Memory Gap . . . . .                 | 3          |
| 2.2 Memory System Architectures . . . . .              | 4          |
| 2.2.1 Traditional Memory Hierarchies . . . . .         | 4          |
| 2.2.2 Alternative Memory Architectures . . . . .       | 5          |
| 2.2.3 Memory Systems in Chip Multiprocessors . . . . . | 7          |
| 2.2.4 GALS Systems . . . . .                           | 8          |
| 2.3 Memory Requirements for DSP Applications . . . . . | 8          |
| 2.3.1 Memory Access Patterns . . . . .                 | 8          |
| 2.3.2 Memory Size Requirements . . . . .               | 9          |
| 2.4 Summary . . . . .                                  | 9          |
| <b>3 The AsAP Architecture</b>                         | <b>11</b>  |
| 3.1 Target Applications . . . . .                      | 11         |
| 3.2 The AsAP Array . . . . .                           | 12         |
| 3.3 The AsAP Processing Element . . . . .              | 13         |
| 3.3.1 Dual-clock FIFOs . . . . .                       | 13         |
| 3.3.2 Clocking . . . . .                               | 14         |
| 3.3.3 Configuration . . . . .                          | 14         |
| 3.3.4 Memory Limitations . . . . .                     | 14         |
| <b>4 Design Space Exploration</b>                      | <b>17</b>  |
| 4.1 AsAP Memory Requirements . . . . .                 | 18         |
| 4.2 Physical Memory Parameters . . . . .               | 18         |
| 4.2.1 Capacity . . . . .                               | 19         |

|          |                                                   |           |
|----------|---------------------------------------------------|-----------|
| 4.2.2    | Density . . . . .                                 | 19        |
| 4.2.3    | Distribution . . . . .                            | 20        |
| 4.3      | Memory Processor Interface . . . . .              | 21        |
| 4.3.1    | Clock Source . . . . .                            | 22        |
| 4.3.2    | Address Source . . . . .                          | 22        |
| 4.3.3    | Buffering . . . . .                               | 22        |
| 4.3.4    | Sharing . . . . .                                 | 23        |
| 4.3.5    | Inter-parameter Dependencies . . . . .            | 23        |
| 4.4      | Degree of Configurability . . . . .               | 24        |
| 4.5      | Design Selection . . . . .                        | 24        |
| <b>5</b> | <b>FIFO-Buffered Memory Design</b>                | <b>27</b> |
| 5.1      | Overview . . . . .                                | 27        |
| 5.2      | Theory of Operation . . . . .                     | 28        |
| 5.3      | Processor Interface . . . . .                     | 30        |
| 5.3.1    | Memory and Configuration Address Spaces . . . . . | 31        |
| 5.3.2    | Request Types . . . . .                           | 31        |
| 5.3.3    | Input Port Modes . . . . .                        | 37        |
| 5.3.4    | AsAP Processor Instruction Set Mapping . . . . .  | 40        |
| 5.4      | Memory Module Implementation . . . . .            | 41        |
| 5.4.1    | Memory Core . . . . .                             | 41        |
| 5.4.2    | Input Port Implementation . . . . .               | 41        |
| 5.4.3    | Arbiter Implementation . . . . .                  | 47        |
| 5.4.4    | Mutex Implementation . . . . .                    | 50        |
| 5.4.5    | Address Generator Implementation . . . . .        | 51        |
| 5.4.6    | Output Port Implementation . . . . .              | 54        |
| 5.4.7    | Module Clocking . . . . .                         | 54        |
| 5.4.8    | Module Configuration . . . . .                    | 56        |
| 5.4.9    | Processor Port Implementation . . . . .           | 56        |
| 5.5      | Design Summary . . . . .                          | 60        |
| <b>6</b> | <b>Implementation Results</b>                     | <b>61</b> |
| 6.1      | Performance Results . . . . .                     | 61        |
| 6.1.1    | Peak Performance . . . . .                        | 61        |
| 6.1.2    | Actual Performance . . . . .                      | 63        |
| 6.2      | Area and Power Trade-offs . . . . .               | 69        |
| 6.2.1    | Synthesis Methodology . . . . .                   | 70        |
| 6.2.2    | Area Results . . . . .                            | 70        |
| 6.2.3    | Power Results . . . . .                           | 72        |
| <b>7</b> | <b>Conclusion</b>                                 | <b>77</b> |
| 7.1      | Summary . . . . .                                 | 77        |
| 7.2      | Future Work . . . . .                             | 78        |
|          | <b>Glossary</b>                                   | <b>79</b> |
|          | <b>Bibliography</b>                               | <b>81</b> |

# List of Figures

|      |                                                                           |    |
|------|---------------------------------------------------------------------------|----|
| 2.1  | The processor/memory gap. . . . .                                         | 4  |
| 3.1  | AsAP processor block diagram. . . . .                                     | 13 |
| 4.1  | Various topologies for distribution of memories in an AsAP array. . . . . | 20 |
| 5.1  | FIFO-buffered memory interface. . . . .                                   | 28 |
| 5.2  | FIFO-buffered memory block diagram. . . . .                               | 29 |
| 5.3  | Configuration address space. . . . .                                      | 32 |
| 5.4  | Memory read request command token format. . . . .                         | 33 |
| 5.5  | Memory write request command token format. . . . .                        | 33 |
| 5.6  | Port configuration request command token format. . . . .                  | 34 |
| 5.7  | Address generator configuration request command token format. . . . .     | 35 |
| 5.8  | Burst read request command token format. . . . .                          | 35 |
| 5.9  | Burst write request command token format. . . . .                         | 35 |
| 5.10 | Mutex request command token format. . . . .                               | 36 |
| 5.11 | Mutex release command token format. . . . .                               | 36 |
| 5.12 | Port configuration register formats. . . . .                              | 37 |
| 5.13 | Memory writes in (a) address-data and (b) address-only mode. . . . .      | 39 |
| 5.14 | Single stage input port . . . . .                                         | 42 |
| 5.15 | Input port finite state machine . . . . .                                 | 44 |
| 5.16 | Two stage input port . . . . .                                            | 46 |
| 5.17 | Arbiter priority tracking circuit . . . . .                               | 48 |
| 5.18 | Arbiter priority resolution network . . . . .                             | 48 |
| 5.19 | Parallel arbiter implementation . . . . .                                 | 50 |
| 5.20 | Mutual exclusion primitive (mutex) . . . . .                              | 52 |
| 5.21 | Address generator datapath . . . . .                                      | 54 |
| 5.22 | Processor memory port prefetch buffer. . . . .                            | 59 |
| 6.1  | Workloads used for performance characterization. . . . .                  | 64 |
| 6.2  | Effect of computational load and clock speed on performance. . . . .      | 65 |
| 6.3  | Effect of number of processors on performance. . . . .                    | 66 |
| 6.4  | Effect of address mode on block performance. . . . .                      | 67 |
| 6.5  | Effect of address load on address-only mode performance. . . . .          | 68 |
| 6.6  | Equal area comparison of address modes. . . . .                           | 69 |
| 6.7  | Relative Area Submodules. . . . .                                         | 71 |
| 6.8  | Relative power consumption of submodules. . . . .                         | 73 |

6.9 Relative power consumption, neglecting clocking power . . . . . 73  
6.10 Relative leakage power of submodules. . . . . 74  
6.11 Place and Route results. . . . . 75

# List of Tables

|     |                                                       |    |
|-----|-------------------------------------------------------|----|
| 5.1 | FIFO-buffered memory request characteristics. . . . . | 33 |
| 5.2 | Summary of input port modes. . . . .                  | 37 |
| 5.3 | Memory port mapping to DCmem. . . . .                 | 41 |
| 5.4 | Address generator configuration. . . . .              | 53 |
| 5.5 | Static configuration parameters. . . . .              | 57 |
| 6.1 | Synthesis results. . . . .                            | 70 |



# Chapter 1

## Introduction

The memory subsystem is a key element of any computational machine. The memory retains system state, stores data for computation, and holds machine instructions for execution. In many modern systems, memory bandwidth is the primary limiter of system performance, despite complicated memory hierarchies and hardware driven prefetch mechanisms.

Coping with the intrinsic gap between processor performance and memory performance has been a focus of research since the beginning of computer architecture [1]. The fundamental problem is the infeasibility of building a memory that is both large and fast. Designers are forced to reduce the size of memories for speed, or pay long latencies to access high capacity storage. The primary solution to the memory gap has been the implementation of multi-level memory hierarchies.

In the embedded and signal processing domains, designers may use existing knowledge of system workloads to optimize the memory system. Typically, these systems have smaller memory requirements than general purpose computing machines, which makes alternative architectures attractive. When the designer is not bound to the traditional cache hierarchy, higher performance can be achieved.

### 1.1 Project Goals

This work explores the design of a memory subsystem for an *Asynchronous Array of Simple Processors* (AsAP). AsAP consists of a two-dimensional array of processors with limited local memory resources. Each processor operates independently, and no global address space exists.

To efficiently support applications with large working sets, processors must be provided with higher capacity memory storage.

To maintain design simplicity, scalability, and computational density, a traditional memory hierarchy is avoided. Cache memories provide no additional system functionality and decrease computational density, because caches implement redundant data storage. They also trade system flexibility to ease programming, as memory management is hidden from the programmer. In addition, the fine granularity of the AsAP tiles and low locality in DSP applications makes the cache solution unattractive. Instead, directly-addressible software-managed memories are explored. This allows the programmer to efficiently manage the memory hierarchy explicitly.

The main requirements for the implementation of the memory system are the following:

- The system must provide high throughput access to high capacity random access memory.
- The memory must be accessible from multiple asynchronous clock domains.
- The design must easily scale to support arbitrarily large memories.
- The impact on the processing element should be minimized.

## 1.2 Overview

The remainder of this work is organized as follows. In Chapter 2, the current state of the art in memory systems is reviewed. Chapter 3 provides an overview of the existing AsAP architecture without a memory subsystem. Chapter 4 describes the design space for the addition of memory modules to the AsAP system. Chapter 5 describes the design of a buffered memory module, which has been implemented using a standard cell flow. Chapter 6 discusses the performance of the design, based on high level synthesis results and simulation. Finally, future directions for this work are discussed in Chapter 7.

## Chapter 2

# Background

This chapter discusses the current state of memory systems research, and its application to digital signal processing (DSP). Only a brief survey is provided here. The interested reader should consult the literature or a computer architecture text [2] for additional detail. First, traditional memory hierarchies are discussed, and then alternatives to the cache-based approach are described. Finally, some special considerations for DSP memory systems are enumerated.

### 2.1 The Processor/Memory Gap

The disconnect between achievable system performance, and available memory bandwidth has been a major concern throughout the development of computer architecture. Even computer architecture pioneers Burks, Goldstine, and von Neumann noted the problem in their early discussion of the computing machine in 1946:

Ideally one would desire an indefinitely large memory capacity such that any particular ... word would be immediately available.... It does not seem possible physically to achieve such a capacity. We are therefore forced to recognize the possibility of constructing a hierarchy of memories, each of which has a greater capacity than the preceding but which is less quickly accessible. [1]

Closing the processor memory gap has been a difficult battle for computer designers. As memory densities continue to grow, memory performance has improved only slightly; processor performance, on the other hand, has shown exponential improvements over the years. According to Hennessy and Patterson, processor performance has increased by 55 percent each year, while memory

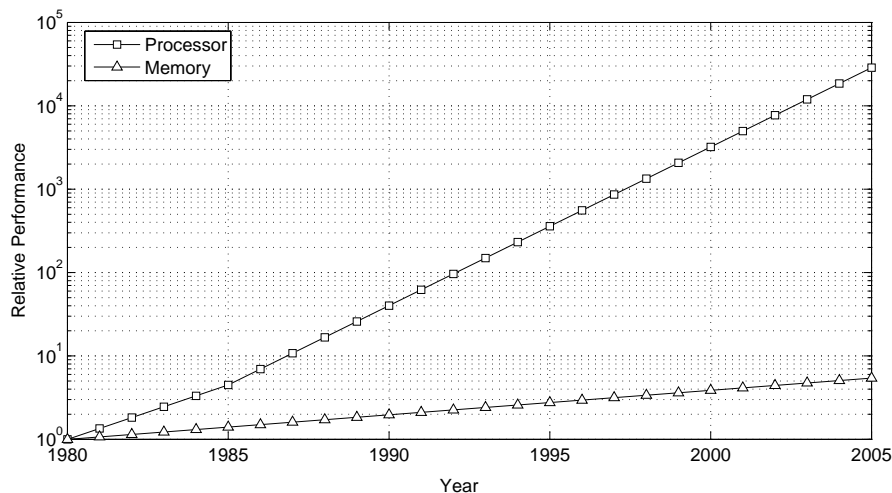


Figure 2.1: The processor/memory gap. Modern processor performance is quickly outgrowing the capabilities of memory storage devices. In this plot from Hennessy and Patterson [2], memory performance increases by 7% each year, while processor performance has increased at 55% per year since 1986, and 35% per year prior to 1986. Values are normalized to 1980 performance.

performance increases by only 7 percent [2]. Figure 2.1 shows the growth of the processor/memory gap.

## 2.2 Memory System Architectures

Although researchers have not been able to stop the growth of the processor/memory gap, they have developed a number of architectural alternatives to increase system performance despite the limitations of the available memory. These solutions range from traditional memory hierarchies to intelligent memory systems. Each solution attempts to reduce the impact of poor memory performance by storing the data needed for computation in a way that is easily accessible to the processor.

### 2.2.1 Traditional Memory Hierarchies

The primary solution to the processor/memory gap has been to introduce a local cache memory, exploiting spatial and temporal locality evident in most software programs. Caches are small fast memories that provide the processor with a local copy of a small portion of main memory. Caches are managed by hardware to ensure that the processor always sees a consistent view of main

memory. Modern machines use virtual memory to add an additional level of hierarchy beyond main memory. The cache solution was first described in 1962 by Kilburn *et al.* [3], and has since become commonplace. A thorough discussion of cache design can be found in any computer architecture textbook, such as that by Hennessy and Patterson [2].

Recently, more aggressive cache designs have been undertaken in an attempt to maximize system performance. Intel's Itanium processors include three levels of on-chip cache. The large L3 cache dominates the die area of the design. The most recent single-core Itanium has a 9 MB L3 cache, which occupies 86 percent of the chip area [4]. Advanced circuit design techniques were used to make a cache of this size feasible. Still, it is likely that ever increasing cache sizes will soon be limited by diminishing returns [2].

The primary advantage of the traditional cache scheme is the ease of programming. Because caches are managed by hardware, the programmer is only concerned with a single large address space. Movement of data from main memory to cache is handled by hardware and is transparent to the programmer. Here, a simple software interface comes at the cost of complicated hardware design.

The primary drawback of the cache solution is its high overhead. While cache memories are effective in providing performance improvements without changing the processor's programming model, the cache memories occupy a significant portion of chip area and consume additional power. Cache memories do not, however, provide any additional functionality to the system. All of the storage provided by a cache is redundant. Identical data must be stored elsewhere in the system: either in main memory or on disk. Patterson *et al.* refer to the overhead of the cache hierarchy as the "memory gap penalty" [5]. Other solutions have tried to address these drawbacks by either removing redundant storage, or by integrating computation hardware with the memory.

### 2.2.2 Alternative Memory Architectures

#### Scratch Pad Memories

A cache alternative prevalent in embedded systems is the scratch-pad memory [6]. A scratch-pad memory is an on chip SRAM with a similar size and access time to a L1 cache. Scratch-pads are unlike caches in that they are uniquely mapped to a fixed portion of the system's address

space. Scratch-pad memory may be used in parallel with a cache or alone [7]. Banakar *et al.* report a typical power savings of 40 percent when scratch-pad memories are used instead of caches [6].

A primary focus for research in this area is the allocation of scratch-pad memories. Panda *et al.* have described automated techniques for scratch-pad and main memory allocation for embedded systems [7, 8, 9, 10]. They report a performance improvement greater than 30 percent over scratch-pad only and cache only approaches [8]. The scratch-pad technique is most applicable in the embedded domain, where the processor runs a single application and memory need not be shared.

### **Intelligent RAM**

Intelligent RAM (IRAM) refers to chips that integrate logic circuits and DRAM onto a single die [5]. DRAM is much denser than the SRAM typically integrated with logic circuits, so more memory can be placed on the die when DRAM is used. DRAM, however, requires a special process which is not commonly used for high performance logic. Integrating high density DRAM near processing logic increases memory bandwidth compared to equivalent multi-chip solutions. Speedups greater than 2x for memory intensive benchmarks have been reported for IRAM based architectures, compared to processors using traditional memory hierarchies [11].

### **Smart Memories**

Smart Memories is a modular reconfigurable architecture targeted at reconfigurable computing applications [12]. A Smart Memories system consists of a reconfigurable fabric of memories and computing elements. Smart Memories attempts to increase computation efficiency by allowing on-chip SRAM resources to be configured as caches, buffers, or scratch-pad memories. The memory configuration can be optimized for each application. The flexibility of the Smart Memories system introduces a 32 percent area overhead, and 23 percent power overhead for a 16-KB SRAM capacity [13]. The average power for a 16-KB module is 125 mW.

### **Imagine Stream Processor**

Imagine is a programmable processor for multimedia processing. It operates on streams of data, and contains 48 ALUs communicating via an on chip network [14]. Imagine's memory

subsystem consists of a stream register file and a memory controller which interfaces to off-chip DRAM. The register file contains 128 KB storage. Data streams of up to 32 KB are loaded from the DRAM into the register file, and operated on using the processors computational resources. Although Imagine provides very high bandwidth access to the off-chip DRAM, access must take place at the stream level. Fine-granularity random access is not directly supported.

### 2.2.3 Memory Systems in Chip Multiprocessors

When a single chip contains multiple processing cores, the possibility for new memory architectures arises. Many commercial multi-core processors have been released. Most of these implement traditional cache hierarchies. Intel's dual-core Itanium contains two multi-threaded processor cores [15]. Each core has a dedicated three-level cache hierarchy, leading to over 26 MB of memory on the die. The global address space is shared between the cores, as in a traditional shared memory multiprocessor system. A recent dual-core SPARC processor also contains two processor cores, but the L2 and L3 caches are shared between the cores [16]. The L3 cache is off-chip, so only 2 MB of memory is integrated on the die. Again, the main memory is accessed via a single address space.

The CELL processor is a multiprocessor targeted at multimedia applications [17]. The processor contains a single power processing element (PPE), and eight additional synergistic processing elements (SPE). The PPE interfaces to an on-chip L2 cache in the typical way. The SPEs implement an alternative memory architecture. Each SPE contains several load-store units, each with 256 KB of memory in a local, untranslated, non-coherent memory space, with respect to the system memory space. This allows each SPE to execute without concern for memory coherency among processors. The memory interfaces to the system memory via a DMA engine.

While most commercially produced chip multiprocessors resemble existing shared memory multiprocessor systems, new multi-core architectures are emerging from academia which are significantly different. These architectures attempt to address the scalability concerns of ever shrinking feature sizes and increasing clock speeds. Tile based architectures, such as MIT's RAW processor, consist of many uniform processing elements. Each RAW tile is a fully functional CPU and contains a local 32 KB data cache [18]. In contrast to traditional systems, this cache may be soft-

ware managed, or treated as a stand alone memory. The system may also access off-chip DRAM, utilizing the data cache in the usual way.

### 2.2.4 GALS Systems

Another approach to addressing the scalability issues of modern technologies is to implement globally asynchronous locally synchronous (GALS) systems. This alleviates the need to distribute a high speed clock across a large die. GALS designs typically consist of a heterogeneous set of processors and peripherals, each in their own clock domain. Clock boundaries are crossed using asynchronous protocols. Smith has described a GALS shared memory multiprocessor system [19], while Gharsalli *et al.* describe the generation of memory wrappers, for integration of memories into the GALS environment [20].

