A Programmable Processor for the Cheops Image Processing System

by

Edward Kelly Acosta

B.A., Boston College (1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1995

Author ....
Department of Electrical Engineering and Computer Science
May 26, 1995

Certified by ....
V. Michael Bove, Jr.
Associate Professor
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

A Programmable Processor for the Cheops Image Processing System

by

Edward Kelly Acosta

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1995, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Science and Engineering

Abstract

We explore the use of dynamically reconfigurable hardware in the CHEOPS image processing system. CHEOPS incurs a substantial performance degradation when executing computational operations for which no dedicated stream processor exists. A new programmable processor that combines the speed of special purpose stream architectures with the flexibility of general purpose processing is proposed and implemented as a solution to this problem. Two SRAM based FPGAs are utilized in conjunction with a PC603 microprocessor to provide a flexible computational substrate. The system allows algorithms to be arbitrarily mapped to some combination of dedicated hardware, and software within the data flow paradigm. Dynamic reconfiguration allows the processor to specialize to each computation while maintaining temporal flexibility. The system is designed and implemented; observations are reported.

This research was supported by the Television of Tomorrow research consortium at the MIT Media Laboratory.

Thesis Supervisor: V. Michael Bove, Jr.
Title: Associate Professor

Acknowledgments

This project was not conducted in a vacuum and there were many people who helped along the way. First and foremost, special thanks go to Prof. V. Micheal Bove Jr. for his patience and guidance and for making this opportunity available. Special thanks also go to John Watlington for countless hours of assistance in the lab and a great deal of technical advice.

Hanoz Gandhi, for help with a multitude of parts and especially for help keeping Oreo and the Cadence software up and running. Thanks also to Andrew Huang for the original Huffman decoder Altera design and to Jeffrey Wong for assistance with parts ordering.

I would like to also thank Henry Holtzman and Klee Dienes for answering many stupid UNIX questions and for occasional intervention when I got in over my head. Many thanks must also go to Dan Gruhl for much help with latex and final document preparation.

I must also thank Steve Wilson and Nam Pham at IBM for providing samples of the PowerPC603 microprocessor, and much technical information, before they were publically available.

Finally, I would like to thank my parents and family for all of their support both moral and financial during this time. And to Lauren Fenton for putting up with me during a very difficult time; I love you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A classic debate in the computer literature between special purpose and general purpose architectures has persisted for decades. On one side, is the desirability of flexible architectures that are capable of implementing a wide range of applications. On the other is the desire, and in some applications the necessity, for very high levels of performance offered only by custom Integrated Circuits (ICs). All other things being equal the ideal solution would be to build a machine that incorporates both. That is, a machine with multiple custom ICs targeted at each application of interest. Unfortunately, all other things are not equal and computer designs are subject to the constraint of cost. Any IC design effort is extraordinary expensive and must be amortized over many possible customers to be cost effective. Custom ICs, desirable as they are, typically do not have a large enough application base to make them economically feasible for a majority of applications. As a result most computers employ general purpose architectures[1] capable of performing a wide range of applications at moderate performance levels.

Custom ICs, have been used only by a limited number of consumers with deep pockets and very specialized applications that demand the highest levels of perfor-

[1]General purpose machine architectures and general purpose processors will be referred to as GPPs in this work.

mance. The effects of these design realities has stratified the design landscape into two camps as indicated in figure 1-1. On the one side are the general purpose designs. They come in several different flavors, mainly, SIMD, MIMD, SISD, etc., and are characterized as follows:

- Implement a general computational architecture onto which a wide variety of applications can be performed.
- Suffer from Von Neumann bottlenecks of one form or another. These are memory bandwidth bottlenecks, ALU computation bottlenecks, control overhead bottlenecks.
- Have the advantage of low cost due to an extremely large pool of consumers. The low cost to the user is possible because the architecture can perform a wide variety of tasks. Thus development costs can be amortized over a very large number of consumers.

At the other extreme are machines that are special purpose and are characterized by the following attributes:

- Have extremely high levels of performance.
- Have only a very limited range of applications.
- Are not constrained computationally by architecture, but still may be constrained by memory bandwidth.
- Have very high costs due to the fact that there are only a small number of consumers for each particular IC.

The void in the middle of this spectrum has begun to be filled by more flexible ICs and by the use of programmable logic devices. Producers of custom ICs have begun to appreciate the importance of versatility in design. As a result most custom ICs being developed recently have been designed to implement multiple functions to the extent that the minimum required performance of the target application is not hindered. An example of this type of IC is video codecs that may implement several different coding schemes [35].

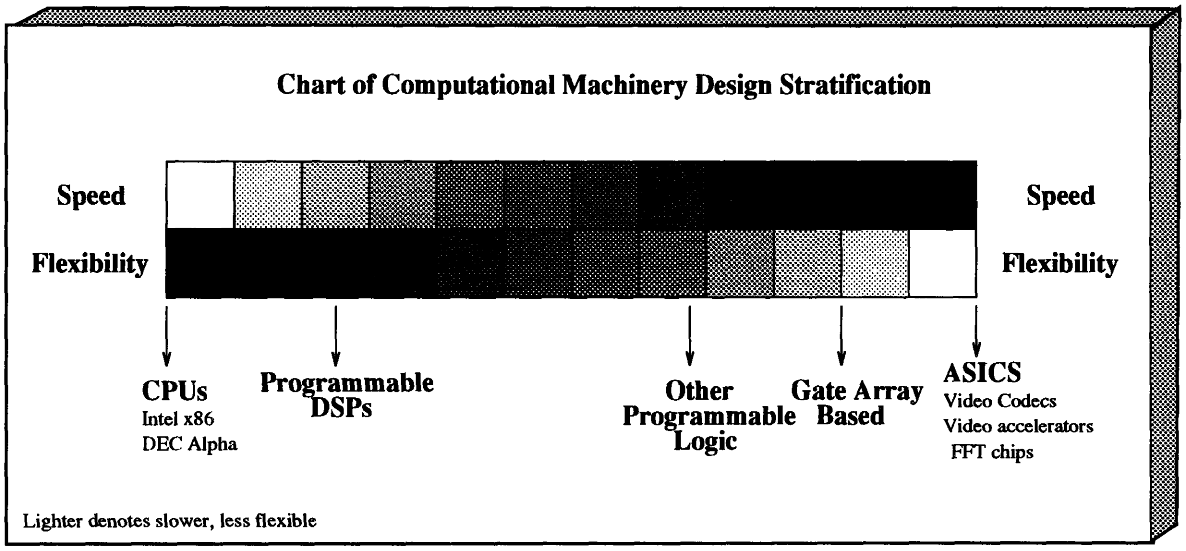


Figure 1-1: Computer Design Stratification

A much more important recent development that has contributed to the filling of this void is the appearance of a new device, the SRAM based Field Programmable Gate Array (FPGA). These programmable logic devices are capable of implementing arbitrary user specified logic functions and can be programmed from a software description of the required logic. Moreover, it is possible to change their configuration dynamically such that the same chip may implement many different functions. The use of such devices has made possible architectures that begin to approximate the speed of custom ICs while maintaining the application flexibility of GPPs. Recently, they have been used both to prototype custom ICs and as flexible coprocessors for GPPs. Several new architectures based on this concept will be discussed in the next chapter. All suggest that the use of dynamically reconfigurable hardware in the design of general purpose computing platforms may allow them to tackle applications that were once the exclusive domain of custom ICs. To this end, the work described here is a further exploration into the use of dynamically reconfigurable hardware, in the form of SRAM based FPGAs, to provide flexible architectures that offer very high performance.

1.1 Image Processing Applications

One class of applications that has traditionally had extremely high computational demands is image processing applications. These typically require the processing of huge amounts of data, and for video applications, in very short time frames. As an example consider the computational requirements of the Discrete Cosine Transform (DCT) of a CCIR 601 resolution image. [2] The DCT is often used in transform based coding methods to achieve high levels of data compression for transmission in low bandwidth channels. Mathematically, the DCT is expressed as in equation 1.1:

$$F(u,v) = \frac{2}{N}\alpha(u)\alpha(v)\left[\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} x(m,n).\cos\frac{(2m+1)u\pi}{2N}\cos\frac{(2n+1)v\pi}{2N}\right] \tag{1.1}$$

where,

$$\alpha(k) = \begin{cases} 1/\sqrt{2}, & \text{if } k = 0 \\ 1, & \text{if } k \neq 0. \end{cases} \tag{1.2}$$

The DCT operation for this image requires approximately 8,110,080 multiply and divides, and 7,096,320 additions and subtractions[32]. [3] Many transform based codecs must process 30 frames (images) per second in this manner to compress full-motion video in real time.

As a point of reference, a 90-MHz Pentium microprocessor is capable of 11 Million Floating Point Operations Per Second (MFLOPS). The 200MHz DEC Alpha

[2]480 x 760 pixel resolution.

[3]These numbers can vary considerably depending on method of implementation of the DCT. The point here is to present a qualitative argument, and for this purpose these numbers are representative.

microprocessor is capable of 43 MFLOPS[16].[4] These processors are representative of the state of the art at this time. Clearly, transform based codecs for real-time full-motion video require more computational power than can be mustered by general purpose CPUs. These types of computational demands exceed the resources of even the most high performance GPPs. Hence, image processing is one application area where the use of custom ICs is demanded by the computational requirements and it is not uncommon to find custom machine architectures to carry out these applications.

1.2 Data Flow Computing and Cheops

Often computers designed for image processing and digital video applications employ the data-flow model as a means of providing special purpose processors to achieve high performance while still providing some application flexibility. Dataflow computers are data driven, as opposed to instruction driven as are GPPs [36]. Their defining feature is that they employ an instruction execution model that enables any particular instruction to execute as soon as all of its operands are available [18]. The evolution of computations in dataflow machines are typified by acyclic directed graphs in which the nodes represent computational elements or instruction execution, and the connecting edges are the datapaths through the graph. Figure 3-4 is an example of a simple dataflow computation described by a directed graph. This type of computational representation is common in digital signal processing applications. The advantage of dataflow architectures is that they are capable of exploiting large-scale parallelism by identifying all tasks that can be performed at the same time. An additional advantage is that memory latency bottlenecks are overcome. This is the result of their data driven nature. However, a significant drawback of such architectures is the extremely complex control and sequencing necessary to identify and exploit this

[4]These numbers were derived based on the Linpack Benchmark. A benchmark widely used to gauge processor performance for complex mathematical operations.

parallelism. Specifically, dataflow machines require a very large synchronization name space.

The Cheops Image processing system is a hybrid dataflow architecture that was designed and constructed to facilitate the immense computational demands of digital video applications while avoiding some of the inherent complexities associated with true dataflow machines. Cheops is really a hybrid dataflow/MIMD machine. Like dataflow machines, it computations are specified by directed graphs. However, in Cheops the computational nodes are dedicated special purpose stream processors that perform a narrow class of computations with very high throughput. Instructions in Cheops do not require complex analysis and scheduling as do more traditional dataflow machine instructions. Instead, they simply identify the source and destination memory regions. Algorithms are implemented by having the data flow through these processors from memory bank to memory bank. All memory banks and all stream processors are connected through a full cross-bar switch whose configuration can by dynamically changed. Like MIMD machines, more than one instruction thread may be operational at one time. Unlike most MIMD machines though, each processor is special purpose and can only implement a single function. By utilizing this stream architecture, Cheops is able to offer very high performance. Additionally, dynamic configuration of the cross-bar allows Cheops to provide algorithmic flexibility. It is thus capable of implementing a wide range of digital video applications with the speed of special purpose hardware and flexibility approximating that of general purpose processors. The architecture of Cheops will be discussed in much more detail in chapter III.

1.3 Contribution of Thesis

Cheops is capable of offering very high performance provided that there is a Special Purpose Stream Processor (SPSP) available for any given computation. However, practical constraints place limitations on the number of SPSPs that can be present in the system at any one time. A Cheops system may be configured with many different types of SPSPs and these can be changed by replacing small modular Printed Circuit Boards (PCBs). But, these changes cannot occur at run time, the system must be powered down to make the changes.

As a result of these limitations, it is possible for an algorithm to specify an operation for which no SPSP exists in the system. When this occurs Cheops can exhibit severe performance degradation as the computation must then take place on a traditional GPP. In this thesis I propose a new SPSP for Cheops called the State Machine. Specifically, I propose a dynamically reconfigurable processor that can be configured to perform an arbitrary computation. The State Machine will utilize a general purpose processor closely coupled to two large SRAM based FPGAs to provide a flexible computational substrate. It will be used to overcome the practical constraints to the architectural flexibility of Cheops by extending this flexibility to the stream processor element level. In the course of this work I hope to accomplish two tasks. First to show that dynamically reconfigurable hardware can be used within Cheops to successfully implement an arbitrary computation such that Cheops is not forced to execute these on its general purpose platform. In the course of so doing, the performance of Cheops will be improved by enabling it to implement complex real-time video algorithms. These performance improvements will be noted and compared against the performance of an unimproved Cheops.

Additionally, I investigate the utility of such dynamically reconfigurable hardware for providing architectural flexibility at run-time. While many groups have already demonstrated the use of FPGAs to provide architectural flexibility in co-processor design, few take into account reconfiguration time in their performance improvement calculations. Thus these efforts provide little if any information for run-time applications, and in particular, in a multi-tasking environment. In this work I address this issue directly and present results obtained with Cheops.

1.4 Organization of Thesis

This thesis is organized into several chapters as follows. Chapter II discusses the history of reconfigurable computing and provides the background for the work attempted here. Chapter III is an in-depth explanation of the Cheops Image Processing System and its software environment. Chapter IV introduces the programmable processor and indicates how it is used to extend the architecture of Cheops to provide increased performance for complex applications. Chapter V reviews the hardware implementation and its computational abilities. Chapter VI discusses programming issues and the software environment associated with the State Machine. In chapter VII, the results of this research are presented. Finally Chapter VIII presents insights gained during this project along with directions for further research. Several appendices follow that provide additional information about the programmable processor.

Chapter 2

Background: Dynamically Reconfigurable Hardware

Dynamic reconfiguration of hardware to customize a machine is not a new idea although the current generation of reconfigurable logic devices has generated renewed interest in the concept. The desire to implement custom hardware to improve the performance of special instructions and applications within GPPs has been prevalent throughout the history of processor design. The trend towards instruction specific specialization has only very recently ceased with the wide spread acceptance, both academic and commercial, of the superior performance offered by RISC architectures. But many computers today still employ special purpose hardware, in the form of application specific integrated circuits (ASICs), for specific applications such as floating point mathematics, graphics rendering, and MPEG video coding/decoding. Additionally, network and storage interface controllers are increasingly using ASICs to keep pace with the storage bandwidth requirements of modern processors. These ASICs provide very high levels of performance but generally only for a very limited number of applications. Much of the current research in the use of dynamic reconfiguration of hardware is targeted at these ASICs. It attempts to approximate the performance of ASICs but with greater application flexibility to make better use of the silicon area and thus achieve better cost/performance design points. In the following chapter we provide a brief history of dynamic hardware configuration and discuss the current generation of programmable logic devices that has generated renewed interest in this area. Several systems based on these devices will be discussed to convey a feeling for the current state of affairs. Finally we conclude with a discussion of some of the technological limitations of these devices and the systems based on them.

Many early computers had microcoded control units that allowed their instruction sets to be altered, or tailored to specific applications, to improve performance [15]. The early 70's saw extensive research on dynamically altering the instruction set of a machine at compile time to improve performance by reducing bus traffic to main memory [45] [1]. Most notably, the IBM 360 series of computers popularized this method of CPU control in the middle 70's. These machines allowed not only improved performance through custom modifications to the instruction set, but also the ability to emulate other instruction sets and hardware platforms [26]. Among the most well known (or notorious) of application specific microcoded instructions was the polynomial evaluation instruction of early VAX computers. This instruction was designed to improve performance by reducing instruction fetches to main memory[37].

These machine designs had the advantage of a somewhat flexible instruction set. Custom instructions could be created for special applications or the instruction set could be tailored to the application as desired. They provided the ability to customize the architecture at the instruction level. That is, they provided instruction specific flexibility. Still, this method of specialization was severely limited by the fact that microcode only allowed changes of the sequencing of control signals and data-flow in a fixed datapath. The datapath itself was static and unalterable.

Even after improved memory and cache technology eliminated most of the traditional advantages of microcoding, the desire to implement application specific instructions continued. There are countless examples of the implementation of application specific instructions that contributed to the proliferation of CISC machines in the late 80's and early 90's [46] [23]. Thus it is clear that even within the realm of general purpose processing there is still a desire to employ custom hardware whenever it is economically feasible, or at least custom instructions for specific applications. Research has shown however, that for comparable semiconductor process technology and clock rate such CISC architectures are two to three times slower than RISC architectures that take advantage of simpler control design, larger register sets, and improved memory hierarchies [8]. Further, RISC instruction sets eliminate the overhead of supporting complex instructions that are infrequently utilized by compilers at best [31].

Even within current designs of GPPs there is more hardware specialization than one might imagine. This is not immediately obvious if only the single design dimension of application specificity is considered. Consider instead a two dimensional design space in which one dimension represents application specificity, as before, while the second represents data type specificity. Clearly, as figure 2-1 indicates, in this space, most processors we think of as GP are very special purpose. However they are not specialized to applications, like ASICs, but instead to data types. That is, they are specialized to and provide very good performance for, applications that deal primarily with 16-bit or 32-bit data objects[1]. Consequently, just as special purpose machines exhibit poor performance on tasks other than their target task, most GPPs exhibit poor performance for data types other than their target data types. This fact has been noted by several researchers in dealing with bit-serial applications[40][7].

The recent dominance of RISC architecture, while relegating traditional forms of hardware reconfiguration to the depths of history, is of no consequence to the renewed interest in custom computing due to the emergence of a new type of device. These devices are SRAM based FPGAs. [2] These are programmable logic devices

[1]Some of the very latest processor designs are specialized to 64-bit data types, like the DEC Alpha and the Intel Pentium and i860.

[2]The concept of custom computing refers to the use of specialized hardware for executing applications. Systems that now employ dynamically reconfigurable hardware of one form or another to provide a flexible custom computing environment are referred to in the literature as custom

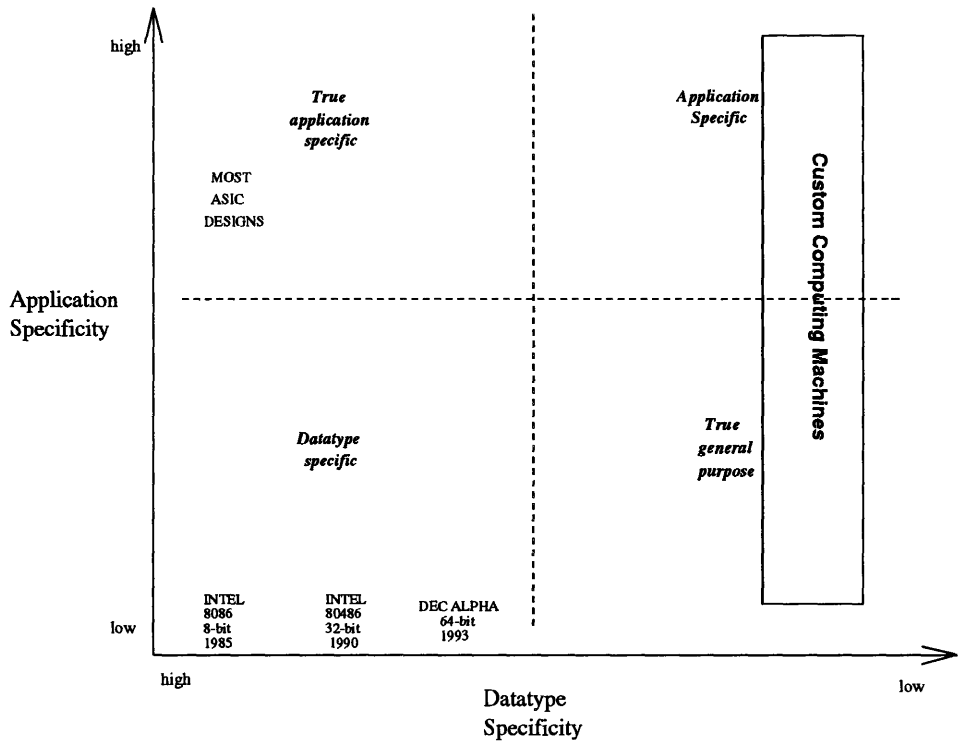


Figure 2-1: Two Dimensional Design Specificity Space

that can be completely reconfigured under external control.[3] This means that they are capable of assuming completely different architectures at different times. Their presence has motivated new research into dynamic hardware configuration. Unlike earlier attempts though, these are aimed not at the instruction level, but instead at the application level. A datapath can be designed for each particular application and the devices can be reconfigured immediately before run-time. Thus SRAM based FPGAs offer an additional advantage over earlier forms of dynamic machine configuration. It is not only possible to change the order of occurrence of control signals for a fixed datapath, as with microcoded machines, it is possible to completely change the datapath itself. Moreover, they offer true dynamic flexibility along both dimensions of the two dimensional design space mentioned above in that the custom datapath can be specific to an arbitrary data type. The custom computing machines box in

computing machines (CCMs). This convention will be used in this work as well.

[3]They may also reconfigure themselves under external signal.

figure 2-1 is meant to be representative of this fact. While CCMs can provide very high degrees of both application and data type specificity, they are also very general in the sense that they can be quickly specialized to a wide range of possible applications.

2.1 Field Programmable Gate Arrays

Field Programmable Gate Arrays are programmable logic devices that can be reprogrammed any number of times. In this respect they are similar to PAL devices, but offer much higher logic density and are capable of implementing more complex logic. The most advanced FPGAs currently offer up to 10,000 logic gates.[4] The name is descendent from the earlier mask programmable sea of gates devices from which these are derived. Their objective is to obtain the density of sea of gates technology and the flexibility of PAL/PLD devices. Most FPGAs are constructed as an array of very fine grained logic resource blocks surrounded by routing resources which are used to wire logic together. Figure 2-2 is a block diagram of the typical internal organization of an FPGA.

The exact architecture and size of the blocks varies from vendor to vendor but generally consists of a 2-8 input,1-2 output combinational logic function, 1-2 flip-flops, and some control and multiplexing circuitry. The blocks labeled IOB in figure 2-2 are I/O buffer cells that typically consist of a few tri-state buffers, a line driver for driving signals off chip, and possibly a register for latching either inputs or outputs. The logic block of an Altera FLEX8000 logic block, known as a Logic Element (LE), is shown in figure 2-3. In these devices 8 LEs are grouped together in Logic Array Blocks (LABs) to provide both fine and coarse grained logic elements. These devices have been used in the research work discussed here.

Most FPGAs to date have been based on technologies like CMOS EPROM or EEP-

[4]However, a more appropriate measure of logic resources for these devices is the number of logic blocks offered. This will become clear shortly.

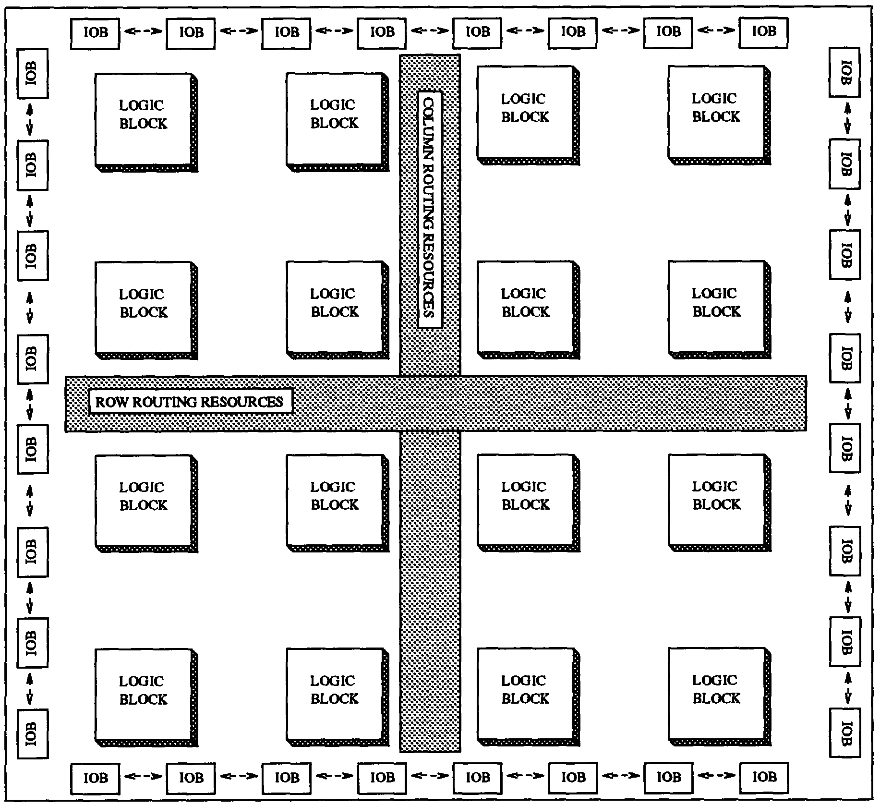


Figure 2-2: Typical internal organization of an FPGA

ROM, which is electrically programmable read-only memory or electrically erasable programmable read-only memory respectively. Both require special programming apparatus that utilize voltage levels not normally seen in the application circuit. Once programmed these devices are not re-programmable without being removed from the target circuit. As a result of the physical manner in which they are reprogrammed, they are typically only good for 100-10,000 programmings and start to fail after that[17]. Their use is thus limited to application development. Once the design has been debugged these devices become static in function. In this respect they are similar to PAL devices.

The general trend of the current generation of FPGAs is towards SRAM based configuration methods. These FPGAs store their configuration information in standard static RAM cells interspersed throughout the chip. In contrast to their predecessors,

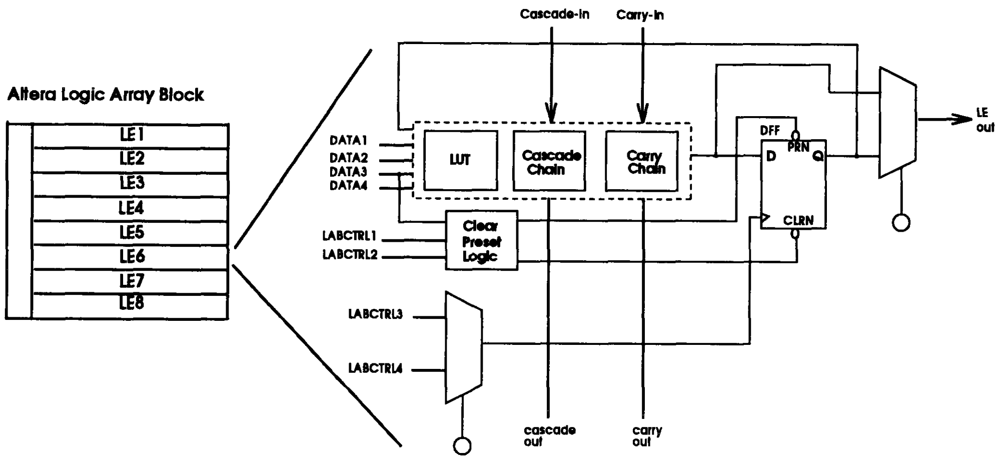


Figure 2-3: Altera Logic Array Block (LAB) and Logic Element (LE)

they may be reconfigured any number of times without any wear or tear on the device. They use simple RAM based look-up tables (LUTs) to implement logic functions instead of gates made from transistors. The inputs to a logic function are treated as an address, the contents of the address is the logic function result. Since no special voltage levels are required for configuration, these devices can be configured without removing them from the circuit. In fact, it is possible for the target circuit itself to control the configuration. It is this last possibility that is most interesting and offers great potential for dynamic reconfiguration of application specific hardware.

When an SRAM based FPGA exists in a circuit designed to control it and utilize its resources, a whole new dimension of design flexibility is introduced. It becomes possible to build systems that begin to approximate the performance of application specific ICs while offering the flexibility of general purpose processors. The performance comes from the fact that hardware can be custom tailored to each application. For many applications it becomes possible to achieve the highly desirable attribute of a result per clock cycle. [5] The flexibility comes from the dynamic reconfigurability mentioned above. Theoretically, it is possible to implement a custom architecture for

[5]Result is used instead of instruction since instructions begin to lose their significance. Thus the metric of Cycles Per Instruction (CPI) is less meaningful with these machines.

each individual application. Machines that embody this philosophy are commonly referred to in the literature as custom computing machines. While a diversity of systems architectures exists, they all employ this basic concept.

2.2 Current Systems Exploiting Dynamically Reconfigurable Hardware

As mentioned earlier, the potential offered by SRAM based FPGAs [6] has ignited renewed interest in dynamic hardware configuration. A large number of researchers have very recently, and concurrently, begun to reexamine the feasibility from a cost/performance perspective of providing application specific hardware in light of the recent developments in FPGA technology. However, the emphasis is not on instruction specific specialization, but instead on application specific specialization. Most research in this area is currently exploring two general trends. [7] The first is the use of FPGAs in complete systems that allow the user to arbitrarily configure the hardware for specific applications to obtain super-computer type performance. Most of these target a specific application, like logic emulation, and do not have dynamic reconfiguration as a primary goal. Rather, they exploit the low cost and short design cycles of these devices as a welcome alternative to full custom ICs in obtaining improved performance. Many groups have reported spectacular success at speeding up various applications with FPGAs without the expense or time commitment of ASICs[43][25][11]. A more interesting current area of research is the use of FPGAs in add-on boards and other closely coupled co-processors that assist GPPs in performing certain parts of applications. Typical designs attempt to move two distinct types of operations to hardware to speed applications, these are:

[6]From this point forward SRAM based FPGAs will be referred to as simply FPGAs.

[7]There are notable exceptions to this generalization. An example is the Spyder processor, a superscalar processor with reconfigurable execution units implemented with FPGAs, constructed by Iseli and Sanchez et al.[28]

- Bit parallel computations. Computations that exhibit a high degree of parallelism and employ non-standard data types. GPPs generally provide poor performance in each of these cases and thus benefit greatly from co-processor assistance.

- Computationally intensive inner program loops. Takes advantage of the fact that typically less than 10% of program code accounts for 90% of program execution time [26]. Attempts to implement these sections of code in hardware to improve performance.

In both cases designs of this nature attempt to use FPGAs to provide a flexible computational substrate that is dynamically adaptive to the application of interest. We chose to discuss a few of these systems further, as their presence has been a motivating factor in the design of the programmable processor for Cheops.

Perhaps the first custom computing machine was the Anyboard, a PC plug-in board that provided five Xilinx FPGAs together with local RAM store to provide a rapid-prototyping environment for digital system development [14]. Anyboard, constructed in 1992, utilized Xilinx XC3042 FPGAs and took over five seconds to reconfigure[13]. For this proprietary system, dynamic reconfiguration at run time was too computationally costly. Anyboard's chief contribution was the creation of a pioneering reconfigurable hardware platform for custom computing. Although its purpose was not to augment the functionality of a GPP, its presence served as the inspiration for other custom computing machines with this goal in mind.

Another notable custom computing machine is SPLASH, constructed at the IDA Supercomputing Research Center. This architecture consists of up to sixteen SPLASH boards that connect to a Sun Sparcstation through a separate interface board. The interface board provides four DMA channels capable of providing a total bandwidth of 50 MBytes/sec. to the SPLASH boards. Each SPLASH board consists of 16 Xilinx XC4010 FPGAs fully connected through a 16x16 crossbar switch. Additionally, 36-bit paths run between adjacent FPGAs which are laid out in a linear array[5].

Figure 2-4 is a block diagram of the layout of a SPLASH board. The linear bus that connects adjacent FPGAs may also be extended across multiple boards should they be employed. SPLASH has several key advantages over its predecessors. First the FPGAs used are high density, 10,000 gates per chip, and thus provide enough hardware resources to implement large scale architectural designs. In addition, the programming environment is vastly improved over earlier machines through the use of HDLs (Hardware Description Languages) which allow designs to be specified at the behavioral level. Finally, the newer FPGAs have configuration times that are several orders of magnitude less than Anyboard. Several applications have been successfully demonstrated on this machine and SPLASH2 is under development to overcome I/O and internal connect bandwidth problems encountered[39].

