DESIGN AND IMPLEMENTATION

OF

AN EMBEDDED MULTIPROCESSOR SYSTEM, SAM-II

by
Rodoslav Dervishev

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
October 1995

ISBN 0-612-16858-1

Approval

Name:	Rodoslav S. Dervishev
Degree:	Master of Science
Title of Thesis:	Design and Implementation of An Embedded Multiprocessor System, SAM-II
Examining Committee:	Dr. F. David Fracchia Chair

Dr. Rick Hobson
Senior Supervisor

Dr. Slavomir Pilarski
Supervisor

Dr. Glenn Chapman
Examiner

Date Approved: October 11, 1995

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENSE

Title of Thesis/Project/Extended Essay

Design and Implementation of An Embedded Multiprocessor System, SAM-II.

Author: _____

(signature)

Rodoslav S. Dervishev

(name)

October 20, 1995

(date)

To my parents

Acknowledgments

I thank Dr. Rick Hobson for his supportive supervision and help. Without his ideas and comments, this thesis would not have been possible. I also thank Rick for supporting me financially for the duration of the project, allowing me to concentrate on the actual research.

I also thank Dr. Slawek Pilarski and Dr. Glenn Chapman for their careful reading of my thesis and their comments that helped improve it.

Thanks go to my friends and affiliates from the VLSI Lab Dr. Slawek Pilarski, P.S. Wong, Don Smith, George Vodarek and Alicja Pierzynska for making the lab such an exciting place to be at and to the people from Hardware Support Group Peter Corps, Frank Manuel and Steve Nix for helping with equipment in times of need.

Finally, I would like to thank Simon Fraser University and the School of Computing Science for providing financial support for four semesters and thank all faculty members, administrative personnel and students in the department, who made my time at SFU such a rewarding and unforgettable experience.

Abstract

The Structured Architecture Machine (SAM-II) board-level hardware and system software design and implementation concepts are discussed. SAM-II is An Embedded Multiprocessor Vector-Oriented Computer System based on a custom 32 bit RISC chip set designed at the VLSI and Computer Design Laboratory and intended for low-cost array processing and computationally intensive applications.

The prototype is designed to accommodate up to five processing units connected to a DS80C320 microcontroller based mother-board bus. FPGA based logic is designed to implement bus control and processing units control interface functions and also to realize selection, deselection and broadcasting interfacing capabilities, allowing efficient access to the multiprocessor environment within a single DS80C320 machine cycle. The use of the SJCP custom data/control interface incorporating interfacing and boundary-scan testing capabilities is also discussed in the context of system hardware concepts, testing, and processing unit resource access.

The custom use of The Small Computer System Interface (SCSI) Protocol for efficient program, data, and control information transfer is discussed in detail. Menu-driven front-end software provides multiprocessor system configuration, system management, and program debugging capabilities. It also allows flexible real-time access to and use of the system resources. Low-level system resource management is realized by the mother-board monitor program built around a SCSI interface driver communicating with the host in a way similar to the concepts of the remote procedure call.

Discussion about the next generation system and alternative use and applications of the system is also included.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1. Embedded Systems

1.1.1. Architectural Concepts

Embedded Systems (Figure 1.1) represent a subclass of Computer Systems with well-defined features [16, 17, 18].

- Fixed Resources - The system resources, hardware and firmware, are fixed and predefined. They do not increase or decrease during system operation. The way the embedded system interacts with the surrounding environment is predetermined.