## 2.3 Memory Requirements for DSP Applications

There has been a large amount of research concerning memory systems for general purpose computing machines. These systems are typically multi-user systems, which run a variety of different types of software. The system requirements for DSP applications are typically much different. The key elements that differentiate memory systems for DSP workloads are the regularity of memory access and the size of the working set.

### 2.3.1 Memory Access Patterns

DSP applications are characterized by data-driven processing. This leads to many small kernels which access the working set in a regular and iterative fashion. DSP designers can significantly improve the performance of these applications by providing dedicated address generation hardware, reducing the computation required for the most common access patterns. Current DSPs from Texas Instruments support auto-increment and auto-decrement addressing in circular or linear modes [21]. Similarly, a recently published DSP from Huang *et al.* includes an address generation unit to support modular and circular addressing, as well as bit-reversed addressing for FFT computation [22].

To reap the benefits of sophisticated address generation hardware, the address generators must support the access patterns of the applications being mapped. Supporting many different addressing modes, however, increases the design complexity of the address generators, increasing area, power consumption, and cycle time. Circular addressing modes, as implemented in most commercial DSPs are heavily used in FIR filters, one of the most common DSP applications. Additionally, work by Lee *et al.* has explored the access patterns common to multimedia applications [23]. They report fixed stride, two-way fixed-stride, and two-dimensional access patterns as being the most prevalent in the multimedia kernels explored. Finally, systems such as APEX attempt to characterize applications and customize the system's memory architecture based on access patterns observed [24].

### 2.3.2 Memory Size Requirements

The memory required for a DSP algorithm depends on the coding approach and the amount of parallelism exploited. For most algorithms a theoretical lower bound is easily determined. For example, a double-buffered 1024 point complex FFT requires approximately 4096 words of memory storage [25]. An FIR filter requires a minimum of two words per tap, in general. Requirements for other algorithms can be estimated similarly. Work by Grun *et al.* attempts to characterize applications to find the optimal trade-off between performance and memory size [26]. This work reports system memory requirements on the order of 10 K words for a number of multimedia applications. These requirements are orders of magnitude smaller than the typical memory capacity of general purpose machines and commercial DSPs. There is a potential for dramatic area and power savings by reducing the memory capacity of DSP processors.

## 2.4 Summary

Most general purpose computing systems cope with the memory gap by introducing a complex cache hierarchy. This leads to a large memory gap penalty due to the area and power consumed by the cache memories, which store redundant data. Other solutions, such as the CELL processor and MIT's RAW, provide local memory space to each processing element on chip. Because this space is locally addressed, there is no chance for inconsistency between processors, which

leads to more efficient memory usage. In addition, the lack of redundant data storage in these locally addressed memories increases power and area efficiency.

For multi-processor architectures targeting DSP applications with small memory capacity requirements, small locally-addressable memories are attractive. Such a solution is feasible only if a global memory space is not required for inter-process communication. The primary benefits of such a scheme are the avoidance of the memory gap penalty introduced by cache hierarchies, and the area and power efficiency provided by reduced memory capacity.

## Chapter 3

# The AsAP Architecture

The target architecture for this work is a chip multiprocessor called AsAP. AsAP (Asynchronous Array of simple Processors) is an extreme example of the chip multiprocessor paradigm targeted at DSP workloads. AsAP has been previously described in the literature[25, 27, 28]. The architecture consists of a two dimensional array of simple processors. Each processor is clocked asynchronously with respect to the others. The processors are characterized by their minimalist design and extremely limited memory resources. These characteristics provide high computational density, and high energy efficiency for a breadth of DSP applications.

The following sections describe the details of the AsAP architecture. First, target applications are discussed. The second section describes the system architecture, and the third section focuses on the architecture of a single processing element. The emphasis is placed on those elements of the architecture most closely related to the memory module design.

### 3.1 Target Applications

The AsAP architecture targets a variety of DSP applications. Typical DSP applications consist of various kernels, each containing repetitive computation and memory accesses across a dataset. The working sets of DSP applications are fairly small, which allows for easy exploitation of parallelism. The various kernels are usually independent of one another so they are easily pipelined. Although total system latency is important, sustainable throughput is usually the primary performance metric.

AsAP is designed to be used as a stand-alone embedded processor, or as a co-processor to a general purpose processor. It can run only one application at a time, and is poorly suited for control oriented tasks, such as process management. With this paradigm in mind, AsAP has little need for large virtual memory spaces, interrupts, or exception handling capabilities. At a system level, these tasks would be managed by a general purpose processor as needed. The role of AsAP is simply to provide a high performance computation engine.

Applications are mapped to AsAP by partitioning computation into many small tasks. Each task is statically mapped onto a small number of processing elements. For example, an IEEE 802.11a baseband transmitter has been implemented on a 22-processor array[28], and a JPEG encoder has been implemented on a nine-processor array.

## 3.2 The AsAP Array

An AsAP system consists of a two-dimensional array of homogeneous processing elements. Each element is a simple CPU, which contains its own computation resources and executes its own locally stored program. Each processing element has a local clock source and operates asynchronously with respect to the rest of the array. The Globally Asynchronous Locally Synchronous (GALS) nature of the array alleviates the need to distribute a high speed clock across a large chip. The homogeneity of the processing elements makes the system easy to scale as additional tiles can be added to the array with little effort.

Inter-processor communication within the array occurs through dual-clock FIFOs on processor boundaries. These FIFOs provide the required synchronization, as well as data buffers for rate matching between processors. The interconnection of processors is reconfigurable, with each processor choosing up to two input sources and four output destinations from its four nearest neighbors. Reconfigurability enables hardware reuse across a variety of applications, while limited communication increases scalability.

Each processor is connected to a global configuration bus. This low-speed bus enables each processor in the array to be programmed at startup. The bus is also used to configure the interconnection of processors. The configuration is stored in registers within each processing element until it is reprogrammed or powered down.

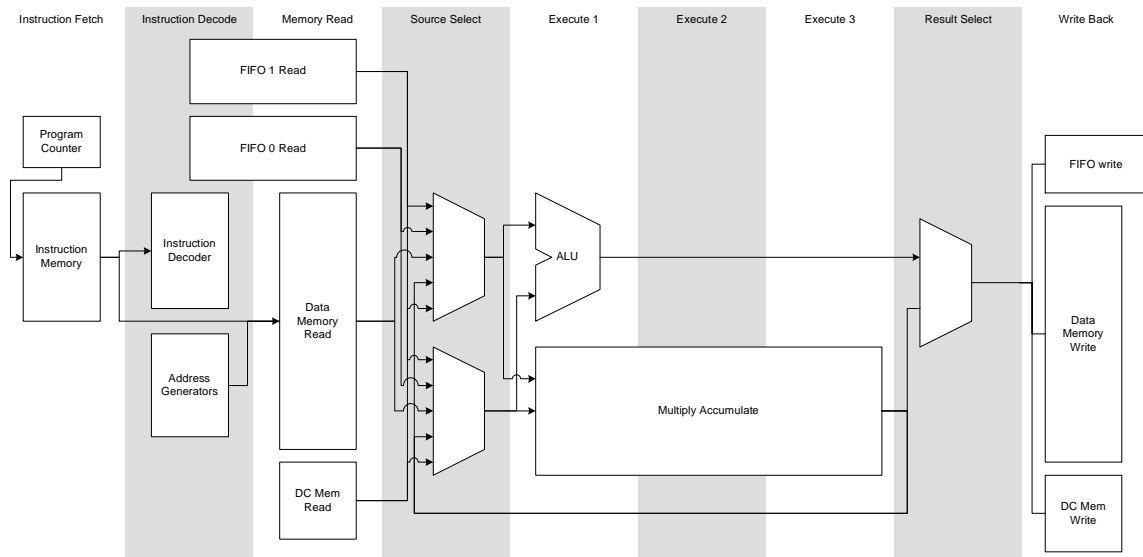


Figure 3.1: Block diagram of the AsAP processor pipeline. The nine pipe stages are shown. Pipe registers are not shown, but are implied. All control signals are generated in the instruction decode stage, and pipelined appropriately. Interlocks are not implemented, so all code must be scheduled prior to execution.

### 3.3 The AsAP Processing Element

The AsAP processor is a small single issue CPU supporting instructions found in most DSP processors. The processor contains a 16 bit fixed-point datapath, including a full-featured ALU and a multiply-accumulate unit with a 40-bit accumulator. The processor instruction cycle is divided into a nine-stage pipeline. Interlocks are not implemented and code scheduling is left to the user. A block diagram of the AsAP processor is shown in Figure 3.1. Each processor is expected to occupy approximately  $0.65 \text{ mm}^2$  in  $0.18 \mu\text{m}$  CMOS.

#### 3.3.1 Dual-clock FIFOs

Each processor contains two dual-clock FIFOs [29]. The FIFOs provide synchronization and rate-matching between adjacent processors. The FIFOs occupy two pipe stages in the processor pipeline. If a processor attempts to write to a neighboring processor with a full FIFO, an output stall will occur. If a processor attempts to read from an empty FIFO, an input stall will occur. When stalled, no instructions are executed. In this way, rate matching between processors is automatic.

In addition, the FIFOs allow a GALS implementation of the array by synchronizing data

across clock boundaries. This provides opportunities for power savings by isolating those portions of an application requiring high clock rates from those that can run more slowly. A GALS design also increases scalability by eliminating a global clock source.

### 3.3.2 Clocking

Each processor contains a local clock source that generates a unique clock for that processor. The clock is generated by a programmable ring oscillator. The frequency of oscillation can be varied for performance tuning or for system characterization. Frequency variation is achieved by changing the delay of each oscillator stage using a digital control word. For a wider frequency range, the number of stages in the oscillator is varied, and the oscillator output is optionally dividable by powers of two. In addition, the oscillator is pausable to increase power savings when a processor is idle.

### 3.3.3 Configuration

Processor level parameters are configured via a global configuration bus. Configuration parameters include the processor clock frequency, FIFO connections, and other processor level parameters. Processor level resets also take place through the configuration interface. All processors are slaves on the configuration bus. A bus transaction consists of an extended address, which includes a processor number, and data word. If the processor number matches the value hardwired into a processor, that processor writes the configuration data to the specified address. Various protocols can be supported on the global bus by introducing hardware protocol adapters between the bus and the configuration module in each processor.

### 3.3.4 Memory Limitations

AsAP processors are characterized by their extremely small memory resources. Memories were chosen to be small to minimize power and area while increasing the computational density of the array. No memory hierarchy exists, and memory is entirely managed by software. Additionally, there is no global address space, and all inter-processor communication must occur through the processors' input FIFOs.

Each processor tile contains four memory spaces: Instruction Memory (Imem), Data Memory (Dmem), Configuration Memory (Cmem) and Dynamic Configuration Memory (DCmem). Imem is limited to 64 32-bit instructions, and is writable only through the global configuration bus. Dmem contains 128 16-bit words, and is used for general purpose data storage. Dmem has a write port and two read ports to avoid structural hazards during pipelined operation. Cmem contains a set of variable width registers, which are used to configure processor parameters at power up. Cmem registers can only be written through the global configuration bus. DCmem contains a number of special function registers, which can be written at runtime. Because these registers affect processor operation, they are not suitable for general purpose storage. An additional 32 words of memory are allocated to each of the processors' input FIFOs.

With only 128 words of randomly-accessible storage in each processor, the AsAP architecture is limited to applications with small working sets. Previous work has addressed the limited memory capacity of the AsAP processor by mapping single kernels across multiple processors to provide additional storage [25, 28]. The efficiency of this solution is limited to a few additional processors due to increased communication costs. For kernels with a working set greater than 1 K, a more scalable solution is needed.



## Chapter 4

# Design Space Exploration

For many DSP applications, the small memory capacity of an AsAP processor is more than adequate. For example, an IEEE 802.11a transmitter requires 22 AsAP processors, but is easily implemented with the 128 words of memory per processor [28]. Some applications, however, demand access to high capacity memory storage by a small number of processors. A 1024-point FFT implementation described by Sattari requires only 6 processors, but 12 K of additional memory space [25]. The addition of large capacity memory modules to the AsAP array has been suggested, but the design of these memories has not yet been explored [25]. A wide variety of design possibilities exist. This chapter describes the design space and the selection of a design based on estimated performance and flexibility.

In exploring the design space, three roughly orthogonal groups of parameters can be defined.

1. Physical design parameters, such as memory capacity and module distribution. These parameters have little impact on the design of the memory module itself, but do determine how the module is integrated into the AsAP array.
2. Processor interface parameters, such as clock source and buffering. This parameter group has the largest impact on the module design itself.
3. Reconfigurability parameters. These parameters allow design complexity to be traded off for additional flexibility.

The remainder of this chapter first discusses the design requirements for the AsAP memory module. The three dimensions of the design space are explored. Finally, an appropriate design selection is made.

## 4.1 AsAP Memory Requirements

There are several system level requirements for the memory module design. First, the memory module must provide the highest possible throughput. To avoid limitations in fetching data from memory, the memory module throughput should be maximized. Requiring that the peak memory module throughput is the same as the SRAM core used in the design will ensure that the memory module is not needlessly limiting the memory bandwidth.

Second, the memory module must be able to interface to processors in different clock domains. To ensure flexibility in application mapping, the AsAP processor used to access the memory must be configurable. Because each AsAP processor has a unique clock, the memory module must have some means to synchronize with different processors.

Third, the memory module design must scale well to large memories. Memories in the range of 8 K words to 64 K words are the primary focus due to addressing concerns, but the design should scale to larger memory sizes as required. This will allow the user to select an appropriate amount of memory based on available die area and application requirements, without requiring significant design effort or creating unnecessary performance penalties.

Finally, the memory must be randomly accessible. Limiting accessibility to a fixed set of access patterns would render the memory module useless for many applications. Random access ensures that any application will be able to use the memory for data storage.

## 4.2 Physical Memory Parameters

Physical memory parameters define the capacity, density, and distribution of memory modules in an AsAP array. Capacity is the amount of storage included in the memory module. Density refers to the number of memory modules integrated into an AsAP array of a particular size. Distribution describes the topology selected for the memory modules within the two-dimensional

processor array. Each of these parameters is largely determined by anticipated application requirements, available die area, and physical layout complexity.

### 4.2.1 Capacity

Memory capacity is driven by application requirements, as well as area and performance targets. The lower bound on memory capacity is given by the memory requirements of targeted applications. Each application requires a minimum amount of memory to function correctly. For example, the double-buffered 1024 point FFT described by Sattari[25] requires a minimum of 4 K words for correct operation. Die area and memory performance limit the maximum size of the memory. Higher capacity SRAMs will occupy more die area, decreasing the total computational density of the AsAP array. Larger SRAMs will also limit the bandwidth of the memory core.

It is desirable to implement the smallest possible memory required for the targeted applications. These requirements, however, may not be available at design time. Furthermore, over-constraining the memory capacity will limit the flexibility of the array as new applications emerge. Hence, the scalability of the memory module design is important, allowing the memory size to be chosen late in the design cycle and changed for future designs with little effort.

### 4.2.2 Density

Memory module density is determined by the size of the AsAP array, available die area, and application requirements. Typically, the number of memory modules integrated in an AsAP array will be determined by the space available for such modules; however, application level constraints may also influence this design parameter. Assuming a fixed memory capacity per module, additional modules may be added to meet minimum memory capacity requirements. Also, some performance increase can be expected by partitioning an application's data among multiple memory modules due to the increased memory bandwidth provided by each module. This approach to increasing performance is not always practical and will not help if the application does not saturate the memory interface. It also requires a high degree of parallelism among data as communication among memory modules is not practical.

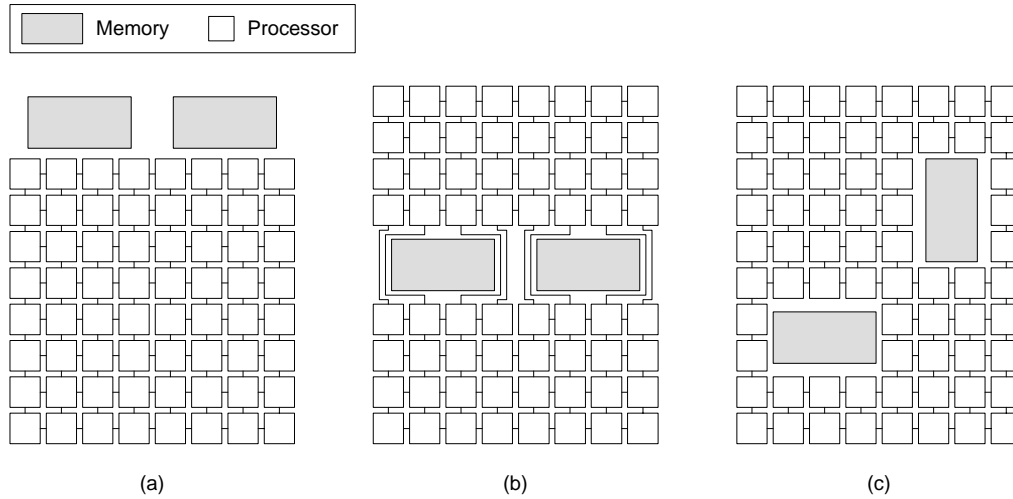


Figure 4.1: Various topologies for distribution of memories in an AsAP array. (a) A row of memories is added to the edge of the array. (b) The array is split to make room for a row of memories. Processor connectivity is maintained. (c) Processor tiles are replaced by memory tiles. Processor connectivity is lost.

### 4.2.3 Distribution

The distribution of memory modules within the array can take many forms. In general two approaches can be used. The first approach leaves the processor array intact and adds memory modules in rows or columns as allowed by available area resources. Processors in the array maintain connectivity to their nearest neighbors, as if the memory modules were not present. The second approach replaces processors with memory modules, so that each processor neighboring a memory module lose connectivity to one processor. Each of these strategies has limitations.

In the first case, a row of memory modules may be added either at the edge of an array, or by splitting the array to make room for the memory modules. These approaches are shown in Figure 4.1a and Figure 4.1b. Although adding memory modules to the edge of the array is fairly benign, this scheme limits memory connectivity to processors on the edge of the array. Because these processors have limited communication to the rest of the array, mapping applications may become difficult. However, the processor-array connectivity is unaffected because insertion of the memory modules does not require movement of any processors. Furthermore, the uniformity of processors can be easily maintained because all processors will connect to the memory from the same side.

If the processor array is split to make room for a row of memory modules, additional physical design concerns may manifest. Assuming that processor connectivity is maintained across the split, processor interconnect must now be routed over or around the memory modules. This interconnect will be longer than other links in the array, and the added latency could adversely affect processor cycle time. Also, splitting the array complicates automatic chip generation. Reduction of physical design complexity tends to favor integration of memory modules on the edge of the array, rather than within the array.

The second approach replaces processors with memory modules. This is shown in Figure 4.1c. The size of the memory module, as determined by its capacity, will dictate how many processors must be replaced. The primary advantage of this approach is its flexibility in selecting memory densities. While the row based approach requires that an entire row be dedicated to memory modules, the tile based approach allows memories to be distributed arbitrarily, in as sparse a configuration as desired. This method, however, is not without its challenges. First, the replacement of processors in the array means that the processor interconnect is no longer uniform and processor connectivity may be reduced. Additionally, the arbitrary placement of memory modules complicates application mapping and will make automation of this task increasingly difficult. Finally, the selection of memory module locations is difficult to generalize and will reduce the flexibility of the array when new applications are considered.

### **4.3 Memory Processor Interface**

The interface between the memory core and an AsAP processor encapsulates the bulk of the memory module design; hence, this dimension of the design space warrants careful consideration. Many different parameters can be considered in this dimension. Some of these parameters include the memory's clock source, the memory's address source, whether memory accesses are buffered, and whether memories can be shared among processors. Although other parameters exist, the current discussion will be limited to those listed for simplicity.