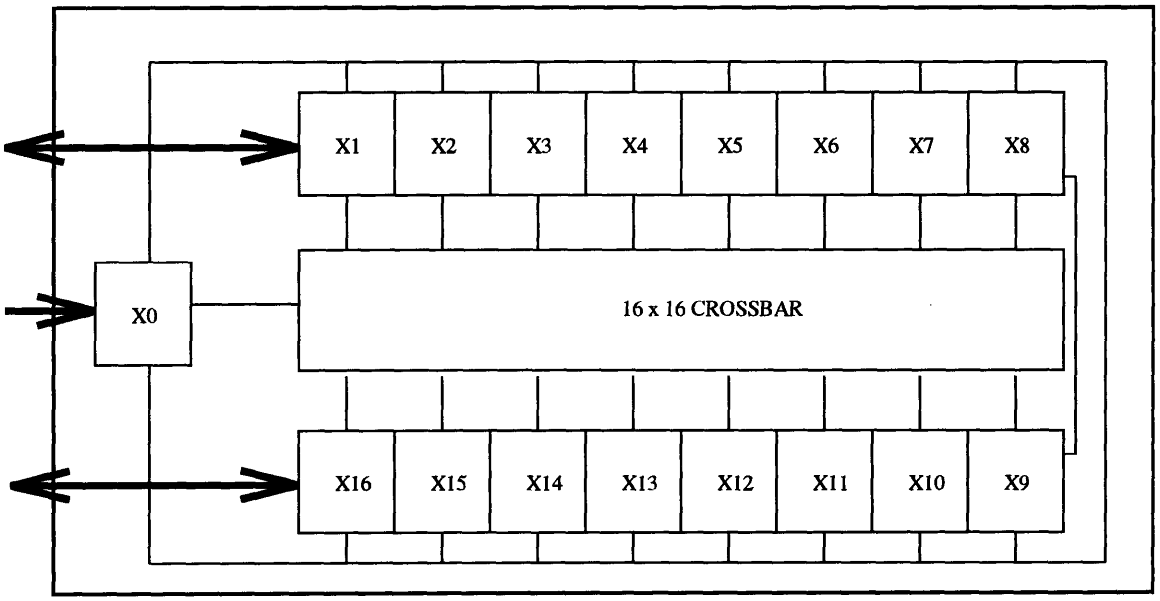


Figure 2-4: SPLASH Custom Computing Machine Architecture

The virtual computer designed by the Virtual Computing Corporation and described by Casselman et al. [9], is not in the class of machines under discussion. It is a complete super-computing platform designed to assist a host workstation. It consists of a large array of Xilinx 4010 FPGAs interspersed with I-Cube IQ160 reconfigurable routing chips. Despite the difference in design, we mention it here because it can be completely reconfigured in 25 milliseconds. A dedicated fast 64-bit I/O port facilities the reconfiguration operation. Although this author feels that this claim is somewhat dubious, it provides an example of what is currently possible in reconfiguration times for the current generation of FPGA devices.[8]

The PRISM-II machine, built at Brown University by Athanas, Agarwal, and Silverman et al. [45], employs three Xilinx 4010 FPGAs closely coupled to a AM29050 processor to provide flexible co-processor support. Figure 2-5 is a block diagram of the system architecture. This project is perhaps the most sophisticated to date in that they have accomplished a considerable amount of work in generation of hardware modules directly from high level C source code [44]. The goal of PRISM is to improve performance of computationally intensive tasks by using information extracted at compile time to automatically synthesize application specific hardware to supplement the computational abilities of the GPP. The FPGAs fulfill the role of a custom co-processor, but an adaptive one that can be specialized to each individual application. The PRISM architecture is very similar to that of the programmable processor as we shall soon see.

As a final note DeHon, et al. [12] has proposed the integration of FPGAs with conventional GPPs in a new device called a DPGA, Dynamically Programmable Gate Array. These augment a core general purpose processor with an array of reconfigurable logic resources on the same chip. Each logic element in the array has local store to store several configurations for fast context switches between applications. While the availability of these devices is many years away, their feasibility is unquestionable. Consequently, they represent a natural extrapolation of dynamically reconfigurable hardware technology and indicate that the paradigm for flexible custom computers has been established.

[8]No other researchers have reported reconfiguration times similar to these for this family of FPGAs.

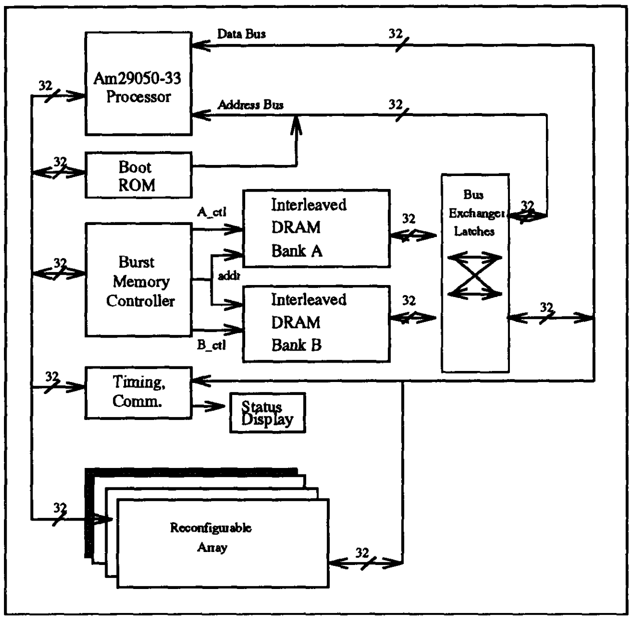


Figure 2-5: PRISM-II Custom Computing Machine Architecture

2.3 Obstacles to Acceptance of Custom Computing Machines

While significant progress has been made in the design and construction of custom computing machines using FPGAs, there are still several technological barriers that have prevented them from establishing a more visible role in the world of computer architecture. Critics often use these to dismiss the relevance of these machines. Specifically, there are three major problems with these machines: a high temporal reconfiguration cost, low silicon density, and a complex programming and design environment. We address each of these points in detail.

2.3.1 Reconfiguration Overhead

The most serious problem is the very high reconfiguration overhead. Typically, the reconfiguration time can exceed the corresponding computation time on a general purpose processor. Thus even though the custom computer may be orders of magnitude faster than a GPP computationally, the total time to perform the computation, reconfiguration plus actual computation, may be longer than the GPPs computation time. Thus it would seem that the use of custom computers is not practical. This is certainly true for most of the devices currently available.

What the critics fail to note though, is that the reconfiguration times of FPGAs are dropping by orders of magnitude each year. As noted above Anyboard took seconds to reconfigure using Xilinx XC3042, an older family of FPGA introduced around 1988. The generation of FPGAs used in this thesis, Xilinx 4000, Altera FLEX8000, etc., have reconfiguration times of 25-100ms. The most recent generation of devices available have reconfiguration times in the 1-10ms range. These reconfiguration times continue to drop at a near exponential rate as FPGA technology improves.[9] Moreover, the currently available devices were not designed with ultrafast reconfiguration in mind. Increased demand for faster reconfiguration for custom computing is already driving down these times further. The next family of FPGAs will likely have reconfiguration times on the order of 100us. Thus reconfiguration overhead is decreasing to the point were it is rapidly approaching the time penalty of a context switch in multiprocessing operating systems. As a result, the comparison to GPPs is beginning to make sense for large data sets. We will have more to say on this subject later.

Another important development that could eliminate the reconfiguration overhead is the appearance of FPGAs that are incrementally reconfigurable. Devices that are incrementally reconfigurable are capable of carrying out computation in one area of the chip while reconfiguration occurs in another. With these FPGAs it may become possible to reduce the reconfiguration overhead to zero if an architecture can be appropriately time division multiplexed. This also introduces another interesting possibility, implementing virtual hardware in an analogous fashion to virtual memory. Babb et al. [6], who has developed a tool for time partitioning a circuit, has discussed this possibility and Ling and Amano et al [33] describe a machine based on it. Algo-

[9]Indeed, product announcements in 1995 surprise even this author.

tronix, a Scottish company, currently makes a family of incrementally reconfigurable FPGAs, the CAL family of chips[19]. If this technique becomes more popular it is likely that other vendors will follow.

The trend of near exponential improvement in reconfiguration time and the availability of incrementally reconfigurable logic devices indicate that the objection to custom computing based on reconfiguration overhead is only a short term objection. It is not a fundamental obstacle to the development of these machines. For large datasets and specific computations, it is already worth paying the configuration overhead. As technology scales this will become true in an increasing number of cases. As such, its worth will be discounted in the remainder of this work. We will choose instead to attempt to develop some empirical insights on when and how to employ these and future devices in an advantageous fashion.

2.3.2 Device Density

Another common objection to the use of FPGAs in custom computing is the device density. Current FPGAs are a factor of 10 less dense than corresponding sea of gates technology[40]. Thus it is hard at this point to fit a design of any reasonable size in a single FPGA. The problem is exacerbated by low device utilization due to routing constraints.[10] Using multiple devices is an option but introduces a whole new set of problems which must be addressed. While not insurmountable, they complicate the design cycle unnecessarily. This is a serious drawback and a fundamental weakness in the cost/performance equation for this class of machines.

Despite this weakness, the advancement of FPGA technology will continue to alleviate these constraints. FPGAs will also continually benefit from improvements in semiconductor process technology in the same manner as other ICs. FPGA gate

[10]FPGAs typically have only a limited number of wire segments in the routing channels to connect the logic blocks with. It is not uncommon for a given design to exhaust the possible connecting paths between two logic blocks. When this happens the logic block becomes unusable to implement logic functions. We will have more to say on this subject in a later chapter.

count is increasing at a rate of better than 100% per year [48]. Newer generation of devices eliminate routing constraints with better architectures and more routing resources. The largest of the current generation of devices has on the order of 50k gates. Xilinx projects FPGA usable gate counts of over 200,000 by the end of the century and a price of under five dollars for the current generation of devices[47]. So although FPGAs may never offer the logic density of custom ASICS, densities will improve to the point where they are acceptable for most applications. Already the largest FPGAs are large enough to accommodate small designs in their entirety. Thus although logic density is a current constraint, it is also only a short term constraint.

2.3.3 Programming Environment

The programming environment of custom computing machines is also a serious concern. Many complain that the use of VHDL and other hardware and circuit languages to program these devices forces the programmer to have detailed knowledge of digital hardware design as well as software design. As long as this is the case, it is likely that they will not gain wide spread acceptance because of the difficulty in programming them. This concern is also a legitimate one, programmers should not be required to specify the architectural details on which their software will run. This is too radical a departure from current practice and introduces more complexity than can be reasonably justified.

A major effort has been under way over the past two years to eradicate this problem. Several groups have made significant progress in synthesizing hardware modules directly from high level programming languages. DEC Paris Research Lab have demonstrated that with a few simple extensions the C++ programming language can be used to describe hardware function at the structural level[7]. While still requiring the user to have knowledge of the underlying architecture, its description can be specified in a familiar environment. Further, many programmers have detailed knowledge of the underlying hardware at the structural level anyway. Software tuning to the run-time architecture is still a widely practiced concept[45].

A more important development is the synthesis of hardware and software directly from a high level programming language. Several groups have begun to automate the various steps of the compilation process. Most of these efforts let a high level language compiler break the program down to the RTL level where control flow graphs (CFGs) can be easily generated to assist with hardware synthesis. Athanas and Silverman et al [45] have made impressive progress in synthesizing hardware directly from an algorithmic description in the C programming language. Their first effort was capable of compiling a subset of the language that included int, char, short, and long types, if-else control structs, and fixed length for loops directly into hardware. They used the GCC compiler as a front end to generate the RTL description of the problem, and then used CFGs, VHDL and the Xilinx tools to directly generate FPGA hardware images. They are currently working on the next generation compiler which will incorporate variable loop count for loops, while and do-while loops, switch-case statements, and break and continue directives. Their efforts have shown that it is already possible to generate synthesized hardware directly from traditional programming languages.

There are many others that are currently conducting research in this area and making significant progress. While the current generation of compilers is rather primitive this is also a temporary condition. It is unquestionable that in the very near future there can be very sophisticated combined hardware software compilers that will synthesize complete systems described in high level languages. DeHon et al [12] has also commented on this possibility. In the short term, architectures and subcomponents can be custom designed by hardware engineers and supplied as libraries. Programmers can then make use of these hardware libraries in the same manner that they use software libraries. In either case the programming environment can be made amenable to current practices so that programming custom computing machines is no more difficult than programming machines based on GPPs.

Dynamic configuration for custom computing has evolved considerably over the past several years. The appearance of SRAM based FPGAs has motivated new research into run-time application specialization. Several experimental systems have demonstrated that their use holds great promise for custom computing. While there are currently some major constraints in using FPGAs to implement custom computing machines, it has been shown that these are at best temporary. There are no fundamental technological boundaries that prohibit the advancement of these machines. Their most major handicap is that the devices that are employed for dynamic reconfiguration were not designed with custom computing in mind. As the growing recognition of the usefulness of these machines-both in terms of flexibility and favorable cost/performance ratios-continues to rise, there will be sufficient market forces present to insure that new devices that overcome the current limitations of FPGAs will be available.

Chapter 3

The Cheops Imaging System

The Cheops Image Processing System is a modular experimental test bed built by the Information and Entertainment Group at the Media Laboratory. It employs a flexible parallel architecture for the execution of real-time image processing applications including image encoding/decoding, motion compensation, model-based image representation, and image compression and decompression[29]. Additionally, the system is capable of sub-band coding and decoding and of generating displays of many different resolutions and frame rates simultaneously in accord with the open-architecture television paradigm proposed by Bove and Lippman, et. al. [30]. Cheops achieves its real-time speed by exploiting the fact that a large class of image processing algorithms and operations employ a small set of basic computational operations that can be implemented in special purpose hardware as dedicated stream processors. These algorithms can then be described in terms of a data-flow representation where data flows between storage elements through a cascaded set of memory buffers and computational nodes [29].

The Cheops system is internally subdivided into three types of modules connected through two high speed buses for image data transfer, and a slower global bus for control and inter-module communications. There are input modules designed to accept coded video data at a wide range of input data rates. Processing modules perform coding/decoding and other image processing operations. Finally, output modules contain several frames of video buffering and are capable of both digital and analog output for many different frame sizes and frame rates. Cheops can be configured with up to four of each type of module all capable of operating simultaneously.[1] The two high speed buses are known as Nile buses, they are 48 bits wide and carry pixel data between modules in a twenty-four bit format (8-bit for each of the RGB channels) at a data rate of 40 M-Pixel/sec. each. A third bus (32 bits wide) carries control and general data communications between the modules at an 8 Mtransfers/sec. data rate. Figure 3-1 is a block diagram of the modular structure of Cheops. The system is controlled by a front-end host computer that supplies a file system and network interface to Cheops. The host is used as a development platform for the cross-compilation of Cheops software and as an execution environment.

3.1 Input and Output Modules

Input and output modules provide the necessary hardware for Cheops to capture and display video data. The input modules pass data to the processor modules for processing, which in turn pass the processed data along to the output modules for display. Video data is passed between the modules using the Nile buses and control and sequencing information is passed via the global bus as mentioned earlier. Since the input and output modules do not have any direct relevance to this research, we do not describe them in this work.

[1]Recently, a Cheops system has been configured to accommodate up to six output modules for the Spatial Imaging Group of the MIT Media Lab for the display of real-time three dimensional holograms.

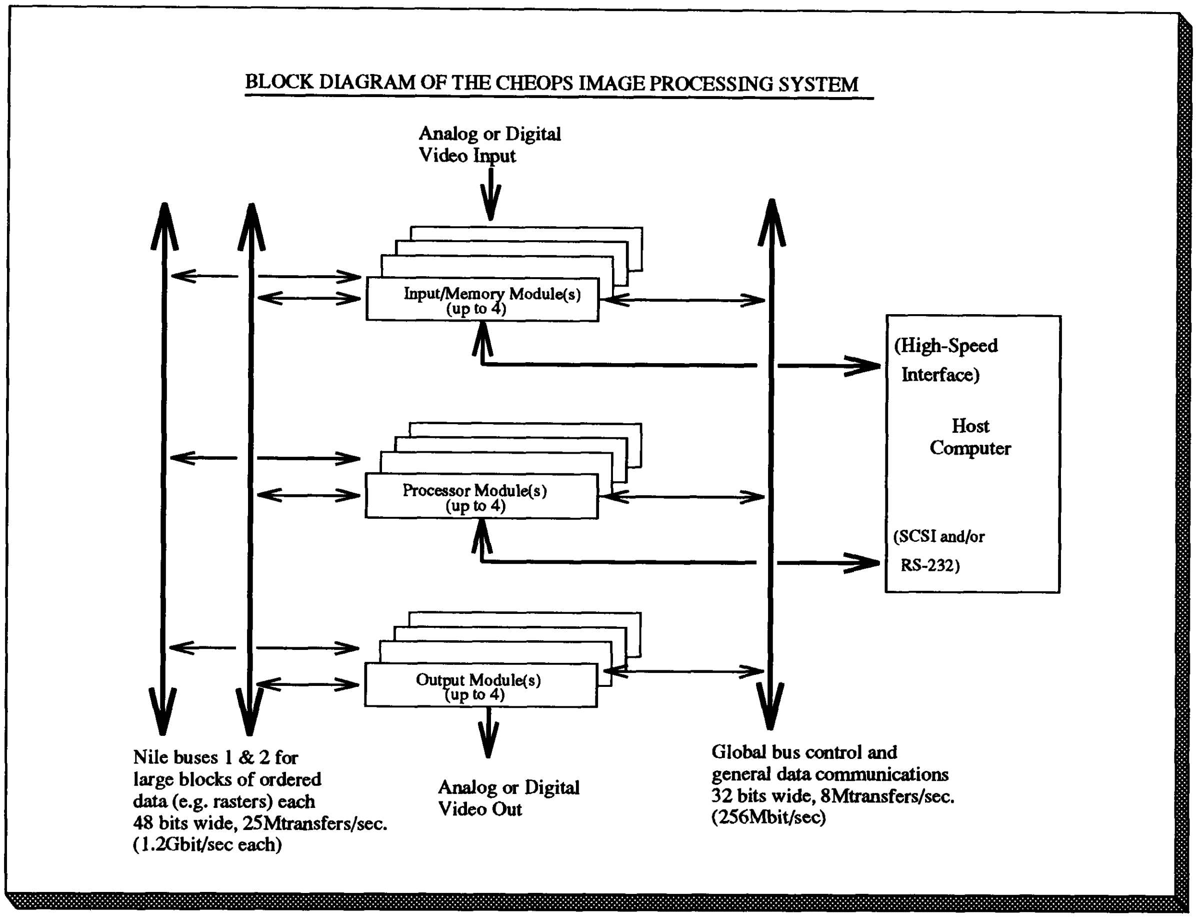

BLOCK DIAGRAM OF THE CHEOPS IMAGE PROCESSING SYSTEM
Analog or Digital Video Input
Input/Memory Module(s) (up to 4)
Processor Module(s) (up to 4)
Output Module(s) (up to 4)
(High-Speed Interface)
Host Computer
(SCSI and/or RS-232)
Nile buses 1 & 2 for large blocks of ordered data (e.g. rasters) each 48 bits wide, 25Mtransfers/sec. (1.2Gbit/sec each)
Analog or Digital Video Out
Global bus control and general data communications 32 bits wide, 8Mtransfers/sec. (256Mbit/sec)

Figure 3-1: Cheops Internal Organization

3.2 The Processor Module Architecture

The majority of the interesting processing in the Cheops system occurs on the processing modules. Each processor module consists of 32 M-byte of dual-ported RAM, a general purpose processor with a small memory, as many as eight special purpose stream processors, a local bus and several external bus interfaces [38]. It is within the processor module that the data-flow model is exploited to achieve real-time speed by performing common computations using dedicated stream processor hardware. It is the nature of this implementation that, under certain conditions, creates a computational bottleneck that degrades system performance.

A block diagram of the processor module is shown in figure 3-2. The memory is subdivided into eight distinct banks of 4M-bytes each. The banks are dual-ported with one port connected to the global bus, and the other through a non-blocking full crosspoint switch, known as "the Hub" [38], to as many as eight special purpose stream processors. The crosspoint switch allows for a 40 M-Sample/sec[2] path to be configured between any memory bank and any stream processor. Connections between memory banks and between cascaded processors are also possible[30]. The general purpose processor is an Intel 80960CF, a 32-bit superscalar RISC processor; capable of executing as many as three instructions per cycle. It is responsible for stream processor coordination, resource management, user interface, and performing computations for which no stream processor exists. Cheops is connected to the host computer through the SCSI and RS-232 interfaces.[3]

3.3 The Stream Processors

Six of the eight stream processors reside on submodules that plug into the processor module system board. The physical and logical interface of the submodules is provided in figure 3-3. Each submodule can contain two stream processors and has two 16-bit inputs and two 16-bit outputs connected to the Hub for the transmission of image data. The submodules are connected to the processor board with a 36-pin dual row header strip (P20) and a 96-pin Eurocard connector (P10). These allow the submodules to be removed and replaced easily for maximum configuration flexibility. Some stream processors require enough hardware to occupy an entire submodule board; others are capable of temporarily utilizing all input and output ports to increase bandwidth, disabling the other stream processor in the process. There are many different types of stream processors for performing common computations in image processing. The following is an exhaustive list of the available stream processor

[2]A sample is a 16-bit wide data type.

[3]A high speed parallel interface (HPPI) that will facilitate a 100M-Byte/sec. transfer rate, is also under construction.

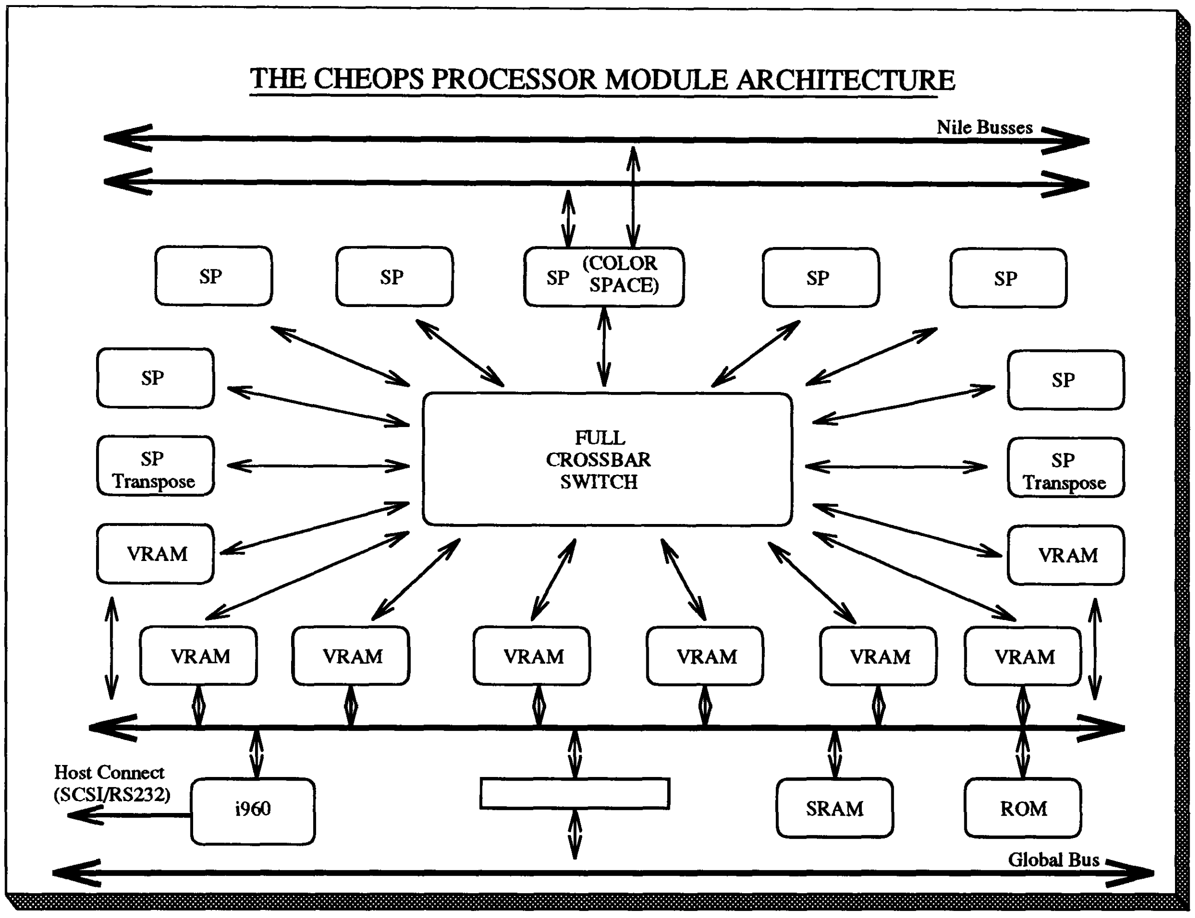


Figure 3-2: Processor Module Organization

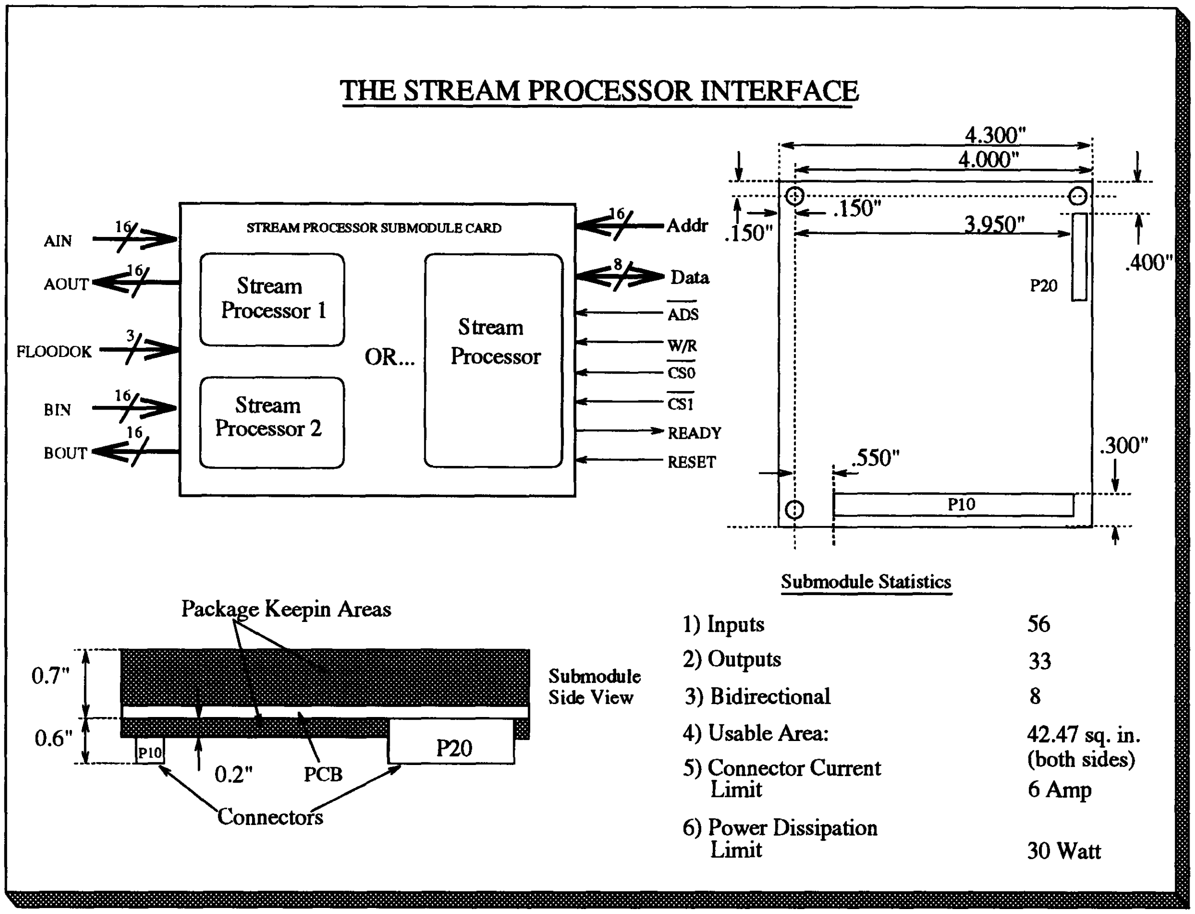
submodules:

- 8x8 discrete cosine transform (DCT/IDCT) unit
- FIR filter engine
- 16x16 full search motion estimation unit
- Sequenced lookup table for vector quantization
- Splotch superposition processor
- Stream multiplier
- Spatial remapping processor

The remaining two stream processors are transpose engines and are permanently attached to the processor module because of their frequent use. Finally, the color space converter is also connected to the memory banks and provides an interface to the Nile system buses.

The flow of data from memory banks to stream processors and vice-versa is controlled by handshaking signals known as OK signals. A processor module has three OK signals that run to each of the submodule cards. A single OK signal governs the transfer of data in a single computational pipeline. The system resource manager may dynamically assign specific OK signals to stream processors to carry out data transfers. Once assigned the stream processor synchronizes its activities to the assertions of the OK channels, which are controlled by the resource manager. Data transfers between memory banks and stream processors occur in one or two phases. For one phase operations data flows from one memory bank through a stream processor and into another memory bank in the same operation. In two phase operations, data flows from a memory bank into a stream processor where it is processed and buffered until the second phase in which the data flows back to a memory bank. Two phase operations use one OK channel in each phase of operation. As a result of this organization, the number of concurrent stream processor computations is limited by the number of OK signals and types of transfers in progress.

The stream processors receive their configuration information over the control interface. A separate 16-bit address and 8-bit data bus and a few handshaking signals are provided for this purpose. A quick look at figure 3-3 shows that the stream processor modules appear as essentially memory mapped objects to the local processor (80960). The only exception is the ready signal which is used by slow devices to extend the normal data transfer cycle. The configuration information stored on the modules includes OK channel assignment, delay value from the assertion of the OK signal to the first valid flood datum, as well as other stream processor specific configuration information.



Submodule Statistics

1) Inputs	56
2) Outputs	33
3) Bidirectional	8
4) Usable Area:	42.47 sq. in. (both sides)
5) Connector Current Limit	6 Amp
6) Power Dissipation Limit	30 Watt

Figure 3-3: Stream Processor Submodule Statistics

3.4 The Stream Bottleneck

The software that runs on the embedded general purpose processor specifies algorithms as data-flow graphs where operations are performed by the stream processors. Data dependencies are also specified so that proper sequencing is maintained. Figure 3-4 is a typical graph describing a simple two step operation. Data flows from buffer to buffer through the stream processors where it is operated on with very high throughput.

The operating system used, Magic7, is cooperatively multitasking allowing several different processes, to execute simultaneously. A separate task, known as Norman, manages the processor module resources and arbitrates between tasks competing

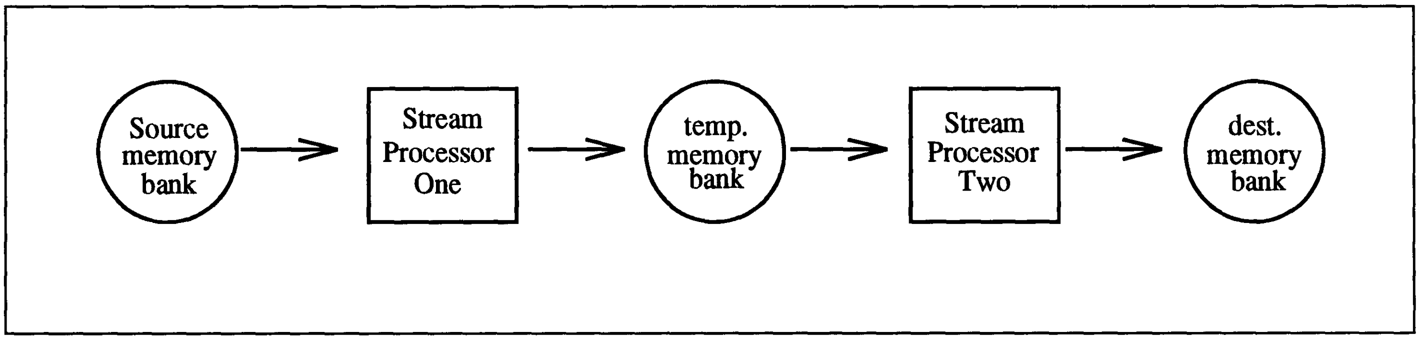


Figure 3-4: Cheops data-flow software description

for the same resources. In this case the resources are the stream processors and the various memory banks. It is the computational performance of these dedicated stream processors that allow Cheops to achieve its real-time performance. Algorithms whose operations can be executed exclusively on the stream processors demonstrate superior throughput. However, algorithms for which no stream processor exists suffer a substantial performance degradation. This is because these must be performed by the general purpose processor, which slows system performance in two ways. First, the general purpose processor, fast as it is, simply does not offer the performance of the specialized stream processors. Additionally, the computation receives only some fraction of the processor's power since it is multitasking and must devote as much time to the resource scheduling task, Norman, as is necessary[30]. Complex algorithms are thus unable to realize the full potential of the Cheops architecture.

In order that this computational bottleneck be removed, all of the computations that would be performed on the main CPU must be moved elsewhere so that it can concentrate exclusively on resource management. A stream processor that combines special purpose speed with general purpose flexibility is required; an apparent contradiction. In the following chapters we discuss the design and implementation of the State Machine stream processor, which is proposed to overcome this fundamental limitation.

Chapter 4

The State Machine Stream Processor

In chapter 2 a detailed explanation of dynamic hardware configuration and SRAM based FPGAs was provided. In addition the term custom computing machine was introduced to describe system architectures that exploit the dynamic reconfigurability provided by SRAM based FPGAs. In this thesis, we endeavor to improve Cheops through the addition of custom computing machinery. The State Machine stream processor is in essence, a custom computing machine. In particular, it is a custom computing machine that has been tailored to the operational requirements of the Cheops imaging system. Custom computing provides an elegant and valid solution to the stream bottleneck problem within Cheops. In this chapter we describe the organization of the State Machine stream processor at a high level, and discuss its use within Cheops. In addition, a model for predicting the reconfiguration time[1] for the State Machine is provided. The State Machine is described in much greater technical detail in the following two chapters.

The State Machine is a new type of computational resource for the Cheops processor module. It appears to the processor module as an ordinary submodule and uses

[1]Here we refer to this time as the reconfiguration penalty.

the standard submodule interfaces. However, the State Machine is really a complete high-performance computational platform. The State Machine satisfies the dual requirements of general purpose flexibility and special purpose speed and is specifically designed to overcome the stream bottleneck problem. The State Machine uses the current generation of dynamically reconfigurable FPGAs under the control of a general purpose processor to satisfy these requirements. The flexibility is provided by the run-time reconfigurability of the FPGAs, the main computational elements. Each State Machine application can have its own specific hardware architecture. The performance is the result of the ability to implement a custom computational architecture for any arbitrary task;[2] in many cases achieving a result per clock cycle.