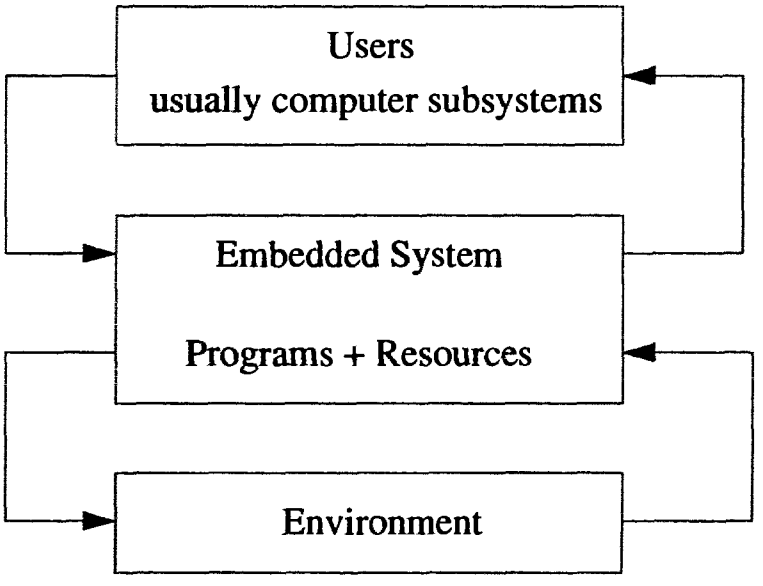


Figure 1.1: Embedded System General Architecture

- Programs - The programs and the coded algorithms do not change and remain the same through the time of operation. In some cases though, part of the code could be downloaded dynamically. For example in multiprocessor configurations, where some of the processors are used as specialized coprocessor units, different code could be loaded dynamically depending on the particular request to the embedded system. In any case though, the code related to the global system resource management and interfacing is fixed and does not change.

- User - The users, which in general are subsystems of more complex

equipment are known and predefined along with the communication protocols.The operator, if any, is getting access to the resources of the embedded system through a directly accessible terminal or an interface unit.

- Environment - The environment where the embedded system performs some control or data processing functions is predefined and finite. This is important to guarantee stable system behavior.

- Data - The data going through the embedded system is the only thing that changes during operation time.

These are the characteristics of a typical embedded system. Of course each application would require features which would make the system unique but in one form or another the general characteristics will be present.

If we have a look at a general purpose desktop computer, every subsystem except the main CPU motherboard computational system represents an embedded system with respect to the user. The hard disk controller for example is a typical representative. The environment, the disk modules, is finite and predictable, the resources are fixed, the firmware is unchangeable, performing predefined disk control functions and servicing the host communication protocol.

1.1.2. Embedded Systems Interfacing

A general characteristic of an embedded system is that the user does not have direct access to its resources. The interface is realized through the host standard or custom built interfaces, Figure 1.2.

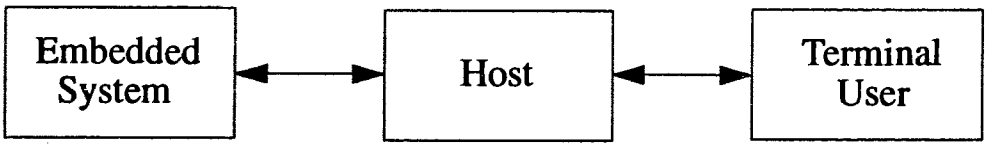


Figure 1.2: Host - Embedded System Configuration

Depending on the timing requirements and architectural concepts the embedded system could be connected to the host through standard external interfaces like RS 232, parallel ports or the Small Computer System Interface (SCSI), or it could be plugged directly on the main CPU local bus. In case an external interface is used,

the data transfer complies to the corresponding interface communication protocol. In some cases though it might be necessary to build a higher-level communication protocol on the top of the standard interface protocol suitable for the particular application.

When an embedded system is attached to a host bus, it could be directly memory mapped using shared memory to communicate with the host, or some more complex protocol could be implemented. In any case shared memory is used to communicate and transfer control information and data in both directions.

In general, we could have several levels of interfacing. Let's consider the case of a multiprocessor embedded system where one of the processors is dedicated to serve the host interface and the rest of the processors are engaged in data processing activities. In this scenario, the first interface level is between the host computer and the embedded system interface processor and the second interface level is between the embedded system interface processor and the data processing processors. The ultimate goal here is to be able to transfer data between the host and the data processing processors. Sometimes a more complex interfacing concept is applied, particularly when the number of processors in the embedded system increases.

Sometimes one needs to access the embedded system directly, particularly during debugging. This is done usually by building local interfaces allowing direct access to the embedded system resources. An interesting concept has been realized in the Power PC 603 microprocessor [43] where dedicated pins in the microprocessor package allow an external processor to get access to 603's resources and monitor the system performance.

1.1.3. Embedded Systems Debugging and Trouble-Shooting

Obviously, with respect to the host computer an embedded system represents a a part of its computational resources regardless of the way it is interfaced. On power-up the system resources are tested including all the embedded systems. The embedded systems initialization procedures should perform local system test and respond in a predefined way providing information about the extent and accessibility and functionality of its local resources. The local testing routines might involve combinations of different board-level testing techniques and Built In Self Tests (BIST) at the chip level. The purpose of a power-up self-test is to determine what part(s) of a presumably working subsystem are available.

A more interesting question is how to test and debug an embedded system in

the process of integrating and building the prototype. The interesting part is that the embedded system, not accessible directly, should be tested at board-level through its interfaces which are also under test and development.