### 4.3.1 Clock Source

Because an AsAP array is a GALS system, the clock source for the memory module becomes a design parameter. In general, three distinct possibilities exist. First, the memory module can derive its clock from the clock of an AsAP processor. The memory would then be synchronous with respect to this processor. Second, the memory can generate its own unique clock. The memory would be asynchronous to all processors in the array. Finally, the memory could be completely asynchronous, so that no clock would be required. This solution severely limits the implementation of the memory module, as most SRAMs provided in standard cell libraries are synchronous.

### 4.3.2 Address Source

The address source for a memory module has a large impact on application mapping and performance. To meet the random access requirement, processors must be allowed to supply arbitrary addresses to memory. The intuitive solution uses the processor producing or consuming the memory data as the address source. The small size of AsAP processors, however, makes another solution attractive.

The address and data streams for a memory access can be partitioned among multiple processors. A single processor can potentially be used to provide memory addresses, while other processors act as data sources and data sinks. This scheme will provide a potential performance increase for applications with complex addressing needs because the data processing and address generation can occur in parallel.

Implementation of hardware address generators provides another means to speed up memory accesses and provides a third address source. This source is limited by the hardware implemented. To avoid unnecessary use of power and die area, only the most commonly used access patterns should be included in hardware. If an unsupported addressing pattern is required, it can be implemented in a processor.

### 4.3.3 Buffering

The implementation of buffers for accesses to the memory module provides another design parameter. Buffers may be used between a processor and a memory module for latency hiding,

synchronization, or rate matching. Without some level of buffering, processors are tightly coupled to the memory interface, and prefetching of data is difficult.

#### **4.3.4 Sharing**

The potentially large number of processors in an AsAP array makes the sharing of memories among processors attractive. In this context, shared memory serves two distinct purposes. First, as in more traditional computing, shared memory can serve as a communication medium among simultaneous program threads. Also, in the context of AsAP, sharing a memory among multiple processors can enable higher utilization of available memory bandwidth in cases where a single thread is unable to saturate the memory bus. In either case, synchronization mechanisms are required to guarantee mutual exclusion when memory is shared.

#### **4.3.5 Inter-parameter Dependencies**

There are strong dependencies among the four parameters described in sections 4.3.1–4.3.4. Selecting a value for one of the parameters limits the feasible values of the other parameters. This results in the existence of two distinct options for the processor interface design. Other design options tend to be a hybrid of these two extremes.

The first design can be defined by forcing a bufferless implementation. Without buffers, there is no way to synchronize across clock boundaries, so the memory module must be synchronous to the interfacing processor. Because AsAP processors are asynchronous to one another, sharing the memory is no longer feasible, and using an alternate processor as an address source is not possible. The resulting design is a memory module that couples tightly to a single processor. Because there is no buffering, memory accesses would either be tightly integrated into the processor's pipeline or carefully timed to avoid overwriting data.

The second design is, in some respects, the dual of the first. We can arrive at this design by requiring that the memories be shareable. In order to share the memories among asynchronous processors, the memory must supply its own clock source. Because it exists in a different clock domain, FIFOs must be used to synchronize across the clock boundaries. An alternate processor could easily be used as an address source with the appropriate hardware in place. This design is

effectively isolated from the rest of the array. The design of this module has few dependencies on the implementation of the AsAP processors.

## 4.4 Degree of Configurability

The degree of configurability included in the memory-processor interconnect, as well as in the memory module itself can be varied independently of the memory module design. To some degree, the level of configurability required in the interconnect is a function of the number of processors in the array, and their distances from the memory module. For small arrays, hardwired connections to the memory module may make sense. For large arrays, with relatively few memory modules, additional configurability is desirable to avoid limiting the system's flexibility.

The configurability of the memory module itself allows the designer to trade off performance, power, and area for flexibility. Examples of configurability at the module level cover a broad range and are specific to the module's design. Some examples of configurable parameters are the address source used for memory accesses and the direction of synchronization FIFOs in a locally clocked design.

## 4.5 Design Selection

Of the three design dimensions previously described, the processor interface has the greatest impact on the design of the memory module itself. As shown in Section 4.3 there are essentially two design extremes to consider.

The unbuffered memory design, which is synchronous to a single processor, has minimal overhead, but also has some disadvantages. Because the memory cannot be shared, the ability to fully exploit the memory bandwidth is limited. This effect is magnified because the processor's clock speed is limited by the memory's cycle time requirements. It follows that although the unbuffered memory solution has minimal hardware overhead, total system performance will be reduced because the memory bus will be difficult to saturate. Additionally, the unbuffered design is highly dependent on the processor implementation. It will have a strong impact on the processor's cycle time, and potentially force the introduction of new instructions.

The buffered memory solution with a local clock source will outperform the unbuffered solution in most cases. Because the memory module generates its own clock, the memory's cycle time does not impact the peak performance of accessing processors. The ability to share the memory, and use alternate processors for addressing will provide additional performance gains by exploiting parallelism in applications and utilizing additional memory bandwidth. In addition, the buffering of requests, which takes place for synchronization, may also be used for software driven prefetching, hiding the memory latency. The impact of physical distance from the memory to an accessing processor can also be reduced with this design.

The primary disadvantages of the buffered memory solution are its high latency and high area overhead. In general, two dual-clock FIFOs are required for each accessing processor. These FIFOs consume significant area and limit the number of processors that can simultaneously access the memory. The FIFOs also introduce additional latency in memory accesses. A memory read will suffer an additional latency of two FIFO delays on top of the compulsory memory latency, in the best case. For the existing ASAP FIFO design, a single FIFO latency is about six cycles, depending on the number of synchronization cycles used. Despite these challenges, the estimated performance, flexibility, and portability make the buffered design an attractive solution.

For the remainder of this work, a design based on the buffered memory solution is described. This design was chosen based on the flexibility in addressing modes and the ability to share the memory among multiple processors. These provide a potential performance increase by allowing redistribution of the address generation workload, and by exploiting parallelism across large datasets. The relative impact of the area overhead can be reduced if the SRAM core used in the memory module has a high capacity. The additional logic required then becomes a small fraction of the total module area. The additional latency can potentially be avoided by appropriate software prefetching at the application level.



## Chapter 5

# FIFO-Buffered Memory Design

This chapter describes the design and implementation of a FIFO-buffered memory module. The module provides additional memory storage to processors in the AsAP array. The module has its own local clock source, and interfaces to the processors of the AsAP array via dual clock FIFOs. As described in Section 4.5, this design was selected based on the flexibility in addressing modes and the potential speedup for applications with a high degree of parallelism across large datasets.

### 5.1 Overview

The FIFO-buffered memory module provides multiple AsAP processors concurrent access to a large SRAM. The prototype described in this chapter allows up to four AsAP processors to access the SRAM. The prototype design supports an SRAM size up to 64 K 16-bit words with no additional modifications. The currently implemented SRAM is 8 K words.

Processors access the memory module via input ports and output ports, as shown in Figure 5.1. An AsAP processor interfaces to an input port via a dual-clock FIFO, as is used for the processor's input ports [29]. The input port also encapsulates the required logic to process incoming requests. Each input port can assume different modes, changing the method of memory access. The memory module returns data to the processor via an output port, which also interfaces to a processor via a dual-clock FIFO. Configuration of the memory module and memory-processor interconnect is achieved through the configuration port, which connects to the AsAP array's global

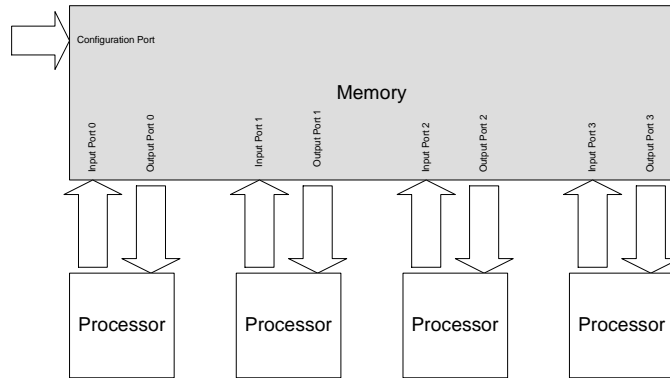


Figure 5.1: FIFO-buffered memory interface. AsAP processors interface to the memory module via dedicated input and output ports. These ports connect to the write and read sides of dual-clock FIFOs. An additional configuration port connects to the global configuration bus for the AsAP array.

configuration bus.

A number of additional features are integrated into the memory module to increase usability. These include multiple port modes, address generators, and mutual exclusion (mutex) primitives. A block diagram of the FIFO-buffered memory is shown in Figure 5.2. This diagram shows the high level interaction of the input and output ports, address generators, mutex, and SRAM core. The theory of operation for this module is described in Section 5.2. The programming interface to the memory module is described in Section 5.3. The hardware implementation details are described in Section 5.4.

## 5.2 Theory of Operation

The operation of the FIFO-buffered memory module is based on the execution of requests. AsAP processors issue requests to the memory module by writing 16-bit command tokens to their memory ports, which are mapped to the processor's DCmem address space. The requests instruct the memory module to carry out particular tasks, such as memory writes or port configuration. Additional information on the types of requests and their formats is provided in Section 5.3. Incoming requests are buffered in a FIFO queue until they can be issued. While requests issued by a single processor execute in FIFO order, requests from multiple processors are issued concurrently. Arbitration among conflicting requests occurs before allowing requests to execute.

In general, the execution of a request occurs as follows. When a request reaches the head

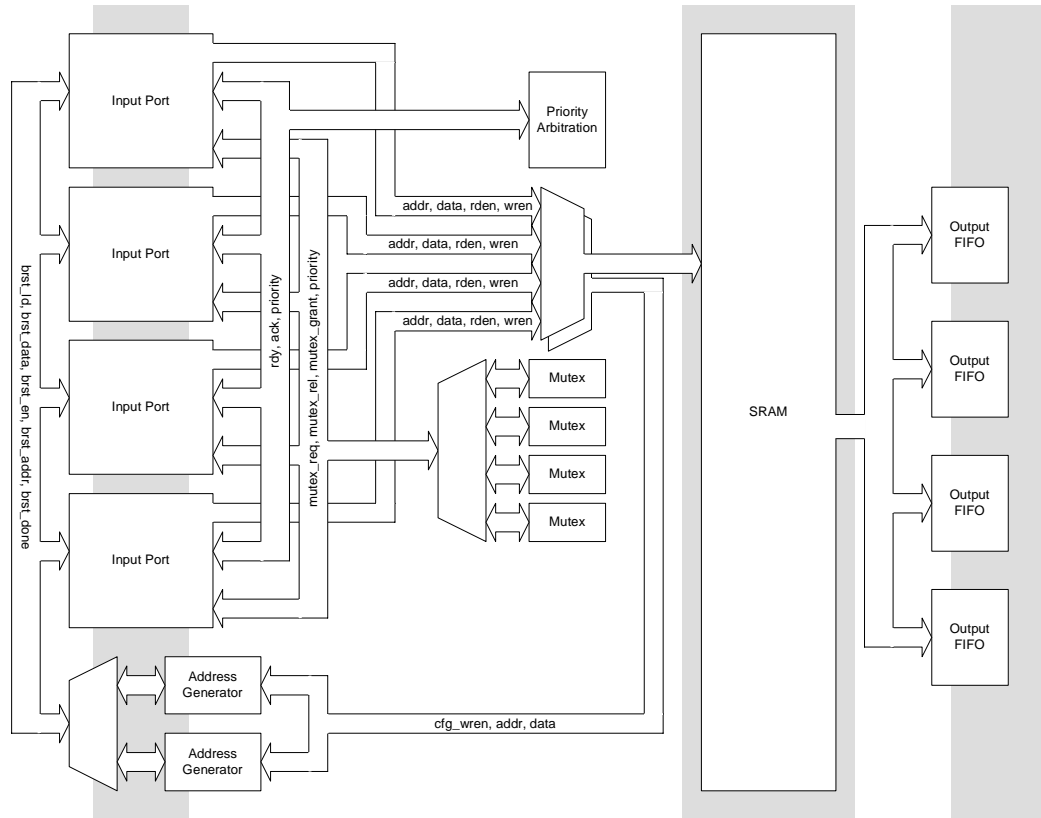


Figure 5.2: FIFO-buffered memory block diagram. Arrows show the direction of signal flow for the major blocks in the design. Multiplexers allow control of various resources to be switched among input ports. The gray bars approximate the pipe stages in the design.

of its queue it is decoded and its data dependencies are checked. Each request type has a different set of requirements. A memory read request, for example, requires adequate room in the destination port's FIFO for the result of the read; a memory write, on the other hand, must wait until valid data is available for writing. When all such dependencies are satisfied, the request is issued. If the request requires exclusive access to a shared resource, it will request access to the resource and wait for acknowledgment prior to execution. The request will block until access to the resource is granted. If the request does not access any shared resources, it will execute in the cycle after issue. Each port can potentially issue one request per cycle, assuming that requests are available and their requirements are met.

The memory module provides many ways for processors to access its memory resources. These different access modes are aimed at improving the efficiency of many common DSP tasks. In the simplest case, a single processor can access the memory by providing a single address, and data where appropriate. For applications requiring complex addressing, the memory can be configured to allow one processor to generate an address stream, while other processors handle data. This behavior is accomplished by setting the input port mode, as described in Section 5.3.3. Address generators are included in the memory module to allow a processor to perform block reads and writes with variable stride and offset. Bursts of 255 memory reads or writes may be issued with a single request. These three address modes provide flexibility in implementing common access patterns without preventing less common patterns from being used.

Because the memory resources of the FIFO-buffered memory are shared among multiple processors, the need for inter-process synchronization is anticipated. To this end, the memory module includes four mutex primitives in hardware. Each mutex implements an atomic single-bit test and set operation, allowing easy implementation of simple locks. More complex mutual exclusion constructs may be built on top of these primitives using the module's memory resources.

### **5.3 Processor Interface**

AsAP processors communicate with the memory module via dedicated memory ports. The design of the memory port is described in Section 5.4.9. Each of these ports may be configured to connect to one input FIFO and one output FIFO in the memory module. These connections are

independent, and which of the connections are established depends on the size of the ASAP array, the degree of reconfigurability implemented, and the specific application being mapped.

An ASAP processor accesses the memory module by writing 16-bit words to one of the memory module's input FIFOs. In general, these words are called *tokens*. One or more tokens make up a *request*. A request instructs the memory module to perform an action and consists of a command token, and possibly one or more data tokens. The requests issued by a particular processor are always executed in FIFO order. Concurrent requests from multiple processors, may be executed in any order. If a request results in data being read from memory, this data will be written to the appropriate output FIFO where it can be accessed by the processor.

### 5.3.1 Memory and Configuration Address Spaces

Within the memory module, two distinct address spaces exist. The first addresses the SRAM core. The second is a special configuration address space, which is used to access the various features and modes of the memory module.

The configuration space provides access to special functions integrated into the memory module. The partitioning of this address space is illustrated in Figure 5.3. Addresses from 0x0000 to 0x01FF map to special function registers used to configure the mode of the input port. The use of these registers is detailed in Section 5.3.3. Addresses from 0x8000 to 0x88FF are used to configure the address generators. Addresses from 0xC000 to 0xC8FF map to burst requests. Addresses above 0xFF00 map to mutex accesses.

To differentiate reads and writes in the two address spaces, two additional bits are added to each token. The first bit is asserted when a request is intended for configuration space and is called *cfgen*. The second bit is asserted to indicate a write request and is called *wren*. The meaning of the write enable bit varies somewhat when used in configuration space. In general, *wren* is asserted when additional data tokens are required to complete a request.

### 5.3.2 Request Types

The FIFO-buffered memory supports eight different request types. These are distinguished by the *cfgen* and *wren* bits, as well as by the content of the command token. Each re-

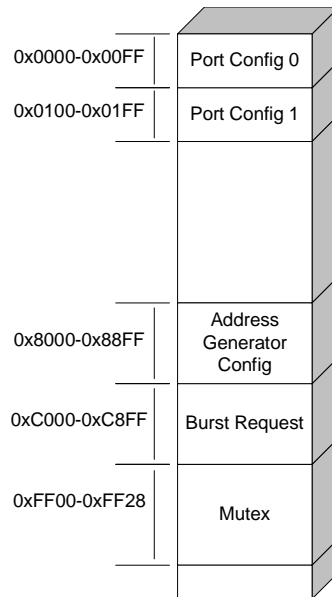


Figure 5.3: Configuration address space.

quest type will utilize different resources within the memory module. In addition, some requests are blocking, meaning that they must wait for certain conditions to be satisfied before they complete. To maintain FIFO ordering of requests, subsequent requests cannot proceed until a blocking request completes. The details for each request type are described in the following sections. The request types are summarized in Table 5.1.

### Memory Read Request

A memory read request causes a single word to be read from memory and written to the appropriate output FIFO. The destination FIFO is determined by the mode of the input port, and the contents of the port configuration registers. Port modes and port configuration registers are described in Section 5.3.3. The format of a memory read request command token is shown in Figure 5.4. The command token for a memory read request consists of the 16-bit memory address to be read. The *cfgen* and *wren* bits must both be set to logic zero to identify a memory read request. No additional data tokens are required. Memory read requests require access to the memory, and block until access is granted.

| Request Type          | <i>cfgen</i> | <i>wren</i> | Blocking | Memory | Mutex | Addr Gen | Data |
|-----------------------|--------------|-------------|----------|--------|-------|----------|------|
| Memory Read           | 0            | 0           | X        | X      |       |          |      |
| Memory Write          | 0            | 1           | X        | X      |       |          | X    |
| Port Config Write     | 1            | 0           |          |        |       |          |      |
| Addr Gen Config Write | 1            | 1           |          |        |       | X        | X    |
| Burst Read            | 1            | 0           | X        | X      |       | X        |      |
| Burst Write           | 1            | 1           | X        | X      |       | X        | X    |
| Mutex Request         | 1            | 0           | X        |        | X     |          |      |
| Mutex Release         | 1            | 0           |          |        | X     |          |      |

Table 5.1: FIFO-buffered memory request characteristics. The blocking column indicates if the request will block. The memory, mutex, and address generator columns show which resources the request requires. The data column indicates which requests require additional data tokens.

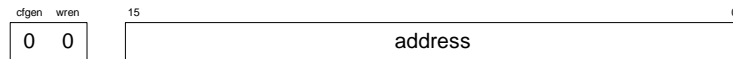


Figure 5.4: Memory read request command token format.

### Memory Write Request

A memory write request causes a single word to be written to memory. The format of a memory write request command token is shown in Figure 5.5. The command token for a memory write request consists of the 16-bit memory address to be written. The *cfgen* bit must be set to logic zero and the *wren* bit set to logic one to identify a memory write request. One additional data token is required. The *cfgen* bit should be logic zero for the data token. This token may be supplied by the same input port as the command token or from a different port, as indicated by the port mode. See Section 5.3.3 for details on input port modes. Memory write requests require access to the memory and a valid data token. The request will block until valid data is available and memory access is granted.

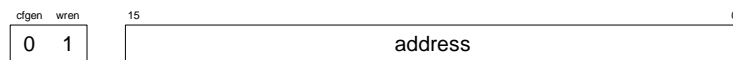


Figure 5.5: Memory write request command token format.

### Port Configuration Request

Port configuration requests allow modification of the 8-bit special function registers available in configuration space. These registers are used to configure the input port mode. Their use is detailed in Section 5.3.3. The format of a port configuration request command token is shown in Figure 5.6. Because the registers are only eight bits wide, the data is included in the lower byte of the command token. The upper byte of the command token contains the address of the register to be written. Addresses from 0x00 to 0x0F are allowed, although not all locations are implemented. The *cfgen* bit must be set to logic one and the *wren* bit set to logic zero to identify a short configuration request. Notice that the *wren* bit is set to zero, even though a configuration register is being written. This protocol was chosen because the write is implied by the command itself, and no additional data token is required. Short configuration requests always execute immediately.