The general purpose CPU controls the configuration process and all communication with the LP.[3] Additionally, the general purpose CPU may augment the computational abilities of the FPGAs for certain configurations. This ability allows the designer of a State Machine application to arbitrarily choose the boundary between hardware and software implementation of a design. The State Machine is able to appear as a dedicated stream processor for an arbitrary application without violating the data-flow model within Cheops. For any given application, throughput and latency characteristics are dependent on the extent to which custom hardware is utilized.

4.1 System and Data-path Design

The objective of the design of the State Machine was to implement a stream processor that would work within the physical, electrical and logical constraints of the current stream processor interface while providing maximum algorithmic flexibility. That is, how best to organize the resources such that the architecture remained general enough

[2]provided, of course, that the architecture does not require more hardware resources than are available in the FPGAs.

[3]In the remainder of this document the LP refers to the Cheops processor module i960 microprocessor.

to accommodate as many algorithms and computations as possible.

Physically, all of the hardware components are constrained to fit on a single submodule card. Figure 3-3 provides diagrams of the stream processor submodule specifications. Components can occupy both sides of the card provided they do not exceed the height restrictions on either side of the card. While most large components are placed on the top side of the board, small surface mount packages and even industry standard PLCC sockets will fit on the bottom side of the board. The only exception on the bottom side is the small package keep-out zone, where the card physically extends over a large heat sink on the processor module.

Electrically the design is constrained by the input/output interface, current constraint, and power dissipation limit. The two connectors are capable of supplying 6 Amps of 5V DC current to the stream processor card. Thus the aggregate peak current requirements cannot exceed 6 Amps. Power dissipation is thus limited to 30 Watts; six large fans in the system chassis insure that the system is adequately cooled.

The physical and electrical constraints restrict the number of computational resources present on the State Machine board. Clearly, to provide maximum application flexibility, it is desirable to include as much memory and as many large dense FPGAs as possible. Unfortunately, large, dense FPGAs take up a considerable amount of board space. Most come in 160-299 pin PGAs or QFPs [2] [48]. In addition, the larger FPGAs can dissipate a considerable amount of power, a maximum power dissipation rate of approximately 3 Watts is typical of these devices. The approach taken here is to provide one large FPGA per Flood port and fill any remaining board space with additional memory.

Logically, to be maximally flexible the State Machine must accommodate three different types of usage. First it must be capable of acting as two completely independent stream processors, each stream processor using one of the two hub ports. It must be able to be used as one large stream processor using one or both flood ports. Finally, it must be utilizable as two stream processors working together. A further logical constraint is that it must be completely configurable from the control interface.

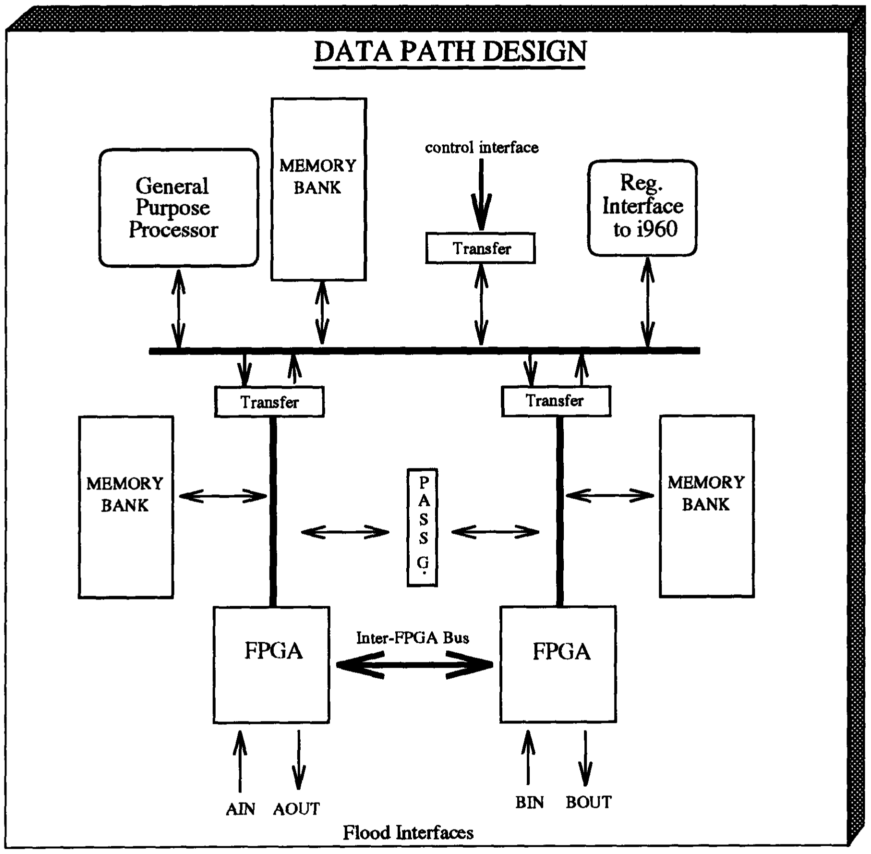


Figure 4-1: State Machine Stream Processor Organization

Figure 4-1 is a block diagram describing the system architecture of the State Machine stream processor. The three busses on the board are capable of supporting concurrent independent transactions. For configurations where the State Machine is to be used as two independent stream processors, each FPGA and its associated memory bank can be operated independently of the other with no interference. For these applications the processor can assist one or both, or none, of the FPGAs with their respective tasks. The inter-FPGA bus allows the FPGAs to pass control signals and data for single processor applications. Also, a set of pass gates connect the two address busses so that they may appear as a single bus when enabled. This feature is useful for applications that require address remapping such as motion compensation and image rendition; allowing one FPGA to generate addresses while the other shuffles the data [10]. For cases where the State Machine is configured as two stream processors working together, the processor can facilitate the transfer of data from one stream processor to the other without interfering with the operation of either one.

The processor bus is clocked at the **Pclk** rate, the processor module system clock speed. It is capable of 896 M-bit/s peak and 224 M-bit/s sustained bandwidth when **Pclk** is 28 MHz.[4] Each FPGA bus is accessed at the Hub clock rate of 32MHz and is capable of 1.024 G-bit/s sustained bandwidth when the FPGA has its memory bus in buslock mode. [5] Data streams flow in from and out to each of the Flood ports at 512 M-bit/s.

4.2 Applications

A State Machine application is divided into two components. There is an architectural description of the custom hardware to be implemented in the FPGAs, and a software image to be executed on the processor. Every application has these two components irrespective of device utilization. Applications that can be implemented entirely in hardware still need a software image to handle communications and transfer phase control information with the LP. Conversely, applications that carry out the entire application with the general purpose processor still need hardware descriptions of the circuitry required to transfer the incoming data from the Flood interface to the memory banks. State Machine application requirements are discussed further in chapter 6.

[4]These peak numbers assume there is no significant interference from the control interface.

[5]Buslock mode enables the FPGA to assume exclusive control of the bus, preventing the processor from accessing the FPGA or the FPGA memory bank.

4.3 Dynamic Reconfiguration

The State Machine is configured by the LP through the control interface. This interface consists of a 16-bit address bus and a 8-bit data bus. State Machine configuration consists of two parts. In the first part, the LP loads both the software image and the device configuration files into the State Machine processor memory bank. In the second part, the State Machine processor initializes the application code and reconfigures the FPGA devices with the new configuration data.[6] The two parts may occur sequentially, or may be decoupled in time.

In the simplest case, one application is loaded and run. Both parts of the configuration process are performed successively. However, sufficient memory exists to allow the LP to load more than one application into the State Machine memory in step one. Thus, timing wise, this simple example represents the worst case. The more typical case, is that multiple applications are loaded into the State Machine memory in step one. Step two is performed for the loaded applications individually, at run time. This method has the advantage of decoupling the loading process from the configuration process so that only the latter occurs at run-time. The best case, minimal reconfiguration time, occurs when two applications can make use of the same FPGA configuration file. In this case, dynamic reconfiguration consists only of initializing the application code.

This configuration methodology introduces several distinct possibilities for device reconfiguration. The following cases are possible:

- Both steps one and two must be performed for the application.
- Both the software image and the FPGA configuration data are already present in the processor memory, only step two needs to be performed.
- Both the software image and the configuration data are already present in the processor memory, only code initialization needs to be performed.

[6]For reconfiguration timing analysis part two is further divided into two parts.

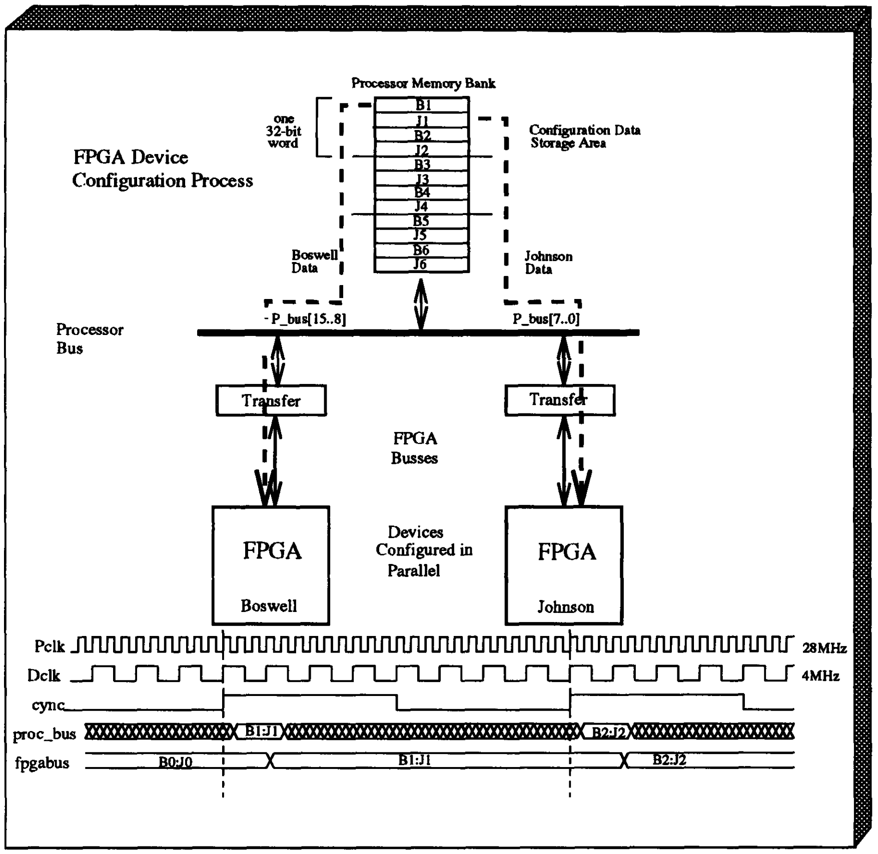


Figure 4-2: FPGA Configuration Process

The FPGA configuration data is stored at the high end of the processor memory bank. In part two of the configuration process, the processor reads this data from memory and forwards it to the FPGA busses. The Boswell FPGA takes its configuration bits from bits $\langle 15:8 \rangle$ of its data bus. The Johnson FPGA takes its configuration bits from bits $\langle 7:0 \rangle$ of its data bus. Since both 32-bit FPGA data busses connect directly to the 32-bit processor data bus through the transfer registers, the devices can be configured in parallel by reading half-words from memory and writing these to the FPGA busses. Figure 4-2 illustrates this process. During configuration the FPGAs require a new configuration data byte every 8 **Dclk** cycles.[7] Using the **cyncclk** as a

[7]**Dclk** is the FPGA configuration clock used to synchronize configuration data input.

reference signal the processor can read a new half-word from memory at every rising edge of **cyncclk** and write the data to the FPGA busses.[8]

4.4 Configuration Timing Analysis

The time required for dynamic reconfiguration of the State Machine is dependent on the type of reconfiguration performed. It may be expressed as the sum of three components as follows:

$$\Upsilon = \tau_d + \tau_i + \tau_c \tag{4.1}$$

where,

$\tau_d =$	time to load State Machine memory with program and configuration data (Part I).
$\tau_i =$	time required to initialize the configured stream processor (Part II).
$\tau_c =$	time required to configure the FPGA devices (Part II).

The first component, τ_d, corresponds to the time required for part one of the configuration process. The second two components, τ_i and τ_c, correspond to part two of the configuration process. As discussed above, one or more of these components can be zero depending on the state of the FPGAs and the number and type of applications already resident in the State Machine processor memory bank. We discuss each of these components in detail.

The first component τ_d is the time required to write the application data into the State Machine's memory bank. The application data includes both program code and configuration data for the FPGA devices. The size in bytes of the application program code varies with the application, and is represented by the variable ρ. The size of the FPGA configuration data is constant for a given device and is represented

[8]**cyncclk** is used by the State Machine processor during the FPGA configuration process to determine when to forward another word of configuration data to the FPGAs.

by η. Thus the transfer time depends on ρ and η, the control bus clock rate, and the number of cycles required to transfer a byte, and can be expressed as:

$$\tau_d(\rho, \eta, \lambda, f) = (\rho + \eta) \cdot \lambda \cdot \frac{1}{f} \tag{4.2}$$

where,

$\lambda =$	Number of **Pclk** cycles required to transfer a byte of data,
$f =$	Frequency of **Pclk**.

The initialization time, τ_i, consists of two components: the time required for the State Machine processor to load the application's parameters into the appropriate place, and to initialize the application code's **.BSS** section. To simplify the calculation, the first component is approximated as constant, to be measured later. The second component is dependent on the size of the application's **.BSS** section. If the time to initialize a word[9] of **.BSS** memory is approximated at six bus clock cycles, then the initialization time can be expressed as:

$$\tau_i(\beta, f) = (\frac{3\beta}{2} \cdot \frac{1}{f}) + T_{prms} \tag{4.3}$$

where,

$\beta =$	Number of bytes in the program **.BSS** section
$T_{prms} =$	Time for OS to load the program parameters.

Finally, the configuration time, τ_c, consists of the time required to load the configuration data into the FPGAs plus the constant FPGA initialization time. This time can be expressed as:

$$\tau_c(d, f_{dclk}) = \frac{d}{f_{dclk} \cdot C_{BW}} + T_I \tag{4.4}$$

[9]The word size is 4 bytes.

where,

d	$=$	Amount of configuration data in bytes,
f_{dclk}	$=$	Configuration clock rate (**Dclk**),
C_{BW}	$=$	Configuration bandwidth of FPGAs,
T_I	$=$	FPGA initialization time.

The total reconfiguration time can thus be expressed as:

$$\Upsilon(\rho, \eta, \lambda, f, \beta, d, f_{dclk}) = \tau_d(\rho, \eta, \lambda, f) + \tau_i(\beta, f) + \tau_c(d, f_{dclk}) \tag{4.5}$$

where, $\tau_d(\rho, \eta, \lambda, f)$, $\tau_i(\beta, f)$, and $\tau_c(d, f_{dclk})$ are expressed as in equations 4.2, 4.3, and 4.4. In the remainder of this work we explore the case where the application data is already resident in the State Machine processor's memory at run time. Thus for dynamic reconfiguration only step two need be performed. Mathematically, we have

$$\tau_d(\rho, \eta, \lambda, f) = 0; \quad for\ all \quad \rho, \eta, \lambda, f. \tag{4.6}$$

and equation 4.5 reduces to

$$\Upsilon(\beta, \alpha, d, f_{dlck}) = \tau_i(\beta, f) + \tau_c(d, f_{dclk}). \tag{4.7}$$

Combining terms gives a model of the dynamic reconfiguration time for the State Machine stream processor:

$$\Upsilon(\beta, f, d, f_{dclk}) = \frac{3\beta}{2f} + \frac{d}{f_{dclk} \cdot C_{BW}} + K. \tag{4.8}$$

where,

$$K = T_{prms} + T_I \tag{4.9}$$

For the case where dynamic reconfiguration consists of performing both parts of step two, a further simplification can be employed to arrive at a "back of the envelope" type calculation for the reconfiguration penalty. For the generation of FPGA devices available at the time of this thesis,

$$f_{dclk} \ll f. \tag{4.10}$$

And if $\beta \leq 16K$ then the contribution due to the initialization of the **.BSS** section is on the order of 100's of μs, while the contribution due to the FPGA configuration is on the order of 10's of ms. And since K is on the order of μs the reconfiguration time can be approximated as,

$$\Upsilon(d, f_{dclk}) = \frac{d}{f_{dclk} \cdot C_{BW}}, \tag{4.11}$$

i.e. the reconfiguration penalty is dominated by the time required to load the configuration data into the FPGAs. Alternatively, if the FPGA configuration file is already resident in the FPGAs, the contribution due to FPGA configuration is zero. For this case the reconfiguration penalty is on the order of μs and is expressed as,

$$\Upsilon(\beta, f) = \frac{3\beta}{2f} + K. \tag{4.12}$$

If the application has a β approaching zero then the reconfiguration penalty reduces to just K. In chapter 7 we will apply this model to the completed State Machine stream processor. The next two chapters provide detailed discussion of the hardware and software implementation. From this discussion will emerge many of the constant factors in the timing model.

Chapter 5

Hardware Implementation

5.1 Overview

In the preceding chapter a data-path diagram was presented to elucidate the application flexibility provided by the state machine architecture. In this chapter we describe the hardware implementation in much greater detail. Figure 5-1 is a block diagram of the state machine system architecture that provides more detail than was provided in the data-path diagram of the previous chapter. It provides an intermediate level of detail for the State Machine system architecture between that of the data-path diagram and that of the system schematics. The system schematics in appendix 8.2.3 provide the highest level of detail. While figure 5-1 shows all logical elements, the schematics break these down further into individual semiconductor and discrete devices. Additionally, all of the data-path and control signals are explicitly shown in the schematics. While the schematics provide the greatest degree of detail, they sacrifice global perspective to do so. Consequently, this chapter will refer mostly to figure 5-1 and only to appendix 8.2.3 when necessary to provide detailed explanations of system operation.

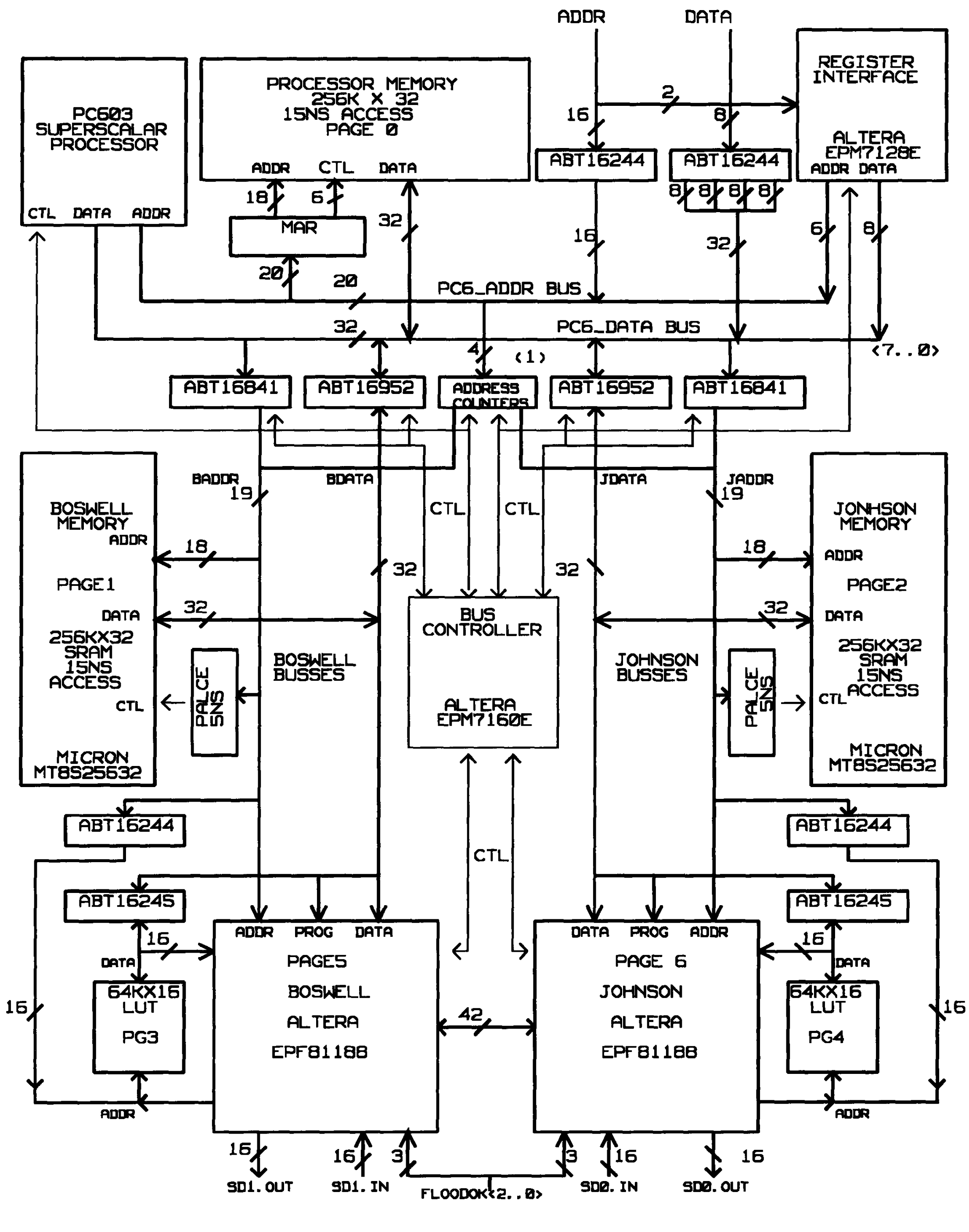


Figure 5-1: State Machine System Architecture Block Diagram

5.1.1 Major Elements

The block diagram begins to reveal the parts used to implement the major functional and logical components. The GPP employed is the PowerPC603 microprocessor built by IBM and Motorola. The PC603 is a superscalar processor with a 32-bit address space and instruction word width. The data bus can be configured in either of a 32-bit or 64-bit mode [27]. The State Machine utilizes the 32-bit mode because the State Machine boards are area limited.[1]

The FPGAs used to provide the platform for dynamically reconfigurable hardware are Altera FLEX EPF81188 devices. The State Machine uses two of these devices, one for each Hub port. These devices provide approximately 12,000 usable logic gates organized in 1008 logic elements and 180 I/O elements[3]. There are sufficient programmable logic resources to implement small to medium sized computational architectures. Each device has access to a 1MB memory bank as well as a 64Kx16 Look-Up Table (LUT) memory. In addition, each device has a direct interface to the Hub. That is, data coming from the VRAM memory banks on the processor module flow directly into the FPGAs on the State Machine submodule from the Hub.

The transfer registers that connect the PC603 busses to the FPGA busses consist of ABT type logic devices and two PAL devices. Because the burst modes of the PC603 processor are supported, the lower bits of the address busses must have the ability to count, and are thus housed in a 22v10 PAL. The **counters** PAL holds bits $\langle 3:1 \rangle$ of each FPGA bus and implements two three bit counters for burst transfers. The second PAL holds control logic necessary to facilitate the bus transfers.

After allowing space for the two FPGAs and other system components, enough room remained to fit three 32-bit wide SIMM modules of fast static RAM. From figure 4-1 one SIMM is dedicated to each of the processing elements, local to each system bus. Although the limited memory provided to the high performance general

[1]Note that word and word-width both refer to 32-bit data types in this document.

purpose processing element does not satisfy the Amdahl/Case Rule [26],[2] this is not a serious limitation because the processor is rarely utilized to its full potential.

5.1.2 Data-paths

There are three major busses on the State Machine board. These are the PC603 bus, the Boswell FPGA bus, and the Johnson FPGA bus. The PC603 bus consists of a 20-bit address bus, a 32-bit data bus, and several control signals. The PC603 bus signals are listed in table 5.1.

Signal Name	Description
PC6_ADDR$\langle 19:0\rangle$	20-bit address bus
PC6_DATA$\langle 31:0\rangle$	32-bit data bus
PC6SIZ$\langle 2:0\rangle$	transfer size bits
PC6ABB	PC603 address bus busy signal
PC6DBB	PC603 data bus busy signal
PC6R/W	PC603 bus transfer read/write signal
TBST	PC603 burst transfer signal
TCO	PC603 instr./data transfer signal
PC6BR	PC603 address bus request
PC6BG	PC603 address bus grant
PC6DBG	PC603 data bus grant
PC6TS	PC603 transfer start signal
PC6AACK	PC603 address acknowledge signal
PC6TA	PC603 transfer acknowledge signal

Table 5.1: PC603 Bus Signals

The PC603 actually has more signals available to describe bus transfers. However, these are not used in the state machine system.[3] In addition to the address bus signals, address bits [27:20] of the PC603 are used to select various devices to read from and write to. Each bit selects a single device when asserted positive true. All devices,

[2]The Amdahl/Case rule states that a machine should have 1 M-byte of memory for each MIPS (million instructions per second) of performance it offers.

[3]For more information on these other transfer bus signals see [27].

including the FPGAs, may be selected and accessed by the PC603. Table 5.2 lists the address bit and device associations.

Address Bit	Board Signal Name	Signal Description
addr[27]	JLUTB	selects the Johnson LUT memory device
addr[26]	BLUTB	selects the Boswell LUT memory device
addr[25]	JFLEXB	selects the Johnson FPGA device
addr[24]	BFLEXB	selects the Boswell FPGA device
addr[23]	REGB	selects the register interface device
addr[22]	PG2B	selects the Johnson FPGA memory bank
addr[21]	PG1B	selects the Boswell FPGA memory bank
addr[20]	PG0B	selects the PC603 local memory bank

Table 5.2: Address Bit Device Selection Mappings

The Boswell and Johnson busses contain identical signals. Consequently, only the Boswell bus will be described here. The bus contains a 19-bit address bus, a 32-bit data bus, and several control signals. Table 5.3 summarizes these signals.

Bus Signal Name	Signal Description
BOS_ADDR⟨19..0⟩	Boswell address bus
BOS_DATA⟨31..0⟩	Boswell data bus
BOS_SZ	Boswell data bus transfer size signal
/BBB	Boswell bus busy signal
/BBR	Boswell bus request signal
/BBG	Boswell bus grant signal
BUSLOCK	Boswell buslock signal
/BMCE	Boswell memory bank select signal
/BFLEXWE	Boswell write enable control signal
/BPC6WE	PC603 Boswell bus write enable control signal

Table 5.3: Flex Bus Signals

The State Machine provides the capability to bridge the two FPGA address busses. Transmission gates can connect the two address busses such that an address asserted by either FPGA is seen by both FPGA memory banks. [4] Each FPGA has a separate signal to enable this capability. The Boswell enable, **/BABUFFEN**, is asserted low, while the Johnson enable, **JABUFFEN**, is asserted high.[5] When either is asserted the signal pairs listed in table 5.4 are directly connected. This functionality is desirable for address remapping where one FPGA is serving as an address generator while the other shuffles the data[10].

Boswell Bus Signal Name	Johnson Bus Signal Name
BOS_ADDR⟨19..0⟩	**JON_ADDR⟨19..0⟩**
/BFLEXWE	**/JFLEXWE**
BOS_SZ	**JON_SZ**
BMCE	**JMCE**

Table 5.4: Flex Bus Signals Connected by Transmission Gates

5.2 Memory Organization

The State Machine board contains five separate memory components, all implemented using SRAM.A single SIMM module is resident on the PC603 bus and provides local storage for the operating system, configuration data, and application programs. The other two SIMMs reside on the FPGA busses and provide storage space for video data. One SIMM is allocated to each FPGA device and serves as local storage. Finally, the two 64Kx16 12ns SRAM devices are present to provide fast LUTs for the FPGA devices.

[4]These devices are not shown in the block diagram of figure 5-1 for the sake of clarity. However, they are present on page 5 of the schematics in appendix 8.2.3.

[5]The Boswell enable is asserted negative true, while the Johnson enable is asserted positive true.

PC603 Memory Bank

The PC603 memory bank is a 1 M-byte 256 x 32 SRAM module. Each memory element has some associated access control logic. The logic for the PC603 memory element is labeled MAR in figure 5-1. It consists of an ABT16952 register device and an Altera EPM7032 programmable logic device. The register holds bits $\langle 19:5 \rangle$ of the address bus while the logic device generates all control signals required by the memory device and the low order address bits. The control signals include an output enable, a write enable, and four chip selects, one for each byte of the 4-byte word width.

All possible data transfer alignments are not supported. Half-word accesses must be aligned to half-word boundaries. This means that the address of a full-word must end in **0b00**, and the address of a half-word must end in either **0b00** or **0b10**. [6] The page0 MAR (EPM7032) logic device holds the necessary logic for performing these alignment translations. In addition, this logic includes the ability to sequence the lower three memory device address bits from **0b000** thru **0b111** to support the burst modes of the PC603 processor. The input signal **/PG0MAR_CNT** is used to initiate a counting sequence for the low order three address bits.

Instructions and data on the PC603 bus are 32-bits wide, and most of the bus transactions will be full-word accesses. However the bus must support byte transfers to enable the transfer of data from the 8-bit control data bus to the memory device and the transfer of byte-wide data to and from the register interface.

FPGA Memory

The FPGAs each have a 256kx32 data memory bank and a 64kx16 LUT memory bank designated for their use. The FPGA memory banks can be accessed in either half-word, or word widths. These data widths are consistent with the video data

[6]Addresses that do not follow these conventions are automatically mapped to the nearest full-word or half-word boundry.

types supported by Cheops. However, the FPGAs have the capability to support other data types at the expense of interface complexity. We support half-word and word widths to minimize this complexity. Data alignment is handled in the same fashion as with the PC603 address bus. Since the FPGA data memory and data bus only support half-word and full-word transfer widths, only nineteen address bits and one transfer size bit are required.

The access control logic for the data memory banks is housed in a 16v8 type PAL device with a 5ns input to output time. The PAL generates the **/OE**, **/WE**, and **/CSE**$\langle 4:1\rangle$ control signals of the memory device. The PAL generates these as a function of **BOS_DATA**$\langle 0\rangle$, **BOS_SZ**, **BMCE**, and two write signals **/BFLEXWE**, **/BPC6WE**. The access control logic for the FPGA data memory requires two write selects to generate the **/WE** signal because the memory may be accessed for writes synchronous to two separate clocks. When the PC603 accesses this memory it does so synchronous to the 28MHz **SYSCLK** signal. However, when the FPGA accesses the data memory it does so synchronous to the 32MHz/40MHz Hub clock. Thus each bus master must have a separate write enable signal.

LUT Memory

The two LUT memories provide only half-word width data and were selected to allow identity mappings using the data size of the data types received from the Cheops Hub. An example of the use of these memories is the table look-up of code words during variable length coding/decoding applications. These devices are read-only memory for the FPGA devices that have very fast (12ns)access times. The LUT memories can be read or written by the PC603 processor which is responsible for initializing the tables for applications that use them. The access control PAL for the data memory bank also generates the **/OE** and **/WE** signals for the LUT memory based on the driven values of the Boswell bus signals and the device chip select. [7]

5.3 PC603 Bus Access

There are two possible bus masters for the PC603 bus. These are the PC603 processor and the LP. Each may generate transfer requests completely asynchronously from the other. Consequently, the state machine must be able to process simultaneous requests in a graceful fashion. Because each processor accesses the bus with a different access protocol, the State Machine bus controller must support both protocols. This section describes how these issues are handled.

5.3.1 Local Processor Access

The local processor accesses the state machine board through the control bus interface.[8] The local control bus signals are provided in table 5.5 [42].

Bus Signal	Signal Description
/ADS	Transfer start signal
W/R	Transfer Read/Write signal
/CS0	Chip Select Zero
/CS1	Chip Select One
ADDR[15:0]	Address Bus Signals
DATA[7:0]	Data Bus Signals
/READY	Transfer Delay Signal (asserted high)

Table 5.5: Local Control Bus Signals

The LP initiates a transfer by asserting the **/ADS** signal for one bus clock cycle. Once the transfer begins, it cannot be stopped because the LP has no arbitration

[7]The chip select is asserted by the PC603 as part of its address generation. See section 5.6.1 for more information on how this is accomplished.

[8]This bus is labeled LxCTL, where x is 0, 1, or 2 and corresponds to the submodule slot the state machine card is plugged into on the processor module [41] [29].

ability in its protocol. But, a transfer may be delayed if the State Machine de-asserts the **/READY** signal, in which case the LP waits until the signal is de-asserted for its results. Figure 5-2 provides the timing for a typical control bus transfer and a delayed transfer in which **/READY** has been de-asserted. If the **READY** signal is asserted the bus signals remain asserted until one cycle after **/READY** is de-asserted. This feature provides the remote device the ability to extend the normal transfer protocol. The two **/CSx** signals may be used to select up to two separate stream processors that reside on a stream processor submodule. The State Machine uses these signals slightly differently. **/CS0** is asserted when the transfer is intended for a register in the register interface. **/CS1** is asserted when the local processor is initiating a transfer to the PC603 memory bank.