Usually during trouble-shooting the designers are using a combination of different debugging tools and mechanisms [45] in order to get the full picture of the problem. One of the least expensive mechanism is ROM monitoring. The ROM monitor allows to set break-points, control program execution, and access memory and registers. A very popular tool is also the ROM emulator. It provides a low-level debug control and is very flexible to use. We used the PROMICE ROM emulator to emulate the microcontroller program storage during debugging.

Probably one of the most favorite tools is the In-Circuit Emulators (ICEs). The ICE provides full execution control and sometimes bus monitoring and performance analysis functions. The problem with the ICEs is that they could be pretty expensive and difficult to build particularly for chips with high complexity and high clock rates.

A fairly new approach which is getting more and more popular is to incorporate an on-chip debugging hardware which could be a Background Debugging Mode (BDM) or scan path implementation allowing program execution control and registers and memory access. The IBM's 403GA Power PC JTAG scan path [45] provides a serial link to the on-chip debugging hardware allowing to set breakpoints, control execution and read and modify registers, memory and cache. A very interesting concept has been realized in the Advanced Micro Devices' 29040 microprocessor. The 29040's on-chip debugger uses scan paths to access registers and memory, and the hardware has been designed in such a way so that two 29040s could work in tandem in a master-slave relationship. The master CPU is executing the code normally. The trace CPU mirrors the master CPU's access addresses on its address bus and they can be picked up by an external hardware and put into a trace buffer.

Of course, we have to mention the traditional tools used during debugging: probes, oscilloscopes, and logic analyzers. In the last years Hewlett-Packard and Tektronix released logic analyzers with disassembling features.

In order to simplify the building and trouble-shooting process, one wants to have a direct access to the resources of the embedded system. The performance of the interface through which the embedded system would be accessed in this case is not of importance since a limited amount of control information is to be transferred.

The important point is for this interface to be simple, reliable, and easy to build.

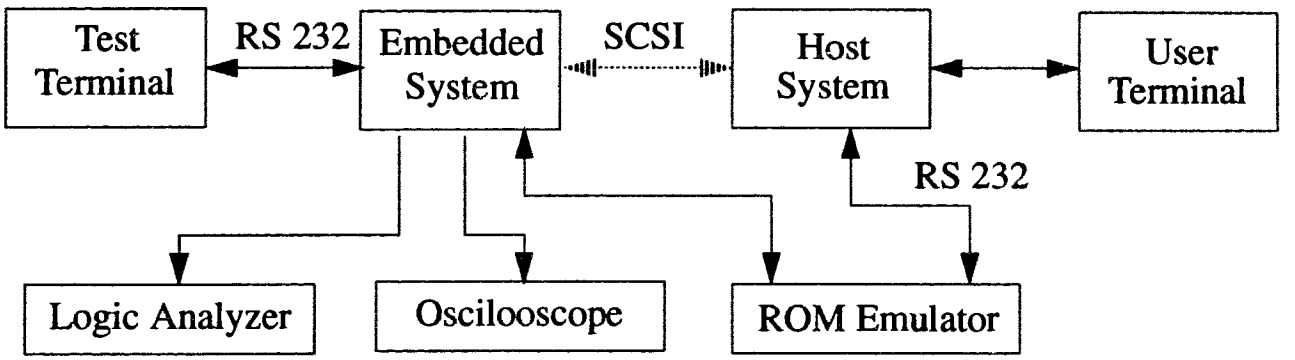


Figure 1.3: Embedded System Trouble-Shooting Configuration

The strategy accepted during SAM-II prototype development is shown in Figure 1.3. Our 8-bit microcontroller, the Dallas DS80C320, has two on-chip serial interfaces and requires very few external resources to build an RS 232 compatible interface to connect the system to a standard ASCII terminal. From this point, we just need to hook up the ROM emulator to get a minimal configuration of a working system. The serial interface provides reliable and simple communication which is vital during the trouble-shooting process. Using the serial communication link, we started adding and troubleshooting the rest of the subsystems like the SCSI interface, the interfacing logic etc.

A typical application of this concept was the trouble-shooting of the SCSI interface.The SCSI interface was dying during system operation sometimes causing total communication failure without being able to continue to figure out what the reason was. Using the direct serial communication we were able to monitor directly the activities on the SCSI bus and in the system as a whole and solve the problem.