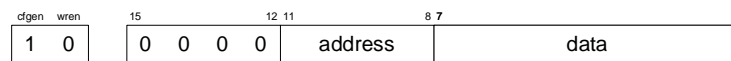


Figure 5.6: Port configuration request command token format.

### Address Generator Configuration Request

An address generator configuration request modifies one of the 16-bit special function registers used to configure the address generators. Because the registers are 16-bits wide, an additional data token must follow the command token. The token is always supplied by the same input port as the command token, regardless of the input port's mode. The format of an address generator configuration request command token is shown in Figure 5.7. The *cfgen* and *wren* bits must both be set to logic one in the command token of an address generator configuration request. In addition, the upper four bits of the command token must be 1000. Bits 11:8 are a one-hot field indicating which address generator is to be configured. The low byte of the command token contains the address to be written within the address generator's configuration space. The configuration of the address generators is detailed in Section 5.4.5. Address generator configuration requests execute as soon as valid data is available. Conflicting address generator requests from different ports may lead to undesirable behavior. The user must synchronize access to address generators to prevent conflicts.

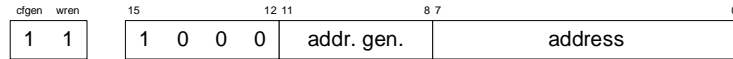


Figure 5.7: Address generator configuration request command token format.

### Burst Read Request and Burst Write Request

A burst request is used to issue up to 255 memory reads or writes using an address generator for addressing. This allows the overhead of issuing a memory request to be amortized over multiple memory accesses. The memory accesses generated by a burst request behave identically to those generated by a memory read or memory write request except that the address source is taken from an address generator, which is incremented after each access. The operation of the address generators is detailed in Section 5.4.5. The data source or destination is determined by the input port mode.

The format of a burst read request and a burst write request command token are shown in Figure 5.8 and Figure 5.9 respectively. The *cfgen* bit must be set to logic one, and the *wren* bit is set to indicate whether the request is for a burst read or a burst write. The upper four bits of the command token must be 1100. Bits 11:8 are a one-hot field indicating which address generator is to supply the burst addresses. The lower byte of the token indicates the length of the burst. A burst write requires a number of data tokens equal to the burst length. A burst request will block until all memory operations in the burst have been executed. During the burst other ports may be granted access to the memory based on the priority scheme used by the arbiter.

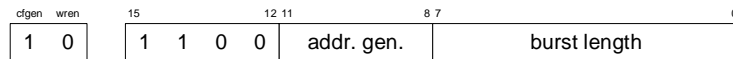


Figure 5.8: Burst read request command token format.

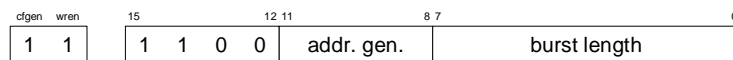


Figure 5.9: Burst write request command token format.

### Mutex Request

Mutex request commands are used to request exclusive control of a mutex primitive. These primitives can be used for synchronization among input ports or in the implementation of more complex mutual exclusion constructs. The operation of the mutex primitives is described in Section 5.4.4. A mutex request blocks until exclusive control of a particular mutex primitive is granted. No other input port will be granted control of the mutex until a mutex release command is issued. While deadlocks are a possibility among ports competing for mutexes, deadlock avoidance is left to the programmer. The format of a mutex request command token is shown in Figure 5.10. The *cfgen* bit must be set to logic one and the *wren* bit set to logic zero to identify a mutex request. In addition, the upper four bits of the command token must be 1111, and bits 7:4 must be set to 0001. The lower 4 bits are a one-hot field indicates which mutex is to be operated on. Bits 11:8 are not used.



Figure 5.10: Mutex request command token format.

### Mutex Release

A mutex release request is used to release control of a mutex primitive held by the issuing input port, allowing other ports to obtain control of the primitive. The mutex selected for release must be held by the issuing port or the request is ignored. Mutex release requests always execute immediately. The format of a mutex request command token is shown in Figure 5.11. The *cfgen* bit must be set to logic one and the *wren* bit set to logic zero to identify a mutex request. In addition, the upper four bits of the command token must be 1111, and bits 7:4 must be set to 0010. The lower 4 bits are a one-hot field indication which mutex is to be operated on. Bits 11:8 are not used.



Figure 5.11: Mutex release command token format.

| Port Mode    | Mode Bits | Mem Read Destination | Mem Write Source |
|--------------|-----------|----------------------|------------------|
| Disabled     | (00)      | -                    | -                |
| Data Only    | (01)      | -                    | -                |
| Address Only | (10)      | data out             | data in          |
| Address-Data | (11)      | out port             | self             |

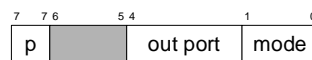
Table 5.2: Summary of input port modes.

### 5.3.3 Input Port Modes

Each input port in the FIFO-buffered memory module can operate in one of three modes. These modes affect how incoming memory and burst requests are serviced. Mode information is set in the port configuration registers using a port configuration request. These registers are unique to each input port, and can only be accessed by the port that contains them. The format of the registers is shown in Figure 5.12.

The mode field in port configuration register 0 sets the mode of the input port. If the mode field is set to 00 the port is disabled. The priority bit in register 0 can be set to increase the priority of the port when accessing memory, or mutex primitives. The other fields specify the source and destination of data tokens for memory requests. Which of the fields is used depends on the port mode. The various modes are summarized in Table 5.2. More detail is provided in the following sections.

Port Config Register (Address 0)



Data Config Register (Address 1)

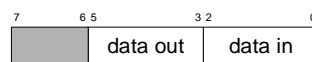


Figure 5.12: Port configuration register formats. These registers are written to change the port behavior. The bit labeled 'p' is the priority bit.

### Address-Data Mode (11)

Address-data mode is the most fundamental input port mode. In this mode, an input port performs memory reads and writes independent of other input ports. The destination for memory reads is specified by the *out\_port* field in the port configuration registers. This field is typically chosen so that the output port and input port connect to the same AsAP processor, but this is not strictly required.

A memory write is performed by first issuing a memory write request containing the write address. This request must be immediately followed by a data token containing the data to be written to memory. In the case of a burst write, the burst request must be immediately followed by the appropriate number of data tokens. Figure 5.13a illustrates how writes occur in address-data mode.

A memory read is performed by first issuing a memory read request, which contains the read address. The value read from memory is then written to the output FIFO indicated by the *out\_port* field. The same destination is used for burst reads.

In address-data mode, a single AsAP processor is required to generate memory addresses, as well as supply data for memory writes. For applications requiring significant computation for both address calculation and data processing, this can cause a decrease in performance. As an alternative, the tasks of address calculation and data processing can be partitioned among different processors by setting the input ports to address-only and data-only modes.

### Address-Only Mode (10)

In address-only mode, an input port is paired with an input port in data-only mode to perform memory writes. This allows the tasks of address generation and data generation to be partitioned onto separate AsAP processors. In this mode, the destination for memory reads is specified by the *data\_out* field in the port configuration registers.

In address-only mode, a memory write is performed by issuing a memory write request containing the write address. In contrast to operation in address-data mode, however, this request is not followed by a data token. Instead, the next valid data token from the input port specified by the *data\_in* field of the port configuration register is written to memory. Synchronization between

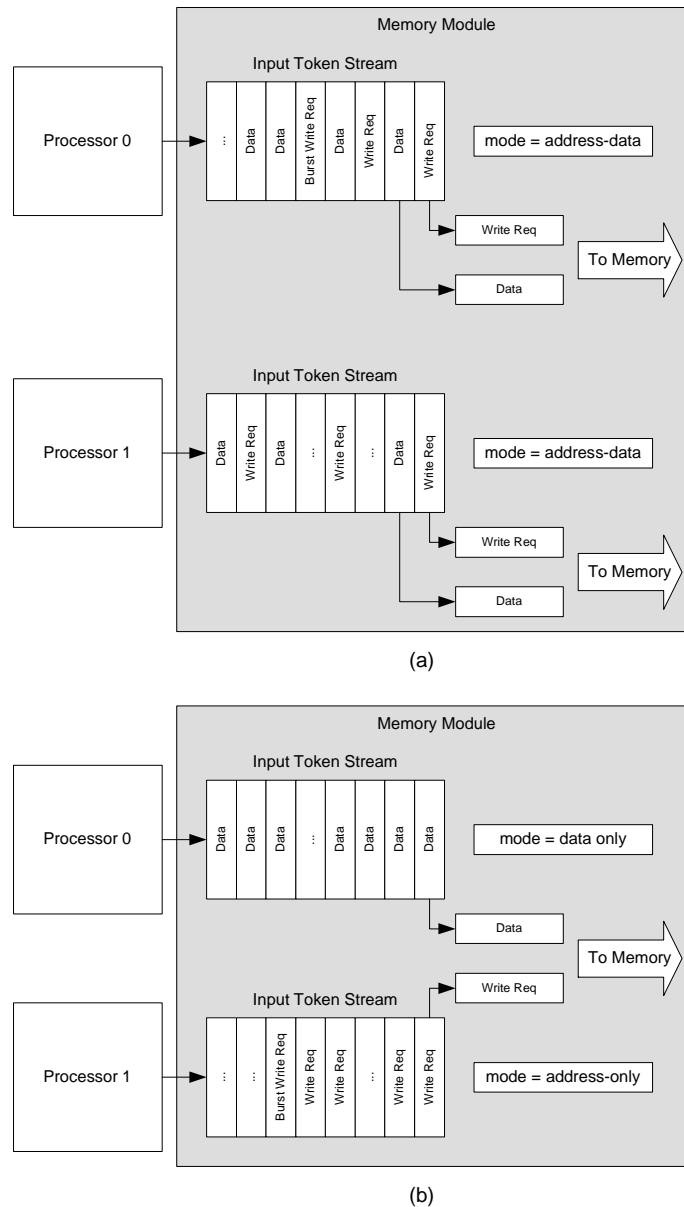


Figure 5.13: Memory writes in (a) address-data and (b) address-only mode. (a) In address-data mode, each port provides both addresses and data. Memory writes occur independently, and access to the memory is time-multiplexed. Two tokens must be read from the input stream to complete a write. (b) In address-only mode, write addresses are supplied by one port, and data are supplied by another. Memory writes are coupled, so there is no need to time-multiplex the memory among ports. One token must be read from each input stream to complete a write.

input ports is accomplished by maintaining FIFO order of incoming tokens. It is the programmer's responsibility to ensure that there is a one to one correspondence between write requests in the address-only port and data tokens in the data-only port. Figure 5.13b illustrates how writes occur in address-data mode.

Reads in address-only mode occur in the same way as for address-data mode, except that the output port is determined by the *data\_out* field instead of the *out\_port* field of the port configuration registers. This facilitates faster switching between the two modes if required at runtime.

### Data-Only Mode (01)

An input port in data-only mode acts as a slave to the address-only input port to which it provides data. All request types, with the exception of port configuration requests, are ignored when the input port is in data-only mode. Instead all incoming tokens are treated as data tokens. The programmer must ensure that at any one time, at most one input port is configured to use a data-only port as a data source.

### 5.3.4 AsAP Processor Instruction Set Mapping

The AsAP processor's memory port is accessed as either a source or a destination in existing AsAP instructions. The memory port is mapped into the existing DCmem space. This allows the addition of the memory port to the processor with no changes to the instruction set. The memory port is mapped to DCmem locations 28-31. A write to one of these DCmem locations causes the instruction result to be written to the input FIFO of the memory module. The *cfgen* and *wren* bits are set according to the address that is written. The lower two bits of the DCmem address correspond directly to the *cfgen* and *wren* bits. This mapping is summarized in Table 5.3. When one of DCmem locations 28-31 is used as a source, the next word in the output FIFO of the memory module is read, regardless of which address is used.

If a processor attempts to write to its memory port, but the destination FIFO is full, the processor will stall. Likewise, if the processor attempts to read from its memory port, but no data is available, a stall will occur. This stall behavior is similar to reading or writing the I/O ports of the processor.

| Access Type  | Destination | <i>cfgen</i> | <i>wren</i> |
|--------------|-------------|--------------|-------------|
| Memory Read  | DCmem 28    | 0            | 0           |
| Memory Write | DCmem 29    | 0            | 1           |
| Config Read  | DCmem 30    | 1            | 0           |
| Config Write | DCmem 31    | 1            | 1           |

Table 5.3: Memory port mapping to DCmem. This mapping applies to writes to the memory port. All reads to the memory port behave the same, regardless of the DCmem location accessed.

## 5.4 Memory Module Implementation

The implementation of the FIFO buffered memory module consists of several hierarchical blocks, each encapsulating a particular functionality. The basic interconnection of these blocks is shown in Figure 5.2. The top level design contains four pipe stages. Two are in the input ports, one is at the memory input, and one is at the memory output. Multiplexers are used at the top level to allow requests to be routed from each input port to each of the resources within the module. The multiplexer select signals are generated by the input ports, and the arbiter. The remainder of this section describes the function and implementation of each submodule.

### 5.4.1 Memory Core

The memory core used for primary storage in the memory module is a generated macro cell provided by Artisan [30]. A single-port synchronous SRAM cell is used. The maximum SRAM capacity available in a single cell is 8 K words. This size is used for the prototype implementation described here.

Additional capacity can easily be added by using multiple SRAM cores and adding additional logic to decode the higher order bits of the address. This additional logic will increase the number of pipe-stages in the module, but should not greatly affect performance. The prototype can address up to 64 K words with no additional changes.

### 5.4.2 Input Port Implementation

The input port encapsulates all the logic required to issue a request. The port consists of a small prefetch buffer, configuration registers, and control logic. The logic in the input port can

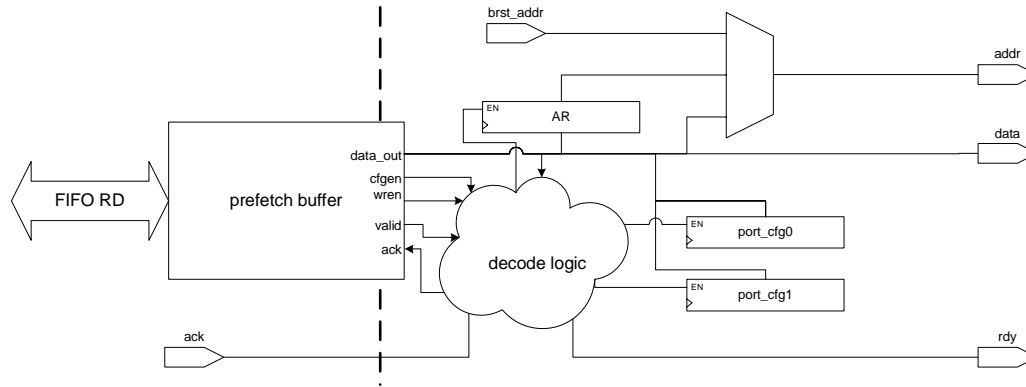


Figure 5.14: Single stage input port. The schematic is simplified for clarity. Most port control logic is omitted.

be placed in a single stage, easing the design of this block. If the design is to run at a frequency comparable to the SRAM cell's minimum cycle time, however, it must be pipelined into two stages. Pipelining this stage takes considerable design effort, but provides the required speed up.

The input port interfaces to the read side of a dual-clock FIFO, which is used to cross the clock boundary between the memory module and the processor. In addition to the request tokens, the FIFO must provide the *cfgen* and *wren* flags that accompany each token. To accommodate the additional bits, the memory module's input FIFOs must be 18 bits wide, while all other FIFOs in the ASAP system are only 16 bits wide. The alternative to widening the FIFO is to reduce each address space to 14 bits. To improve scalability and ease the use of 16-bit data, the wider FIFO is preferred unless area constraints are very high.

### Single Stage Input Port Design

A single stage implementation of the input port is straightforward. Requests that are not followed by data tokens are kept at the front of the queue until they are ready to issue. Requests followed by data tokens are moved to a temporary address register, AR. A finite state machine is used to track the validity of AR. The request is partially decoded, and if the request is blocking, the input port waits for the appropriate conditions to be satisfied before proceeding. When all conditions for execution of a request are met, the request is issued and the request queue is advanced. A schematic of the single stage input port is shown in Figure 5.14.

The majority of the module's control occurs in the input port. The logic is distributed

among the input ports to improve scalability. With control localized with each port, additional ports can be added with only minor design changes. The primary control logic is that which determines when a request is ready to be issued. This includes combinational logic to decode the request and check requirements, as well as a simple finite state machine (FSM) to track the state required for some requests.

Burst requests and requests that are followed by data tokens require the use of a FSM to issue properly. In the case of a burst read request, an additional cycle is required to load the burst counter with the burst length. One FSM state is used to differentiate a burst in progress from the initial burst cycle in which the counter is loaded.

Requests that are followed by data tokens include address generator configuration requests, and memory writes when address-data mode is used. In these cases, the command token of the request is copied to AR so that the data tokens may be accessed. When AR is loaded, a state bit is set to indicate that incoming tokens are to be treated as data and that the current request resides in AR.

The FSM used in each input port is shown in Figure 5.15. Five states and three state bits are used. In the *init* state, requests are issued directly from the head of the queue. In the *cfg\_wr*, *burst\_wr*, and *mem\_wr* states incoming tokens are interpreted as data, and the current request is held in the AR register. The FSM is in the *burst* or *burst\_wr* states when a burst request is in progress. The burst counter is loaded as these states are entered. While in these states, addresses are taken from the address generator indicated by the request rather than the input stream. In the case of a burst write in address-data mode, data is taken from the head of the queue. The FSM will remain in the burst states until the burst counter expires. When a request completes, the FSM returns to the *init* state.