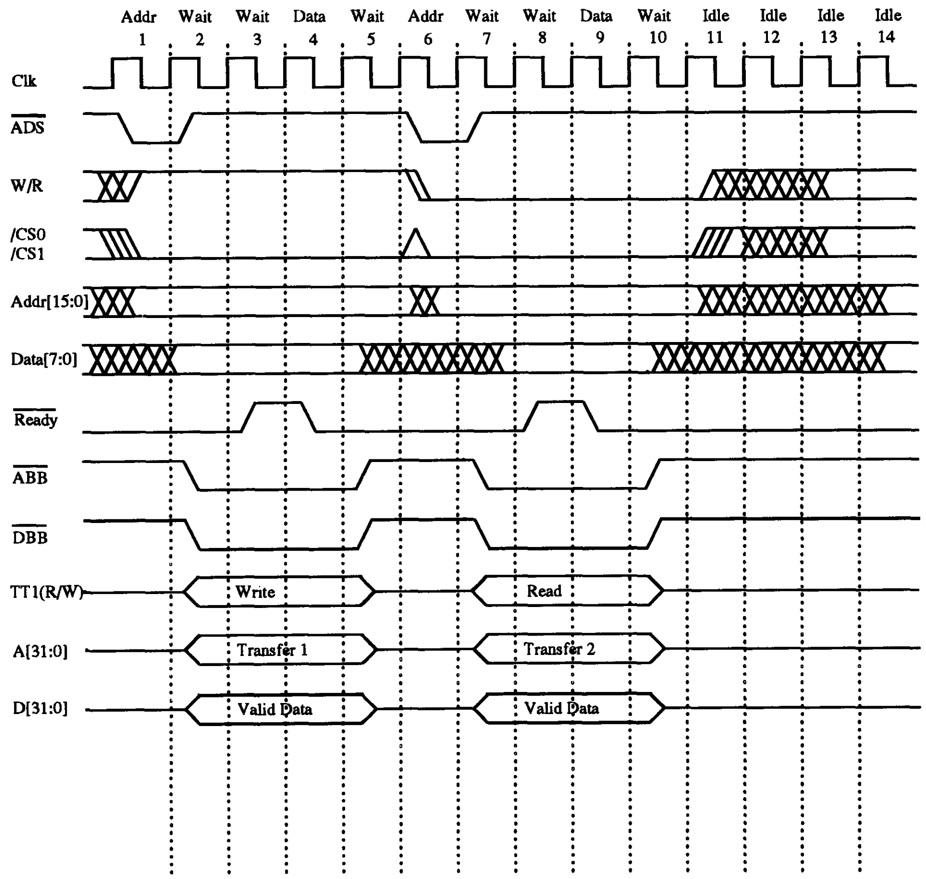


Figure 5-2: Local Bus Transfer Timing

5.3.2 PC603 Access

The PC603 supports split bus transactions and may pipeline transfers two deep internally. In addition, it provides support for symmetric multiprocessing system architectures and implements the MESI memory coherency protocol [27]. A great number of control signals are provided to implement these features. Although the State Machine has no need for multiprocessing capabilities, supporting the split bus transactions, pipeline transfers, and the many transfer modes, make the PC603 bus access protocol significantly more complex than the LP access protocol.

To support split bus transactions, the PC603 has separate arbitration signals for the address and data busses and they may be in a different phase of the transfer process at the same time. Each transfer (both address and data) occurs in three phases, arbitration, transfer, and termination. The tenures of the address and data busses overlap. Figure 5-3 provides an example of how the tenures of each bus overlap when access to the address bus is granted immediately and access to the data bus is delayed one cycle. Although the length of the arbitration and transfer cycles may vary depending on how quickly access to the busses is granted, the arbitration phase of the data tenure always overlaps the transfer phase of the address transfer. This is because the **/TS** signal for the address bus serves as an implicit data bus request. Thus the data bus arbitration phase is always initiated during the address bus transfer phase. There are several other types of transfers that will be discussed further in section 5.6.2.

5.3.3 PC603 Bus Arbitration

Both the PC603 and the LP may access the PC603 bus in an asynchronous fashion. Because this is so, an arbitration protocol must be implemented to insure that the two accesses do not conflict. The specification of this protocol is complicated by the fact that the LP transfer protocol does not have arbitration capabilities built in; it is intended to access the State Machine like a memory element. Once a LP access

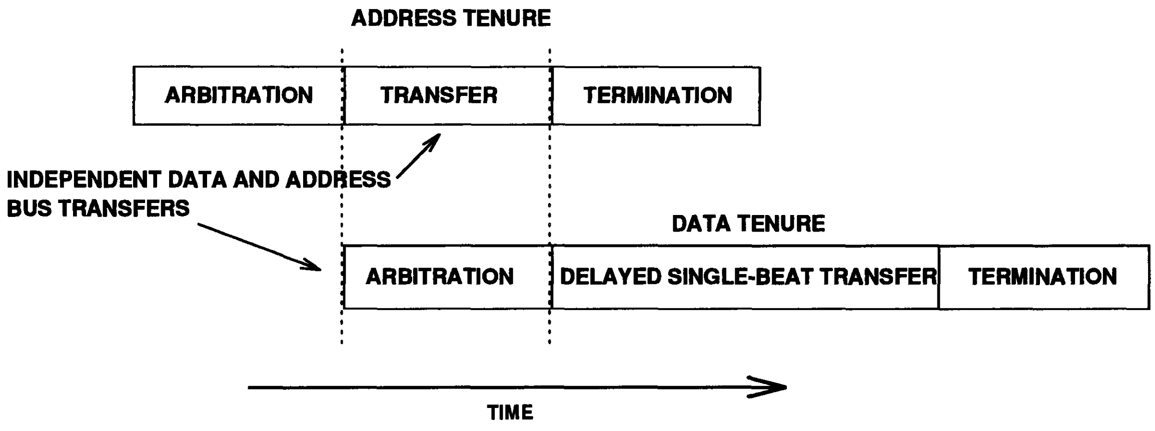


Figure 5-3: PC603 Bus Transfer Tenures

begins it must be allowed to complete otherwise it will bring all of Cheops to a stand still. In order to provide limited arbitration capabilities for the LP protocol, we take advantage of the **/READY** signal to implement a pseudo form of arbitration between the PC603 and the LP.

This arbitration scheme assures that the LP will always complete its transaction within a reasonable amount of time so that it does not cause Cheops to crash. By taking advantage of the **/READY** signal and treating **/ADS** as a bus request instead of a transfer start signal, a makeshift arbitration scheme is implemented. Regardless of who has priority, LP transfers always complete with minimal interference to PC603 transfers.

5.4 Register Interface

The register interface consists of several registers and corresponding control logic that together implement a communications interface between the State Machine and the LP. Through this interface the LP controls the State Machine board. It is housed in an Altera EPM7128 programmable logic device. This device also contains the configuration clock circuitry and the control circuits for the data bus exchanger between the control and PC603 busses.

5.4.1 Registers

There are six 8-bit registers and a 2-bit register in the register interface. There are two general purpose registers that may be read or written by either the PC603 or the LP. A read-only status register may be monitored by both processors as well. The LP controls the state machine through the bus priority register which is write only. Two other write-only registers, the configuration register and the memory page register, also assist in configuring and controlling the board. The 2-bit register allows the PC603 to determine which FPGA generated an external interrupt exception. The role of each register will be discussed in detail. Table 5.6 summarizes these registers.

Register	Offset	Type	Description
Status	0x00	read only	Board status register
GP1	0x01	read/write	General Purpose Register 1
GP2	0x02	read/write	General Purpose Register 2
Bus Priority	0x04	write only	Controls board mode and bus priority
Configuration	0x05	write only	controls the flex configuration process
External Int.	0x06	read only	determines which FPGA caused an interrupt
Memory Page	0x07	write only	selects segments of PC603 memory

Table 5.6: Registers of the Register Interface

Status Register

The status register allows both the PC603 and the LP to monitor the status of the State Machine board. The register is implemented as a 74273 type registered device with each bit being updated on every clock cycle and specifying the status of a different signal. Table 5.7 gives the definitions of each bit.[9]

The first two bits of this register monitor the mode of the board. Board modes and their meaning will be discussed in section 5.5. Bits two and three are used to monitor

[9]For this register and all others in the register interface, bit zero is the least significant bit.

Bit	Description
0	Board config mode bit
1	Board start mode bit
2	Boswell confdone signal
3	Boswell nstatus bit
4	Johnson nstatus bit
5	Johnson confdone bit
6	Configuration clk enable status
7	Configuration clock

Table 5.7: Bit Definitions of the Status Register

the Boswell FPGA status signals. **Nstatus** is used to indicate an error during device configuration while **confdone** indicates that the configuration process is complete. Bits four and five play an analogous role for the Johnson FPGA device. Bit six indicates whether or not the configuration clocks are enabled. Finally bit seven is the configuration synchronization clock. This bit is monitored by the PC603 during FPGA configuration to determine when to write the next bytes of configuration data to the FPGAs.

General Purpose Registers

There are two general purpose registers in the register interface that do not have a specific function. They are both readable and writable by both processors. These are provided to facilitate inter-processor communication and assist the Cheops resource manager, Norman, in configuring the State Machine board. The PC603 operating system, ChameleonOS, makes extensive use of these registers for handshaking protocols with the LP and to provide limited error reporting under conditions of catastrophic failure. They are not intended for general application use.

Memory Page Register

The memory page register is used to assist the LP in addressing the PC603 memory bank. The control **ADDR**$\langle 15:0 \rangle$ bus used by the LP to address the state machine is only 16-bits wide, allowing for a 64 K-byte address range. Since the PC603 memory bank provides 1MB of storage, the LP needs assistance in order to address the full memory space of the PC603. This assistance is provided by the memory page register which provides the upper 4 bits of the 20-bit PC603 memory address. The use of the memory page register essentially divides the PC603 memory bank into 64 K-byte pages with the 4-bit page number coming from the memory page register and the 16-bit offset into the page coming from the 16-bit control address bus.

Configuration Register

The configuration register is used to control the FPGA configuration process and is a write only register. Bit 0 of this register is used to enable and disable the two configuration clocks. Bit 1 of this register is used to start the configuration process. The output of this bit feeds the **nconfig** signal of both FPGA devices. The configuration process is begun shortly after this bit transitions from a zero to a one.

The FPGA configuration clock enable signal is also fed to bit six of the status register so that it may be monitored. The default assumption is that if this bit is asserted, the configuration process is in progress. *This implies that the PC603 must de-assert this bit after the configuration process is complete.* The PC603 determines that the process is complete by monitoring the status register, since the FPGA's **confdone** signals are inputs to this register.

Bus Priority Register

The bus priority register is the primary means by which the LP controls the State Machine board. It is a write only register that may be written by either processor. However, under most normal conditions only the LP should be writing to this register.

It contains five configuration bits used to control board activity. These bits are summarized in table 5.8.

Bit	Description
0	Boswell FPGA bus priority bit
1	Johnson FPGA bus priority bit
2	PC603 bus priority bit
3	Board config mode bit
4	Board start mode bit
5-7	RESERVED

Table 5.8: Bit Definitions of the Bus Priority Register

There are three bus control configuration bits and two board control bits. The bus priority bits determine which of the two possible bus masters on each bus has priority. For both of the FPGA busses, when the bit is zero the FPGA devices have priority on the bus. When the bit is one the PC603 has priority on the bus. For the PC603 bus, the LP has priority when the bit is zero and the PC603 has priority when it is one.

The final two configuration bits in this register control the mode of the board. The mode of the board determines what actions may take place and what functionality is available. The board mode also assists the bus controller in determining what actions to take. The board modes are summarized in table 5.9.

start bit	config bit	Mode	Description
0	0	**Idle**	Board is idle, function undefined
0	1	**Write**	Local Processor writing data to the board
1	0	**Config**	PC603 boots and FPGA devices being configured
1	1	**Normal**	Board configured and ready to compute

Table 5.9: State Machine Board Modes

The final three bits of the register are unused and reserved for future use. The use of these bits and the board modes will be discussed in much greater detail in section 5.5.

5.5 Board Modes and Configuration Control

There are four distinct modes that the State Machine board can be in. These modes determine what board activity can take place and help the bus controller control the various board functions. The mode information is stored in the bus priority register in the register interface.

5.5.1 Board Modes

The four modes are listed in table 5.9, and are controlled by the two mode bits **strt** and **config**. The **IDLE** mode is the default mode for power-up, there is no board activity in this mode. The **strt** bit is used as the hard reset signal for the PC603 processor. Thus when the board is in either of **IDLE** or **WRITE** modes it cannot initiate any bus activity and tri-states all of its bus outputs. The **config** bit for the FPGA devices is also initialized to zero at power-up and thus holds these devices in reset until a command from the PC603 processor changes this bit. Thus when the board is powered up both the PC603 processor and the FPGAs are held in reset.

The board mode is changed to **write** by the LP to download data into the PC603 memory bank. The change in mode enables the bus controller to facilitate this transfer of data from the LP to the memory. Typically, the write mode is used only after power-up to initially write the ChameleonOS operating system code, FPGA device configuration data, and the initial applications into the memory bank.

When the board mode is changed to **config** mode by the LP, the PC603 hard reset signal is released and the 603 bootstraps itself from the code in the PC603 memory bank. The PC603 then performs hardware checking, initialization tasks, and enters the process monitor loop. At this point the State Machine is ready to handle service requests from the LP.

The **NORMAL** mode is the standard mode for processing data. All use of the State Machine by Cheops to process flood data coming from the VRAM banks occurs in this mode. The bus controller is fully enabled in this mode, and FPGA configuration abilities are disabled. Under ordinary conditions the PC603 changes the board mode to normal mode after booting and configuring the FPGAs.

Normal Sequence of Events

The normal sequence of events from power-on until the State Machine is first used for processing, is a progression beginning with **IDLE** mode and finally arriving in **NORMAL** mode. Since all the bits in the register interface are initialized to zeros when the State Machine is powered on, the board defaults to **IDLE** mode with all processing elements disabled. The board remains in this mode until the LP changes the mode by updating the bus priority register, see table 5.8, in the register interface.

When the LP is ready to write programming information to the board it sets the board mode to **WRITE**. This enables the LP to write data to the PC603 memory bank. Since the control address bus is only sixteen bits in width, it must also update and use the memory page register in the register interface to access the upper portions of the PC603 memory bank. The LP fills the PC603 memory bank with the operating system, application code, and FPGA configuration data.

When the LP is finished writing the data, it changes the mode to **CONFIG** mode. In this mode the 603 is released from reset and begins its boot sequence. After the boot sequence is complete, the PC603 begins to program the FPGA devices by manipulating the **config** bits in the register interface and monitoring the two configuration clocks. The PC603 continues to load configuration data bytes into the FPGAs, synchronous to the configuration synchronization clock, until the process is complete or an error is detected. If an error is detected the configuration process is repeated.

Once the PC603 has completed the configuration process it changes the board mode to **NORMAL** mode. At this point the FPGA devices are released for user mode operation and will assume their respective functions as soon as they complete the initialization process. At this point the State Machine is ready to perform stream computations.

5.6 Control Processor

The processor used to control the State Machine board is the IBM PowerPC603 micro-processor. The PC603 is a RISC super-scalar architecture that employs five separate functional units, an integer ALU, a floating-point ALU, a branch processor, a load/store processor, and a system register processing unit [27]. It provides thirty-two 32-bit general purpose registers for integer operations and thirty-two 64-bit floating-point registers for floating-point operations. The PC603 has separate 8K-byte instruction and data caches and supports out-of-order instruction execution and speculative branching. The PC603 also provides dynamic power reduction by shutting down functional units that are not active.

The PC603 is chosen for this application primarily because of its very high performance levels and low power-consumption. Also its highly optimized floating point processor provides single instruction multiplication and division, as well as a multiply and accumulate assembler instructuion. An additional feature that favored it use in the early design phase was its programmable 64-bit/32-bit data bus modes. For the State Machine application the 32-bit data bus mode was selected since it was most consistent both with the image data types to be processed and the 32-bit instruction widths.

5.6.1 Memory Space

The State Machine card has less than 4 M-byte of total memory and memory mapped devices on board. Since the State Machine card has no I/O devices other than a shared memory space, no virtual paging is implemented. However, the processor's block address translation facilities are used to protect certain memory segments from being caching, and to relocate user processes at run-time. Figure 5-4 is a picture of the PC603 memory space that indicates how the various devices are mapped into this space. Note that the very highest addresses that begin 0xFFF are aliased back into the 0x001 range by external logic. This aliasing allows the processor to retrieve instructions from the correct location at boot time and during exceptions.

The PC603 has 1MB of local storage for its operating system, ChameleonOS, user programs, and FPGA configuration data. Figure 5-4 shows the organization of this memory. The memory is divided into eight 128K-byte chunks because this is the smallest block size supported by the block address translation facilities of the PC603. The first block is occupied by the interrupt vector table, the operating system, and system data structures. The last three block is reserved for FPGA configuration data. The middle six slots are available to be allocated to user programs as needed. Chapter 6, Software Environment, will have more to say on the use of this memory and the use of the block address translation facilities.

5.6.2 Memory Access Types

The PC603 performs several different types of data transfers on its address and data busses. These are single beat transfers, double beat bursts, and eight beat bursts. The types discussed in this document represent only the relevant subset of all the transfer types the PC603 is capable of.[10] Single beat transfers refer to the simplest of the transfer types in which a single word or less of data is transfered in a four data

[10]For additional information on the PC603 bus transfer types see chapter 10 of [27].

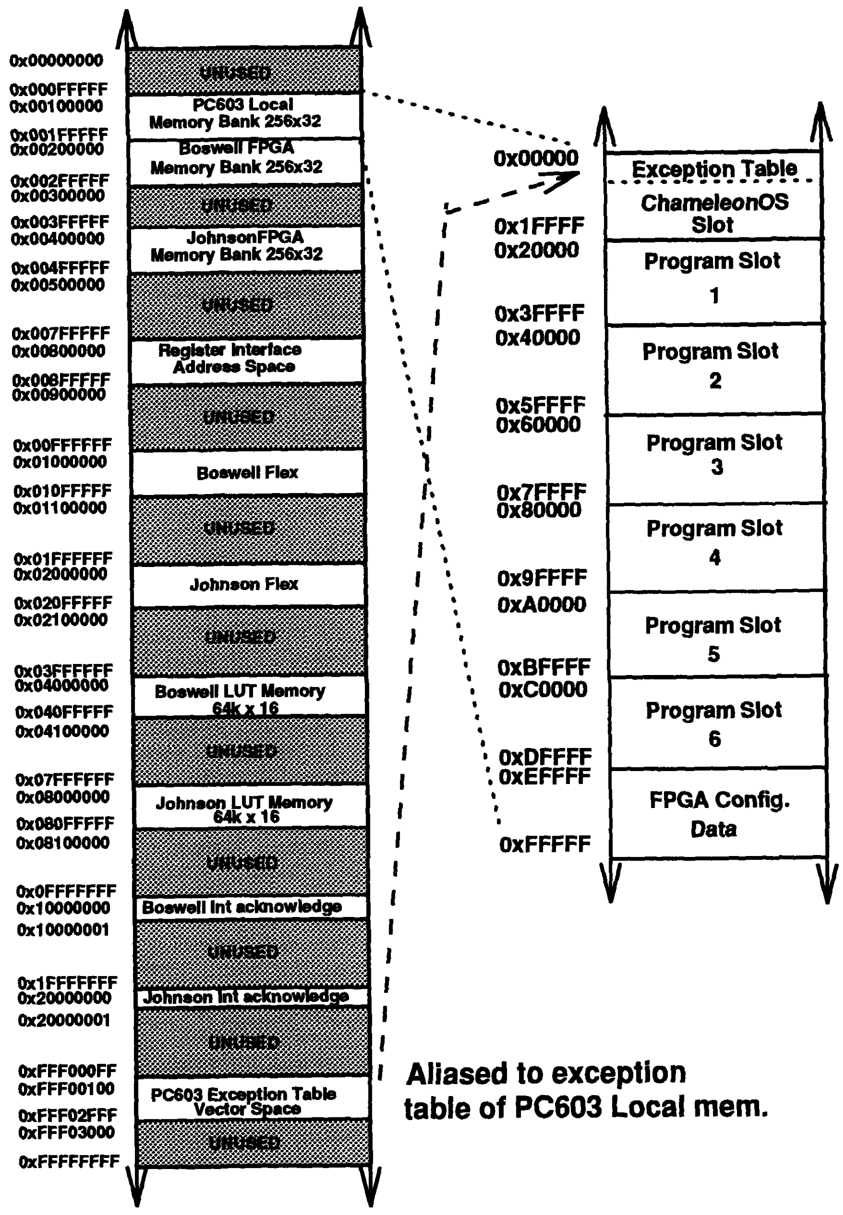


Figure 5-4: PC603 Memory Map

bus clock cycles. The PC603 transfers bytes, half-words, and words in single-beat transfers. Double beat transfers are a special transfer type that take place only when the PC603 is configured for 32-bit data bus mode.[11] Double beat transfers are used to transfer data types that are eight bytes in length, such as the C language types *double* and *long long int.* The PC603 in 32-bit data bus mode performs cache line fills and cache line writes in eight beat burst transfers. These transfers take a full eight bus clock cycles to transfer the data on the data bus.

5.7 Bus Controller and Protocols

The bus controller is the heart of the state machine board and manages all data transfers between the processing elements, PC603 and memory elements. It is by far the most complex element on the board. The bus controller is housed in an Altera EPM7160 logic device. In addition to this chip there are several other logic devices on the board that help facilitate the transfer of data. These are the PC603 to FPGA bus transfer registers, the FPGA memory bank PAL devices, and the PC603 memory bank access logic (EPM7032).

PC603 Bus Protocol

The PC603 bus protocol is outlined in chapter 10, System Interface Operation, of the PowerPC603 Microprocessor User Manual [27]. It is a complex protocol designed for high transfer throughput and supports symmetric multitasking environments. The reader is referred to [27], and section 5.6 for more information. Fortunately, most of the advanced features for this protocol are not required for the State Machine implementation. Consequently, many of the transfer attribute signals are unused and the remaining subset of the protocol implemented is greatly simplified. Despite this

[11]This is the default mode for the State Machine which only supports data transfers as large as 32-bits.

fact, the PC603 bus protocol is still the most complex protocol supported on the State Machine board and requires a large amount of state information to maintain it. This is primarily due to the fact that the PC603 implements split-bus transactions and may pipeline bus transfers two deep internally.

Local Processor Communications Protocol

In contrast, the LP communications protocol, implemented on the control bus, is much simpler and easier to support. However, since the LP can be a bus master on the PC603 bus, some protocol translation must be performed to inform the PC603 of the LP's presence on the bus. Figure 5-2 demonstrates the LP protocol for local bus transfers to and from the stream processor submodule cards [42]. The protocol is very straight-forward as it treats the stream processor submodule as a slave device. That is, the GPP is not capable of initiating a transfer on the control bus.

The presence of the PC603 does introduce some slight complications. The bus controller must handle arbitration issues between the two bus masters, and signal translation must be performed to notify the PC603 of the presence of the LP on the bus. The arbitration protocol has already been described in section 5.3.3 and will not be discussed further here.

5.7.1 FPGA Bus Protocol

The FPGA bus protocols have been designed with absolute simplicity in mind. They have been designed in this manner to minimize the interface overhead that the FPGAs must sustain. The overriding objective is to leave as many logic resources as possible for the implementation of user logic and custom computations. Table 5.4 lists the FPGA bus signals.

When the FPGA desires to use the bus, it asserts the **/BBR** signal. When the **/BBG** signal is given the FPGA may use the bus on the next cycle. The FPGA may not use the bus until **/BBG** has been granted and should continue to assert the

/BBR signal until the **/BBG** is granted. During the cycle that the FPGA is using the bus it must assert the **/BBB** signal to inform the PC603 and the bus controller that the bus is in use.

Under certain conditions, the FPGA may not be able to tolerate long latencies while the PC603 performs extended operations to the FPGA memory, or even spend cycles on bus arbitration. When these conditions exist, the FPGA may assert the **buslock** signal. When the **buslock** signal is asserted, the PC603 is effectively locked out from the FPGA bus and the FPGA may assume complete control ignoring the normal arbitration protocol. If the PC603 attempts a memory access while **buslock** is asserted it will take a transfer error exception. So, for dynamic use of this signal some coordination with the PC603 via software is required.

Bus Arbitration

The bus controller chip also houses all of the bus arbitration circuitry for the busses on the State Machine board. There are three specific arbitration circuits, one for each FPGA bus, and one for the PC603 bus. The FPGA bus arbiters are identical except that each processes different signals. The PC603 arbiter is considerably different and more complex than the FPGA bus arbiters. The arbitration protocol for the FPGA busses is less complex. This is intentional, to avoid wasting precious FPGA logic resources on bus interface issues.

Note that neither the PC603 nor a FPGA device can receive a bus grant when the bus busy signal (**/BBB**) is asserted. This insures there are no bus synchronization problems since the FPGA and the PC603 access the bus with asynchronous clocks. The default mode is each FPGA has priority on its respective bus. *For the case when the two address busses are bridged the* **buslock** *signals from both FPGAs must be asserted to insure that the PC603 does not interrupt address remap processing.*

5.8 FPGA Devices

The State Machine employs two Altera FPGA EPF81188 SRAM based FPGAs that serve as the dynamically reconfigurable computational platform. It is these devices that provide the State Machine's extraordinary power and flexibility. Each FPGA is dedicated to one of the Hub ports on the State Machine stream processor submodule.

Each FPGA provides 1,008 bits of registered storage. Each bit is coupled with a 4 input one output programmable logic function. 180 additional periphery registers are provided in the I/O elements[3].[12] These FPGAs provide space to implement custom architectures so that pieces of complex image processing algorithms can be performed in dedicated hardware. Applications designed for the State Machine consist of both application code and architectural description information, in the form of AHDL files, to be loaded into the FPGAs.

5.8.1 External Signal Interfaces

The State Machine FPGAs have three other interfaces as well as the Hub port interface. There is the FPGA data bus interface that connects each FPGA to its memory bank. This is an arbitrated interface since the bus may also be mastered by the PC603 processor. The signals for this interface are outlined in table 5.3. The LUT memory interface consists of **BLUTADDR**⟨15..0⟩ and **BLUTDATA**⟨15..0⟩, and allows the FPGA device to retrieve LUT data very quickly. Finally, there are 42 signals that run between Boswell and Johnson for inter-FPGA communications. The **BOS_JON**⟨41..0⟩ bus may be used for whatever communication needs that a State Machine application might have.

[12]As of the beginning of this project these devices represented the state-of-the-art in FPGAs. However as of this writing more dense and more advanced SRAM based FPGAs are now available.

5.8.2 Bus Protocol Logic Overhead for FPGAs

Since the State Machine FPGAs are fixed within a larger system architecture it is not possible to utilize all of their logic resources to implement user applications. A small portion of the logic resources within each FPGA must be used to implement the various interface protocols. These protocols have been designed to be as simple as possible to avoid maintaining lots of state internally and thus minimize circuit overhead. The LUT interface and the FPGA interface, if implemented, are considered to be a part of the application and are not considered here. However, each FPGA must use some logic to keep track of system state for the Hub and FPGA memory bus interfaces. Between the two the required logic requires only 18% of the logic resources of the chip.

5.8.3 Flood Interface

The Hub interface requires a minimal amount of logic to monitor the status of the selected OK signal and sense the beginning and end of data flow. A finite state machine, DAM, with four states controls this interface. Figure 5-5 depicts the states and the input output relation of the controller. In addition, a delay value register, an 8-bit counter, a 2-bit OK channel select register, and some combinational logic to generate the OK active signal are used. Figure 5-6 is a register transfer level description of this logic. The controller remains dormant until it detects the assertion of the OK active signal. It then signals the counter to begin counting. When the counter has counted through the delay count, signaling the end of the wait period before valid data, the controller transitions to the **FLOWING** state and produces the data flowing signal. When the OK active signal is de-asserted the controller again signals the counter to count through the delay value. At the end of the count, the controller de-asserts the data flowing signal and returns to the **DRY** state. This logic implements a basic Flood Interface. [13] Applications may implement more complex interface logic if desired.

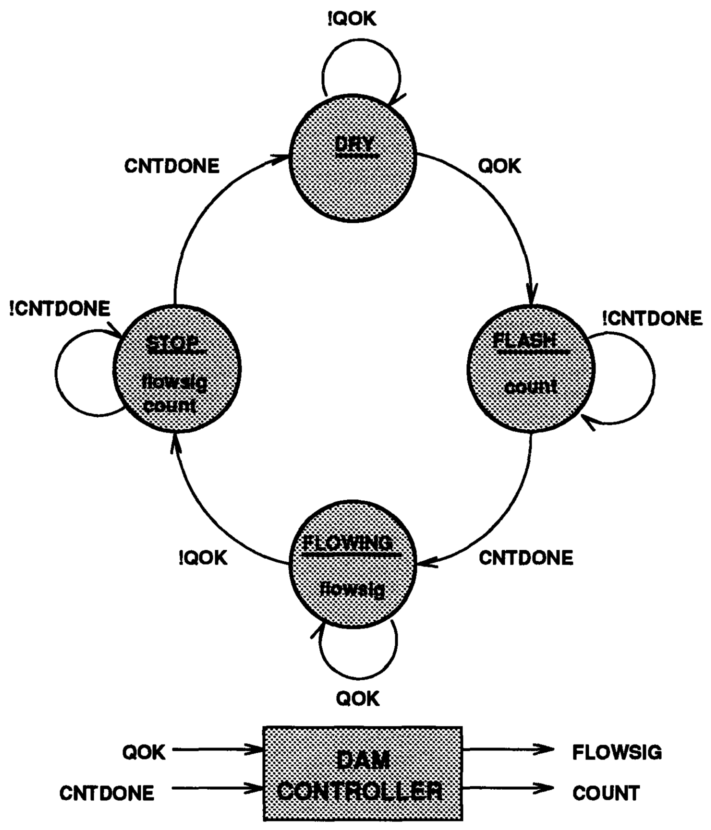


Figure 5-5: FPGA Flood Interface State Machine, DAM

Memory bus interface

The FPGA memory bus interface is the most complex of the FPGA interfaces. It requires a fair amount of logic to preserve the state of the bus. The logic for this interface includes a 7-state finite state machine, several flip-flops for control signals, registers for the address and data, and a 19-bit address counter. Figure 5-7 describes the state machine and figure 5-8 is a block diagram of the bus controller logic. The data and address registers are implemented in the periphery registers of the I/0 elements of the FPGA devices and thus do not use any internal logic resources.

[13]This example was taken from the AHDL file dell.tdf

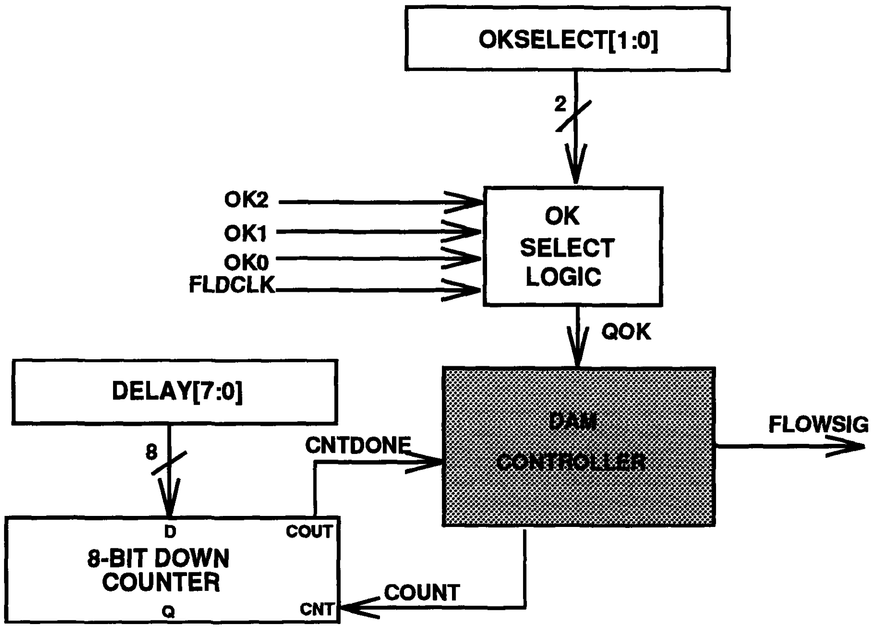
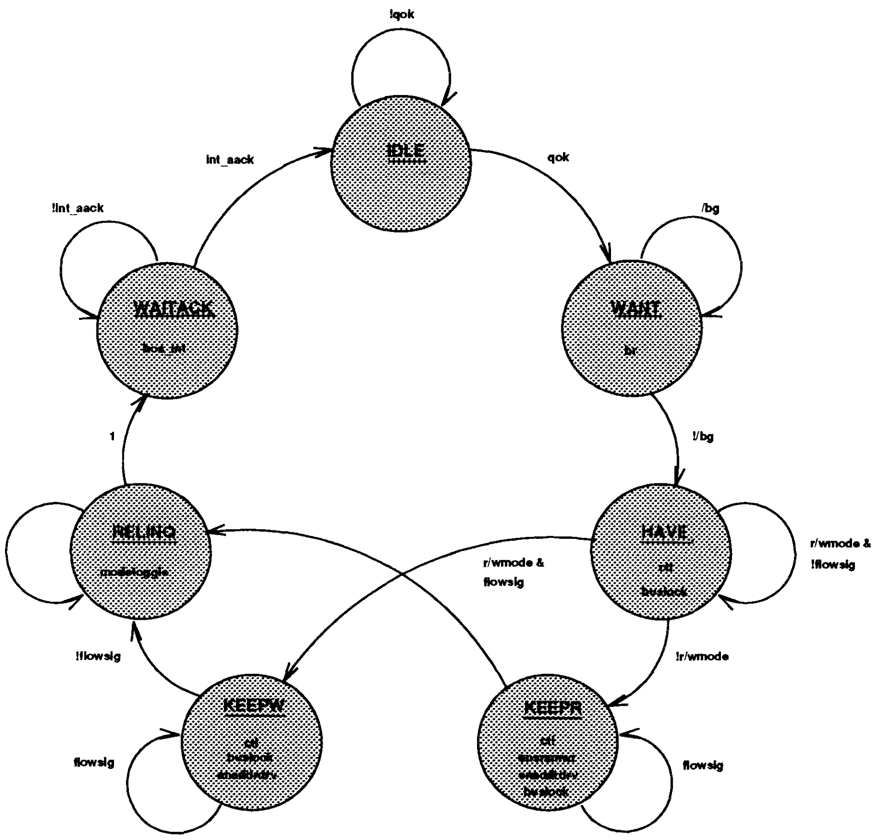


Figure 5-6: FPGA Flood Interface Logic

In addition to saving logic resources, the data and address registers are placed in periphery registers so that fast bus accesses may be accommodated. Specifically, the Hub clock, which is the system clock for the FPGAs may be operated at up to 40MHz. In order to read from the FPGA memory banks at this speed, the memory address and the return data must be registered in the I/O elements to allow for fast clock to output and data setup times.