1.1.4. SAM-II Embedded System

Even though SAM-II can be classified as an embedded system it has features which are relevant to servers and stand-alone computers. The overall system configuration has five basic components, Figure 1.4. A host computer, PC or workstation, used as a front-end interface, a high-performance SCSI bus connecting the host with SAM-II, a DS80C320 based microcontroller, a microcontroller local bus used to connect the SAM Junior (SJ) processing units, and the SJ processing units.

With respect to the host the microcontroller is a typical embedded system. Its resources, the program and hardware are fixed, the host is defined and finite, the environment, the SJ-boards, is finite and predictable. The information transferred through the SCSI is a mixture of data and commands to the microcontroller.

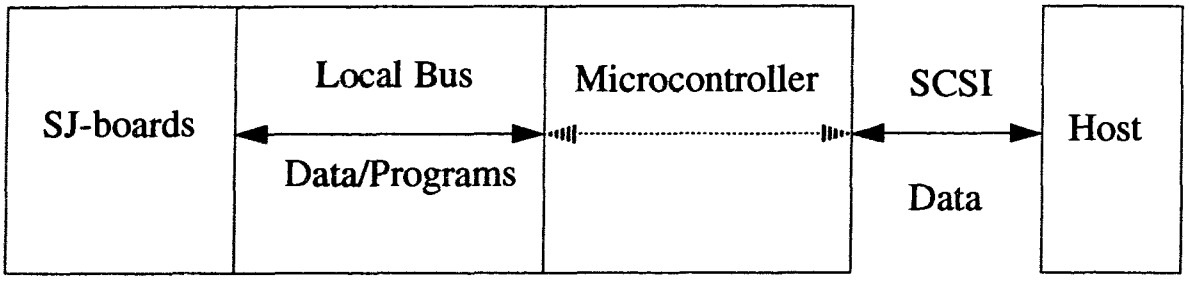


Figure 1.4: SAM-II Embedded System Architecture

The uniqueness is coming from the presence and status of the SJ-boards which are active microprocessor subsystems. The SJ-boards are embedded with respect to the microcontroller. The data transferred through the SCSI to the microcontroller now can be interpreted as data or executable code. In this sense the SJ computing environment could be considered as a remote vector arithmetic server. By replacing the SCSI with an Ethernet or ATM interface, SAM-II can be connected to a network and used as a remote server by remote hosts running array processing applications.

1.2. Test Methods

1.2.1. General Concepts

During the process of system development and later during system operation we have several levels of testing which naturally form a hierarchy of testing procedures: chip or component level test, board-level test and system level test.

At the lowest level we have chip-level test. The chip tests are conducted in two stages. First the chips are tested after manufacturing on special analyzing equipment and after that they are tested as a part of a real system. From a system development point of view we are interested in chip-level test where the chips are a part of functional system. Different chips are tested in different ways, for example the CPUs are tested differently from the memories, but the general method is to supply an input test sequence and check the response. Unfortunately, this might not

be good enough since the chip is a part of a system and its output might be influenced by the other components. Also, with advances in the microelectronic technology it became possible to accommodate a number of fairly independent functional units, sometimes with a high-level of complexity like memories register files, ALUs, interface systems etc. on a single chip. In this case we want to have a better understanding of what is going on in the chip particularly when we are building a prototype. This imposes the necessity of special on-chip hardware for performing or providing the means to test chip functionality.

Embedded BIST circuitries are becoming very popular. Special on-chip circuitry is designed to perform complete chip functionality test. The BIST circuit is interfaced to provide some control if necessary and to check the result of the test. The BISTs are fast, very convenient when there are many chips to be tested, and they save main CPU time during the testing routine. Sometimes it is not possible or not necessary to design complete BIST circuits and other approaches should be used. A new technique called Boundary-Scan allows access to control points inside the chip and chip boundaries [25] and it simplifies the integration of different test mechanisms at chip- and board-level. A variation of Boundary-Scan is used in SAM-II [23] and we will discuss it in the following sections.

Board-level tests use the results from chip-level tests to verify chip functionality and they have more to do with intercomponent interactions, signal propagation, power distribution, and timing problems. They are testing the way the components interact and how their interaction affects overall board functionality. Board-level testing is not a trivial problem and a lot of interesting points could be discussed here. One of the problems we encountered during SAM-II prototype development was connected with the power distribution. The signals on the SCSI bus, for example, turned out to be very sensitive towards the board power distribution and power serge at one place even on different board due to invalid operation happened to affect them causing SCSI communication failure.

The system-level tests are testing the overall system functionality. They are usually a set of testing routines providing the system test control algorithm and checking system resources availability and access. Sometimes the results from the system tests are used by the operating systems, monitor programs and device drivers during system operation.