Requests that access the memory (see Table 5.1) must check that all data dependencies are met before the request is issued. In short, these dependencies amount to having data available before performing a write operation and to having adequate space in the destination FIFO before performing a read operation. These conditions are complicated, however, by the introduction of multiple port modes and burst requests. The logic which requests access to the memory is described in Equations 5.1–5.4. Equation 5.1 checks that a valid memory or burst request is at the front of the request queue. Equations 5.2–5.3 check the data dependencies of the request. The final

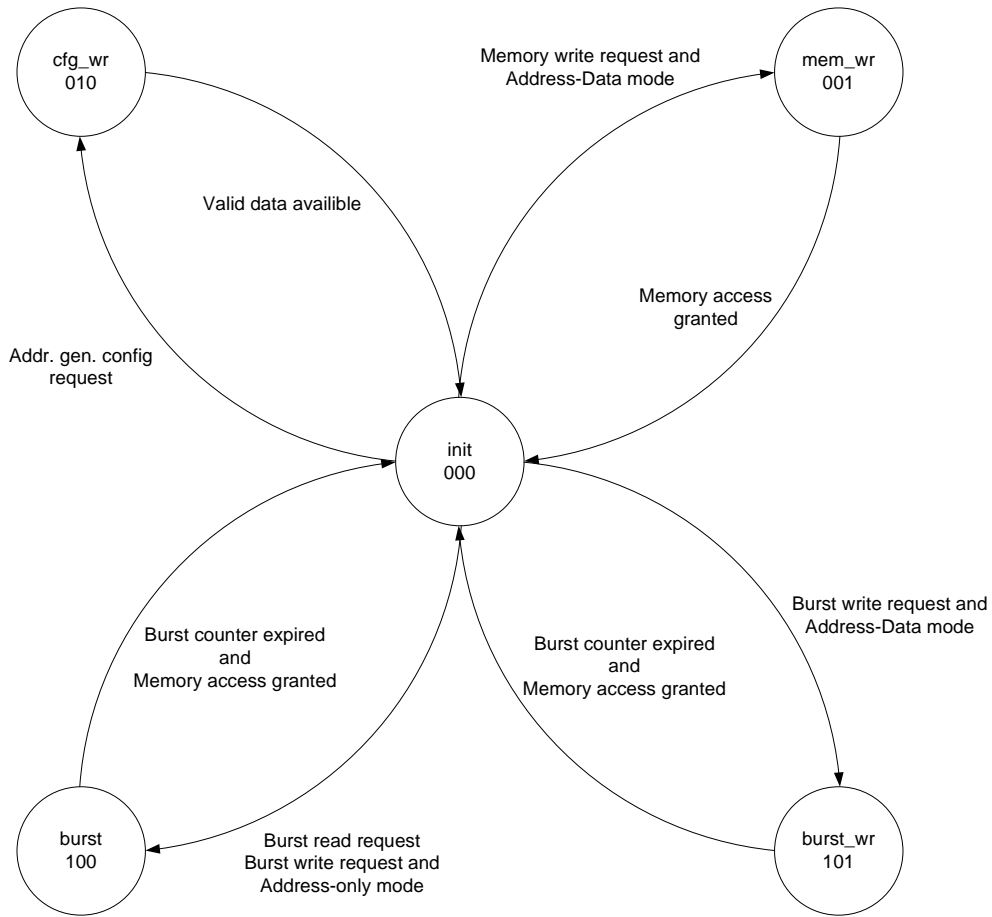


Figure 5.15: Input port finite state machine. The FSM tracks the state of burst requests and requests which are followed by data tokens.

equation for the *rdy* signal, which indicates that a request is prepared to access the memory is given in Equation 5.4. In these equations, *port\_mode* refers to the two-bit port mode field described in Section 5.3.3; *port\_ps* refers to the present state of the FSM shown in Figure 5.15.

$$mem\_request = (cfgen' + port\_ps_2) \cdot valid \cdot port\_mode_1 \quad (5.1)$$

$$rdy_{addr-data} = mem\_request \cdot (wren' \cdot dataout\_full' + port\_ps_0) \quad (5.2)$$

$$rdy_{addr-only} = mem\_request \cdot (wren' \cdot dataout\_full' + wren \cdot datain\_valid) \quad (5.3)$$

$$rdy = port\_mode'_0 \cdot rdy_{addr-only} + port\_mode_0 \cdot rdy_{addr-data} \quad (5.4)$$

### Two Stage Input Port Design

In the prototype design using the Artisan TSMC 0.18 $\mu$ m cell library and an 8 K-word generated memory, the single stage input port implementation was much slower than the SRAM cycle time. The critical path in the single cycle design includes decoding the request, checking execution conditions, requesting access to shared resources, and finally being granted access to the desired resources and advancing the request queue. To improve performance, the input port is divided into two stages.

The intuition for partitioning the input port comes from realizing that there are essentially two independent phases to the issue process. In the first stage, a request is decoded, and its execution conditions are checked. This includes checking for space in an output FIFO in the case of a memory read, or checking for valid source data in the case of a memory write. This stage is fundamentally the same as the single stage design, except that the conditions for advancing the queue are modified to accommodate the second stage.

In the second stage, the request is issued, and waits until access to any shared resources, such as mutex primitives or the memory bus, is granted. Once granted, the request leaves the input port and executes. This stage consists of additional registers to hold the required values. The critical path is reduced with the multistage design because the value of *rdy* is precomputed in stage one. Rather than having to first decode the request, and then wait for arbitration to finish, these two operations occur in parallel on consecutive requests. The data path for the two stage input port is shown in Figure 5.16.

The primary challenge in implementing a two-stage version of the input port is in deciding

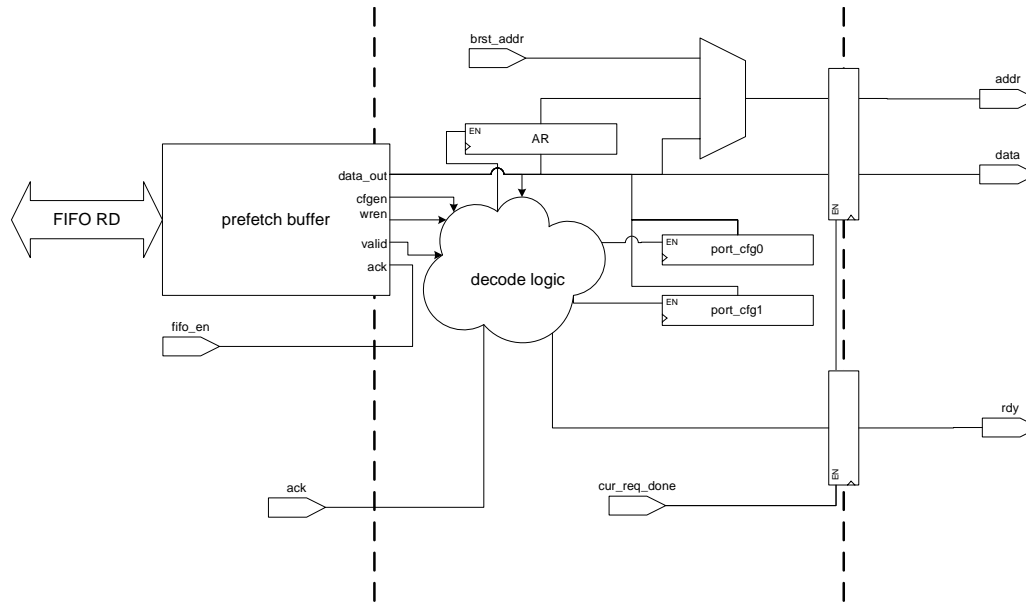


Figure 5.16: Two stage input port. The schematic is simplified for clarity. Most port control logic is omitted. The primary differences between the one and two stage designs are the additional pipe stage registers, and the stage enable signals.

when each pipe stage should be advanced. Advancing a stage too early could cause requests to be overwritten before they are issued. Advancing a stage too late will degrade the port's throughput. In addition, requests cannot affect the state of the input port prior to the execution of preceding requests as this would violate the in order execution semantics of the system. This logic adds significant overhead and limits the performance gain achievable from pipelining.

The enable logic for the output stage must hold the last issued request until it is accepted for execution. After execution, the stage must be advanced to prevent repeated execution of a single request. If the executing request completes before the following request is ready for issue, the request in stage one will be repeatedly sampled by the second stage until it is ready for issue.

The enable logic for the first stage, which advances the request queue, has slightly different requirements. This stage is advanced when two conditions are satisfied. First, the request currently at the head of the queue must be ready to issue. Second, the output stage of the input port must contain either an invalid request or a request that has been accepted for execution. This ensures that a request will only issue when all dependencies are met and prevents incomplete requests from being overwritten.

### **Coping with FIFO Latency**

A small prefetch buffer is used in the input port to hide the request latency of the dual-clock FIFO. In order to allow an input port to issue a request every clock cycle, the request at the front of the queue must be readily available for decoding. To prevent dropped requests the queue cannot be advanced until the currently pending request is executed. These requirements prevent the input port from tolerating any latency between the request of the next token from the queue and the availability of that token. The existing dual-clock FIFO, used in the ASAP processors, has a read latency of two cycles. Adding an additional buffer with a depth of three (one more than the latency to be hidden) allows data to be prefetched from the dual-clock FIFO, so that it is readily available to the input port's issue logic. A similar technique is used to hide the latency from the output port to the processor. This buffer design is discussed in Section 5.4.9.

### **Input Port Configuration**

In addition to the issue logic, the input port also contains logic to support different modes. The port's current configuration is stored in two port configuration registers, which are shown in Figure 5.12. These registers are written by port configuration requests. Writing these registers changes the port mode and priority, as well as the data source and sink used for memory access.

### **5.4.3 Arbiter Implementation**

Because requests from different input ports may attempt to access the memory module's shared resources simultaneously, arbitration is required to grant or deny access. An arbiter based on a least-recently-served scheduling policy is used to control access to the memory core. Each cycle, the least recently chosen port issuing a request is granted access to the resource. The arbitration may be overridden with a user-defined priority. Conflicting requests from ports of the same priority are resolved with the least-recently-served scheme. The same arbiter implementation is included in each mutex. Requests for other resources, such as the address generators, are not arbitrated. It is the responsibility of the user to ensure that conflicts do not occur with these resources by implementing mutual exclusion where appropriate.

The arbiter design consists of two parts. The first portion is a small sequential circuit

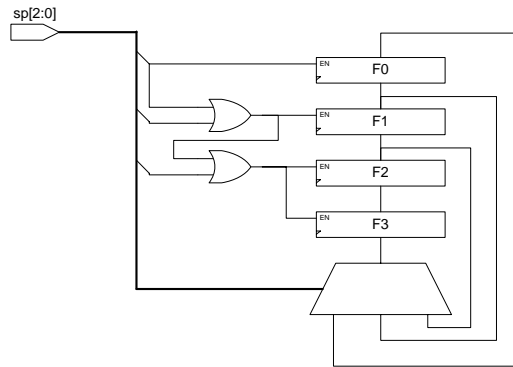


Figure 5.17: Arbitrator priority tracking circuit. A small sequential circuit is used to track the current priority of each port. Each port is assigned a number 0-3. The location of the port's number in the stack of registers determines the ports priority. When a port is serviced, it moves to the bottom of the stack, and the other ports shift up.

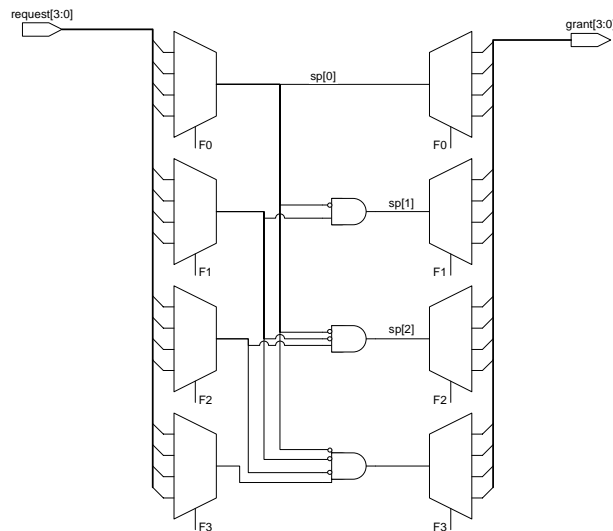


Figure 5.18: Arbitrator priority resolution network. A combinational network is used to select the highest priority port of all ports requesting access to a shared resource. The leftmost muxes reorder the requests based on the current priority of each port. The combinational network then selects the highest order request, and the de-muxes on the right shuffle the grant signals. The shuffled signals ( $sp[0]$ – $sp[2]$ ) are used to update the priority tracking circuit. The network is strictly combinational to allow requests to be granted in the same cycle in which they are issued.

which maintains the current priority of each input port. The circuit is shown in Figure 5.17. The current priority of an input port is indicated by storing that port's number in one of four two-bit registers F0-F3. The port corresponding to the number stored in F0 is granted the highest priority, F1 the second highest, etc. At reset, the circuit is initialized such that registers F0 through F3 contain the values zero through three, with each value appearing in exactly one register. When an input port is granted access to the resource, its value is moved from its current position to the lowest priority position, F3. The positions of the ports with lower priority than the one selected are increased.

The second portion of the arbiter is a combinational network that selects the highest priority request. This network is shown in Figure 5.18. Each input port has a dedicated request line, which is hardwired to the arbiter. When an input port needs access to the resource controlled by the arbiter, it asserts its request signal. The selection then occurs in three phases. In the first stage, an array of multiplexers controlled by the registers in the sequential portion of the arbiter re-order the request lines based on the relative priorities of the input ports. The requests then pass through a priority resolution network, which generates a one-hot output corresponding to the highest priority request signal with a value of '1'. Finally, a stage of demultiplexers undoes the shuffling carried out in the first stage. The resulting output is a one-hot vector indicating which port is granted access to the resource. This computation is completely combinational, which allows the input port to access the resource in the same cycle that it requests access.

The input ports' priority bits can be used to override the default arbiter priority. This is achieved by masking the request bit for each port with the corresponding priority bit. If any of the priority bits are asserted, then the masked version of the request bits is forwarded to the arbiter; otherwise, the unmasked version is used. The speed of this approach can be increased slightly by using two copies of the combinational network. One copy operates on the the request bits after masking them with the priority bits from the input port; the other operates on the unmasked version of the request bits. The end result is chosen at the output of the two networks. This allows the detection of asserted priority bits to occur in parallel with the priority resolution. This scheme is shown in Figure 5.19. This strategy becomes more important as the number of input ports increases.

The scalability of the arbitration unit is a limiting factor in the memory module design. The complexity of the priority resolution network grows quadratically with the number of inputs. If the arbitration is to complete in a single cycle, introducing many more than four input ports will

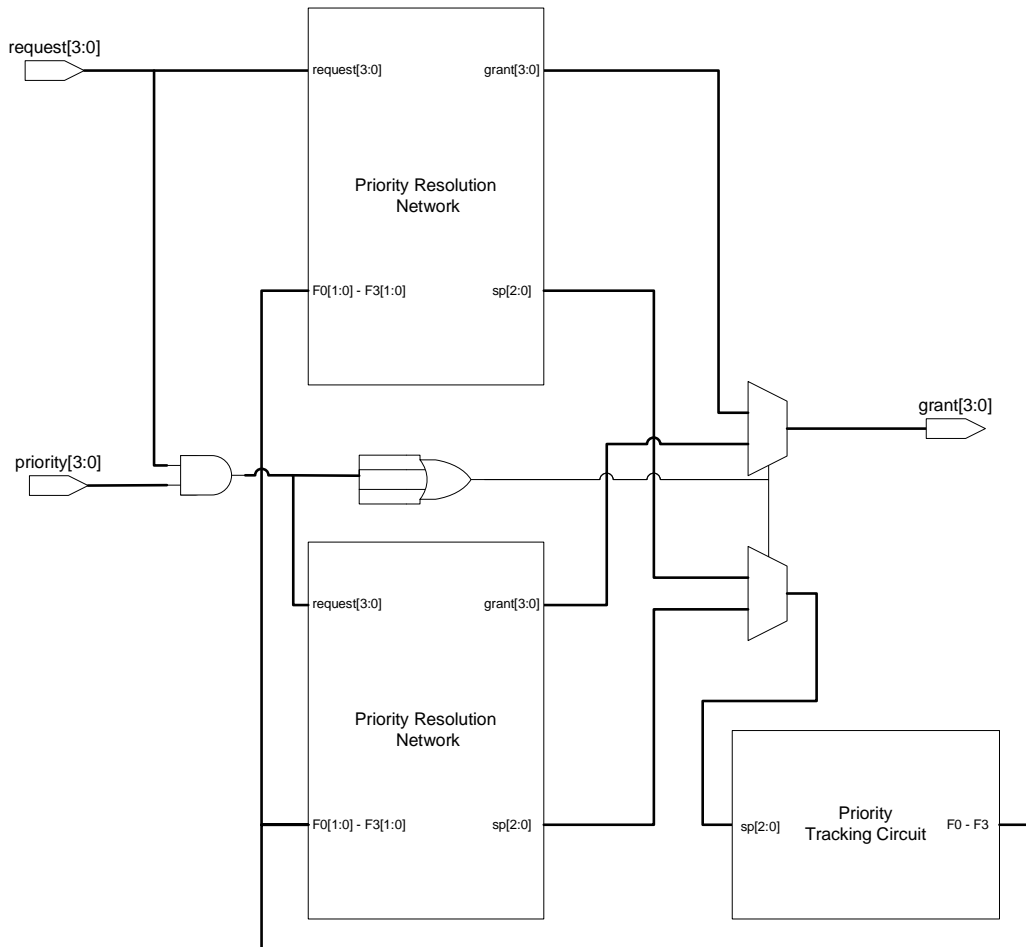


Figure 5.19: Two priority resolution networks can be used in parallel to gain a slight speedup when resolving priority override bits.

severely limit the module's cycle time.

#### 5.4.4 Mutex Implementation

The FIFO-buffered memory module shares its memory resources among multiple asynchronous threads, so inter-process synchronization is required. Because each port operates concurrently and asynchronously with respect to the others, there is no reliable method of implementing mutual exclusion in software without allowing the processes to communicate outside of the memory module. External communication is not always feasible, so dedicated mutual exclusion primitives are integrated in hardware to enable synchronization.

The mutual exclusion primitives implement simple test and set locks. The lock is either

available, or held by a particular input port during any given cycle. Input ports request and release a lock using the appropriate mutex requests, as detailed in Section 5.3.2.

When an input port issues a mutex request, it asserts its *mutex\_req* signal. The request blocks until the mutex is granted. This ensures that the port is holding the lock before any requests following the mutex request are executed. An arbiter is used to resolve multiple simultaneous requests for a single mutex. The arbiter design is identical to that used to resolve concurrent memory access requests. The mutex is released when the owner asserts its *mutex\_rel* signal. The *mutex\_rel* signal is asserted when a mutex release request is executed. The release request does not block, and is guaranteed to finish execution in a single cycle. If there are additional requests pending for the mutex, the highest priority request will be immediately granted. Otherwise, the mutex will become available. The implementation of a single mutex is shown in Figure 5.20. There are four mutexes included in the design, and more can easily be added if required.

In the initial design, the path from *mutex\_req* to *mutex\_grant* was in the critical path of the system. This introduced a major challenge for timing closure, because of the large number of critical paths. To reduce the length of the combinational path from *mutex\_req* to *mutex\_grant*, the grant process is divided into two stages. This introduces a trade-off between cycle time and cycles per request. With a two-stage mutex request, the minimum time in which a mutex request can be granted is two cycles. A single cycle grant would be possible with a single stage design. However, with a single stage design the cycle time for all requests would be reduced. Because mutex requests are rare compared to memory accesses, choosing to increase the latency of mutex requests while boosting the maximum cycle time of the system yields higher performance than keeping the single stage design and taking a cycle time hit.

#### 5.4.5 Address Generator Implementation

In general, a single memory access requires at least two instructions. One instruction must be used to issue the memory address. The second instruction is required to read or write the data. Additional instructions are required if the address or data must be calculated on the fly. If the cycle time of a processor is similar to the cycle time of the memory module, this overhead will limit the potential throughput of the memory. Hardware address generators are included in the memory

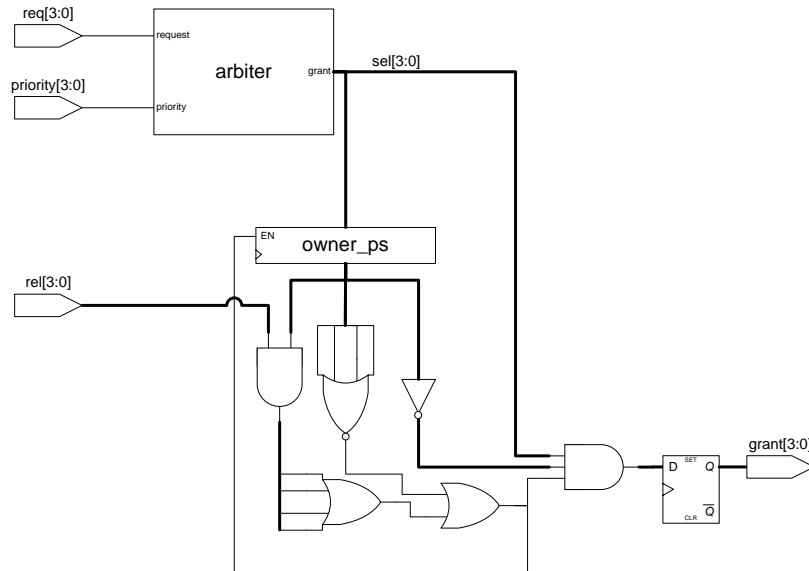


Figure 5.20: Mutual exclusion primitive (mutex). The mutex implements an atomic test and set lock, allowing a means for inter-process synchronization and for building more complex mutual exclusion constructs. The current owner of the mutex is stored in *owner\_ps*. The arbiter manages simultaneous mutex requests.

module to allow blocks of data to be read from or written to memory with a single burst request. When using an address generator for a burst access, the average number of processor instructions per memory access is reduced. For a burst size of 100, only 1.01 processor instructions are required per access.