The interface circuit described in figure 5-8 is a generic example taken from a flood controller design. This application simply streams data into the FPGA memory bank so that the PC603 can operate on it, and then streams the results back onto the Hub at the appropriate time. Other applications may require a more, or less, complex controller circuit depending on their requirements. The logic for this interface can be simplified and reduced for applications that are going to make exclusive use of the FPGA memory bus and do not have to contend with bus arbitration issues.



ctl = tri-state enable for bus busy & bus_sz memory bus signals
ensramwr = tri-state enable for data bus bits
enaddrdrv = tri-state enable for address bus bits and cnt enable for addr register
qok, flowsig - signals from the flood interface controller

Figure 5-7: FPGA Memory Bus Controller

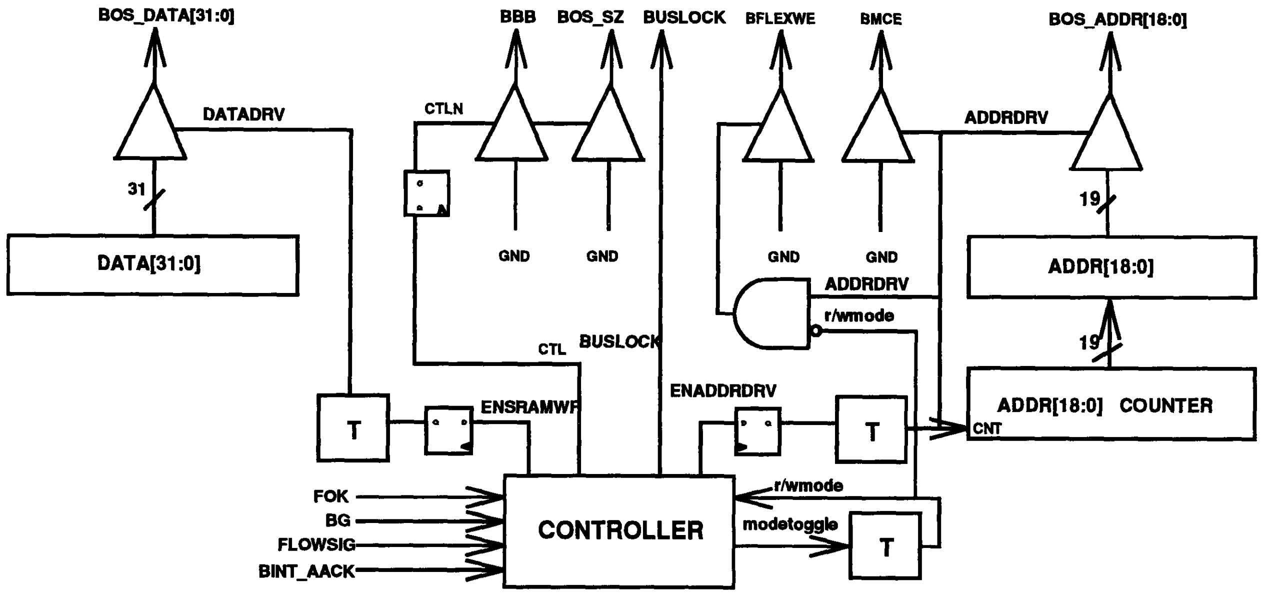


Figure 5-8: FPGA Memory Bus Logic

5.8.4 Timing Issues

The State Machine FPGAs use three clock signals internally. All application logic is clocked from the Hub clock, **BEHUBCLK**. The Hub clock serves as the system clock for the FPGAs. The **FLOODOK** signals are synchronous to the **FLDCLK** and this clock is used to validate the assertion of these signals. The PC603 system clock, **SYSCLK**, is also available as a reference to handle any synchronization issues that might arise in State Machine applications.

The FloodOK signals are used by the LP as handshaking signals for the stream processors to indicate the presence of valid data on the flood ports. Stream processors either read data from the ports or write data to the ports in response to these signals. The FloodOK signals are asserted synchronous to the Floodclk, which operates at half the frequency of **BEHUBCLK**. The FloodOK signals are only to be considered valid at the rising edge of **BEHUBCLK**, iff **FLDCLK** is at its valid low logic level at this time. Figure 5-9 illustrates the condition for validating the **FLOODOK** signals. **BEHUBCLK** is a buffered and delayed version of **EHUBCLK** that is intended to be a close approximation to the actual **HUBCLK**. **EHUBCLK** is an early version

HUBCLK that is distributed to the stream processor submodules to compensate for delays in the clock signal caused by clock distribution and buffer delays.

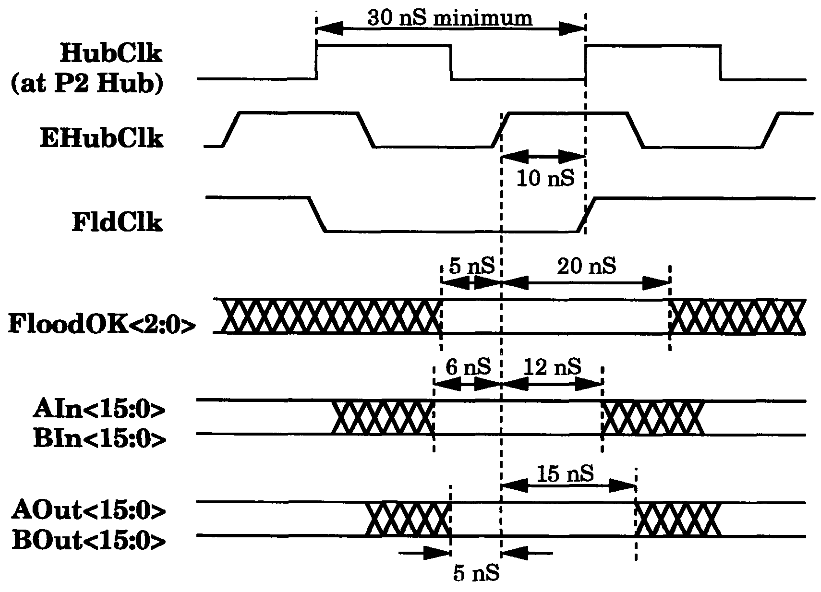


Figure 5-9: Flood Interface Timing

When the Hub clock is operated at 40MHz, the timing of the FPGA memory reads and writes becomes critical. At this speed the clock period is 25ns while the access time of the SRAM is 15ns. Because of the considerable I/O buffer delays of the FPGAs it is not possible to read from memory directly to an internal register. To perform the read in one clock cycle the address must be registered in, and the return data latched in, the registers in the I/O elements. The clock to output times for the address bits, and the register setup times for the data bits, is approximately 4ns [4]. [14] If the Hub clock is operated at 32MHz or less this requirement is not necessary. However, it is a good idea to register the transfer address and the incoming data in the I/O elements as this cuts down on internal logic resource usage and uses registers that would otherwise be wasted. Figure 5-10 provides the timing parameter information for this interface.

[14]For precise, current values for these timing parameters consult the Altera Data Book 1994

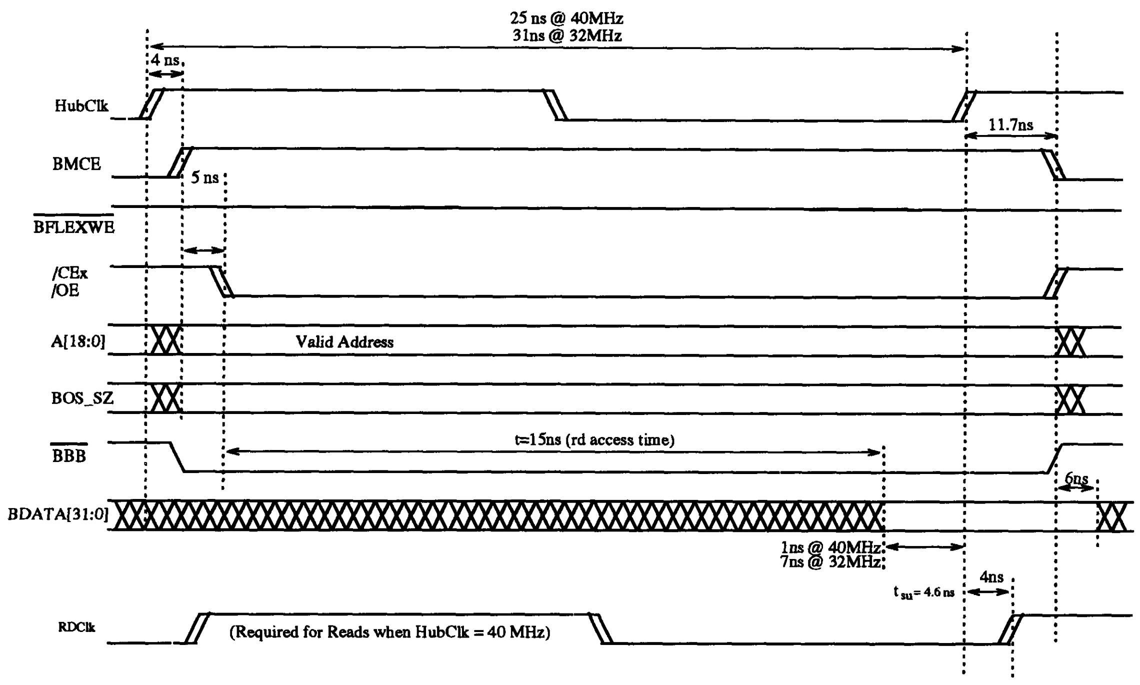


Figure 5-10: FPGA Memory Access Timing

5.8.5 LUT tables

Each FPGA has an associated LUT memory bank for designs requiring table look-ups. This memory is a Micron MT5C64K16A1 64Kx16 SRAM device with a 12ns access speed[34]. It allows FPGA applications to have table look-up access in a single clock cycle even when the Hub clock is operated at 40MHz. This allows the FPGAs to carry-out applications like Huffman decodes and table-lookup multiplies at the Hub clock rate.

To maximize the speed of access these devices are always enabled when the State Machine is in **NORMAL** mode. The FPGAs need only drive an address on **BLUTADDR** to perform a table look-up. This configuration also has the additional advantage of requiring very little logic to implement this interface.

Data is written into the LUT memories by the PC603 processor while the board is in **CONFIG** mode. The FPGAs do not have the ability to write to these memory elements. Once the PC603 changes the board mode to normal mode the chip selects are asserted and the write enables are de-asserted placing the devices in read only state. The PC603 may write to the LUT memories in **NORMAL** mode but this is not advised since the FPGAs may be reading at the same time and there is no mechanism for bus arbitration.

This concludes the hardware description of the State Machine stream processor. All of the major components have been addressed, along with the more subtle interface issues between them. The focus of this chapter has been on providing commentary on the more important aspects of the architecture without overwhelming the reader with technical detail. Most issues not addressed here can be resolved with a casual review of the schematics provided in appendix 8.2.3. In the next chapter we continue in this spirit with a discussion of the supporting software that provides the programming environment for the State Machine.

Chapter 6

Software Environment

6.1 Overview

The State Machine software environment includes three distinct components. A small operating system kernel manages the PC603 processor and launches applications. There is the application code itself. In addition, each State Machine application has an associated FPGA configuration file. These files are software descriptions of the hardware architectures to implement in the FPGAs for the application. Finally, the Cheops P2 resource manager is responsible for managing the State Machine stream processor memory resources. In this chapter, we begin with a description of the software development environment. Next, each of the above components will be discussed in detail. The final section of this chapter will discuss the integration of the P2-side State Machine library code into the Cheops resource manager, Norman.

6.2 Development Environment

All of the code for the State Machine stream processor was developed on Unix workstations, mostly in the C programming language. All code was compiled, assembled and linked with the GNU C compiler and binary utilities[21]. Code targeted for the P2 local processor, i80960, was compiled with GCC running on a DECstation 5000/200 configured as a cross-compiler for the i80960 architecture. Code targeted for the PC603 processor was compiled with GCC in native mode running on an IBM RS6000 workstation using the PowerPC compiler options.

Some of the code for the PC603 operating system was developed in PowerPC assembly language. Specifically, the bootstrap and process swap facilities were coded in PowerPC assembler. In addition the **log_event**() function, the State Machine equivalent of **printf**() was also coded in assembler. All assembly code segments were preprocessed with the GNU C preprocessor[20], to resolve macros and remove C style comments, and then assembled with the **as** assembler[22].

6.3 ChameleonOS Operating System

The ChameleonOS operating system is responsible for controlling and managing all on board functions of the State Machine stream processor. These responsibilities include initialization, FPGA configuration, application process launch, and LP communications. Before a State Machine can be used for any purpose, the operating system must be loaded into the PC603 memory bank and the processor allowed to complete the boot sequence. With this task complete, the State Machine is ready to service requests from the local processor. The ChameleonOS operating system handles these requests and any required communication with the local processor.

6.3.1 Design Objective

The design objectives of the ChameleonOS operating system were threefold. The operating system should be designed in such a way that it is simple, does not require much space, and is highly efficient in switching applications. To simultaneously meet these goals ChameleonOS has been designed to be as simple as possible and includes only the bare minimum functionality required to launch applications and communicate with the LP.

As a consequence of these goals this operating system differs substantially from typical operating systems in several ways. First, there is no memory management whatsoever. The PC603 local memory bank is a shared memory region, with the PC603 having word access and the LP having byte access. The LP is responsible for all memory management of this memory region. Thus the LP loads and keeps track of all code and data on the card, including the ChameleonOS operating system code. The operating system keeps no internal information about memory usage other than that required by its system tables and that provided to it by the local processor in the course of launching an application.

The PC603 is an embedded processor and has no access to a file system of any type. Thus all standard library functions that require file I/O are not supported. This includes functions such as **printf**() and **sprintf**(). However, the operating system does implement a ramlog facility for passing text messages back to the local processor. The **log_event**() function is provided as an alternative to **printf**() for State Machine applications.

The ChameleonOS operating system does take advantage of some of the virtual memory management features offered by the PC603 processor to keep the process of loading application code simple. The block address translation facilities of the PC603 are used to provide a common address space for application code to run in. This allows application code to be linked to start at address zero with no run-time relocation required of the loader.

6.3.2 OS Initialization

The start of execution of the operating system occurs on the transition of board mode from **Write** to **Config** mode, caused by the assertion of the **strt** bit of the bus priority register in the register interface. At this time the PC603 attempts to fetch instructions and begin executing code. If the operating system has been loaded into the PC603 local memory bank, execution begins, otherwise the processor thrashes about until it takes a machine check exception and halts.

Figure 6-1 shows the system specific tasks performed by the **main**() function. These consist mostly of hardware tests and data structure initialization. The processor begins by checking to insure that it can write/read to/from the registers in the register interface. Since the register interface is the primary means by which the PC603 communicates with the local processor, a failure during these tests causes the processor to attempt to report the failure to the LP and then halts. If the register interface test is passed, the processor attempts to test the PC603 local memory bank to insure its integrity. If the memory test is unsuccessful, the processor reports this fact through the register interface and halts.

Once the hardware tests are complete, the processor proceeds by initializing all of the operating system data structures. The structures used to track the current application process are cleared, the memory semaphore table is cleared, and the semaphore structures initialized. Finally, the ramlog facility is initialized and started. The data structures used by the operating system will be discussed in section 6.3.3. The final task the **main**() function executes is to call the **monitor**() function which, under normal operating conditions, never returns.

Figure 6-2 describes the major operation of the operating system kernel. It is a simple process monitor that continually waits on and then services requests from the LP. Requests for service are made using the **STATEMCH_GP1_REG** register in the register interface and the system management exception vector of the PC603. When idle the monitor sits in a tight loop checking its **sleeping** variable. The monitor stays in this loop as long as **sleeping** is set.

When the LP wants to request a service, it places the service request byte in the **STATEMCH_GP1_REG** register of the register interface. It then causes a system management interrupt to the PC603 to occur by writing a 0x02 to the **STATEMCH_SERVREQ** register in the register interface. On taking the excep-

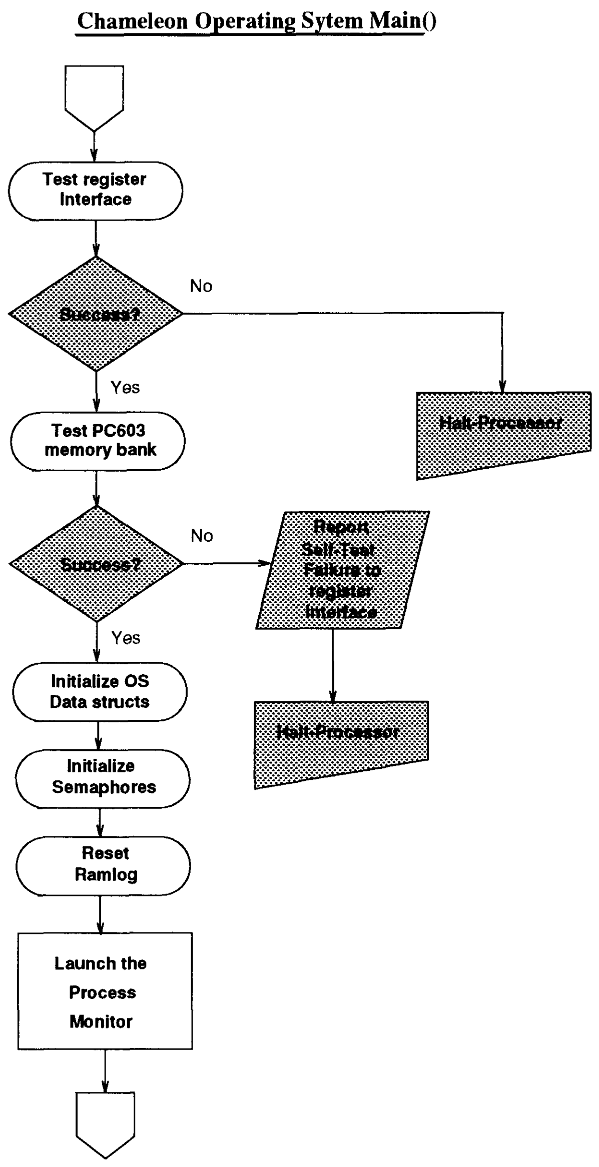


Figure 6-1: Chameleon Operating System Flowchart

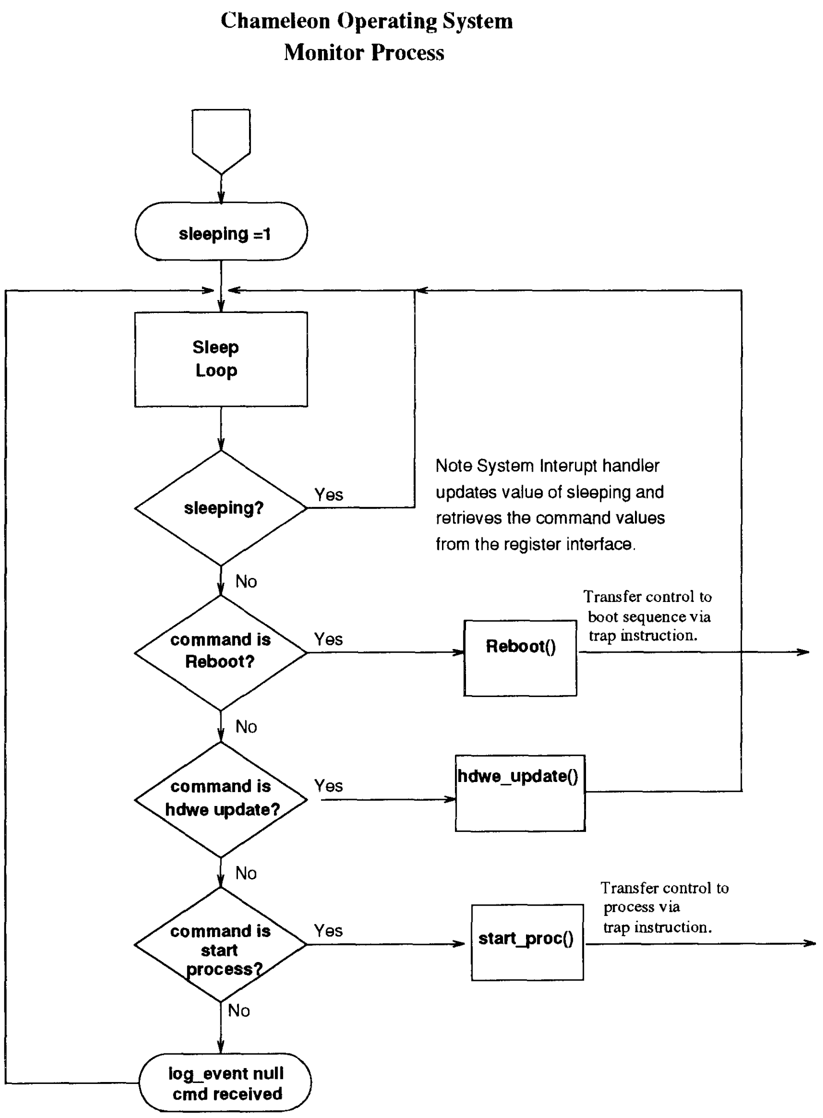

Chameleon Operating System
Monitor Process
sleeping =1
Sleep Loop
sleeping?
Yes
No
Note System Interupt handler updates value of sleeping and retrieves the command values from the register interface.
command is Reboot?
Yes
Reboot()
Transfer control to boot sequence via trap instruction.
No
command is hdwe update?
Yes
hdwe_update()
No
command is start process?
Yes
start_proc()
Transfer control to process via trap instruction.
No
log_event null cmd received

Figure 6-2: Chameleon Operating System Flowchart

tion, the PC603 unsets the **sleeping** variable and reads the **STATEMCH_GP1_REG** register to retrieve the service request. On return from the exception the monitor leaves the sleeping loop and processes the request.

There are three types of service requests that the LP can make. The **Reboot** command causes the PC603 to immediately jump to the start of the bootstrap code and causes all process state to be lost. The processor then continues with the normal bootstrap sequence, reinitializing the operating system environment.

The **start_process** command causes the PC603 to transfer control from the operating system to the application process. The operating system has no knowledge of the application prior to the service request. All memory management and loading is performed by the local processor. When the local processor makes the request, it places all the information that ChameleonOS requires to start the process in a special structure at the end of the application's address space. This process will be described in greater detail in subsequent sections.

The last type of request that the LP can make is the **hdwe_update** service request. This request is used to pass a new set of data parameters through to the FPGA devices. The local processor places the parameter set in the system semaphore space[1] and requests the **hdwe_update** service. On honoring the service request the PC603 takes the parameter set and copies it into the appropriate FPGA device. The first parameter determines which FPGA device the update is for. A parameter set is a generic structure that consists of a number of address and value pairs. The PC603 performs a simple transfer of these parameters from the semaphore area to the FPGA and has no knowledge of the content or purpose of the parameter set.

If the monitor exits the sleeping loop without recognizing the service request an anomaly has occured. This fact is reported with a text message that is logged in the ramlog space. The LP can detect this message and report it to the attached workstation display console. After logging the message, the monitor will return to

[1]System memory semaphores will be discussed in more detail in section 6.3.3

the sleep loop and go back to sleep until the next service is requested. The monitor main loop has no exit condition, and should never return to the **main**() function during normal operation. If it does, something is terribly wrong and the processor will probably take a machine check exception and halt.

6.3.3 Data Structures

The Chameleon operating system uses several data structures to keep track of its internal state, application processes, the ramlog, and shared memory regions. Stack space is kept separate for normal operation and exception processing. The following section describes the important structures used.

internal structures

The *sleeping* variable controls monitor activity and keeps the monitor in a tight loop waiting for LP requests. The system management exception handler unsets the sleeping variable when it processes a LP service request, causing the monitor to exit its sleep loop and initiate the service. The two bytes *command1* and *command2* are filled with the contents of the general purpose registers **STATEMCH_GP1_REG** and **STATEMCH_GP2_REG** at the time of the system management exception. These registers are used by the LP during the service request to indicate what type of service is requested.

The **flood_update** service uses the following data structure to facilitate the transfer of flood parameters to the FPGA devices. Its use enables the operating system to handle the transfer without knowing anything about the order or nature of the parameters transferred.

```
typedef struct {
  unsigned long addr;          /* address of parameter */
  long          val;           /* flood parameter */
}flood_list_el;
```

Tables

The operating system uses several tables to implement the virtual memory model and keep track of user process code slots. ChameleonOS uses the block address translation facilities of the PC603 to protect certain regions of the address space from caching, and to simplify the linking and loading process of application code. The tables used are as follows:

- *unsigned long os_ibattbl[]* - Contains the configuration words for the upper and lower BAT registers for the Operating system. The data from this table is placed in the instruction BAT registers at boot time, and whenever control is returned to the operating system from application code.
- *unsigned long proc_ibattbl[]* - Plays an analogous role for application code to the os_ibattbl[] for the operating system. This table is used to fill the BAT registers of the processor immediately before control is transferred to the application code.
- *unsigned long dbattbl[]* - Specifies the block address translation configuration information for data accesses. Since data accesses for both the application code and the operating system are handled in a like manner, separate tables for applications and the operating system are not required.
- *unsigned long bat_addr_tbl[]* - This table holds the physical addresses of the start locations of each of the application process code slots. When an application is

started the address in this table, corresponding to the code slot of the application, is loaded into the BPRN field of the lower block address translation register zero. The virtual memory manager of the PC603 subsequently maps logical address zero to this address during application code.

Process handling structures

Application code is loaded into the PC603 memory bank by the LP according to a predetermined convention that will be discussed shortly. Each application has an associated process parameter structure that is used to communicate the process parameters. This structure is located at the bottom of the process's last code slot.

```
typedef struct {
    unsigned long   configcode;   /* FPGA configuration command */
    unsigned long   stackbase;    /* ptr. to process stack base */
    unsigned long   TOCptr;       /* ptr. to process TOC */
    parameter_st   procparms;   /* process parameters structure */
} proc_rec;
```

The *configcode* word determines if and what FPGA configuration action is to be taken. The code may specify that the processor is to reconfigure the FPGAs from one of its two configuration data memory areas, or to perform no FPGA configuration before starting the process. The *stackbase* and *TOCptr* fields are pointers to the stack base of the stack and the program table of contents start. These pointers are relative to the start of the 128k application address space. The *procparms* element provides the actual arguments to the application function. The supporting data structures are provided below.

```
#define MAX_PARAMETERS  6
typedef union {
  float           f;   /* single precision floating point  */
  unsigned long   ul;  /* 32 bit integers, signed and unsigned */
  signed long     l;
  unsigned short  us;  /* 16 bit integers, signed and unsigned */
  signed short    s;
  unsigned char   uc;  /* 8 bit integers, signed and unsigned  */
  unsigned char   b;
  signed char     c;
  smem_id         m;   /*  shared memory reference id  */
} parameter_t;
typedef struct {
  unsigned long   num_parameters;      /* number of parameters   */
  parameter_t     p[ MAX_PARAMETERS ]; /* array of parameters    */
  unsigned long   shared_mem;          /* bit field indic.  shared mem param. */
} parameter_st;
```

The operating system uses six variables to keep track of the application code to be executed. These six variables are:

- *void *proc_code;* - A pointer to the start of the executable code.
- *unsigned long *proc_stack;* - A pointer to the stack space base for the application.
- *proc_rec *proc_parms;* - A pointer to the process parameter structure for the application process.
- *unsigned long proc_TOC;* - the Table of Contents pointer value for the process.
- *unsigned long proc_index;* - The process index number of the application. This index is the same as that used to reference the bat_addr_tbl to determine the

physical start address of the application.

- *int proc_state;* - a variable used to record the state of the current application process.

Ramlog structures

The ramlog facility is used to communicate status and application program execution information to the LP. It allows text and data to be logged into memory in a compressed mode, for later interpretation and reporting. The ramlog area is essentially a circular queue and uses a simple data structure to manage the queue. This header structure sits at a fixed location in shared memory so that both the LP and the PC603 can access it. The **log_event**() function is used by both the operating system and applications in the same way that the C **printf**() function would be used on a workstation. The difference is that the **printf**() would write the message to a file, while the **log_event**() function writes it to the ramlog space. The LP ramlog monitor can then pick up the message to use internally, or display to the console window on the attached workstation.

Semaphore structures

A single variable is used to manage the semaphore memory. The semaphore memory is a dedicated 2KB section of memory divided into 16 128 byte chunks that are dynamically allocated as needed. These chunks can be used to pass information back and forth between the LP and the PC603 on an ad hoc basis. The *unsigned long sema_stat* variable is used to keep track of which semaphores are available or in use. The semaphore usage is tracked by the value of its corresponding bit in the low order 16-bits of the *sema_stat* word.

6.3.4 Memory Usage

Figure 6-3 describes the memory organization of the PC603 local memory bank. The memory is divided into 8 128K bytes regions. The divisions are chosen to correspond to the smallest BAT register block size mask supported by the PC603[27]. The first 128K block is dedicated to operating system usage. The next six are reserved for application code,which may begin only at the beginning of any of the six blocks. The last block is used to store FPGA configuration for both FPGAs for up to two different configurations. Alternatively, fewer application code slots can be reserved to make room for more FPGA configurations should this need arise. This allocation is controlled by and can be dynamically changed by the LP.

The operating system block, block 0, is divided into five sections as shown by the dotted lines in figure 6-3. The processor exception table occupies the first 12 K-byte of space and is immediately followed by the operating system code. The ChameleonOS kernel takes up less than 20 K-byte of space, however, 50 K-byte is reserved for future expansion. The operating system is immediately followed by the system ramlog space. This space includes the 24 byte header and a 66,536 byte circular queue area. The final 2 K-byte section is the semaphore table that contains space for dynamically allocated memory chunks as discussed in section 6.3.3.

Figure 6-4 describes the memory usage of an application code slot. The code begins at the beginning of the slot at logical address 0x0000. The last 44 bytes of the slot are dedicated to the **proc_rec** parameters structure used to communicate essential information for starting the process. The process stack space begins immediately before this structure and grows from higher address to lower address. The boundary between code space and stack space is arbitrary and determined by each individual application. Note that application processes are not limited to a single code slot. Larger applications may take two or more slots as required. The only overriding restrictions are that the process parameter structure and stack space be at the end of the last code slot used, and the code must start at the beginning of one of

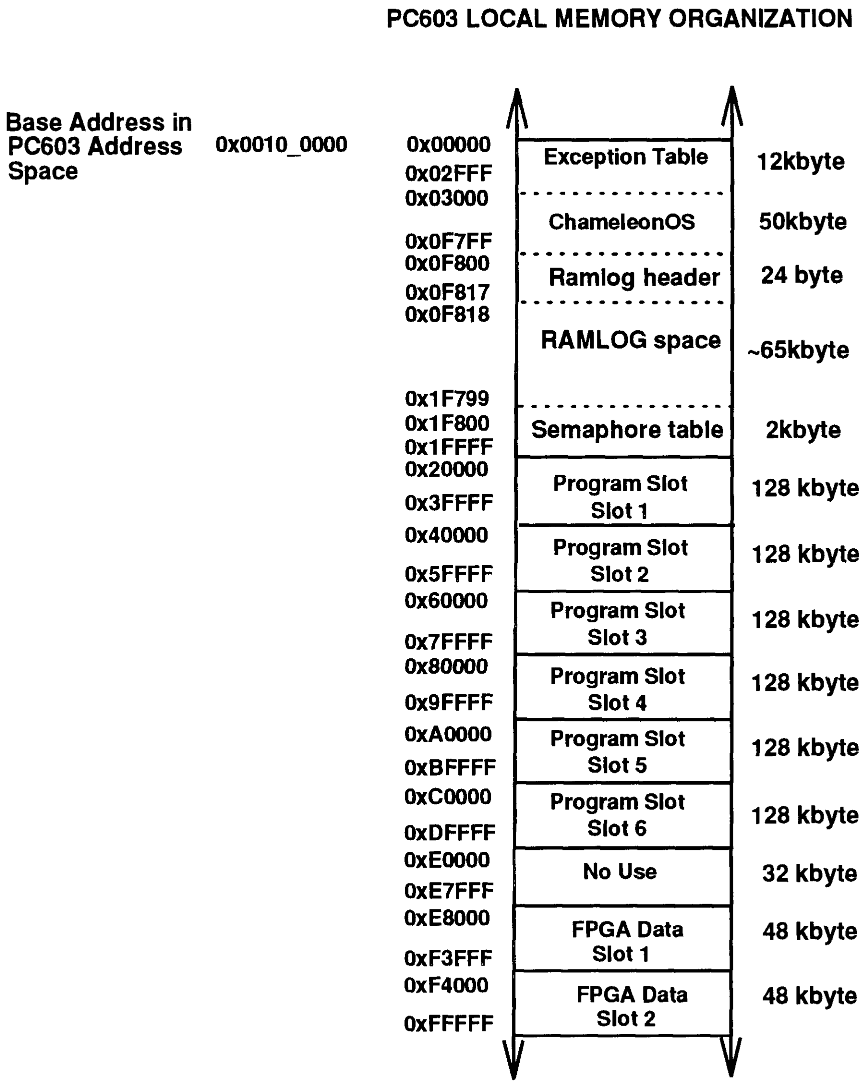


Figure 6-3: PC603 Local Memory Usage

the first code slot used.