1.2.2. Boundary-Scan Testing Principles

One approach, which seems to be responding pretty well to the requirements of the new technologies and getting an wide acceptance is the Boundary-Scan. It requires little hardware resources, allows easy integration of chip-, board- and system-level tests and it also can be combined with BIST techniques for automated testing.

The idea of the Boundary Scan test is to connect all I/O pads under test, usually the pads at the boundaries of the chips, in a scan chain [IEEE-1149.1, 24, 25, 28]. It allows access to each individual chip but at the same time certain pads could be bypassed during testing to shorten the scan path. The technique could be used successfully at chip- board- and system-level and it also allows sample testing making possible to test control points on the board at a certain instant of time. Boundary Scan can be used to test points inside the chip as well as to conduct external tests on points between different chips' I/O pads.

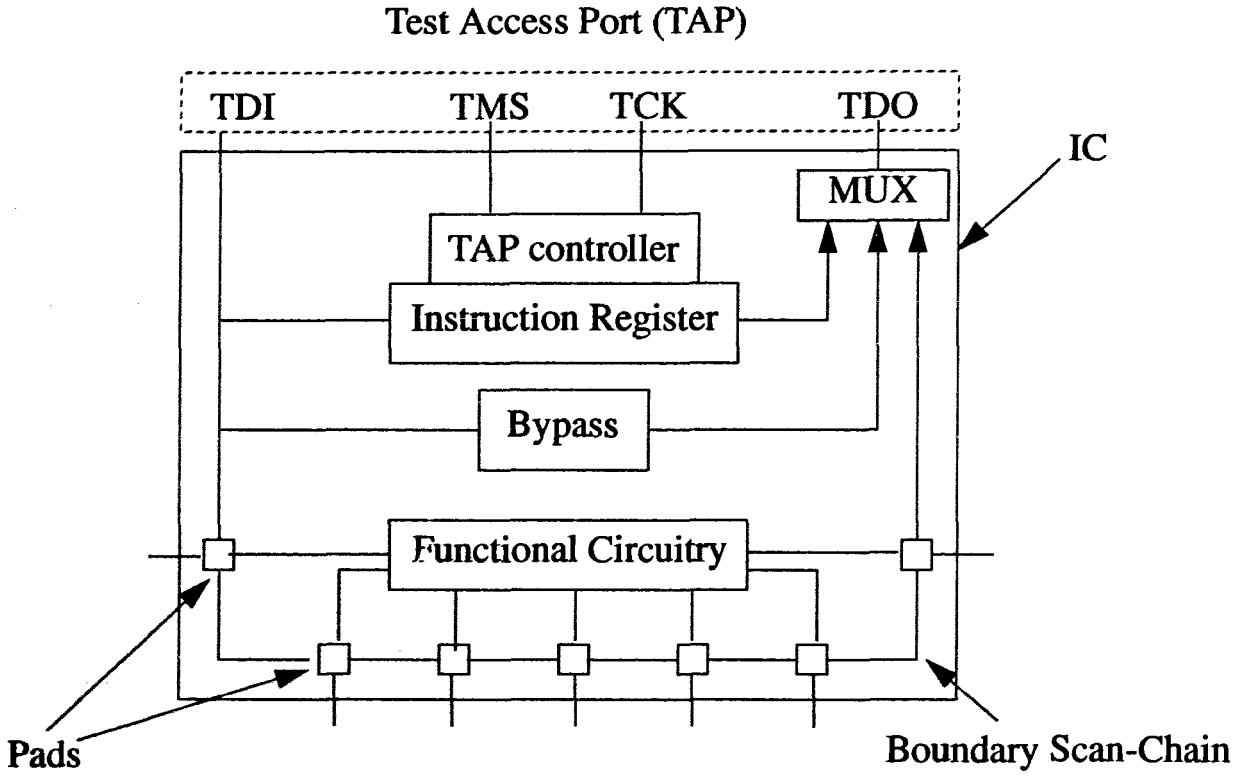


Figure 1.5: Single-Chip Boundary-Scan Architecture

The single-chip Boundary-Scan Architecture submitted by JTAG to IEEE in 1988 for a standard review is shown in Figure 1.5. Four additional pins are

necessary to configure the chip in a standard testing mode and for interfacing purposes - Test Data Input (TDI), Test Data Output (TDO), Test Mode Select (TMS) and Test Clock (TCK). TDI and TDO pins allow the chip to receive and send data from/to other chips. TMS and TCK are used to configure the chip for testing. These four pins constitute the Test Access Port (TAP).