The applicability of the burst request to a particular application depends on the access patterns used in the data. Some DSP algorithms, such as the FFT [25] and Viterbi decoding [31] require complex address generation. Many applications, however, require only sequential or strided access into memory [23, 32]. It is not practical to support every possible access pattern in hardware. This would lead to degradation in system performance and a large increase in configuration overhead. As a compromise, the address generation hardware supports variable-stride modular access within an arbitrary block size. This provides a reasonable degree of flexibility while keeping the hardware design very simple. More complex access patterns should be implemented using a dedicated processor in address-only mode.

Each address generator implements a variable-stride counter, which is configurable based on three configuration registers. These registers are addressable using an address generator config-

| Parameter  | Address | Description                                                                                                                               |
|------------|---------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Offset     | 0       | The start address of the block accessed by the generator. The generator is reset when this location is written.                           |
| Block Size | 1       | The number of words in the block accessed by the generator.                                                                               |
| Stride     | 2       | The distance to move the generators address with each access. This value should not be greater than block size for predictable operation. |

Table 5.4: Address generator configuration. The parameters listed in this table determine the behavior of the address generators. The parameters are set using address generator configuration requests to the address shown in the address column.

uration request. The registers are described in Table 5.4. When *offset* is loaded the address counter is reset to zero. The count is increased by the *stride* with each access. When the count reaches or exceeds *blocksize*, it wraps back around past zero. The address output for the  $n^{\text{th}}$  access is given recursively by Equations 5.5– 5.6. The generator hardware consists of three 16 bit adders, and a count register. The hardware is shown in Figure 5.21.

$$count[n] = (count[n - 1] + stride) \bmod blocksize \quad (5.5)$$

$$address[n] = count[n] + offset \quad (5.6)$$

Each address generator also contains an 8-bit burst counter. This counter is loaded when an input port issues a burst request. Each memory access in the burst is handled like a regular memory request, except that the request queue is not advanced while the burst is in progress. The counter decrements each time the port is granted memory access. The burst counter indicates the completion of the burst request by asserting the *burst\_done* signal. Upon assertion, the input port completes the final access of the burst and advances the request queue. The maximum burst request size is 255.

The maximum burst length can be increased to 256 by loading the burst counter count with *burst\_length* + 1 when a burst request is received, instead of with *burst\_length* as in the current design. This will require the burst counter to be extended by one bit. In addition, the change will alter the specification for burst requests, preventing backward compatibility with existing software.

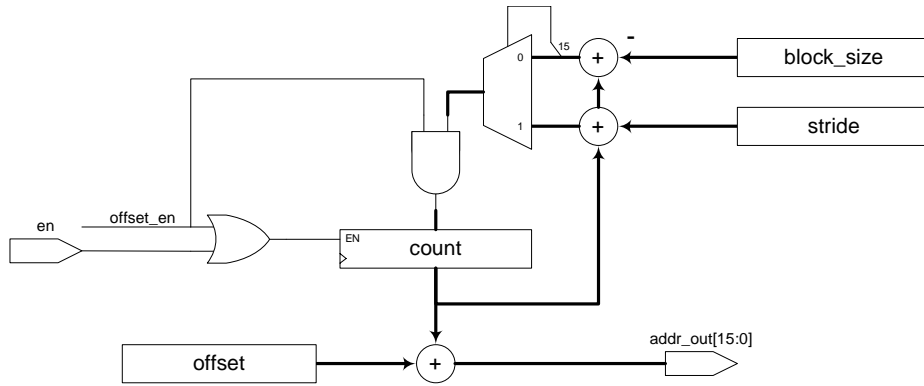


Figure 5.21: The address generator datapath consists of three adders and a register to hold the current address. The parameters used to generate the addresses are set in the address generator configuration space.

### 5.4.6 Output Port Implementation

The only outputs of the FIFO-buffered memory module are read data. This greatly simplifies the output port implementation. Each port is simply a dual clock FIFO. The *wr\_valid* signal for each FIFO is generated by a decoder, based on the destination of the required request. The primary concern in the output port implementation is the prevention of writes to a full output port. To prevent FIFO overflow, memory read requests are not allowed to issue unless there is adequate space in the destination FIFO. The latency between the issue stage and the actual FIFO write is handled by adjusting the reserve space, as described by Apperson [29].

### 5.4.7 Module Clocking

The FIFO-buffered memory module is asynchronous to the rest of the ASAP system, and therefore must receive its own local clock. The requirements on this clock are not strict, as long as minimum cycle time requirements are met. The easiest solution is to provide a clock signal from off chip. This solution is not feasible, however, if many modules on the chip must receive their own clocks, as the routing of these clocks will become a limiting factor. In addition, the maximum speed of an off-chip clock will be limited by the capacitance of the input pin. Instead, an oscillator is integrated into the module and generates a local clock signal. The oscillator is adjustable for module characterization and power/performance tuning. In addition, the oscillator may be paused to reduce power consumption during idle time.

### **Oscillator Implementation**

The oscillator used in this module is borrowed from the AsAP processor design. Because a standard cell flow is used, the oscillator is constructed from standard cells. The frequency is changed by controlling the drive strength of each oscillator stage, while its load remains relatively fixed. This is accomplished by placing a number of tristate buffers in parallel with an inverter, as described by Olsson and Nilsson [33]. The number of tristates which are on is inversely proportional to the delay of the stage.

To safely pause the clock, a mutually exclusive element is used. This circuit is commonly used in asynchronous circuit design. The circuit consists of a latch, followed by a metastability filter. Typically, a full-custom circuit is used to create the metastability filter. This is not possible in a standard cell design, so a wide NOR gate, with all inputs tied together, is used instead. This solution is described by Kessels and Marston [34].

### **Clock Stall Conditions**

If a pausable oscillator is used to generate the system clock, the conditions for stalling the clock must be derived. In general, the clock should be stalled whenever the system is idle to maximize power savings. The actual frequency of stalls, however, should be determined by weighing the overhead of stalling the clock with the time that the clock will remain stalled.

The stall conditions used for this system check that all request queues are empty and that no requests are in execution. When these constraints have been met the clock is paused. The clock is restored when any of the dual clock FIFOs are no longer empty. The restore condition must be calculated asynchronously, because it must be asserted in the absence of the local clock.

A more aggressive clock stalling strategy might attempt to stall the clock when there are requests to be issued, but all are waiting for their execution requirements to be met. This approach was not taken because it would cause the clock to be stalled more often, for shorter periods of time. It is estimated that a high frequency of clock stalling could reduce performance.

### 5.4.8 Module Configuration

Certain parameters must be configured at startup for the proper operation of the memory module. These parameters are static during the standard operation of the module, but may change from one application mapping to another. Following the model set forth by the AsAP processor design, the global configuration bus is used to perform the configuration. The configuration space also provides module level reset signals.

The design for the configuration block is borrowed from the AsAP processor. The only modifications made to this design are the addition of new parameters specific to the memory module, and the omission of parameters that are not applicable. The configuration block listens to the global bus for transactions that are addressed to the memory module, and writes the appropriate configuration registers. The registers are written asynchronously to the local clock, and configuration signals are synchronized where appropriate. The various parameters in the static configuration space are shown in Table 5.5. The addresses for the parameters are chosen to avoid conflicts with the parameters in the AsAP processor configuration space.

### 5.4.9 Processor Port Implementation

In addition to the design of the memory module itself, some modifications to the AsAP processor are required to interface the memory. The alternative to adding a dedicated memory port to the processor is to interface the memory with the existing I/O ports. This solution reduces the overhead of memory support, but also limits the flexibility of the design. If existing I/O ports are used, processors must dedicate one of two input FIFOs to the memory interface, limiting connectivity. Also, the processor's output port must be controlled in software, adding additional code complexity. Finally, the memory interface must be adapted to utilize a 16 bit input, rather than the current 18-bit design. This motivates the addition of a dedicated memory port.

The processor uses the memory port to access to the memory module's input and output FIFOs. As described in Section 5.3.4, the memory ports are mapped to the DCmem address space in the AsAP processor. When the processor writes to a memory port location, the memory port writes the result to the memory module's input port, with the *cfgen* and *wren* flags set appropriately. When the processor reads from a memory port location, the next word from the memory module's output

| Parameter           | Address[bits]<br>(Hex) | Description                                                                                                                                                                         |
|---------------------|------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| cfg_halt            | 00[0]                  | Stop the operation of the memory module. This bit is kept for compatibility, but currently has no effect.                                                                           |
| cfg_clk_enable      | 00[1]                  | Enable the local oscillator. Even if this bit is de-asserted, the external clock source can be used to clock the module.                                                            |
| cfg_stall_disable   | 00[3]                  | Prevent the clock from stalling. When asserted, the oscillator will continue to run when the system is idle.                                                                        |
| cfg_reset           | 00[3]                  | Active-high synchronous reset for blocks within the module.                                                                                                                         |
| cfg_reset_fifo      | 00[6]                  | Provides a reset signal to all input FIFOs. Output FIFOs are reset by the processors configured to read them.                                                                       |
| cfg_freq            | 01[7:0]                | Configures the clock frequency of the local oscillator. The absolute frequency value depends on the oscillator implementation, and may vary with process, temperature, and voltage. |
| cfg_clk_ext         | 04[0]                  | Allows an external clock source to be selected as the local system clock.                                                                                                           |
| cfg_connect_in0..3  | 10[2:0]..13[2:0]       | Determines which processor connects to input port 0..3 in a system with reconfigurable interconnect.                                                                                |
| cfg_connect_out0..3 | 14[2:0]..17[2:0]       | Determines which processor connects to output port 0..3 in a system with reconfigurable interconnect.                                                                               |
| cfg_fifo_rsrv_in    | 18[4:0]                | Configures the reserve space parameter for the input FIFOs [29]. This parameter is determined by the latency from the memory module to the processor.                               |
| cfg_fifo_rsrv_out   | 19[4:0]                | Configures the reserve space parameter for the output FIFOs [29]. This parameter is determined by the depth of the module pipeline.                                                 |

Table 5.5: Static configuration parameters. These parameters are configured at startup via the global configuration bus.

port is provided as a source operand.

To ease physical design of the AsAP array, it is desirable to use uniform processing elements. To this end, every processor in an array contains a memory port, whether or not it connects to a memory. This adds additional area overhead to processors which do not use the memory. The design was chosen to minimize the area of the memory port, while still providing the desired functionality.

The primary challenge in the design of the memory port is coping with physical delays due to potentially long interconnect between the processor and memory module. In AsAP processors, latency between a processor's output and another processor's input can be accommodated by adjusting the FIFOs reserve space. This solution is viable because the read side of the FIFO is integrated into the processor itself. In the case of the memory module, both the input and output FIFOs are in the module. A reserve space change can accommodate additional latency on the write side of the FIFO, but is not applicable to the read side. Because the read side of the output FIFOs for the memory module may require additional latency, an alternative solution is proposed in the following section.

### **Coping with Output Port Latency**

When laying out an AsAP array with a FIFO-buffered memory module, it may become necessary to add additional pipeline stages between the processor's memory port and the memory's input and output ports to accommodate the latency of long wires. The added latency to the input port can be accommodated by adjusting the reserve space of the input FIFOs. This solution is not viable for the output ports, because reserve space is only applicable in the context of a FIFO write. Instead, a small prefetch buffer is included in each processor, and is used to hide the latency of reads from the memory port.

Rather than waiting for a read request to occur, the prefetch buffer constantly requests data from the memory module's output port. Each cycle, a delayed version of the FIFO's empty flag is checked to determine if the incoming data is valid. If the word is valid, it is added to the buffer; otherwise it is ignored. To prevent buffer overflow, the number of pending requests is compared against space available in the buffer. If the number of requests is less than the space available, a request is issued. This results in the buffer maintaining the fullest possible state, provided that data

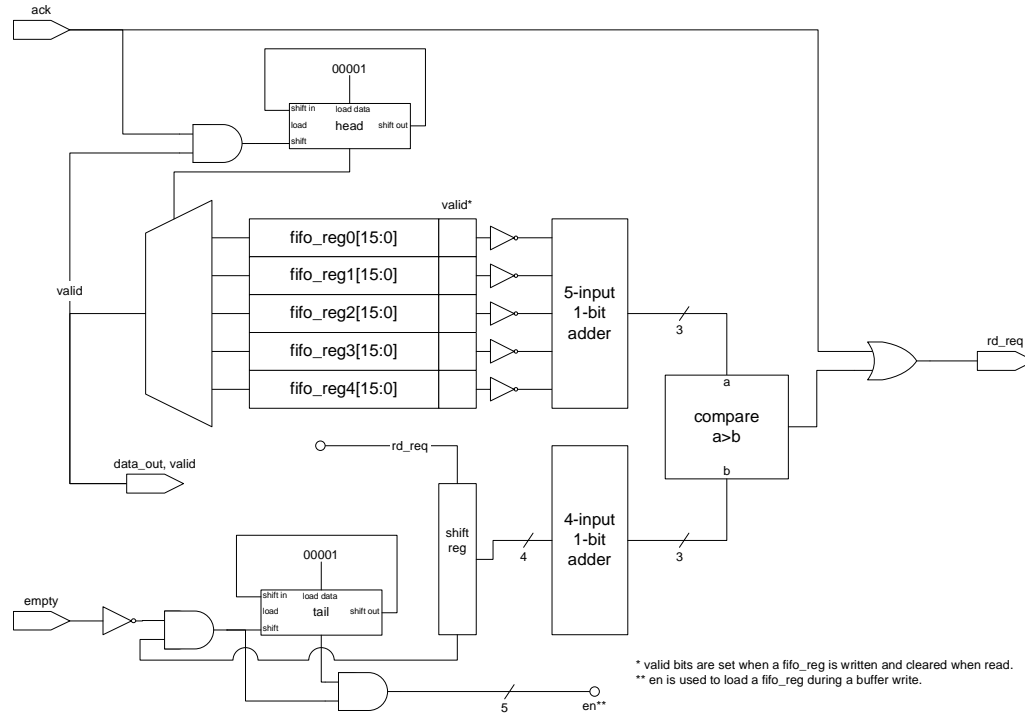


Figure 5.22: Processor memory port prefetch buffer. A small prefetch buffer is included in the memory port to accommodate additional latency from the memory module output FIFO to the processor memory port. The buffer requests data from the FIFO as long as space is available in the buffer. The requests are generated by counting the number of pending requests, and comparing it against the number of empty registers in the buffer. The buffer itself is managed as a circular FIFO, using shift registers to manage the head and tail of the queue. The buffer depth is determined by the request latency.

is available at the buffers input.

Unless the source FIFO is empty, the processor should be able to read one word per cycle from the memory port without stalling. This is achievable by selecting an adequate buffer depth. Assuming adequate time is given to fill the buffer, the buffer depth required to sustain maximum throughput is determined by the latency from the memory port to the memory module. If there are  $L$  pipe stages between the memory port and the source FIFO, the round trip time of a request is  $2L$  cycles. An additional cycle must be allocated to latch the request into the buffer. This implies a buffer depth of  $2L + 1$  for maximum throughput.

The buffer design is shown in Figure 5.22. The figure shows a buffer for  $L = 2$ . Modifying the design to support different latencies is straight forward. To support additional latency, registers must be added to increase the buffer depth, and the width of the shift registers and adders

used in the control logic must be increased. To reduce the latency, bits can be masked to prevent buffer slots from being used. The buffer design implemented for the AsAP design is configurable for  $L = 1$  or  $L = 2$ , allowing for zero or one pipe stages between the processor and memory module. This provides additional flexibility in physical memory placement without requiring multiple processor designs. For proper operation, the latency must be known at design time and the buffer configured accordingly. Otherwise, it is impossible to synchronize the outgoing requests with the incoming responses.

## 5.5 Design Summary

The FIFO-buffered memory module is designed to provide additional memory capacity to processors in an AsAP array. The module fits well into the AsAP paradigm, because the module's interface, configuration, and clocking are the same as the AsAP processors themselves. The memory module contains additional features aimed at increasing the performance and usability of the design. These include hardware support mutual exclusion and address generation, as well as flexible port modes.

The programming interface to the memory module is built around a small set of requests. This allows the memories to be easily interfaced by any processor, without substantial changes to the instruction set. Requests allow random access to the memory core, as well as manipulation of the additional hardware features. The simple interface and flexible hardware make the buffered memory module an effective solution for expanding AsAP's memory capacity.

## Chapter 6

# Implementation Results

The FIFO-buffered memory module described in Chapter 5 has been modeled in Verilog and synthesized with the Artisan 0.18  $\mu\text{m}$  SAGE-X<sup>TM</sup> standard cell library [35]. Speed, power and area results were estimated from high level synthesis. In addition, the design's performance was analyzed with RTL level simulation. This chapter discusses these results.

### 6.1 Performance Results

System performance was the primary metric motivating the memory module design. Two types of performance are considered. First, the system's peak performance, as dictated by the maximum clock frequency, peak throughput, and latency is calculated. A more meaningful result, however, is the performance of actual programs accessing the memory. Both of these metrics are discussed in the following sections.

#### 6.1.1 Peak Performance

##### Throughput

The peak performance of the memory module is a function of the maximum clock frequency, and the theoretical throughput of the design. The FIFO-buffered memory module is capable of issuing one memory access every cycle, assuming that requests are available and their data dependencies are met. In address-data mode, memory writes require a minimum of two cycles to issue, but this penalty can be avoided by using address generators or the address-only port mode,

or by interleaving memory requests from multiple ports. If adequate processing resources exist to supply the memory module with requests, the peak memory throughput is one 16-bit access per cycle. Synthesis results report a maximum clock frequency of 555 MHz. At this clock speed, the memory's peak throughput is 8.8Gb/s. The performance will be slightly lower after place and route.

## Latency

The worst case memory latency is for the memory read request. There are contributions to this latency in each of the system's clock domains. In the processor's clock domain, the latency includes one FIFO write latency, one FIFO read latency, and the additional latency introduced by the memory port. In the memory's clock domain, the latency includes one FIFO read latency, the memory module latency, and one FIFO write latency.

The minimum latency of the memory module is given by the number of pipe stages between the input and output ports. The current memory implementation has four pipe stages. The number of stages may be increased in the future to add address decoding stages for larger memories.

The latency of FIFO reads and writes is dependent on the number of pipe stages used to synchronize data across the clock boundary between the read side and the write side. In the current FIFO design [29], the number of synchronization stages is configurable at runtime. When a typical value of three stages is used, the total FIFO latency is four cycles per side. When the minimum number of stages is used, the latency is reduced to three cycles per side. A latency of four cycles is assumed in this work.

The latency of the memory port depends on the number of stages introduced between the processor and the memory to account for wire delays. The minimum latency of the memory port is two cycles. This latency could be decreased by integrating the memory port more tightly with the processor core datapath. This approach hinders the use of a prefetch buffer to manage an arbitrary latency from processor to memory, and is only practical if the latency can be constrained to a single cycle.