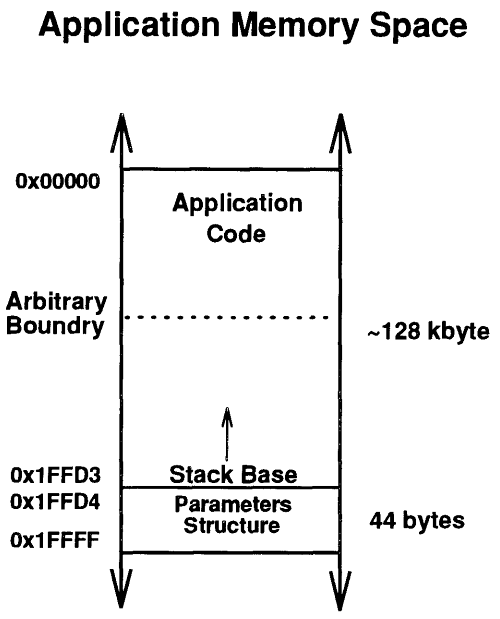


Figure 6-4: PC603 Local Memory Usage

6.3.5 Process Switching

Process switching occurs when the operating system transfers control to an application code segment, or when an application code segment exits and control is returned to the operating system. Both transfers involve a change of address space which is facilitated by use of the system call instruction (**sc**) of the PC603. This instruction generates a 603 exception and allows the transfer to occur in a different machine context. Much of the low level process switching code is implemented in assembly language. Figure 6-5 is a pictorial representation of the process switching operation. The following is a description of this process.

Transfer of control to applications

If a system service has been requested by the LP, the monitor executes the **start_process**() routine handing it the process index number and size contained in the **command1**

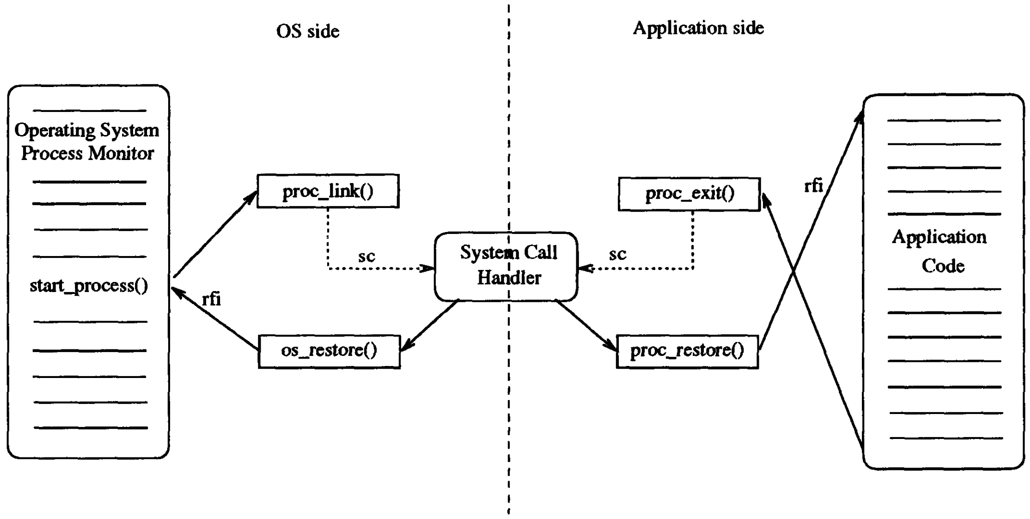


Figure 6-5: ChameleonOS Process Switching

byte. **start_process**() is the monitor entry and exit point for launching applications. **start_process**() then calls **proc_start**() which updates the process data structures before calling **proc_link**(). **proc_start**() is the C level exit and entry point for the operating system. **proc_link**() is an assembly language stub used to generate the **sc** instruction, causing a system call exception to be taken.

After taking a system call exception, the PC603 exception handler analyzes the byte in the GPR0[2] register. Upon sensing a process index and size in this byte, the handler jumps to the **proc_restore**() routine. The **proc_restore**() function is a low level assembly routine that saves the operating system processor state and initializes the processor to begin the application code segment. It obtains the process data from the process data structures. The **proc_restore**() function ends in an **rfi** instruction which causes the PC603 to jump to the start of the application code by way of an exception return.[3] The **proc_restore**() function initializes the application stack frame in such a way that when the application finishes, it automatically returns

[2]This notation refers to the general purpose register set, GPR0-31, of the PC603 processor. This notation will be used throughout this work.

[3]See [27], pages 6-1 thru 6-15 for detailed information on PC603 exceptions and exception return information.

to the **proc_exit**() routine.

Transfer of control back to the operating system

Transfer of control back to the operating system from the application code segment occurs automatically, *when the application code executes a return.* The **proc_exit**() routine is called automatically, having been placed in the return linkage by the **proc_restore**() routine. **proc_exit**() plays an analogous role for the operating system that **proc_link**() link plays for application code, it is a stub that generates a system call exception. The difference is that **proc_exit**() loads register GPR0 with the GETOS command byte so that the system call interrupt handler calls the **os_restore** routine. The **os_restore** () routine restores the operating system state and returns to the end of the **proc_start**() routine, the C level entry and exit point for launching application code. **proc_start**() returns to **start_process**() which immediately calls **proc_halt**(). **proc_halt**() removes the application information from the process management data structures and then returns. **start_process**() then returns to the main monitor loop and returns to the sleep state, until the next service request.

6.3.6 Local Processor Communications

There are several different ways in which the LP and the PC603 on the State Machine board communicate. The most basic method, and the default method in case of catastrophic failure, is through the general purpose registers in the register interface. These registers are used for several different types of communication. At boot time the PC603 periodically writes a status byte to **STATEMCH_GP1_REG** to inform the LP of its progress. After the boot sequence is complete, **STATEMCH_GP1_REG** is used by the LP to indicate the type of service request. The 603 writes return information to both **STATEMCH_GP1_REG** and **STATEMCH_GP2_REG** to handshake with the LP.

The semaphore space in the operating system memory segment can be used to pass information back and forth between the PC603 and the LP. The **hdwe_update** service uses this method to pass a list of parameters to the PC603 to update the FPGA devices with. It is anticipated that additional required services will pass information in this manner. Finally, application process parameters are passed with the **proc_rec** structure by the LP to the PC603 in their own dedicated space at the end of the process memory slot.

6.3.7 FPGA Configuration

The operating system configures the FPGAs at the direction of the LP in accord with the requirements of the desired application. The LP communicates the FPGA configuration requirements to the operating system through the **configcode** field of the **proc_rec** record associated with each application code segment in memory. The operating system may then configure the FPGAs from either of two configuration data slots reserved in the last 96 K-byte of the PC603 local memory bank. A special routine, **flexconfig**, is provided to perform the configuration operation. This occurs as one of the operations carried out by the **proc_start** routine as it initializes the other process structures for application execution.

6.4 Application Code

State Machine application code can include a wide range of functionality. It can perform the majority of the computation for the designated stream processor, or it can do almost nothing, letting the computation take place entirely in the FPGAs. Many applications will fall somewhere in the middle of these two extremes, carrying out the desired computation partly in hardware with the FPGAs, and partly in software, on the 603. The State Machine allows the boundary between hardware and software solution to be arbitrarily set. This choice is naturally influenced by the available hardware resources, the computational demands of the application, and the developer's knowledge of hardware/software development.

The application code for the PC603 is developed in the same manner as any other C program with a few important exceptions. These exceptions include both coding conventions and linking and loading conventions. We address each point here in detail.

6.4.1 The main() function

State Machine applications do not have a **main**() function like ordinary C programs. Instead, State Machine applications use the name of the application as the name of the top level function. That is, the name of the application serves as the entry point into the code. This convention was chosen to reinforce the model of the State Machine as a resource for arbitrary function execution. A constraint of this convention is that the entry point for the code must be explicitly provided to the linker. Command line options of the **ld** command are used in the application make file to accomplish this purpose.

6.4.2 Malloc and Printf

The PC603 does not have storage peripherals of any kind available to it, including a file system. The only I/O capability it has is to SRAM memory and hardware registers in the register interface and FPGA devices. Consequently, none of the standard C library functions that implement file or buffered I/O are supported for State Machine applications. These include all file handling functions as well as **printf**() and **sprintf**().

In addition, memory allocation functions such as **malloc**() and **calloc**() are not supported. As has been discussed before, to keep the operating system as simple as possible, no memory management of the PC603 memory bank is provided. This memory management, of the application code slots and FPGA configuration slots, is handled by the LP. Thus application code cannot call these functions. This does not pose difficulties for most applications since most memory requirements can be predetermined and the storage declared at compile time. Applications for which this is not true can rely on the Cheops resource manager to allocate the required memory.

6.4.3 FPGA Communication and Interrupt Handlers

State Machine applications for which the PC603 plays more than a trivial role typically require some form of communication between the PC603 and the FPGA devices. There are several methods by which this communication may take place. The important thing to remember is that neither part of the application is developed without knowledge of the other and implicit communication can and should occur.

memory based methods

The PC603 can communicate with the FPGA devices through special designated memory locations in the FPGA memory banks that both know about. That is a special location in the FPGA memory bank can be reserved, that the PC603 can read from and the FPGA can write to, or vise-versa. This method can be used so that both the FPGA and PC603 can communicate their progress in completing a computation.

Alternatively the PC603 can read and write the FPGA device itself, as if it were a memory element. This, of course, only works as long as the FPGA design files support this activity. There is usually enough prior knowledge about each part of the application such that this is known. Thus the PC603 can poll either location to communicate with the FPGAs.

However, this form of communication only works if the FPGA devices are not locking their busses with their respective buslock signals. When this occurs the PC603 cannot access the FPGA busses and this type of communication cannot occur.

The external interrupt

Another method is provided which allows limited signaling between the FPGA and the PC603. Each FPGA device has the capability to interrupt the PC603 using the PC603's external interrupt vector. The FPGAs interrupt signals, **bos_int** and **jon_int**, are logically ORed in the register interface with the resulting output signal wired to the PC603's **/INT** input. The PC603 upon taking an external interrupt can read from a location in the register interface to determine which FPGA caused the interrupt. Two of the PC603's high order address lines act as interrupt acknowledge signals. Thus when a FPGA generates an interrupt it holds its interrupt signal asserted until it receives an interrupt acknowledge signal from the PC603.

Note that the FPGA does not have to generate an interrupt to receive an interrupt acknowledge. The PC603 can assert the interrupt acknowledge signal to signal the FPGA of certain events or synchronize operations. For this type of signaling it is assumed that there is implicit knowledge between the FPGA and the PC603 of the others activities. That is, the meaning of the interrupt has been previously agreed upon.

To use this form of communication productively, State Machine applications must have a method of installing their own interrupt handlers for the external interrupt vector. The installed handler may then carry out the required operations of the application when the external interrupt is signaled by a FPGA. The operating system provides a special mechanism for registering interrupt handlers. The following function is used for this purpose:

void *register_handler(* **void** *(*p_funct)(***void***));*

The **register_handler** function installs the handler as the function to execute when an external interrupt is taken. Application interrupt handler functions should take no arguments and return nothing; they should only operate on global variables or local variables declared on the stack.

6.4.4 Linking Applications

It was stated earlier that State Machine applications do not have a **main**() function at the top level, using the name of the application instead. This convention has the advantage of allowing the use of all the RS6000 PowerPC code development tools for State Machine application development without suffering the injury of having the IBM AIX program linkage code automatically linked into the program. The disadvantage is that the linker is not able to determine the entry point into the code unless it is explicitly stated at link time. Thus State Machine applications must explicitly state their entry point at link time.

This task can be accomplished by using the **-e** option of the **ld** command in the application Makefile. At link time, the name of the application, which should also be the name of the top level function, is used with the **-e** flag to explicitly state the entry point into the code. As an example, for the application **matrix_mul**, the top-level function name is **matrix_mul**() and the linker command in the make file is:

```
ld -omatrix_mul -D-1 -ematrix_mul -H8 -S4096 -T0x00000000 \
   -L${STATEMACH_USER_LIBS} -L${STMACH_C_LIB} ${APPFILES}
```

There are two other important **ld** options present in this command line that are essential for linking State Machine applications. The first is the **-T** option, which tells the linker to link the program as if it were to start at address 0x0000. This is extremely important since the Chameleon operating system uses the block address translation virtual memory features of the PC603 to execute all application code as starting at logical address zero. The memory management facilities of the PC603 maps the logical addresses to the correct physical address of the application code slot where the program resides in the PC603 memory bank. The **-T** option is used with the linker to insure that all address references in the program will be correct and relative to a start logical address of 0x0000.

The other is the **-D-1** option, which tells the linker to place the .data section of the program immediately following the .text section. This option is used to economize on memory usage and simplify the memory management task. Other options used in the command line include:

- **-o** - Tells the linker what to name the resulting executable output file.
- **-H** - Tells the linker what alignment to use for the start of program sections.
- **-S** - Tells the linker the maximum size the program stack is allowed to grow.
- **-L** - Specifies the directories to use to look for libraries that are to be linked with the application.

6.5 FPGA Configuration Files

Each application has an associated FPGA configuration file. This file is actually a composite consisting of the contents of two Altera ***.ttf** files with their contents interleaved. The contents of the configuration file is loaded into the State Machine at the same time the configuration code is. However, the mapping between application code and FPGA configuration files is not isomorphic, and many applications may use the same FPGA configuration file. Consequently, it is possible for the applications configuration file to be already resident in the State Machine at the time the application code is loaded.

The process of generating the FPGA configuration file is described by figure 6-6. The application's hardware architecture is first designed and described in the AHDL hardware description language using the Altera MaxPlusII development environment. A ***.tdf** file is generated for each of the two FPGAs, Boswell and Johnson. The AHDL files are then compiled using the MaxPlusII design compiler which performs logic database generation, logic synthesis, netlist extraction, device mapping, routing, and finally configuration file generation. An option of the compiler generates the ***.ttf** form of the configuration files which consists of an ASCII file containing the comma separated list of configuration bytes. Thus the output of the MaxPlusII development tools is a pair, one for Boswell and Johnson, of ***.ttf** files.

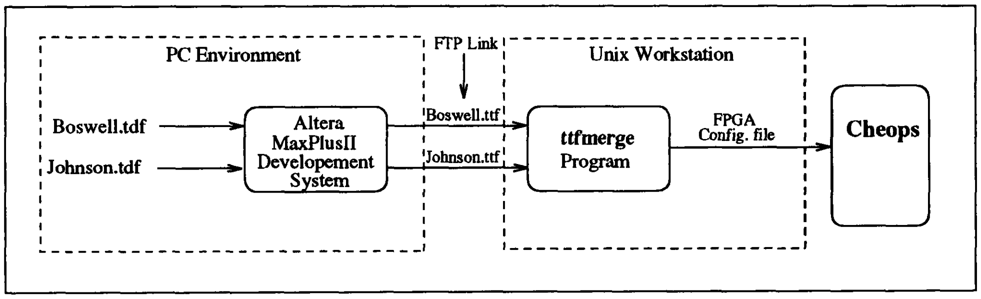


Figure 6-6: FPGA Configuration File Generation

These files are then transferred into their appropriate position in the application directory tree on the Cheops host Unix workstation. Finally, the program **ttfmerge** is run, taking the two ***.ttf** files as input and generating the FPGA configuration file associated with the application. It is this file, in binary format, that is loaded into the State Machine along with the application code.

6.6 Resource Manager Integration

Currently, only a low-level interface library for communicating with the State Machine exists. These low level functions are used by the diagnostic and test code and are unsuitable for general application use. However this is merely a short term condition until support for the State Machine is built into the Cheops resource manager.

Eventually, the Cheops resource manager, NORMAN, will handle these tasks. At this point, the low-level interface library will be used only by RMAN functions so that the use of State Machine resources becomes transparent to the end user. Cheops applications will simply refer to "virtual functions" without regard to which State Machine they run on, or whether or not the function is presently loaded in a State Machine. The resource manager is responsible for maintaining and modifying the state of all State Machine stream processors so that these details need not be addressed by user applications.

This allows applications to refer to State Machine functions in the same manner as other stream processors. Users merely include references to functions implemented on the State Machine into their data-flow algorithm description using the RMAN library functions[38]. Upon initial parsing of the algorithm description, the resource manager will load instructions and configuration data (from disk) into local memory, if it is not already present. At run time, the resource manager configures the State Machine with the user requested function using a special **start_proc**() routine and **hdwe_config** messages.

Thus all details of State Machine usage are hidden from user applications. We do not attempt a detailed explanation of the Cheops resource manager in this work; as it is beyond the scope of this thesis. However, the curious reader is referred to [38].

Chapter 7

Research Results

We have designed and implemented the entire State Machine stream processor including the hardware described in chapter 5 and the software described in chapter 6. Unfortunately, the State Machine stream processor is only partly functional at this time. Consequently, many of the proposed analysis are not described here. These will be reported in a later work, after the State Machine is fully functional. The results described in this chapter include the difficulties encountered in the construction of the State Machine stream processor, a report on the achievable FPGA resource utilization factors that determine the maximum size of State Machine hardware architectures, and a refining of the reconfiguration timing model proposed in chapter 4.

7.1 Implementation Difficulties

Two major factors contributed to the delay in the design and implementation of the State Machine stream processor: fabrication and assembly problems due to the extremely high density of the State Machine board, and delays in getting information on and samples of the IBM PowerPC603 microprocessor. The State Machine design packs an extraordinary amount of ICs in a very small PCB area. The final design includes 33 ICs in a 5.25"x4.3" double sided area, including 3 large 240-pin QFPs. The

PCBs are 12 layer boards and have approximately 3000 holes, 1900 SMT pads, and a board density factor of .102. The routing and layout of the State Machine board took several attempts and two separate layout design houses before an acceptable board fabrication was achieved. The low board density factor also complicated the testing process and contributed to increases in testing and debugging time.

The other source of delay in implementation was the availability of the IBM PowerPC603 microprocessor. Although not generally available yet, we were able to obtain several samples, probably from one of the first foundry runs. This version of the PC603 had bugs and was not reliable. As a result we were forced to abandon our initial samples and obtain new ones from a later run. In addition detailed technical information for the PC603 was not made available until 9/94.

7.2 FPGA Resource Utilization

In the course of designing the State Machine stream processors, two sample applications were created to assist with the design process. A simple Huffman decoder designed to operate at 40 MHz using both FPGAs was implemented and synthesized. In addition, a simple flood controller was designed and implemented. The flood controller design transfers data from the flood ports to the FPGA memory banks so that it can be operated on by application code running on the PC603 processor. The flood controller also returns the data to the Hub via the output flood ports.[1]

The purpose of this effort was to gain insight into the optimal pin-out assignment for the various bus interfaces and control signals that each FPGA had connectivity to. The optimal pin-out is the pin-out assignment that provided maximum flexibility in compiling and routing different designs in the Altera EPF81188 devices. In this work, we discovered that fixing the pin-out of these devices during design compilation and

[1]The flood controller is generic and is used by all State Machine applications that intend for all processing to be handled by the PC603.

routing produced major device routing problems that adversely affected the device resource utilization levels achievable.

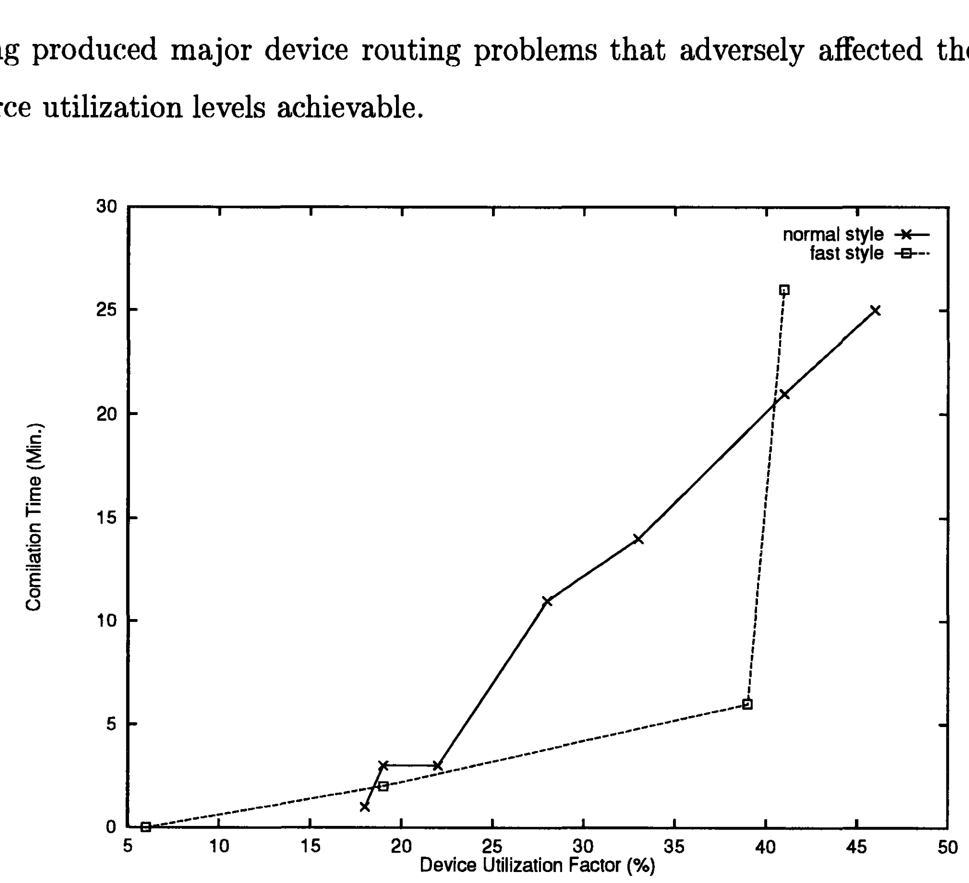


Figure 7-1: Compilation Time vs. Device Utilization Ratio

The results of this work are presented in figure 7.2. This graph plots design compilation time vs. device utilization as reported in the Altera compilation report files. All compilations were performed using Altera MaxPlusII design compiler v4.02. These tests were performed for a fixed pin-out design of the Huffman decoder and the flood controller. The two separate plots represent two different logic synthesis styles. The *Normal* style causes the compiler to optimize the design for area, packing the synthesized logic as closely together as possible. The *Fast* style causes the compiler to optimize for speed, minimizing interconnect and logic delays, which generally requires that logic resources be used less efficiently.

The test points show that the *Fast* logic style gives better compilation performance at low densities, while the *Normal* logic synthesis style provides better compilation performance at higher densities. The compilation time for the *Fast* logic style increases at an exponential rate with increased device utilization, and at an apparent linear rate for the *Normal* logic synthesis style. Moreover, for both the *Fast* and *Normal* logic synthesis styles, the design compilation time is dominated by the routing time. The most disturbing finding is that we were unable to successfully compile either of the two designs for devices utilization factors above 46%. This suggests that the compilation time for the *Normal* logic synthesis style is also probably increasing at an exponential rate.

These results can be attributed to the Altera FLEX architecture. The routing resources of these FPGAs are optimized for speed and employ long lines in the routing channels, called FastTrack interconnect, that run the full length of the chip. The output of each logic element (LE) connects to a single FastTrack channel in the adjacent row and column routing channels. However, the I/O pins of the FPGA can only be connected to a small subset of the FastTrack channels. The result is that each I/O pin can only be connected to a very small number of the chips LEs. This fact severely restricts the ability of the compiler to fit and route a design with a fixed pin assignment. In addition, our results provide no indication as to what extent the compiler's ability to route the FPGAs contributed to the low device utilization ratios.

The conclusions that can be drawn from these experiments are that the Altera FLEX architecture is ill-suited for in circuit re-programmability, and by extension, dynamic reconfiguration, with fixed pin assignments. Designs are limited to less than 50% [2] device utilization by routing constraints and compiler performance. Because of these constraints, design compilation time increases exponentially with device utilization. [3] This implies that State Machine applications are limited to less than 6,000 usable gates, including communications protocol logic, per FPGA device. Thus for

[2]This figure includes the logic required to implement the bus communication protocols. So application designs are limited to less than 32% of the chips logic resources.

[3]Admittedly, this must be a qualitative argument, as not enough data points were collected to rigorously defend this analysis. Although, other researchers have reported similar results.

designs that use the entire State Machine as a single stream processor, the hardware architecture is limited to 12,000 usable gates. Designs that use the State Machine as two independent stream processors, have each of the hardware architectures limited to 6,000 usable gates.

7.3 Timing Analysis

In chapter four a timing analysis model was presented for determining the reconfiguration penalty required for switching State Machine applications. This model is further clarified here in accordance with the completed design of the State Machine stream processor. In particular, several variables are replaced with their nominal values, and several constants are defined as a result of the FPGA technology and other system parameters.

The choice of the Altera FLEX EPF81188 FPGAs determines several of the key timing parameters. These devices require 24 K-byte of configuration data when programmed in the passive parallel synchronous mode; the mode used in this work. Thus the size of the configuration data for both the Boswell and Johnson FPGAs is $\eta = 49,152$. The maximum allowable configuration clock rate for these devices is 6 MHz. However, the State Machine uses a 4 MHz multiple of the **EHUBCLK** signal for the configuration clock, thus $f_{dclk} = 4$. For the EPF81188, a new configuration byte is required every 8 **DCLK** cycles, thus $C_{BW} = \frac{1}{8}$ byte/cyc. Finally, the EPF81188 requires 10 global clock cycles to initialize its internal logic before use. The FPGAs run synchronous to the 32 MHz **BEHUBCLK** clock signal, so $T_I = 312$ ns.

The other timing parameters are determined by the State Machine clock frequencies, the LP communications protocol, and the choice of control processor. The nominal system clock rate is the same as the P2 clock rate, so $f = 28$ MHz. Figure 5-2 shows the nominal LP to State Machine communications protocol requires six **Pclk** cycles to transfer a byte, so $\lambda = 6$. The control processor, PC603, is a super-scalar pipelined processor that can launch or retire as many as three instructions per clock cycle. In addition a normal single beat transfer of up to 4 bytes requires 4 **Pclk** cycles. To estimate T_{prms} it is assumed that a constant six words of memory are moved from one location to another. It is further estimated that each transfer requires 12 **Pclk** cycles so that $T_{prms} = 2.6\mu$s.

Summarizing these parameters, we have

$$\begin{array}{lcllcl} \lambda & = & 6 & f & = & 28 \text{ MHz} \\ \eta & = & 49{,}152 \text{ (48K)} & T_{prms} & = & 2.6\ \mu\text{s} \\ T_I & = & 312 \text{ ns} & K & = & 3.4\ \mu\text{s} \\ f_{dlck} & = & 4 \text{ Mhz} & C_{BW} & = & \frac{1}{8} \text{ bytes/cyc.} \end{array}$$

With these values defined we can refine the equations for the components of the delay given in equations 4.2, 4.3, and 4.4. Incorporating this new information, these reduce to:

$$\tau_d = \rho \cdot 216ns + 10.6ms \tag{7.1}$$

$$\tau_i = \beta \cdot 54ns + 500ns \tag{7.2}$$

$$\tau_c \approx 49.2ms \tag{7.3}$$

As in chapter 4 we are interested only in the case where the applications have been loaded ahead of time so that the load time τ_d is ignored. In addition we note for these parameters that,

$$f_{dclk} \ll f. \tag{7.4}$$

Thus for all reasonable β ($\beta \leq 16K$), if FPGA configuration must be performed, the configuration time is dominated by the time to reconfigure the FPGA devices and may be approximated by

$$\Upsilon \approx 50ms. \tag{7.5}$$

Conversely, if the FPGAs are already correctly configured, then the reconfiguration time is dominated by τ_i and is exclusively a function of β,

$$\Upsilon(\beta) = \beta \cdot 54ns + 812ns. \tag{7.6}$$

For any reasonable sized β this is on the order of 100's of μs. Thus the reconfiguration penalty for the State Machine stream processor is approximately 50ms for applications requiring FPGA reconfiguration. For applications that require only code initialization, the reconfiguration penalty is a function of β and is on the order of μs.

Chapter 8

Suggestions for Further Research and Future Directions

The research described in this thesis represents a "first cut" at using custom computing within the Cheops Imaging System to improve performance. The results have been inconclusive to date but show promise for considerable performance gains within Cheops. Several areas that are deserving of further research can be immediately suggested. These will serve to realize the full potential of the State Machine. More generally, the State Machine demonstrates an affective use of dynamic hardware configuration and provides further evidence that custom computing machines can be used affectively for certain computational problems. To this end, directions for future research are suggested.

8.1 Further Research for the State Machine

A principal result of this work suggests that dynamic reconfiguration within Cheops is possible. Much work remains to be done to make it a reality. The author suggests continued development of the current State Machine design to bring it to full functionality. The detailed performance comparisons suggested at the beginning of this work can then be completed and conclusions drawn. In addition, the research performed in this thesis suggests additional improvements that could be made to the State Machine design to improve performance and facilitate usage.

8.1.1 Adding Newer Xilinx FPGAs

As discussed earlier a severe limitation for the State Machine is the low logic resource utilization factors achievable within the FPGAs due to the fact that their pin assignments are fixed by the external system architecture. This is a limitation of the Altera FPGA devices employed that trade rout-ability for improved signal path delay and signal delay predictability. This limitation restricts the size of user applications that can be implemented with custom hardware.

To overcome this limitation a State Machine stream processor could be modified to use Xilinx XC4000 family FPGAs. Xilinx FPGAs provide much improved rout-ability in fixed pin assignment designs over Altera FPGAs. This improvement is due to an increased number of physical routing resources on the chip and better synthesis tools that allow the user greater control over the logic synthesis, placement, and routing processes. In addition, as of this writing, Xilinx has introduced a new family of FPGAs, the XC5000 family, that are specifically designed to overcome the constraints of fixed pin assignment designs. These devices have the potential to offer resource utilization ratios approaching those for designs with unconstrained pin assignments (90-100%).

8.1.2 Hardware/Software Compiler

While the State Machine provides an efficient computing platform for implementing computations in custom hardware, the process of developing these applications is quite complex. This process involves both software programming and hardware design using AHDL (Altera's variant of the more popular VHDL). It is desirable to simplify this process so that users are not discouraged from using the State Machine in their

Cheops programs.

To automate this process a hardware/software compiler is needed to compile State Machine applications directly from *C* or other high level programming languages. Such a compiler would allow programmers to develop applications for the State Machine without detailed knowledge of digital hardware development and without changing their current programming environment. This compiler would make all decisions about the division of labor between the FPGAs and the PC603, handle all synchronization and communications issues between the FPGAs and the PC603, compile PC603 code, and synthesize hardware architectures for the FPGAs. It would essentially make the use of the State Machine stream processor transparent to the end-user.

The author strongly believes this is an essential step that must be taken in order for this class of machines to gain wide-spread acceptance as an architectural paradigm.

8.1.3 Resource Manager Support for Dynamic Reconfiguration

Towards this same goal, it is desirable to have resource manager support within NORMAN to facilitate transparent usage of the State Machine stream processor. These facilities would allow the programmer to simply use function calls from a generic library without knowledge of how the function is actually implemented or executed. The resource manager would handle all low level operations required to configure and initialize the State Machine for the desired operation, carry out the operation, and return the results. It is not difficult to imagine a Cheops bestowed with only State Machine stream processors that becomes essentially unconstrained in its ability to implement user computations in custom hardware. Resource manager support for this functionality needs to be developed.

While this scenario is wonderful in theory, practical limitations of current FPGAs restrict its viability. However, these limitations are merely short term in nature, as discussed in chapter 2. Within the next generation of FPGAs, these limitations will begin to disappear. Alternatively, a custom reconfigurable device could be designed and fabricated that meets the run-time processing and throughput requirements of Cheops.

8.2 Future Directions for Research

Although considerable progress has been made in developing custom computing machines in this work and by other research groups, it is merely the foundation for a new class of machines whose potential does not suffer from the physical and economic constraints of general purpose machines. As fabrication costs and design cycle times increase exponentially to extract the next incremental improvement in performance from general purpose architectures, interest in alternative methods of implementing computing machinery will rise. Custom computing machines will satisfy that interest and will be the next evolutionary step for computing machinery.

In order for this prediction to become a reality many outstanding issues still must be addressed. This will require further research to find the best solution to these issues. The author can suggest several areas that are deserving of further intensive study.

8.2.1 Improving Configuration Data Bandwidth

Currently the primary limitation of custom computing machines in general purpose computing environments is the lack of reconfiguration data bandwidth. Clearly it is desirable to minimize the amount of time required to configure and initialize the devices with a specific architecture such that the total time for the computation, configuration time plus computation time, is significantly less than that of general purpose processors. The work in this thesis has begun to address this issue directly. If custom computing machines are ever to be used as the basis for general purpose computing platforms these reconfiguration times need to drop to the order of a process context switch, or approximately 5-10 μs.

There are no inherent reasons, other than lack of popularity and demand, why shorter configuration times cannot be achieved. Indeed, since the beginning of this project, Xilinx has introduced a new family of devices that reduces the reconfiguration time from on the order of 100ms to approximately *10ms.*[1] This is an order of magnitude improvement in a very short time period (1-2 yrs.). Further research in this area will be required to reduce these times even further.

8.2.2 Continued Work on Custom Computing in General Purpose Machines

So far most research and commercial development has focused on simply providing the hardware platform for custom computing. With many of these issues solved, interest and resources are shifting towards research designed to incorporate these resources in general purpose computing platforms. This is a very exciting direction and suggests that computers may implement a custom machine architecture for each application in the very near future. Much research is still required in order to complete the progression of custom computing machines from their current state to a mainstream general purpose computing platform. There are many exciting opportunities in this area.