The Instruction Register is used to store the test mode to be used and to select one of the possible data paths from the Boundary Scan-Chain, Bypass Register and user defined registers. The Boundary Scan-Chain can be used for internal as well as external tests.

1.2.3. Scan Paths Implementations

The Boundary-Scan testing concept undergoes continuous development and different variations are coming out sometimes in combination with other approaches. Texas Instruments has released a chip set compatible with the IEEE-1149.1 standard [28] allowing long chains to be broken into several shorter ones, easier to manage. There are several methods which integrate the Boundary Scan concept with BIST techniques for automated board-level testing [24, 26]. Also an interesting design [27] has been implemented using Boundary Scan to test for interconnection and power supply problems using on-chip amplifiers.

A standard JTAG/IEEE-1149.1 boundary scan interface has been incorporated into the Power PC 602 and 603 microprocessors to facilitate board-level testing [43]. Besides for testing purposes the standard JTAG port can be used to access a special interface that allows an external processor to read or write memory or any of the 603's internal registers.

The AMD's 29040 microprocessor has a JTAG scan-path based debugging hardware allowing to modify registers, memory and cache [45]. The microprocessor does not have dedicated pins to access the scan chains but rather the hardware has been designed in such a way so that two microprocessors could run in tandem in master-trace relationship.

ARM's ARM-7 microprocessor's JTAG scan-chain based debugger [45] allows to modify registers and memory, set hardware breakpoints and also the JTAG port can be used for ROMless boot-up and as an output serial port to drive an external device.

Intel's Pentium and P6 microprocessors also have on-chip JTAG scan-path

debuggers.

A variation of Boundary Scan has been realized in the SAM-Junior Control Processor (SJCP) and used in implementing SAM-II testing strategy. The scan chains provide the means to integrate chip- and board-level testing and also they are used for interfacing purposes as well, allowing access to the SJCP internal resources.

1.3. Debugging Tools and Concepts

Development of a program debugging system is an essential part of the system development process. In case of building a prototype the debugger serves two major purposes. First it is essential to be able to write correct programs and second in our case when the software development tools are not completely debugged we want to be able to verify syntactical and logical correctness of the programs used in the system testing procedures. Also, the debugger can be used for low-level resource manipulation by initializing memory blocks or performing some hardware configuration tasks.

1.3.1. General Features

A conventional debugging system [29, 31] performs tasks in two main areas - program management and program environment management. Also it can perform some real time source code manipulation. In the program management area the debugger controls the overall program execution and some of the following or similar options are generally available:

- Stepping - it allows to step through the program a specified number of instructions one or more at a time. One might also be able to initialize the program counter.

- Breakpointing - it is used to suspend program execution at specified location and observe the program status. Usually one is able to edit the breakpoint table in real time. Some debuggers have a breakpoint counter associated with each breakpoint showing how many times a certain breakpoint has been encountered. An interesting feature is the so called programmable breakpoint. There is an expression associated with each breakpoint and a particular breakpoint is valid if the expression is true at this particular point of execution.

- Run - it allows the program to run full-speed and stop at any moment.

- Tracing - It allows information about all procedures and data structures or statements to be reported as the program executes.

- Continue - it starts program execution or resumes it after a breakpoint or a step.

- GoTo - it moves the execution pointer to a certain statement.

- Return - it moves the execution point to the exit point of the current procedure.

The above options provide the user with the necessary means to control the execution program flow. Beside that one needs tools to manage the program environment, to handle variables, data structures, procedure arguments etc. which directly affects the control flow.

- Environment Control Options - these are used to initialize the environment before or during a program execution. It provides scope to the debugger for identifying variables and statements.

- Stack - it allows to manipulate the stack, printing the current status of the stack or a traceback of certain number of stack frames.

- Symbolic Access Options - these options allow to refer to variables, tables or elements of arrays.

- Arguments - gives access to the arguments of an active procedure.

- Evaluate - it allows to evaluate a certain expression from the program.

- Assign - it assigns a value of expression to a name.

- Return - allows to set a return value from a procedure and moves the execution pointer to the exit point of the procedure.

Some debuggers also have the tools to examine the source code or the history of the program execution.

- Find - locates a line in the source file.

- Wh[illegible] - reports a specified location or the current point of execution.

- Point - locates a line number within the debug listing file.

A debugger could have some or all of these features or even some specialized debuggers could have features specific to the particular application. For example debuggers running on parallel computers need to be able to handle concurrently running processes. This is relevant to the SAM-II architecture and is discussed in the following sections.