Summing the latency contributions from each clock domain, the total latency of a memory read is found, as shown in Equations 6.1–6.3.

$$L_{proc} = L_{FIFO-wr} + L_{FIFO-rd} + L_{mem-port} \quad (6.1)$$

$$L_{mem} = L_{FIFO-rd} + L_{mem-module} + L_{FIFO-wr} \quad (6.2)$$

$$L_{total} = L_{mem} + L_{proc} \quad (6.3)$$

For the current design configuration, the latency is 10 processor cycles and 13 memory cycles. If the blocks are clocked at the same frequency, this is a minimum latency of 23 cycles. Additional latency may be introduced by processor stalls, memory access conflicts, or data dependencies. The latency is slightly higher than typical L2 cache latencies, which are on the order of 15 cycles [2], due to the communication overhead introduced by the FIFOs. This high latency can be overcome by issuing multiple requests in a single block. Because the requests are pipelined, the latency penalty only occurs once per block.

### 6.1.2 Actual Performance

To better characterize the design's performance, the memory module was exercised with two workloads. The single-element workload performs a copy of a 1024-element array. Multiple implementations of the workload were tested to exploit different features of the memory module. The block workload first writes a block of 1024 words. After the write completes, the block is read back. The number of instructions in each kernel is varied to simulate the effect of varying computation loads for each application. Figure 6.1 shows pseudo-code for the two workloads.

The single-element workload performs a simple array copy and consists of three phases. First, a burst write is used to load the source array into the processor. Second, the array is copied element by element, moving one element per loop iteration. Finally, the resulting array is read out with a burst read. The number of instructions in the copy kernel is varied to simulate various computation loads.

The primary limitation of the single-element kernel is the memory read latency. Because each memory read is forced to complete before others are issued, the latency affects every access. To reduce the effect of latency on the test workload, the block test is used. This workload first writes 1024 memory words, and then reads them back. Three coding approaches are compared. The first uses a single processor, executing a single read or write per loop iteration. The second uses

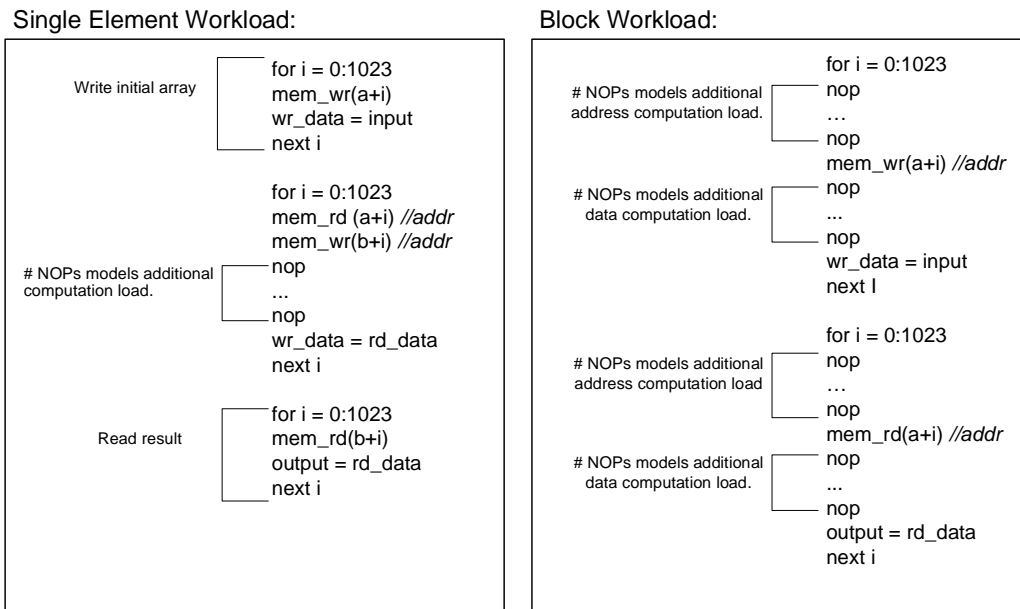


Figure 6.1: Two workloads are used for performance characterization. Pseudo-code for the two workloads is shown for processors in address-data mode. In each workload, the computation load is simulated by adjusting the number of NOPs in the main kernel. The block workload is also used in address-only/data-only mode. In this mode, the code that generates memory requests, and the code that reads and writes data is partitioned appropriately. Note that *rd\_data* indicates data being read from the processor's memory port. *wr\_data* indicates data being written to the port. *mem\_rd()* and *mem\_wr()* indicate that a read or write request is being issued with the specified address.

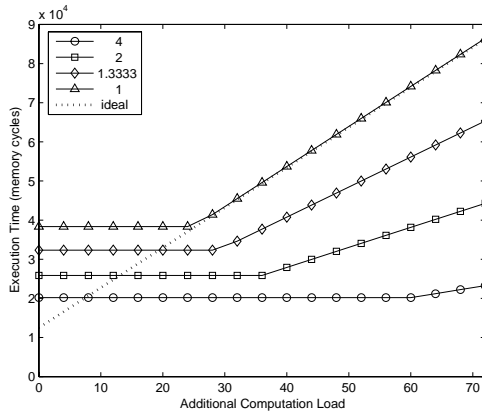


Figure 6.2: Effect of computational load and clock speed on performance. The figure shows the execution time of the single-element workload for a single processor clocked at 1, 1.33, 2 and 4 times the memory speed. The dotted line represents the theoretical maximum performance for the workload operating on a single processor clocked at the same speed as the memory.

burst requests to perform the memory accesses. The final approach partitions the task among two processors in address-only and data-only modes. One processor issues the request addresses, while the other manages data flow. Again, the number of instructions in each kernel is varied to simulate various computation loads.

### Computation Load and Clock Speed

Figure 6.2 shows the performance results for the single-element workload running on a single processor at different clock speeds. For small workloads, the performance is dominated by the memory latency. This occurs because each iteration of the loop must wait for a memory read to complete before continuing. A more efficient coding of the kernel could overcome this latency using loop unrolling techniques. This may not always be practical, however, due to limited code and data storage. The bend in each curve occurs at the location where the memory latency is matched to the computational workload. Beyond this point, the performance scales with the complexity of computation.

The processors clock speed has the expected effect on performance. At high frequencies, the performance is still limited by memory latency, but larger workloads are required before the computation time overcomes the read latency. The latency decreases slightly at higher processor frequencies because the component of the latency in the processor's clock domain is reduced. The

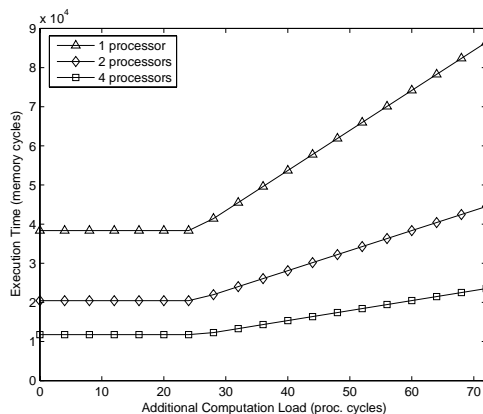


Figure 6.3: Effect of number of processors on performance. The figure shows the execution time of the single-element workload for 1, 2, and 4 processors clocked at the same frequency as the memory. The execution time for each case includes some fixed overhead to initialize and read the source and destination arrays. Multiple processor cases have additional overhead for synchronization among processors.

slope of the high-workload portion of the curve is reduced because the relative impact of each additional instruction is less at higher frequencies.

### Computation Load and Multiple Processors

For highly parallel workloads, the easiest way to improve performance is to distribute the task among multiple processors. Figure 6.3 shows the result of distributing the single-element workload across one, two, and four processors. In this case, the 1024 copy operations are divided evenly among all of the processors. When mapped across multiple processors, one processor performs the initial array write, and one processor performs the final array read. The remainder of the computation is distributed uniformly among the processors. Mutexes are used to ensure synchronization between the initialization, copy, and read-out phases of execution.

When the single-element workload is shared among processors, the applications performance is increased at the cost of additional area and power consumed by the additional processors. For small computation loads, the effective read latency is reduced. Although each read still has the same latency, the reads from each processor are issued concurrently. Hence, the total latency suffered scales inversely with the number of processors used. For loads where latency is dominated by computation cost, the impact of the computation is reduced, because multiple iterations of the

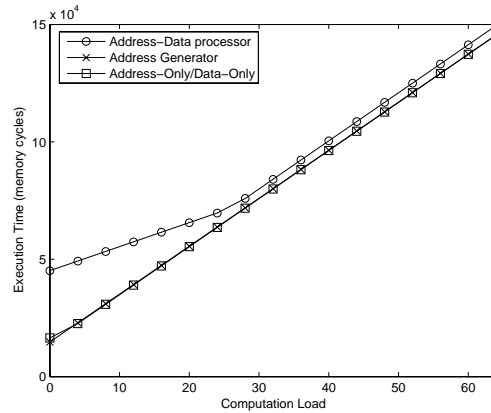


Figure 6.4: Effect of address mode on block performance. The figure shows the execution time of the block workload for a single processor in address-data mode, a single processor utilizing address generator hardware, and two processors, one in address-only mode and one in data-only mode. Both the address generator and address-only mode solutions outperform the address-data mode solution if the work load is dominated by the memory latency. Note that the address generator and address-only performances are roughly equal.

application kernel run concurrently on the various processors. Note that the point where computation load begins to dominate latency is constant, regardless of the number of processors used. The relative latency depends only on the relative clock speeds of the processors and memories, and not on the distribution of computation.

### Computation Load and Address Mode

Figure 6.4 shows the performance of the three schemes for the block workload when the processors and memory are clocked at the same frequency. For small workloads, the address-data mode solution is dominated by read latency, and write workload. Because writes are unaffected by latency, the computation load has an immediate effect. For large workloads, the execution time is dominated by the computation load of both reads and writes.

The address generator and address-only/data-only solutions decouple the generation of memory read requests from the receipt of read data. This allows requests to be issued far in advance, so the read latency has little effect. There is also a slight performance increase because the number of instructions in each kernel is reduced.

The address generator solution outperforms the single cycle approach, and does not re-

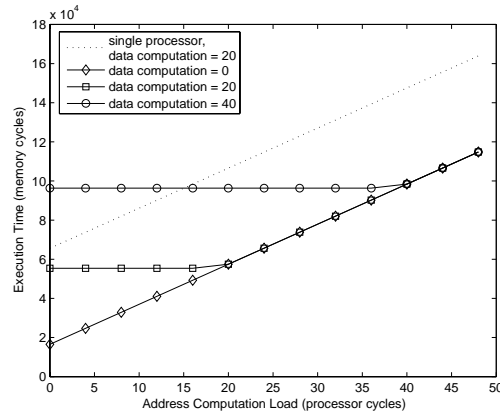


Figure 6.5: Effect of address load on address-only mode performance. The figure shows the execution time of the block workload for a single processor in address-data mode, and two processors, one in address-only mode and one in data-only mode. The address calculation workload is varied for each case. Each curve represents a fixed data computation workload. The memory module and processors share the same clock frequency.

quire the allocation of additional processors. This is the preferred solution for block accesses that can be mapped to the address generation hardware. For access patterns not supported by the address generators, similar performance can be obtained by generating the addresses with a processor in address-only mode. This requires the allocation of an additional processor, which may limit the solution's practicality.

Address only mode allows arbitrary address generation capability at the cost of an additional processor. This method eases implementation of latency insensitive burst reads without requiring partitioning of the data computation. This method is limited by the balance of the address and data computation loads. If the address and data processors run at the same speed, whichever task carries the highest computation load will dominate the system performance. This can be seen in Figure 6.5.

Partitioning an application among multiple processors in address-data mode will typically outperform a mapping using the same number of processors in address-only or data-only mode. This occurs because the number of iterations of the application kernel required per processor is reduced. This reduces the application's sensitivity to computation loads. Address-only mode is most useful when address computation and data computation are of similar complexities, when code space limitations prevent the two tasks from sharing the same processor, or when the application

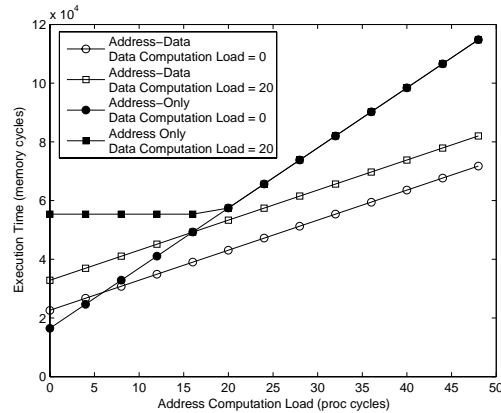


Figure 6.6: Equal area comparison of address modes. The figure shows the execution time of the block workload for two parallel processors in address-data mode, and two processors, one in address-only mode and one in data-only mode. The address calculation workload is varied for each case. Each curve represents a fixed data computation workload. The memory module and processors share the same clock frequency.

lacks adequate data parallelism to distribute the computation otherwise.

Figure 6.6 compares the performance of the block workload when computation is distributed across two processors. A mapping with two address-data mode processors outperforms address-only and data-only partitioning in most cases. If address and data computation loads are mismatched, the greater load will dominate the execution time for the address-only/data-only mapping. When the address and data computation loads are similar, the performance gap for the address-only mode mapping is small. Furthermore, for very small computation loads, the address-only mode mapping outperforms the address-data mode mapping because each loop iteration contains fewer instructions.

## 6.2 Area and Power Trade-offs

As with most digital IC designs, area and power are closely related to performance. Generally, performance can be increased at the expense of area and power by using faster devices or by adding parallel hardware. Although the performance of the FIFO-buffered memory module was the first design priority, the power and area results are acceptable.

| Parameter   | Synthesis Constraint | Result               |
|-------------|----------------------|----------------------|
| Cycle Time  | 1.70 ns              | 1.80 ns              |
| Design Area | 0                    | 1.28 mm <sup>2</sup> |
| Power       | None                 | 1.83 W               |

Table 6.1: Synthesis results. The results of the synthesis flow are summarized here. These results are for high-level synthesis. Some increase is expected during the back-end flow. Note that the power result is likely high, due to overestimation of the clock net capacitance.

### 6.2.1 Synthesis Methodology

The FIFO-buffered memory design was modeled with synthesizable Verilog. The design was synthesized using Synopsys Design Compiler [36]. A high effort design flow, consisting of multiple incremental compilation steps was used to obtain better performance results. To generate a netlist for place-and-route, the design was ungrouped, allowing optimizations to occur across the design hierarchy and improving system performance. The constraints used for synthesis are specified in Table 6.1.

To obtain more meaningful results for power and area, a separate synthesis run was executed, keeping the entire design hierarchy intact. This negatively impacts the design's performance and total area, but allows measurement of the contributions of each submodule to the total area and power measurements.

To increase the accuracy of power estimation, the flow described in the Synopsys Power Compiler User Guide [37] was followed. This flow consists of three steps. First, a forward-annotation file in Switching Activity Interchange Format (SAIF) is generated by Design Compiler. This file is loaded into NC Verilog and used to capture switching activity during simulation of the design. Finally, a back-annotation SAIF file generated in simulation is read during the Design Compiler synthesis flow, and is used to estimate power. Switching activity was obtained from RTL simulation. Greater accuracy could be achieved by capturing switching activity at the gate level.

### 6.2.2 Area Results

The results of the synthesis flow provide a reasonable estimate of the design's area. The design contains 9713 cells, including hard macros for the SRAM core and FIFO memories. With an

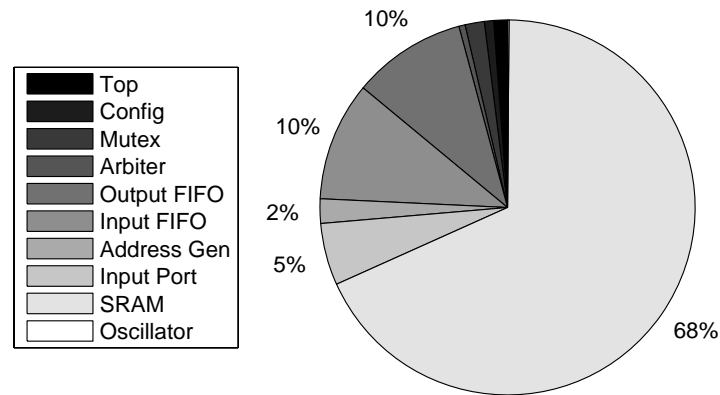


Figure 6.7: Area Distribution among Submodules. The relative cell area of each group of submodules is shown. The SRAM used in this design is 8 K words. The SRAM consumes 68% of the area; the four input FIFOs occupy 10%; the four output FIFOs occupy 10%.

8 K-word SRAM, the cell area of the synthesized design is  $1.28 \text{ mm}^2$ . This is roughly equivalent to three AsAP processors. Assuming 70 percent utilization in the back-end flow, the final module area will be about  $1.8 \text{ mm}^2$ . This estimation is conservative, as the area occupied by the SRAM core is independent of routing utilization.

The area of the FIFO buffered memory module is dominated by SRAM core, which takes up 68.2 percent of the module's cell area. This implies a 32.8 percent area overhead to implement the FIFO-buffered design, rather than a simpler SRAM interface. This overhead is similar to that reported by Mai *et al.* for a Smart Memories system with the same memory capacity [13]. The distribution of area among the major blocks of the design is shown in Figure 6.7. In the design described, the SRAM is only 8 K words. The synthesized design, however, is capable of addressing up to 64 K words. Assuming that the memory size scales linearly, a 64 K-word memory would occupy 94.5 percent of the module's area. This implies an overhead of only 5.5 percent, which is easily justified by the performance increase provided by the ability to easily share the memory.

The potential for further area reduction is limited. If cycle time requirements are relaxed, area could be improved by using smaller gate sizes and more aggressive area optimization during synthesis. From an architectural standpoint, reduction in area comes at the cost of a reduction either

throughput, flexibility, or functionality. Functionality can be sacrificed by eliminating features such as address generators, or mutexes. Flexibility can be traded off for area by reducing the configurability of the design, and decreasing the number of ports provided in the memory module. Throughput can be sacrificed by reducing the number of pipe stages in the design, or by removing the buffers from the input ports, which are required to maintain throughput at the FIFO interface. All of these options will yield only a small decrease in area. Because the SRAM cell will continue to dominate the area of the design, further attempts at area reduction are not practical.

Some additional area overhead results from the addition of the memory port to each AsAP processor. With a five entry prefetch buffer, the memory port occupies  $0.012 \text{ mm}^2$ . This is slightly more than three percent of the total processor cell area. This overhead is reasonably insignificant, but could be reduced by minimizing the allowable latency between the processor and the memory. Some additional area cost is introduced in the processor by the additional stall logic required to support the memory interface. This integrates well into the existing stall logic, and results in only a negligible increase.

### 6.2.3 Power Results

In general, accurate power estimation is difficult without physical design details. A reasonable estimate of the design's power consumption can be taken from high level synthesis results and library information. The main limitation of power estimation at this level is obtaining accurate switching activity for the nodes in the design. The flow described in Section 6.2.1 was run on the design to obtain an accurate high-level power estimate. Switching activity was recorded for the execution of a four processor application that computes  $c_j = a_j + 2b_j$  for 1024 points. This application exercises most of the design's functionality.

Power Compiler reports the average power consumption of the entire module as 1.83 W, with a 1.8 volt supply and 1.8 ns cycle time. The relative power of each submodule is shown in Figure 6.8 and Figure 6.9. Of the total power consumption, 57.1 nW is attributed to cell leakage power. The breakdown for leakage power is shown in Figure 6.10.