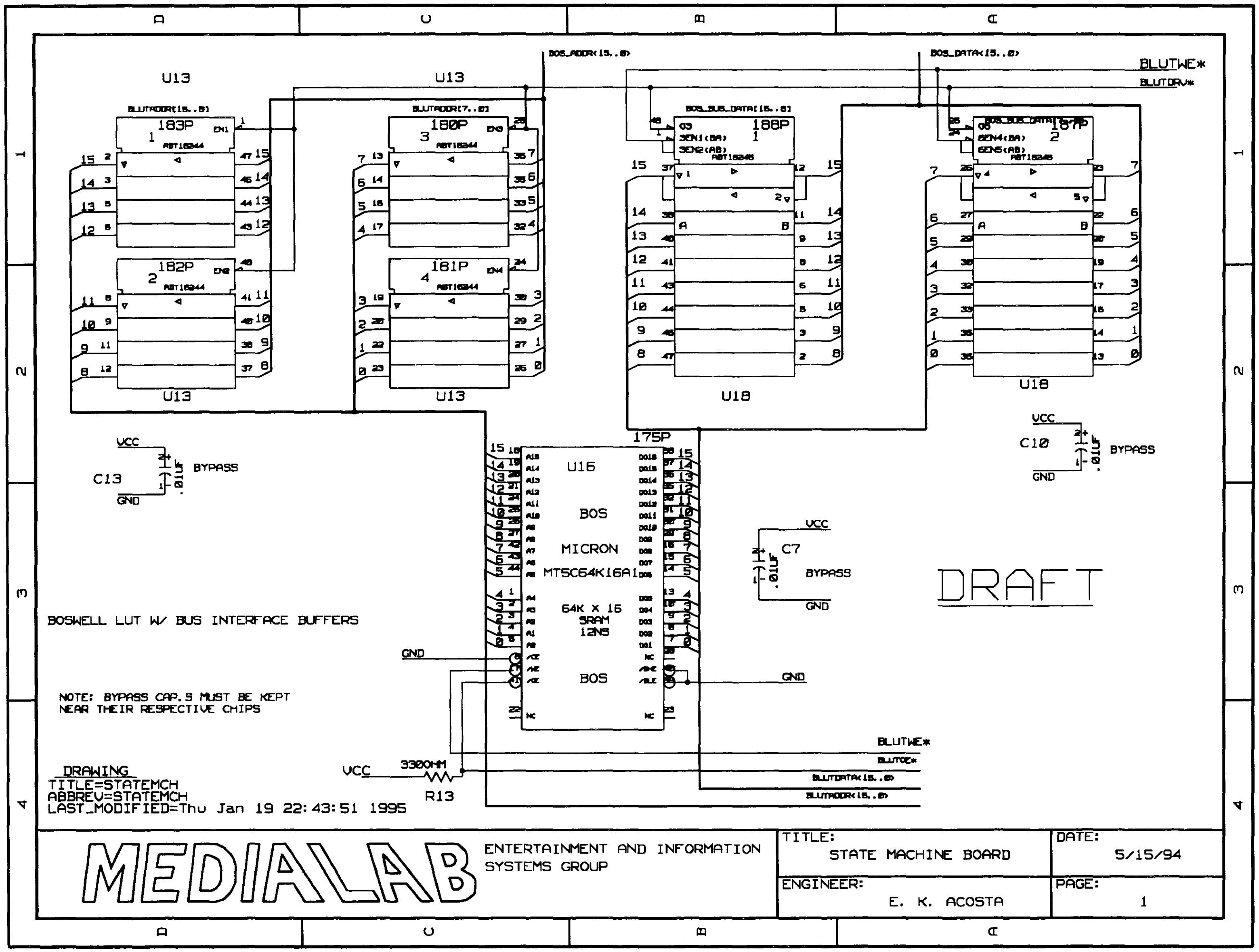
8.2.3 Self-Altering Computing Machinery

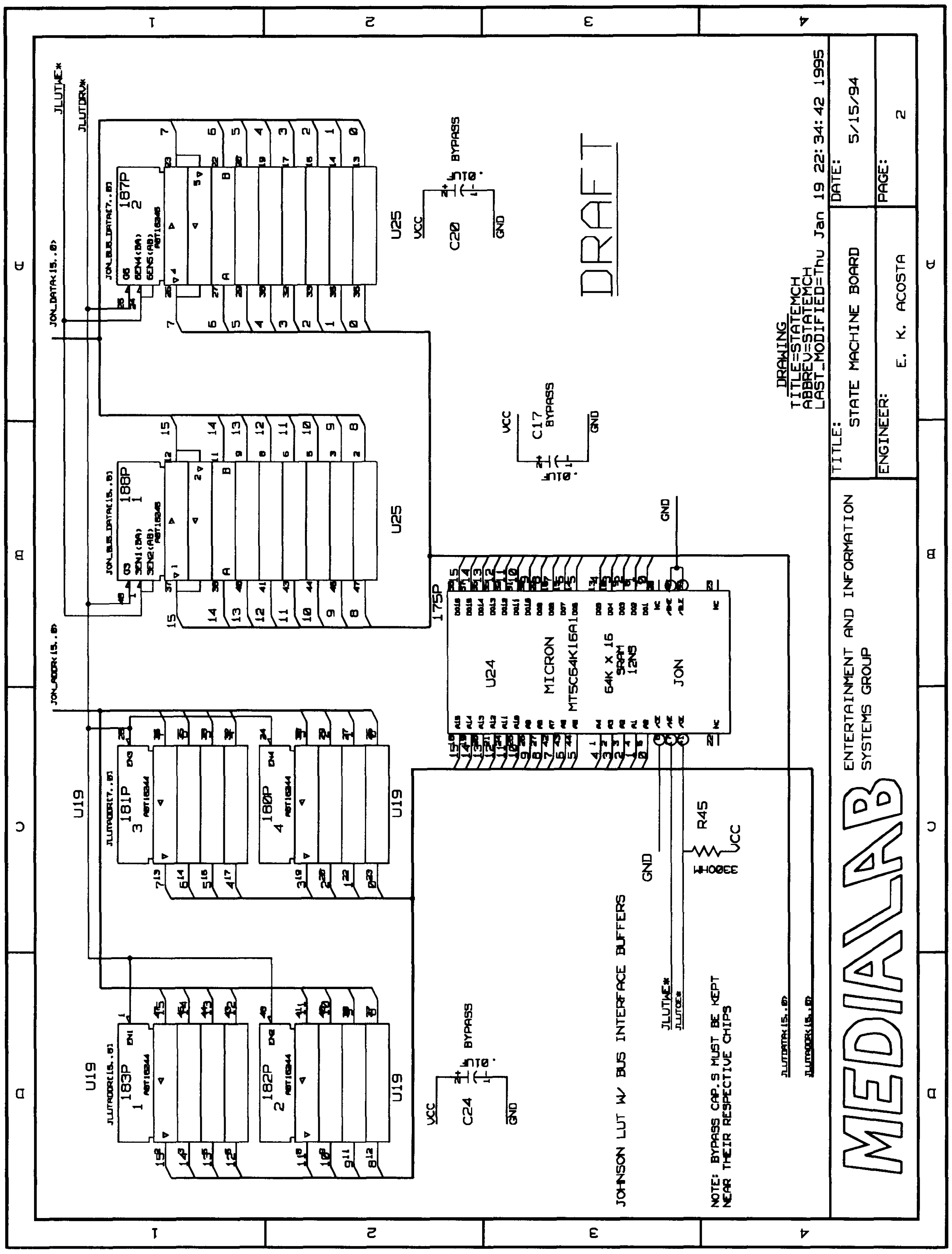
Finally, perhaps the most exciting future direction for research, is the exploration of computers that employ dynamic hardware configuration to alter, in real-time, their machine organization to adapt to their target problem. That is the machine archi-

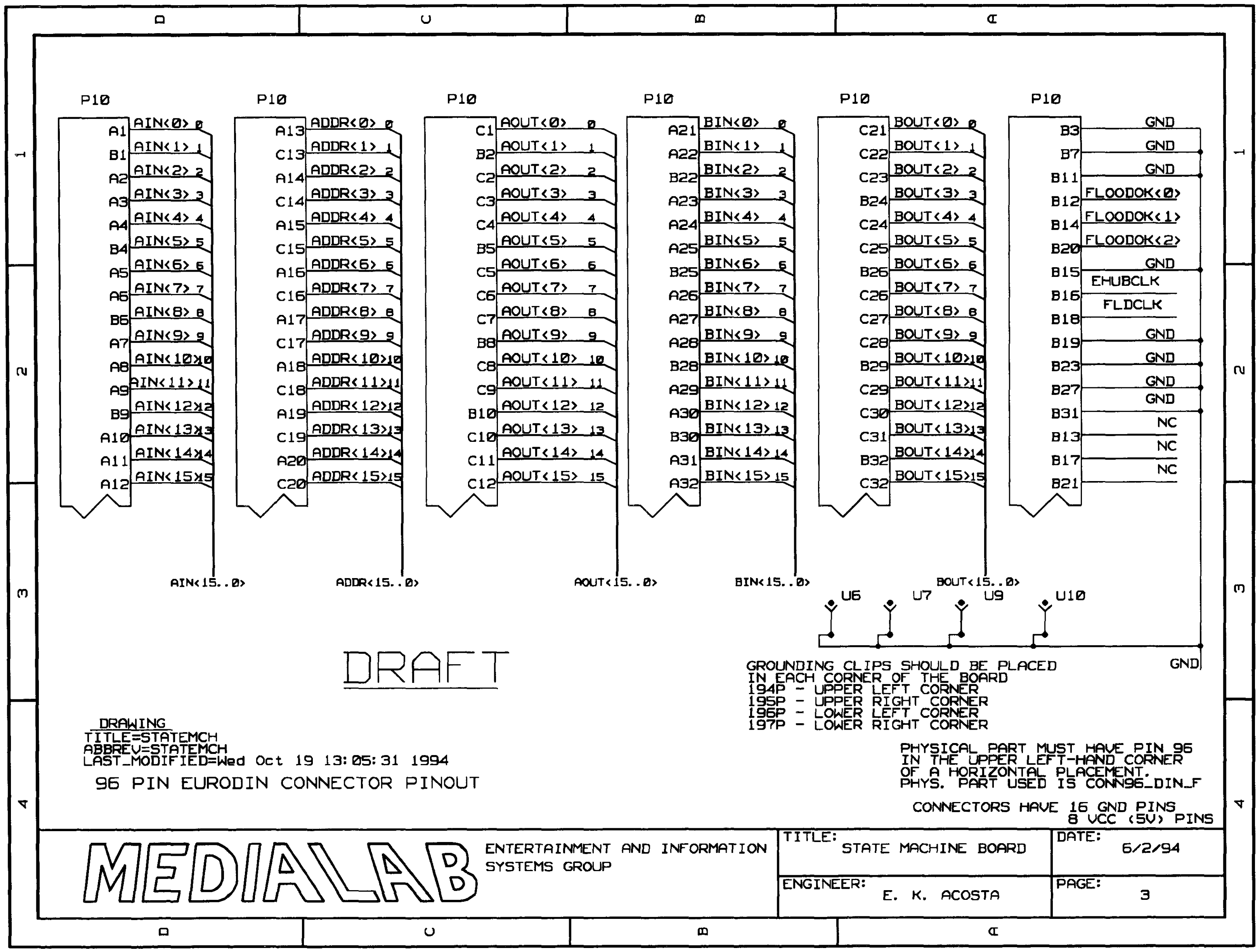
[1]The introduction of the XC5000 family provides evidence of the growing renewed interest in dynamic hardware configuration that will provide the commercial impetus to reduce reconfiguration times

tecture for the next time segment is a function of not just the inputs, but also of the architecture, outputs, and performance metrics of the current time segment. Such machines could be use to address problems with ambiguous characteristics or applications where the lack of standardization introduces hardware/software compatibility issues. Such machines would alter their architecture over time to provide increased computational performance. Machines designed in this manner, when used with current neural networks might even demonstrate machine learning that more closely approximates that of the human brain.

Appendix A: State Machine Schematics


BOSWELL LUT W/ BUS INTERFACE BUFFERS
NOTE: BYPASS CAP.S MUST BE KEPT NEAR THEIR RESPECTIVE CHIPS
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Thu Jan 19 22:43:51 1995
MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
TITLE: STATE MACHINE BOARD
DATE: 5/15/94
ENGINEER: E. K. ACOSTA
PAGE: 1
DRAFT
U13
U18
U16
BOS
MICRON
MT5C64K16A1
64K X 16
SRAM
12NS
175P
183P
182P
180P
181P
188P
187P
ABT16244
ABT16245
BOS_ADDR<15..0>
BOS_DATA<15..0>
BLUTWE*
BLUTDRV*
BLUTOE*
BLUTDATA<15..0>
BLUTADDR<15..0>
BLUTADDR[15..8]
BLUTADDR[7..0]
BOS_BUS_DATA[15..8]
BOS_BUS_DATA[15..0]
C13
C10
C7
.01UF
BYPASS
VCC
GND
R13
330OHM


JOHNSON LUT W/ BUS INTERFACE BUFFERS
NOTE: BYPASS CAP.S MUST BE KEPT NEAR THEIR RESPECTIVE CHIPS
U24
MICRON
MT5C64K16A1
64K X 16 SRAM 12NS
JON
U19
U25
C17
C20
C24
R45
330OHM
BYPASS
.01UF
VCC
GND
JLUTWE*
JLUTDRV*
JON_DATA<15..0>
JON_ADDR<15..0>
JLUTDATA<15..0>
JLUTADDR<15..0>
DRAFT
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Thu Jan 19 22:34:42 1995
DATE: 5/15/94
PAGE: 2
TITLE: STATE MACHINE BOARD
ENGINEER: E. K. ACOSTA
MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP



GROUNDING CLIPS SHOULD BE PLACED IN EACH CORNER OF THE BOARD
194P - UPPER LEFT CORNER
195P - UPPER RIGHT CORNER
196P - LOWER LEFT CORNER
197P - LOWER RIGHT CORNER

PHYSICAL PART MUST HAVE PIN 96 IN THE UPPER LEFT-HAND CORNER OF A HORIZONTAL PLACEMENT.
PHYS. PART USED IS CONN96_DIN_F

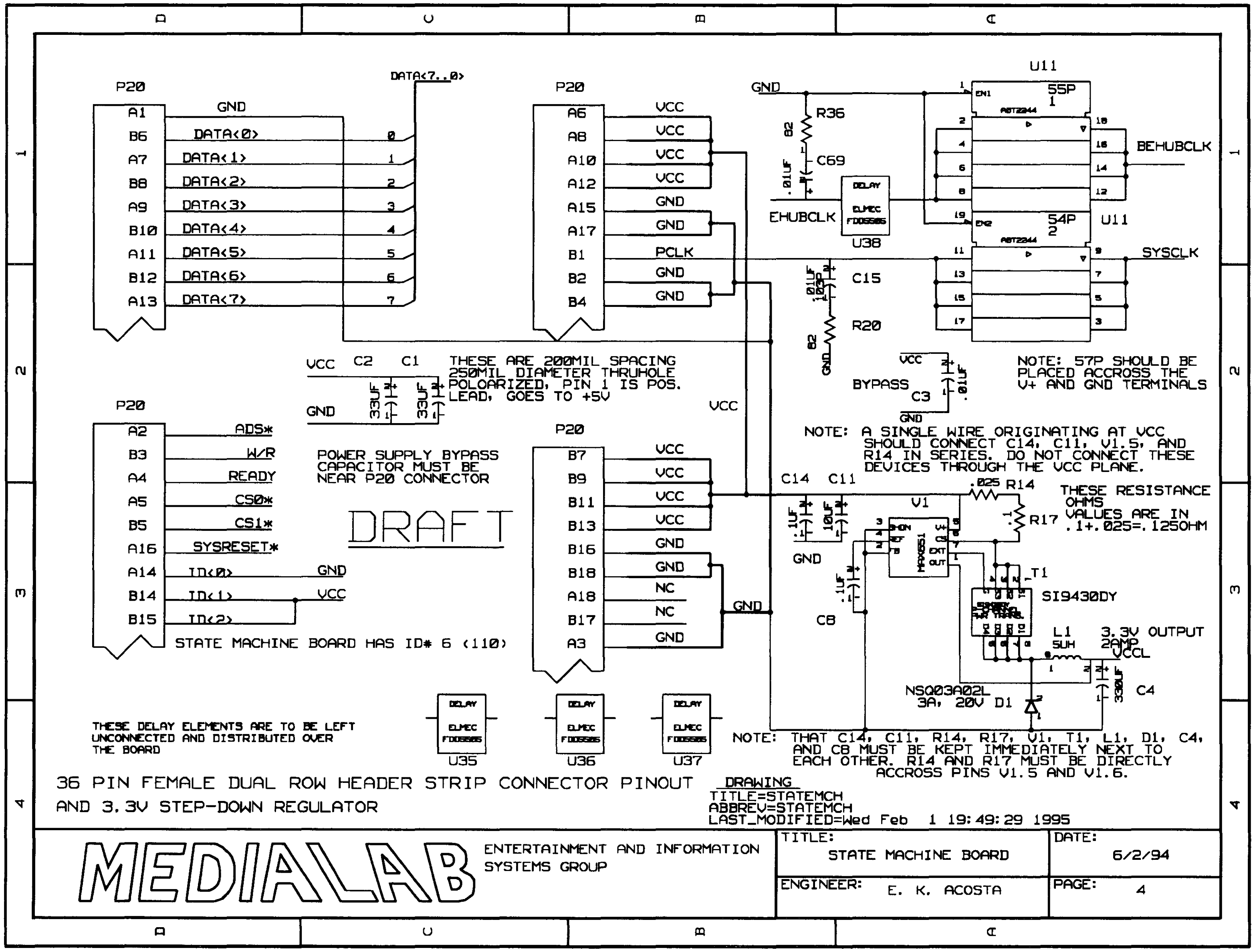
CONNECTORS HAVE 16 GND PINS
8 VCC (5V) PINS

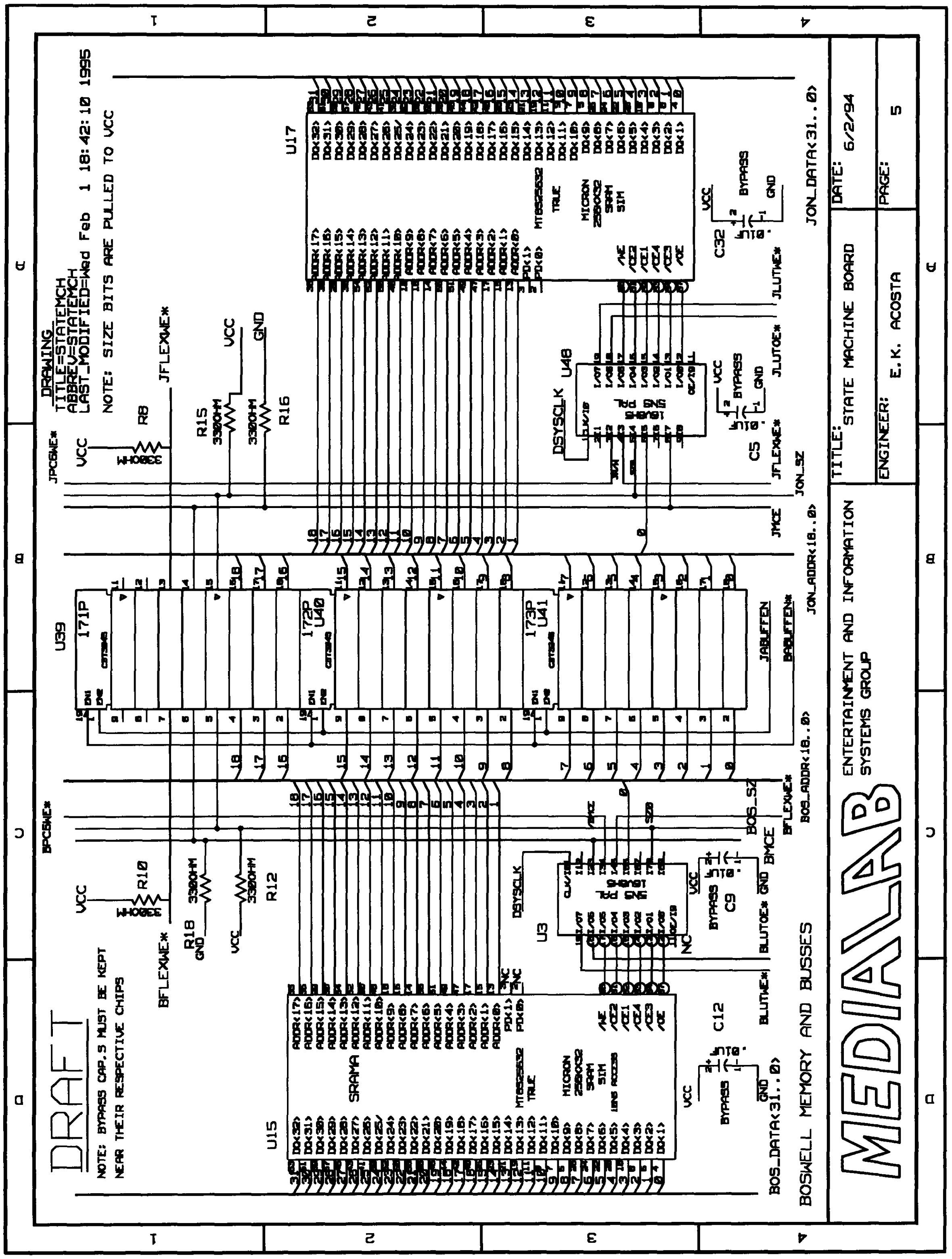
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Wed Oct 19 13:05:31 1994