1.3.2. Debugger System Interfacing

Initially the debugger systems were accessed through not very flexible and most of the time non-interactive command-line interfaces. The corresponding commands were typed at the system prompt in a way similar to typing commands at DOS or UNIX system prompts. In some cases also, the programmers had to write some kind of batch files or even to modify the source code to control the debugging process.

Modern debugging systems are highly interactive windowed tools employing advanced Integrated Development Environments (IDEs). The IDEs allow developers to edit, compile, link, load, execute programs in different modes, manipulate the environment in real time etc. They are proving to be the favorites of the program developers and the tendency is that they will stay on the market.

IDEs have one shortcoming [45], they are not real-time systems. Usually, the embedded software is meant to run on a real-time kernel or monitor reacting to external events in real time. Most of the IDEs do not provide real-time features during debugging. But recently, some companies released debugging systems which incorporate the so called OS-awareness. The debugger has features which let the developers obtain information about the OS status. There is an ongoing research in this direction and debugging systems are getting more responsive to real-time events.

1.3.3. Simulators vs. Target System Debugging

The traditional approach is to debug the software on the target system. This approach seems to be the natural one. The software is debugged on the system it is supposed to execute on and all the tune-ups and fixes can be done during debugging. The developers are using conventional tools like ROM emulators and ICEs and most probably some form of IDE.

Developing the software on the target system looks to be the straightforward way to go but, sometimes it might not be the most efficient one, particularly when we are talking about embedded system software development. In the last years simulators are becoming more and more popular. They provide a stable environment for program developers and basically full control over all the aspects of program development and debugging process. A special simulation tool, Basic RISC Architecture Timer (BRAT) [43], was designed to serve the purposes of Power PC microprocessors based software development and performance estimation. BRAT has been used for performance modeling of a number of Power PC microprocessors, it provides what-if analysis capabilities and sufficient accuracy during simulation. As far as debugging is concerned it allows to watch the state of the system cycle by cycle, to run certain number of cycles, backtrack certain number of cycles, run to certain address or instruction etc. BRAT provides both command line and windowed user interface.

1.3.4. Debugging Parallel Programs

Debugging parallel programs involves aspects of a new range of problems relevant to distributed and parallel computation The problems have to do with data consistency when the parallel processes are running in a shared memory or there is an active interprocess data exchange and also, with the so called race and deadlock conditions between processes running in parallel. There are several ways to guarantee process synchronization. One way is to use timestamps where a timestamp is associated with every shared data item. Another way is to use semaphores controlling the access to specific data items.

Using process synchronization techniques or not, debugging a parallel program could represent a challenge and requires additional attention and tools. A common approach is to replay the program execution. But, the program might have been written nondeterministicly or to have race conditions in which case we might get different results for different runs. One way to handle this is to trace the program execution by recording all the access to the shared data and use the trace later to reproduce the program execution. Many debuggers are recording all the traces but it could be a problem sometimes, the trace could be in order of tens of Mbytes. There are algorithms presented [32] which can decrease the trace length by 2 - 4 orders of magnitude recording only certain critical data.

Also, the debugger might incorporate race condition detectors. These detectors work in different ways depending on the data consistency protocols used. They

might keep track of the timestamps of the shared data or the state of the programs or semaphores.

It is very difficult to create a universal algorithm because data consistency in general is not just a software problem. Data consistency protocols reflect the architecture of the machine, for example if we have a cache at each processor or not. If we have a shared memory and no cache, there is only one copy of the data in the memory and the consistency protocol would have to take care only of the memory. If we have a cache at each processor and distributed memory then things are getting more complicated because we have several copies of the same data in the memory and in the cache.

A very optimistic example would be considering a multiprocessor system with relatively independent processing units. In this case, we only need to be able to address the different units dynamically and debug every single executable image as in case of uniprocessor system.

1.4. Objectives

In the attempts of researchers and engineers to build more and more powerful systems there are two fundamental approaches: distributed computation where more autonomous computer systems usually connected in a network are used to work in parallel on a certain application, and multiprocessor systems where the processing units are connected with high-speed buses or specialized interface networks. A multiprocessor system could be interfaced as a specialized server connected to a network, it might provide its own front-end interface or some more complex interface mechanism might be used. Usually the multiprocessor data processing environment is accessed by the user through a specialized interface controller(s) handling the interface protocol and managing the multiprocessor environment. The particular implementation of the multiprocessor environment, the hardware and software interfacing concept, the system software etc. depends on the target application of the system, performance requirements, selected chip set and most probably on the price requirements if it is intended for the market.