The bulk of the reported power (94 percent) is consumed in switching the clock net. Rough calculations indicate that a wire capacitance on the order of 1 nF is needed to produce

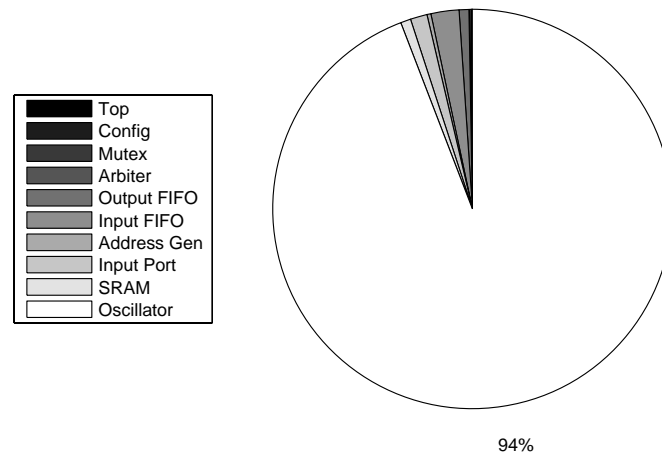


Figure 6.8: Relative power consumption of submodules. The power consumption of the memory module, as reported by Design Compiler, is dominated by clock switching power (94%). The power estimate for the clocking power is unreasonably high. More realistic estimates can be obtained in the physical flow.

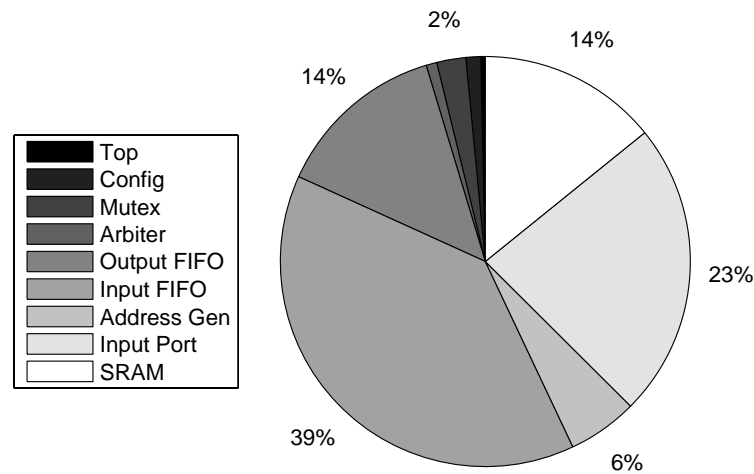


Figure 6.9: Neglecting clocking power, the relative power consumption of the submodules is more apparent. The power is dominated by the input FIFOs (39%) and input ports (23%) as these are the most active blocks in the design. The dynamic power of the SRAM cell is relatively low, but matches well with the datasheet value.

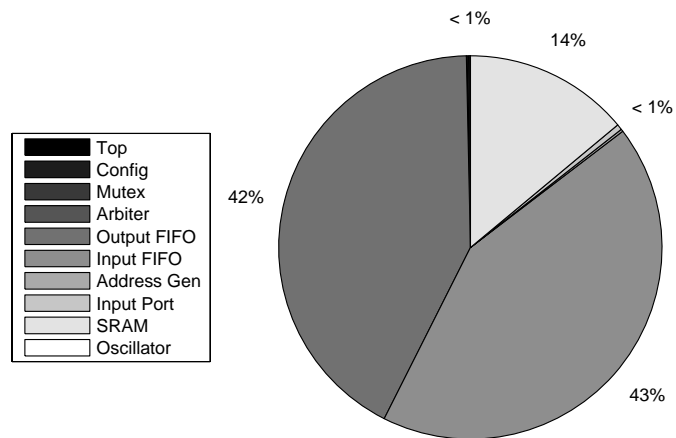


Figure 6.10: Relative leakage power of submodules. As expected, the leakage power is dominated by the memory cells. The eight FIFOs(85%) and the SRAM core(14%) consume nearly all of the systems leakage power.

this result. For the TSMC 0.18  $\mu\text{m}$  process, typical wire capacitance values are no greater than 0.25 fF/ $\mu\text{m}$ . A wire 2.5 m long would be required to produce a capacitance of this magnitude. The erroneous result is likely caused by limitations of Design Compiler when dealing with high fan-out nets. A more realistic estimate might be an order of magnitude smaller. While the absolute power number is questionable, it is not unreasonable that the system clock will dominate dynamic power consumption.

Because the system power is dominated by clock switching, the most effective approach for power reduction is clock gating. The low activity of many blocks during typical design operation implies a significant power savings from coarse clock gating. For example, if the address generators in the module are not being used, the clock to the generators can be gated off, reducing the clock load by about 10 percent. Due to the design complexity required for implementation, this possibility was not explored. Support for clock gating in synthesis is, however, built in to Synopsys Power Compiler, and warrants further consideration.

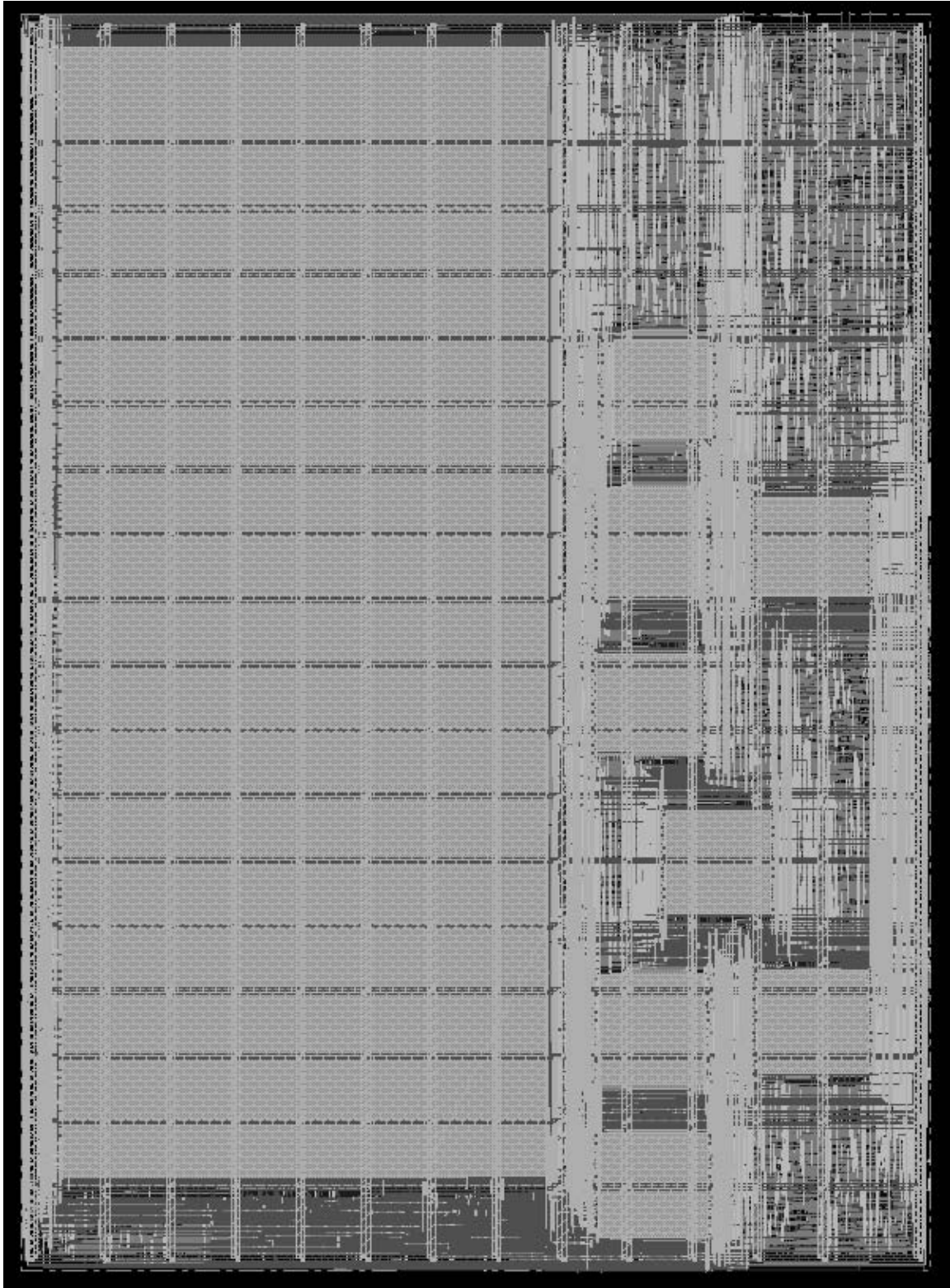


Figure 6.11: Preliminary Place and Route results of the FIFO-buffered Memory Module. This layout was provided by Zhiyi Yu from VCL.



## Chapter 7

# Conclusion

### 7.1 Summary

The design of an asynchronously sharable FIFO-buffered memory module to increase the data storage capacity of the AsAP architecture has been described. The design allows a high capacity SRAM to be shared among the processing elements of the AsAP array, which have limited local memory resources. The purpose of the module is to enable applications with large working sets on the AsAP architecture.

Many features are added to the design to increase performance and flexibility. The memory module shares its memory resources with up to four AsAP processors. This allows the memory to be used for interprocess communication or to increase application performance by parallelizing computation. The addition of addressing modes and hardware address generators increases the systems flexibility when mapping memory intensive applications. These schemes also have the potential to improve system performance when used appropriately.

The FIFO-buffered memory module was described in Verilog, and synthesized with a  $0.18\mu\text{m}$  CMOS standard cell library. A design with an 8K-word SRAM has a cycle time of 1.8 ns, and occupies  $1.2\text{ mm}^2$  based on high level synthesis results. These values will increase slightly during place and route. Synopsys Power Compiler estimates the system power consumption at 1.8W for a typical workload. The accuracy of this estimate is questionable due to apparent overestimation of capacitance for the clock net. A more accurate estimate can be obtained from the final placed design. The memory module can service one memory access each cycle, leading to a peak memory

bandwidth of 8.8 Gb/s.

## 7.2 Future Work

Potential opportunities for expansion of this work lie in four major areas: circuits and micro-architecture, system architecture, CAD, and platform expansion.

Areas for micro-architecture enhancement include power optimization, and increased configurability. Potential power optimization targets include aggressive clock gating, dynamic voltage and frequency scaling, and low power SRAM design. As more advanced process technologies are targeted, leakage power will become an issue. Implementing drowsy SRAM schemes [38] or utilizing other low power techniques may provide significant power improvements. Additional configurability can be implemented at the cost of design complexity. Areas of interest include reconfigurable interconnect, variable latency port implementations, and reconfigurable buffer directions.

The architecture of the memory module could be better optimized through a quantitative study of the system's hardware/software interface. Due to time constraints, only a few benchmark applications were considered and most design decisions were made based on intuition and past experience. A more focused architectural study should be performed to determine the optimal number of ports, degree of sharing, and size of memory for wide variety of target applications. Such a study could also determine memory module density and distribution parameters.

A third area of research is CAD. Currently, limited tool support exists for the AsAP platform. Ideally, tools will be developed to map high level application descriptions on to the AsAP array. These tools should carefully consider available memory resources, and determine an optimal application mapping and memory allocation. Such tools would be required to ensure the usability of the AsAP platform.

Finally, this work can be expanded by introducing additional functionality to the AsAP platform using the proposed memory module as a framework. The design of the memory module can be extended to support a wide variety of peripheral devices, which would be mapped into the memory space of the module. Possibilities include special functional units, analog-digital and digital-analog converters, and PCI or USB controllers. The buffered memory module described can be viewed as the starting point for an endless number of peripheral devices.

# Glossary

**address-data mode** An input port mode which causes both addresses and data for a memory write to be taken from the same input port. This mode is selected by setting the mode field of the port's configuration registers to 11.

**address-only mode** An input port mode which causes only addresses for a memory write to be taken from the input port. Write data is taken from a port in data-only mode. This mode is selected by setting the mode field of the port's configuration registers to 10.

**AsAP** For *Asynchronous Array of simple Processors*. A parallel DSP processor consisting of a 2-dimensional array of very simple CPUs.

**data-only mode.** An input port mode which causes only data for a memory write to be taken from the input port. Write addresses are taken from a port in address-only mode. This mode is selected by setting the mode field of the port's configuration registers to 01.

**dual-clock FIFO.** A *First In First Out* queue with a read clock and a write clock, used to synchronize data across a clock boundary.

**GALS.** For *Globally Asynchronous Locally Synchronous*. A design methodology in which major design blocks are synchronous, but interface to other blocks asynchronously.

**input port.** A block within a FIFO-Buffered memory which encapsulates a dual-clock FIFO and logic required to issue requests to the memory module. An input port interfaces to a single AsAP processor and represents a single thread of execution in the memory module.

**least-recently-serviced.** A scheduling policy used in the buffered memory arbiters. The policy schedules the requesting process which was least recently granted access to the resource. If

all processes are requesting access to the resource, the policy is equivalent to a round-robin policy.

**mutex.** For *Mutual Exclusion*. A hardware or software construct used to guarantee mutual exclusion to processes competing for shared resources. In the context of the FIFO-buffered memory, a “Test-and-Set” lock used to implement mutual exclusion.

**reserve space.** Additional space provided in a FIFO buffer to prevent overflow in case of non-unity write latency.

**request.** An ordered set of tokens that make up a single command to a FIFO-buffered memory module.

**token.** A 16-bit word accompanied by two control bits written to an AsAP processor’s memory port. One or more tokens make up a request.

# Bibliography

- [1] A. W. Burks, H. H. Goldstine, and J. von Neumann. Preliminary discussion of the logical design of an electronic computing instrument. In A. H. Taub, editor, *Collected Works of John von Neumann*, volume 5, pages 34–79. The Macmillan Company, New York, NY, 1963.
- [2] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach*, chapter Memory Hierarchy Design. Morgan Kaufmann, San Francisco, CA, third edition, 2003.
- [3] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level strage system. *IRE Tansactions on Electronic Computers*, pages 223–235, April 1962.
- [4] J. Chang, S. Rusu, J. Shomaker, S. Tam, M. Huang, M. Haque, S. Truong, M. Karim, G. Leong, K. Desai, R. Goe, and S. Kulkarni. A 130-nm triple- $V_t$  9-MB third-level on-die cache for the 1.7-GHz Itanium 2 processor. *IEEE Journal of Solid-State Circuits*, 40(1):195–203, January 2005.
- [5] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, and C. Kozyrakis. A case for intelligent RAM. *IEEE Micro*, 17(2):34–44, March-April 1997.
- [6] R. Banakar, S. Steinke, Bosik Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Symposium on Hardware/Software Codesign*, pages 73–38, May 2002.
- [7] P. R. Panda, N. D. Dutt, and A. Nicolau. On-chip vs. off-chip memory: The data partitioning problem in embedded processor-based systems. *ACM Transactions on Design Automation of Electronic Systems*, 5(3):682–704, July 2000.
- [8] P. R. Panda, N. D. Dutt, and A. Nicolau. Efficient utilization of scratch-pad memory in embedded processor applications. In *IEEE European Design and Test Conference*, pages 7–11, March 1997.
- [9] P. R. Panda, N. D. Dutt, and A. Nicolau. Local memory exploration and optimization in embedded systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 18(1):3–13, January 1999.
- [10] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle, and P. G. Kjeldsberg. Data and memory optimization techniques for embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 6(2):149–206, April 2001.
- [11] B. R. Gaeke, P. Husbands, X. S. Li, L. Loiker, K. A. Yelick, and R. Biswas. Memory intensive benchmarks: IRAM vs. cache-based machines. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 30–36, April 2002.

- [12] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. A. Horowitz. Smart Memories: a modular reconfigurable architecture. In *Proceedings of the International Symposium on Computer Architecture*, pages 161–171, June 2000.
- [13] K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, and M. A. Horowitz. Architecture and circuit techniques for a 1.1-GHz 16-kb reconfigurable memory in 0.18- $\mu\text{m}$  CMOS. *IEEE Journal of Solid-State Circuits*, 40(1):261–275, January 2005.
- [14] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, March-April 2001.
- [15] S. Naffziger, B. Stackhouse, and T. Grutkowski. The implementation of a 2-core multi-threaded Itanium-family processor. In *IEEE International Solid-State Circuits Conference*, pages 182–183, February 2005.
- [16] J. Hart, S. Y. Choe, L. Cheng, C. Chou, A. Dixit, K. Ho, J. Hsu, K. Lee, and J. Wu. Implementation of a 4<sup>th</sup>-generation 1.8GHz dual-core SPARC V9 microprocessor. In *IEEE International Solid-State Circuits Conference*, February 2005.
- [17] B. Flachs, S. Asano, S. H. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S. M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, and N. Yano. A streaming processing unit for a CELL processor. In *IEEE International Solid-State Circuits Conference*, February 2005.
- [18] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, f. Ghodrati, B. Greenwald, H. Hoffman, P. Johnson, J. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpfen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, March-April 2002.
- [19] S. F. Smith. An asynchronous GALS interface with applications. In *IEEE Workshop on Microelectronics and Electron Devices*, pages 41–44, 2004.
- [20] F. Gharsalli, S. Meftali, F. Rousseau, and A. A. Jerraya. Automatic generation of embedded memory wrapper for multiprocessor SoC. In *IEEE Design Automation Conference*, pages 596–601, June 2002.
- [21] Texas Instruments. *TMS320C6000 CPU and Instruction Set Reference Guide (Rev. F)*, September 2000.
- [22] Yuan-Hao Huang, Hsi-Pin Ma, Ming-Luen Liou, and Tzi-Dar Chiueh. A 1.1 G MAC/s sub-word-parallel digital signal processor for wireless communication applications. *IEEE Journal of Solid-State Circuits*, 39(1):169–183, January 2004.
- [23] J. Lee, C. Park, and S. Ha. Memory access pattern analysis and stream cache design for multimedia applications. In *IEEE Asia and South Pacific Design Automation Conference*, pages 22–27, 2003.
- [24] P. Grun, N. Dutt, and A. Nicolau. Access pattern-based memory and connectivity architecture exploration. *ACM Transactions on Embedded Computing Systems*, 2(1):33–73, February 2003.
- [25] O. Sattari. Fast fourier transforms on a distributed digital signal processor, 2004.

- [26] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *Symposium on Hardware/Software Codesign*, pages 145–9, 1998.
- [27] B. M. Baas. A parallel programmable energy-efficient architecture for computationally-intensive DSP systems. In *Signals, Systems and Computers, 2003. Conference Record of the Thirty-Seventh Asilomar Conference on*, November 2003.
- [28] M. Meeuwsen, O. Sattari, and B. Baas. A full-rate software implementation of an IEEE 802.11a compliant digital baseband transmitter. In *IEEE Workshop on Signal Processing Systems*, pages 124–129, October 2004.
- [29] R. W. Apperson. A dual-clock FIFO for the reliable transfer of high-throughput data between unrelated clock domains, 2004.
- [30] Artisan Components, Sunnyvale, CA. *Artisan Standard Library SRAM Generator User Manual*, 2003.
- [31] G. D. Forney Jr. The viterbi algorithm. *Proceedings of the IEEE*, 61(3):268–278, March 1973.
- [32] C. Kozyrakis and D. Patterson. Vector vs. superscalar and VLIW architectures for embedded multimedia benchmarks. In *IEEE/ACM International Symposium on Microarchitecture*, 2002.
- [33] T. Olsson and P. Nilsson. A digitally controlled PLL for SoC applications. *IEEE Journal of Solid-State Circuits*, 39(5):751–761, May 2004.
- [34] J. Kessels and P. Marston. Designing asynchronous standby circuits for a low-power pager. *PIEEE*, 87(2):257–267, February 1999.
- [35] Artisan Components, Sunnyvale, CA. *TSMC 0.18 $\mu$ m Process 1.8 Volt SAGE-X<sup>TM</sup> Standard Cell Library Databook*, 2003.
- [36] Synopsys. *Design Compiler User Guide*, June 2004.
- [37] Synopsys. *Power Compiler User Guide*, June 2004.
- [38] K. Flautner, Nam Sung Kim, S. Martin, D. Blaauw, and T. Mudge. Drowsy caches: simple techniques for reducing leakage power. In *Proceedings of the International Symposium on Computer Architecture*, pages 148–157, May 2002.