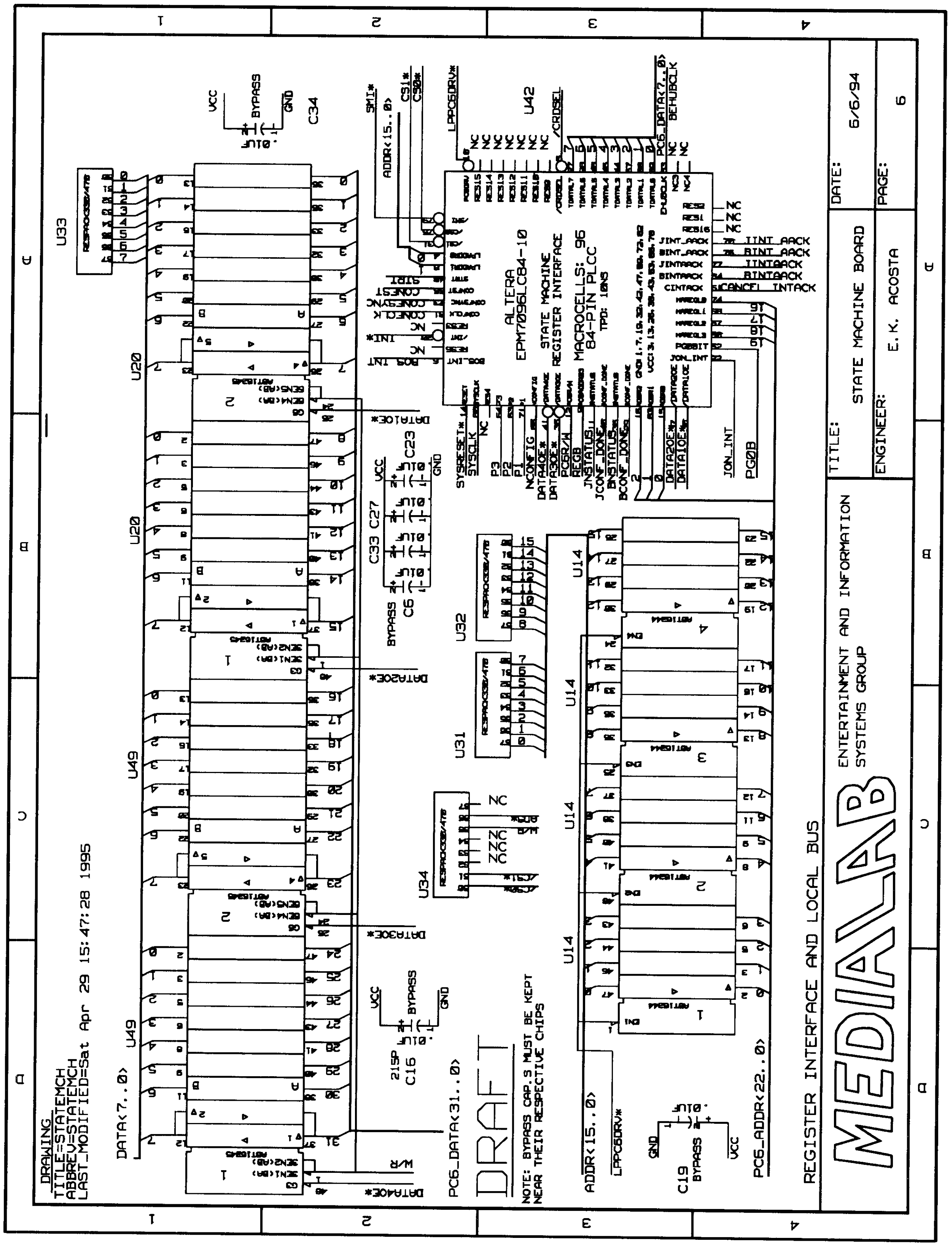
96 PIN EURODIN CONNECTOR PINOUT

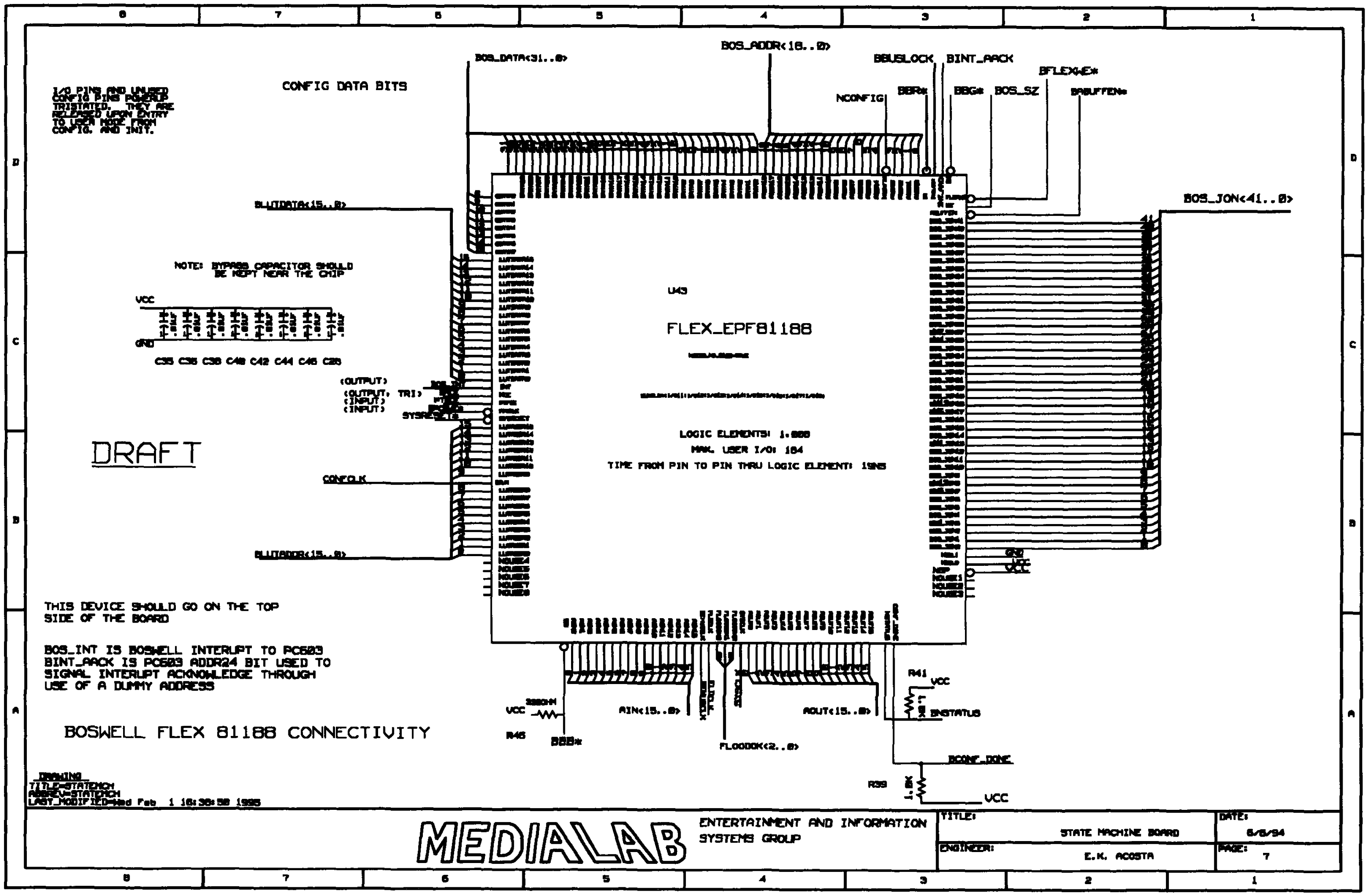
MEDIALAB ENTERTAINMENT AND INFORMATION SYSTEMS GROUP

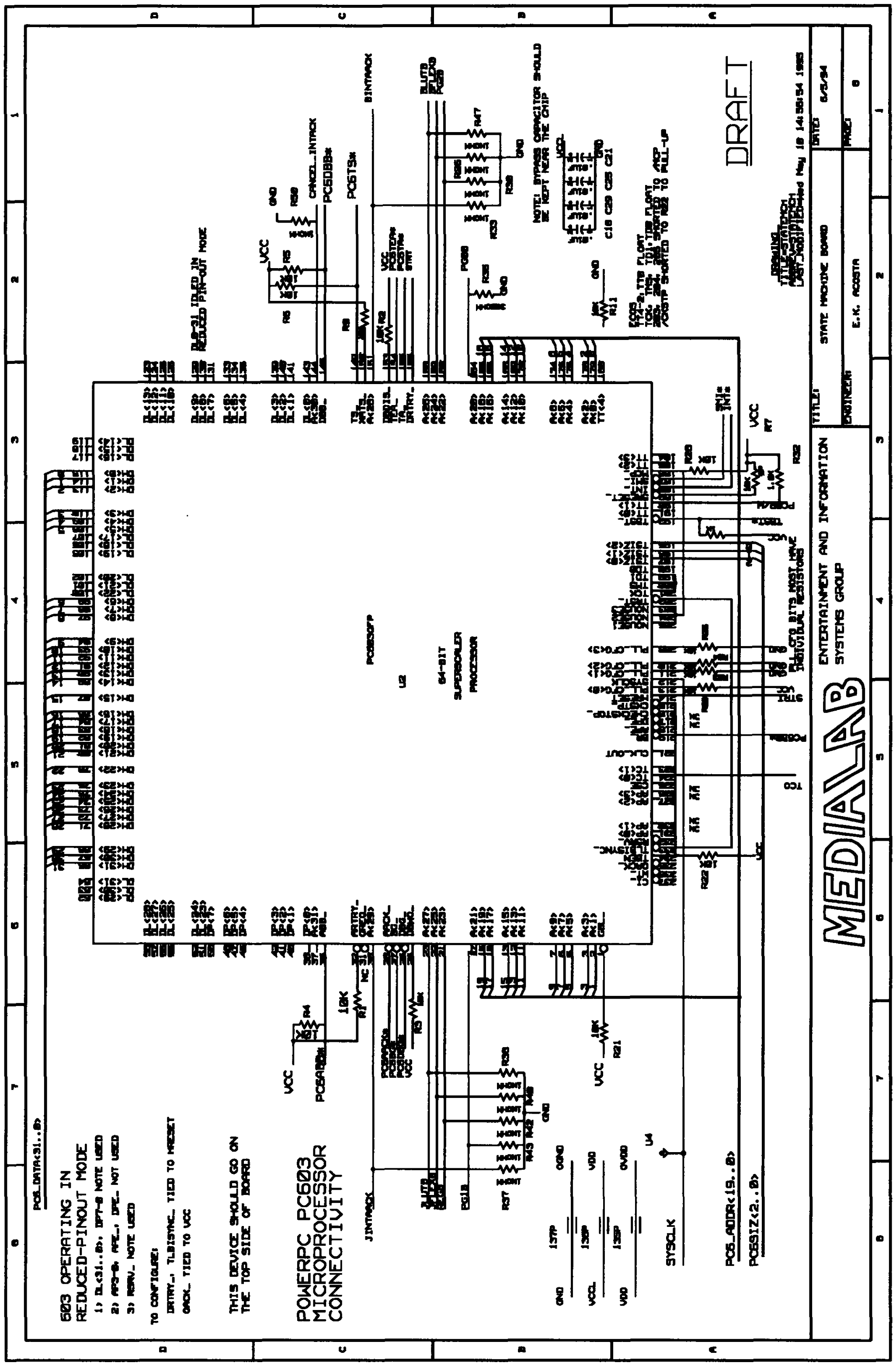
TITLE: STATE MACHINE BOARD	DATE: 6/2/94
ENGINEER: E. K. ACOSTA	PAGE: 3

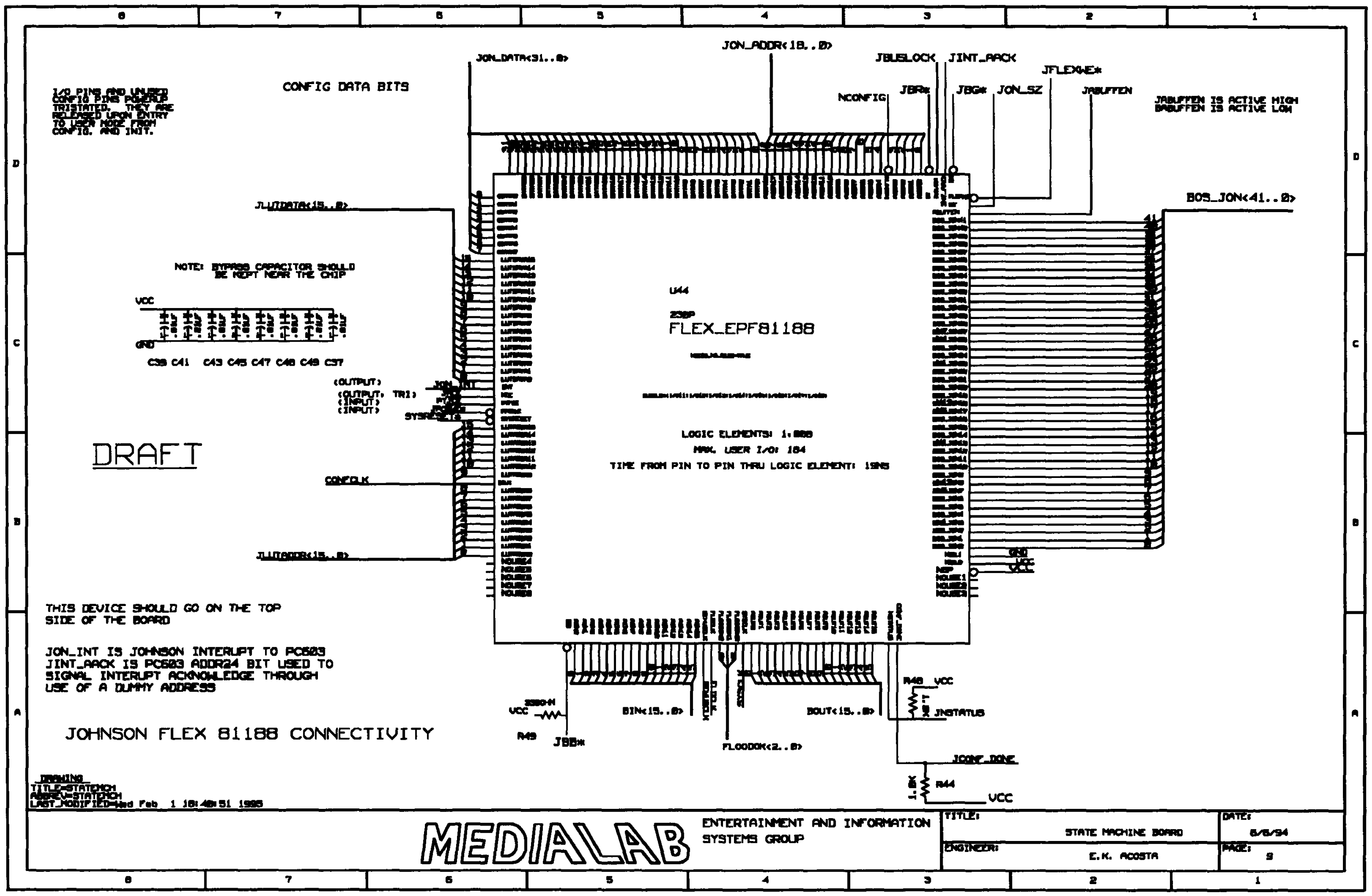

P20
A1 B6 A7 B8 A9 B10 A11 B12 A13
GND
DATA<0>
DATA<1>
DATA<2>
DATA<3>
DATA<4>
DATA<5>
DATA<6>
DATA<7>
DATA<7..0>
0 1 2 3 4 5 6 7
A6 A8 A10 A12 A15 A17 B1 B2 B4
VCC VCC VCC VCC GND GND PCLK GND GND
A2 B3 A4 A5 B5 A16 A14 B14 B15
ADS*
W/R
READY
CS0*
CS1*
SYSRESET*
ID<0>
ID<1>
ID<2>
GND
VCC
STATE MACHINE BOARD HAS ID# 6 (110)
B7 B9 B11 B13 B16 B18 A18 B17 A3
VCC VCC VCC VCC GND GND NC NC GND
VCC C2 C1
GND
33UF 33UF
THESE ARE 200MIL SPACING 250MIL DIAMETER THRUHOLE POLOARIZED, PIN 1 IS POS. LEAD, GOES TO +5V
POWER SUPPLY BYPASS CAPACITOR MUST BE NEAR P20 CONNECTOR
DRAFT
VCC
GND
R36
82
C69
.01UF
EHUBCLK
DELAY ELMEC FD0S585
U38
U11
55P 1
54P 2
ABT2244
EN1 EN2
BEHUBCLK
SYSCLK
C15
.01UF
R20
82
BYPASS
C3
.01UF
NOTE: 57P SHOULD BE PLACED ACCROSS THE V+ AND GND TERMINALS
NOTE: A SINGLE WIRE ORIGINATING AT VCC SHOULD CONNECT C14, C11, V1.5, AND R14 IN SERIES. DO NOT CONNECT THESE DEVICES THROUGH THE VCC PLANE.
C14 C11
.1UF 10UF
.025 R14
R17 .1
THESE RESISTANCE OHMS VALUES ARE IN .1+.025=.125OHM
V1
MAX651
SHDN REF FB V+ CS EXT OUT
GND
C8
.1UF
T1
SI9430DY
L1 5UH
3.3V OUTPUT 2AMP VCCL
330UF
C4
NSQ03A02L 3A, 20V D1
DELAY ELMEC FD0S585
U35 U36 U37
THESE DELAY ELEMENTS ARE TO BE LEFT UNCONNECTED AND DISTRIBUTED OVER THE BOARD
NOTE: THAT C14, C11, R14, R17, V1, T1, L1, D1, C4, AND C8 MUST BE KEPT IMMEDIATELY NEXT TO EACH OTHER. R14 AND R17 MUST BE DIRECTLY ACCROSS PINS V1.5 AND V1.6.
36 PIN FEMALE DUAL ROW HEADER STRIP CONNECTOR PINOUT
AND 3.3V STEP-DOWN REGULATOR
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Wed Feb 1 19:49:29 1995
MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
TITLE: STATE MACHINE BOARD
DATE: 6/2/94
ENGINEER: E. K. ACOSTA
PAGE: 4

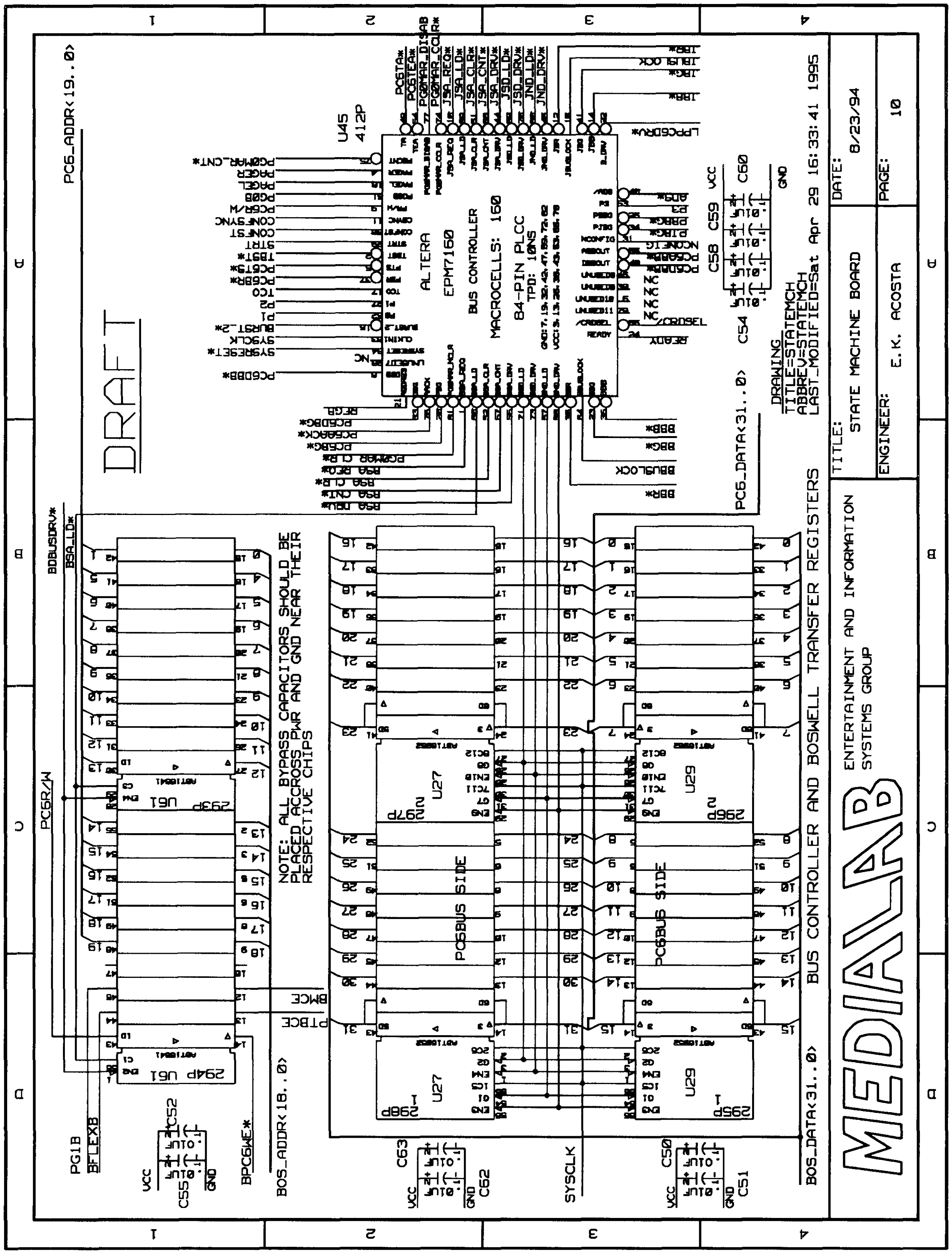

DRAFT
NOTE: BYPASS CAP.S MUST BE KEPT NEAR THEIR RESPECTIVE CHIPS
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Wed Feb 1 18:42:10 1995
NOTE: SIZE BITS ARE PULLED TO VCC
U15
SRAMA
U17
U39
171P
U40
172P
U41
173P
U3
U48
DSYSCLK
BOS_DATA<31..0>
JON_DATA<31..0>
BOS_ADDR<18..0>
JON_ADDR<18..0>
BOS_SZ
JON_SZ
BFLEXWE*
JFLEXWE*
BPCSWE*
JPCSWE*
BLUTWE*
BLUTOE*
JLUTWE*
JLUTOE*
BMCE
JMCE
JABUFFEN
BABUFFEN*
R8
R10
R12
R15
R16
R18
330OHM
C5
C9
C12
C32
BYPASS
VCC
GND
BOSWELL MEMORY AND BUSSES
MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
TITLE: STATE MACHINE BOARD
ENGINEER: E.K. ACOSTA
DATE: 6/2/94
PAGE: 5

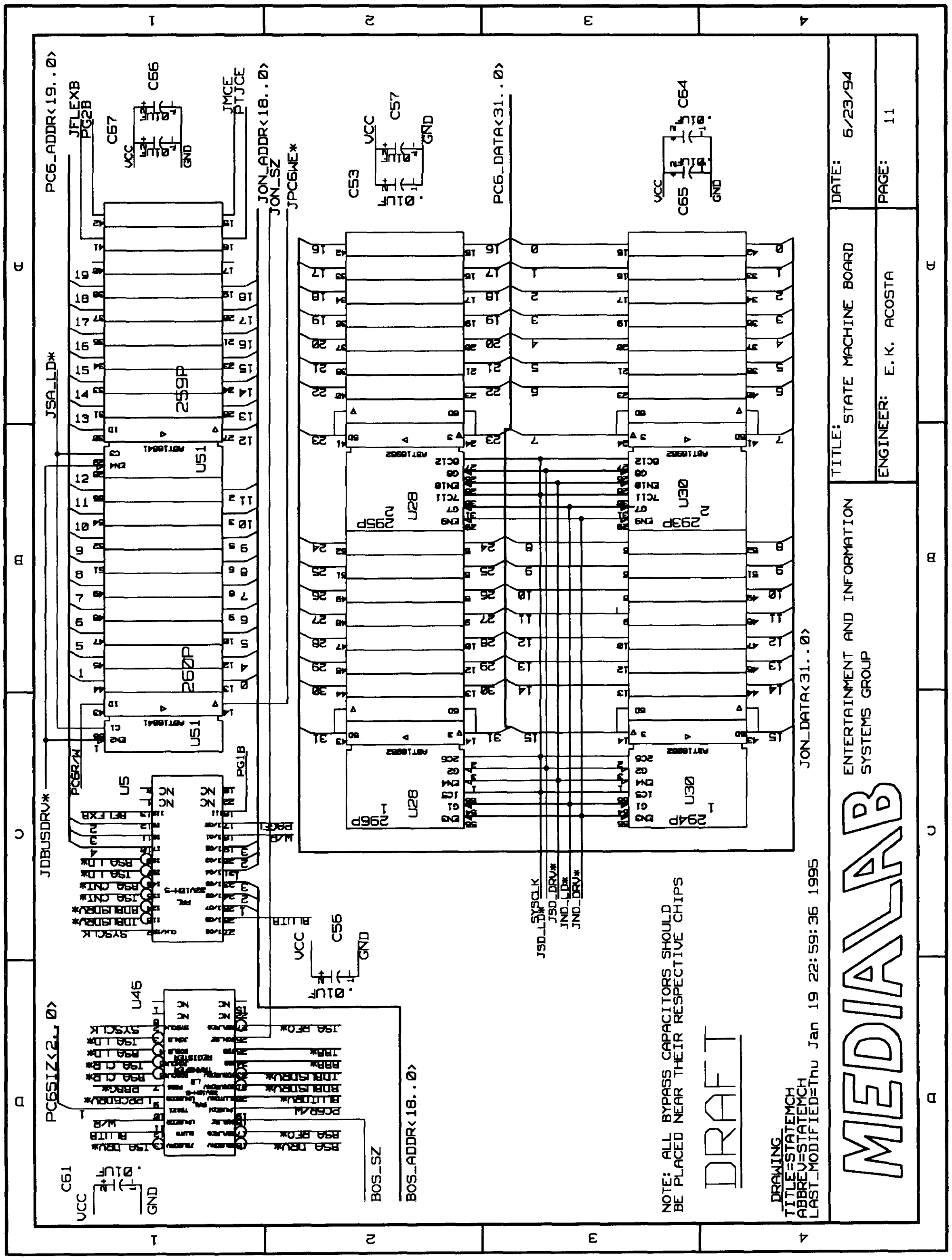

DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Sat Apr 29 15:47:28 1995
DATA<7..0>
U20
U49
U33
C34
VCC
BYPASS
GND
.01UF
DATA1OE*
DATA2OE*
DATA3OE*
DATA4OE*
W/R
PC6_DATA<31..0>
C16
21SP
DRAFT
NOTE: BYPASS CAP.S MUST BE KEPT NEAR THEIR RESPECTIVE CHIPS
C33 C27 C23 C6
U31
U32
U34
NC
ADDR<15..0>
LPPC6DRV*
C19
PC6_ADDR<22..0>
U14
U42
SMI*
CS1*
CS0*
ALTERA
EPM7096LC84-10
STATE MACHINE
REGISTER INTERFACE
MACROCELLS: 96
84-PIN PLCC
TPD: 10NS
STRT
CONEST
CONESYNC
CONECLK
INT*
BUS_INT
SYSRESET*
SYSCLK
P3
P2
P1
NCONFIG
DATA4OE*
DATA3OE*
PC6R/W
REGB
JNSTATUS
JCONF_DONE
BNSTATUS
BCONF_DONE
DATA2OE*
DATA1OE*
JON_INT
PG0B
/CRDSEL
PC6_DATA<7..0>
BEHUBCLK
IINT_AACK
BINT_AACK
IINTAACK
BINTAACK
CANCEL_INTACK
REGISTER INTERFACE AND LOCAL BUS
MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
TITLE:
STATE MACHINE BOARD
ENGINEER:
E.K. ACOSTA
DATE:
6/6/94
PAGE:
6

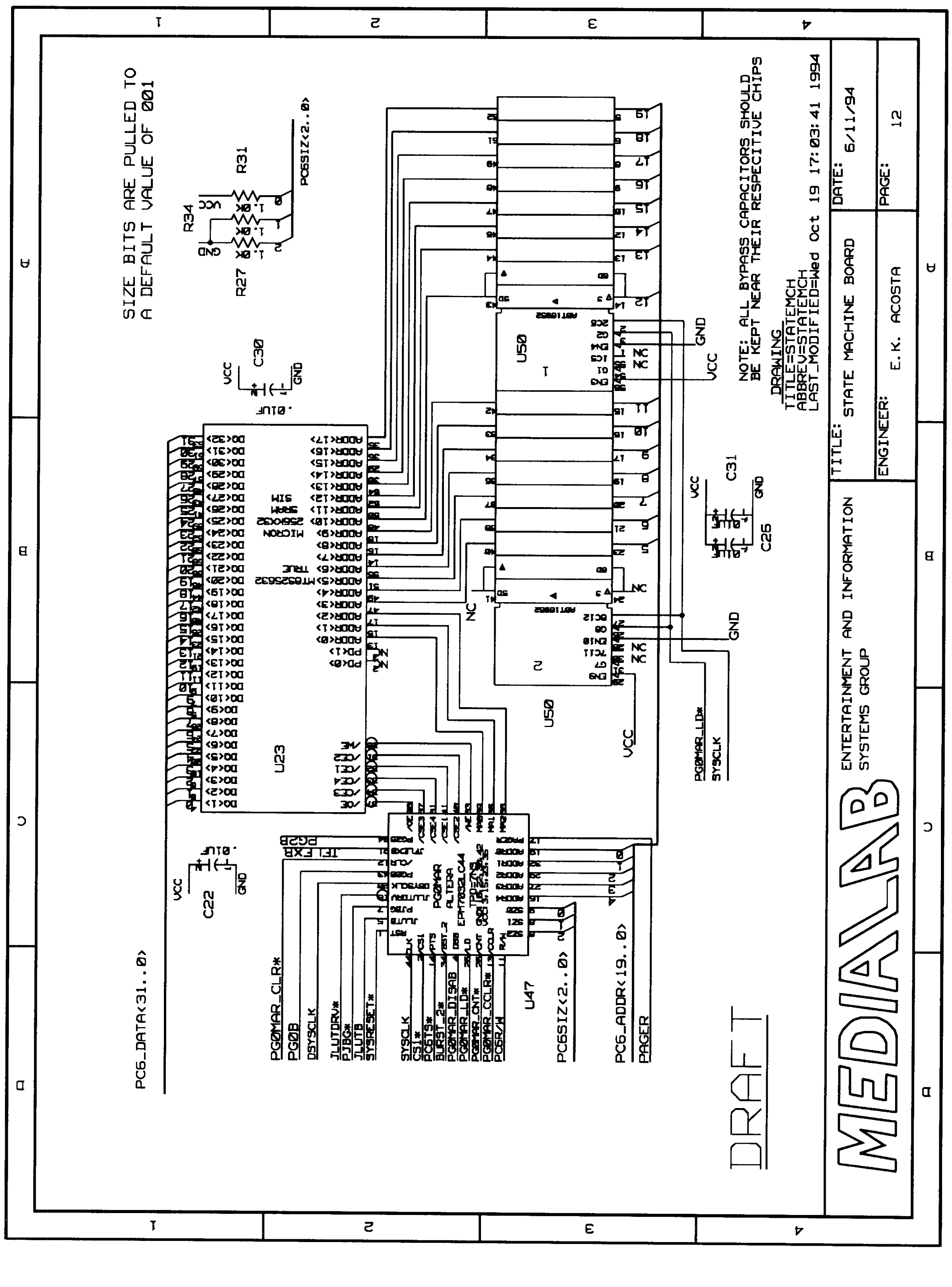

I/O PINS AND UNUSED CONFIG PINS POWERUP TRISTATED. THEY ARE RELEASED UPON ENTRY TO USER MODE FROM CONFIG. AND INIT.
CONFIG DATA BITS
BOS_DATA<31..0>
BOS_ADDR<18..0>
BBUSLOCK
BINT_AACK
BFLEXCE*
NCONFIG
BBR*
BBG*
BOS_SZ
BABUFFEN*
BOS_JON<41..0>
BLUTDATA<15..0>
NOTE: BYPASS CAPACITOR SHOULD BE KEPT NEAR THE CHIP
VCC
GND
C35 C36 C38 C40 C42 C44 C46 C28
(OUTPUT)
(OUTPUT, TRI)
(INPUT)
(INPUT)
SYSRESET
U43
FLEX_EPF81188
LOGIC ELEMENTS: 1,008
MAX. USER I/O: 184
TIME FROM PIN TO PIN THRU LOGIC ELEMENT: 19NS
DRAFT
CONFCLK
BLUTADDR<15..0>
THIS DEVICE SHOULD GO ON THE TOP SIDE OF THE BOARD
BOS_INT IS BOSWELL INTERUPT TO PC603 BINT_AACK IS PC603 ADDR24 BIT USED TO SIGNAL INTERUPT ACKNOWLEDGE THROUGH USE OF A DUMMY ADDRESS
BOSWELL FLEX 81188 CONNECTIVITY
VCC
330OHM
R45
BBB*
AIN<15..0>
FLOODOK<2..0>
AOUT<15..0>
R41
VCC
BNSTATUS
BCONF_DONE
R39
1.8K
VCC
GND
VCC
VCC
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Wed Feb 1 16:38:58 1995
MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
TITLE:
STATE MACHINE BOARD
DATE:
6/8/94
ENGINEER:
E.K. ACOSTA
PAGE:
7


603 OPERATING IN REDUCED-PINOUT MODE
1) DL<31..0>, DP7-0 NOTE USED
2) AP3-0, APE_, DPE_ NOT USED
3) RSRV_ NOTE USED
TO CONFIGURE:
DRTRY_, TLBISYNC_ TIED TO HRESET
QACK_ TIED TO VCC
THIS DEVICE SHOULD GO ON THE TOP SIDE OF BOARD
POWERPC PC603 MICROPROCESSOR CONNECTIVITY
PC6_DATA<31..0>
PC6_ADDR<19..0>
PC6SIZ<2..0>
SYSCLK
JINTACK
BINTACK
CANCEL_INTACK
PC6DBB*
PC6TS*
U2
PC603FP
64-BIT SUPERSCALER PROCESSOR
NOTE: BYPASS CAPACITOR SHOULD BE KEPT NEAR THE CHIP
DRAFT
MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
TITLE: STATE MACHINE BOARD
ENGINEER: E.K. ACOSTA
DATE: 5/5/94
REV: 0
VCC
GND
CLK_OUT


I/O PINS AND UNUSED CONFIG PINS POWERUP TRISTATED. THEY ARE RELEASED UPON ENTRY TO USER MODE FROM CONFIG. AND INIT.
CONFIG DATA BITS
JON_DATA<31..0>
JON_ADDR<18..0>
JBUSLOCK
JINT_AACK
JFLEXWE*
NCONFIG
JBR*
JBG*
JON_SZ
JABUFFEN
JABUFFEN IS ACTIVE HIGH
BABUFFEN IS ACTIVE LOW
BOS_JON<41..0>
JLUTDATA<15..0>
NOTE: BYPASS CAPACITOR SHOULD BE KEPT NEAR THE CHIP
VCC
GND
C39 C41 C43 C45 C47 C48 C49 C57
(OUTPUT)
(OUTPUT, TRI)
(INPUT)
(INPUT)
JON_INT
SYSRESET*
U44
238P
FLEX_EPF81188
LOGIC ELEMENTS: 1,008
MAX. USER I/O: 184
TIME FROM PIN TO PIN THRU LOGIC ELEMENT: 15NS
DRAFT
CONFCLK
JLUTADDR<15..0>
THIS DEVICE SHOULD GO ON THE TOP SIDE OF THE BOARD
JON_INT IS JOHNSON INTERUPT TO PC603
JINT_AACK IS PC603 ADDR24 BIT USED TO SIGNAL INTERUPT ACKNOWLEDGE THROUGH USE OF A DUMMY ADDRESS
JOHNSON FLEX 81188 CONNECTIVITY
VCC
330OHM
R49
JBB*
BIN<15..0>
FLOODOK<2..0>
BOUT<15..0>
R48
VCC
1.8K
JNSTATUS
JCONF_DONE
1.8K
R44
VCC
GND
VCC
VCC
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Wed Feb 1 10:40:51 1995
MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
TITLE:
STATE MACHINE BOARD
DATE:
6/6/94
ENGINEER:
E.K. ACOSTA
PAGE:
9


MEDIALAB
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
TITLE: STATE MACHINE BOARD
DATE: 8/23/94
PAGE: 10
ENGINEER: E.K. ACOSTA
BUS CONTROLLER AND BOSWELL TRANSFER REGISTERS
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Sat Apr 29 16:33:41 1995
DRAFT
NOTE: ALL BYPASS CAPACITORS SHOULD BE PLACED ACCROSS PWR AND GND NEAR THEIR RESPECTIVE CHIPS
U45 412P
ALTERA EPM7160
BUS CONTROLLER
MACROCELLS: 160
84-PIN PLCC
TPD: 10NS
PC6_ADDR<19..0>
PC6_DATA<31..0>
BOS_ADDR<18..0>
BOS_DATA<31..0>
SYSCLK
PC6R/W
BDBUSDRV*
BSA_LD*
PG1B
BFLEXB
BPC6WE*
BMCE
PTBCE
U61 293P
U61 294P
U27 297P
U27 298P
U29 296P
U29 295P
PC6BUS SIDE
C50 C51 C52 C54 C55 C58 C59 C60 C62 C63
VCC
GND


TITLE: STATE MACHINE BOARD
DATE: 6/23/94
ENGINEER: E.K. ACOSTA
PAGE: 11
ENTERTAINMENT AND INFORMATION SYSTEMS GROUP
MEDIALAB
NOTE: ALL BYPASS CAPACITORS SHOULD BE PLACED NEAR THEIR RESPECTIVE CHIPS
DRAFT
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Thu Jan 19 22:59:36 1995
PC6_ADDR<19..0>
JON_ADDR<18..0>
JON_SZ
JPC6WE*
PC6_DATA<31..0>
JON_DATA<31..0>
JSA_LD*
JDBUSDRV*
PC6SIZ<2..0>
BOS_SZ
BOS_ADDR<18..0>
SYSCLK
JSD_LD*
JSD_DRV*
JND_LD*
JND_DRV*
U51
U5
U46
U28
U30
C53
C55
C57
C61
C64
C65
C66
C67
VCC
GND
.01UF



SIZE BITS ARE PULLED TO
A DEFAULT VALUE OF 001
R27
R34
R31
1.0K
GND
VCC
PC6SIZ<2..0>
C30
.01UF
U23
MICRON
256KX32
SRAM
SIM
MT8S25632
TRUE
U50
ABT16952
NC
GND
VCC
NOTE: ALL BYPASS CAPACITORS SHOULD
BE KEPT NEAR THEIR RESPECTIVE CHIPS
DRAWING
TITLE=STATEMCH
ABBREV=STATEMCH
LAST_MODIFIED=Wed Oct 19 17:03:41 1994
C31
C26
PG0MAR_LD*
SYSCLK
PC6_DATA<31..0>
C22
PG0MAR_CLR*
PG0B
DSYSCLK
JLUTDRV*
PJBG*
JLUTB
SYSRESET*
PG2B
TEIEXB
PG0MAR
ALTERA
EPM7032LC44
U47
SYSCLK
CS1*
PC6TS*
BURST_2*
PG0MAR_DISAB
PG0MAR_LD*
PG0MAR_CNT*
PG0MAR_CCLR*
PC6R/W
PC6SIZ<2..0>
PC6_ADDR<19..0>
PAGER
DRAFT
MEDIALAB
ENTERTAINMENT AND INFORMATION
SYSTEMS GROUP
TITLE:
STATE MACHINE BOARD
ENGINEER:
E.K. ACOSTA
DATE:
6/11/94
PAGE:
12

Bibliography

[1] A.K. Agerwala and T.G. Rauscher. *Foundations of Microprogramming Architecture, Software, and Applications*, chapter 1. Academic Press, 1976.

[2] Altera Corporation. *Altera MAX+plus TTL Macro Functions*, August 1990.

[3] Altera Corporation. *Altera Data Book*, August 1993.

[4] Altera Corporation. *Altera FLEX 8000 Handbook*, May 1994.

[5] Jeffrey M. Arnold. The SPLASH 2 software environment. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 88–93, April 1993.

[6] Jonothan William Babb. Virtual Wires: Overcoming Pin Limitations in FPGA-based Logic Emulation. Master's thesis, Massachusetts Institute of Technology, February 1994.

[7] Patrice Bertin and Herve Touati. PAM Programming Environments: Practice and Experience. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 133–137, April 1994.

[8] Dileep Bhandarkar. RISC Architecture trends. In *Proceedings Advanced Computer Technology, Reliable Systems and Applications*, Bologna, Italy, May 1991.

[9] S. Casselman. Virtual computing and the virtual computer. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 43–49, April 1993.

[10] O. Cholwon. A stream processor for motion compensation and image rendition. Master's thesis, Massachusetts Institute of Technology, 1993.

[11] C.P. Cowen and S. Monaghan. A reconfigurable monte-carlo clustering processor. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 59–65, April 1994.

[12] Andre DeHon. DPGA-coupled mircroprocessors: Commodity ICs for the early 21st century. Transit note #100, MIT Artificial Intelligence Laboratory, January 1994.

[13] David E. Van den Bout. The Anyboard: Programming and enhancements. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 68–76, April 1993.

[14] Dave Van den Bout Joe Morris Douglas Thomas Scot Labrossi Scott Wingo and Dean Hallman. Anyboard: An FPGA-based, reconfigurable system. In *IEEE Design & Test of Computers*, pages 21–30, September 1992.

[15] Digital Equipment Corporation. *VAX Hardware Handbook*, 1982.

[16] Jack Dongarra. PDS: The performance database server. Web page: http://performance.netlib.org/, University of Tennessee, April 1995.

[17] Richard C. Dorf. *The Electrical Engineering Handbook*, pages 1654–1656. CRC Press, Boca Raton, FL, 1993.

[18] Ralph Duncan. A survey of parallel computer architectures. *Computer*, February 1990.

[19] Patrick W. Foulk. Data-folding in SRAM configurable FPGAs. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 163–171, Napa, California, April 1993.

[20] The Free Software Foundation. *The C preprocessor*, 1990.

[21] The Free Software Foundation. *Texinfo*, 1992.

[22] The Free Software Foundation. *Using AS, The GNU Assembler*, 1994.

[23] K. Ghose. On the VLSI realization of complex instruction sets using RISC-like components. In *Proceedings of VLSI and Computers. First International Conference on Computer Technology, Systems and Applications*, Hamburg, West Germany, May 1987.

[24] Maya Gokhale and Ron Minnich. FPGA computing in a data parallel C. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 94–101, April 1993.

[25] Regina L. Haviland Greg J. Gent, Scott R. Smith. An FPGA-based custom coprocessor for automatic image segmentation applications. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–179, Napa, California, April 1994.

[26] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, San Mateo, California, 1990.

[27] IBM Microelectronics. *PowerPC 603 RISC Microprocessor User's Manual*, 1994.

[28] Christian Iseli and Eduardo Sanchez. Beyond superscalar using FPGAs. In *1993 IEEE International Conference on Computer Design: VLSI in Computers & Processors*, pages 486–490, October 1993.

[29] V. Michael Bove Jr. and John A. Watlington. Cheops: A data-flow system for real-time video processing. Technical report, MIT Media Laboratory, June 1993.

[30] V. Micheal Bove Jr. and Andrew B. Lippman. Scalable open-architecture television. *SMPTE Journal*, January 1992.

[31] D.V. Klein. RISC vs. CISC from the perspective of compiler/instruction set interaction. In *Proceedings of the Autumn 1989 EUUG Conference*, September 1989.

[32] Adnan H. Lawai. Scalable coding of HDTV pictures using the MPEG coder. Master's thesis, Massachusetts Institute of Technology, 1994.

[33] X.-P. Ling and H. Amano. WASMII: A data driven computer on a virtual hardware. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 33–42, April 1993.

[34] Micron Semiconductor, Inc. *Micron 1994 SRAM Data Book*, December 1993.

[35] Richard Nass. Competitive video compression-decompression schemes forge ahead. *Electronic Design*, 42(13), June 1992.

[36] Karin Schmidt Reiner W. Hartenstein Alexander G. Hirschbiel Micheal Riedmuller and Michael Weber. A novel ASIC design approach based on a new machine paradigm. *IEEE Journal of Solid-State Circuits*, 26(7), July 1991.

[37] S. Seshadri. Polynomial evaluation instructions, a VAX/VMS assembly language instruction. *VAX Professional*, 10(2), 1988.

[38] Irene J. Shen. Real-time resource management for Cheops: A configurable, multi-tasking image processing system. Master's thesis, Massachusetts Institute of Technology, 1992.

[39] Daniel V. Pryor Mark R. Thistle and Nabeel Shirazi. Text searching on SPLASH 2. In *IEEE Workshop on FPGAs for Custom Computing Machines*, pages 172–177, April 1993.

[40] David M. Lewis Marcus H. van Ierssel and Daniel H. Wong. A field programmable accelerator for compiled code applications. In *1993 IEEE International Confer-*