The main objective of our research is the implementation of a multiprocessor computer system, SAM-II, based on a custom 32-bit vector-oriented RISC chip set intended for low-cost array processing applications. The system is not meant to compete at this point with the commercially available supercomputers, rather we are looking for solutions to certain architectural, interfacing and system software questions, which would let us in the future build a competitive product.

Considering the chip-set and system architecture potentially large space requirements, it would not be possible to attach the system to an existing host's local bus and it should be accessed through an external interface as an embedded system. Also our intention is to make the system accessible by different platforms regardless of the vendor and operating system, which would not restrict the user with respect what kind of host computer to use. Considering these factors and the system performance requirements, we chose to use the SCSI interface which provides a good performance, it is available on most commercial workstations, and would allow the system to be interfaced as a conventional SCSI device.

Right now there is no off-the-shelf controller which would handle the SCSI bus data transfer and provide the access to the multiprocessor environment and we need to build a specialized interface controller (a motherboard) which would handle the SCSI bus protocol and manage the multiprocessor environment. We also need to design an on-board interface logic generating the proper processing unit interface control signals.

The processing units are based on a custom 32-bit chip set. Two of the system components, the control processor and memory manager, just came from the foundry and have not been fully tested yet. The chips have been tested independently under ideal conditions on a specialized testing equipment but the question is how they will behave as a part of a real system. The system will also serve as a test-bed for testing and verifying the functionality of the system components working independently and together as a vector-arithmetic processor. The processing unit board will also need an on-board interface logic allowing the processing unit to be attached to the motherboard and to be interfaced independently and in combination with other processing units.

The SCSI was originally designed to interface hard-disk drive systems and the protocol reflects hard-disk data transfer requirements. A major research point in the project is how to design and use the SCSI to interface a custom built embedded system efficiently. This might involve the design of our own protocol on top of the SCSI protocol.

The system software will consist of two major parts: a front-end system interface software running on the host computer and an embedded system monitor software running on the motherboard and probably partially on the processing units. The host software will provide the user interface and will handle the SCSI bus data transfers. The embedded system monitor will be handling the SCSI bus operations

too and it will also perform the multiprocessor environment resource management.

An essential part of the system development process will be the implementation of a debugging system. The debugger would provide a low-level system resource management, the ability to load programs and data and to execute programs in different modes. It will also allow us to verify the accuracy of the current compiler software which has not been tested completely yet.

Here is the summary of our objectives in the project:

* Design and development of a DS80C320 based motherboard

* Design and development of an SJ-chip-set-based processing unit board

* Verifying that SJCP and SJMI can work together

* Design of a FPGA-based motherboard-parallel processing units interface logic

* Embedded system SCSI interface hardware and firmware development

* System and test software development

* Embedded system debugger development

Chapter 2: Hardware Design

2.1. SAM-II Architecture

SAM-II (Structured Architecture Machine - II) is an embedded multiprocessor vector-oriented computer system intended for low-cost array processing applications. It develops further the concepts and the ideas in the first implementation of the Structured Architecture Machine, SAM-I [46] and is a stepping stone towards the development of a massively parallel computer system.

The prototype (Figure 2.1, Figure 2.2) is designed to accommodate up to five processing units based on a custom 32-bit vector oriented RISC chip set. Each processing unit could have up to 64 Mbytes of four-stage interleaved system DRAM and 64 kwords of high-level 64-bit-wide microcode SRAM used as the CPU external program storage. The microcode is loaded dynamically, which gives some flexibility to the system.

The system could be configured to work as an MIMD (Multiple Instruction Multiple Data) machine in which case the processing units execute independent code or as an SIMD (Single Instruction Multiple Data) computer where one of the processing units plays the role of a program manager broadcasting instructions to the other four data management units through a specialized instruction-pipe interface. The pipe interface has not been implemented yet and right now the processing units program execution is controlled directly by the microcontroller.

The processing units are attached to a motherboard (microcontroller) and each one maps to 256 locations of the motherboard CPU directly accessible memory and can be accessed in a single microcontroller stretched machine cycle. FPGA based logic is designed to perform the necessary bus control and control interface signals generation. Without the pipe interface, the microcontroller plays the role of a program manager and all five units can be used to process data.

Usually in multiprocessing applications there is an intensive data transfer among the processing units during a program execution. A special software and/or hardware support is necessary to guarantee real-time data consistency in case of shared data applications. A special network system will be designed to interconnect directly the processing units data memory systems to handle data exchange and data consistency.

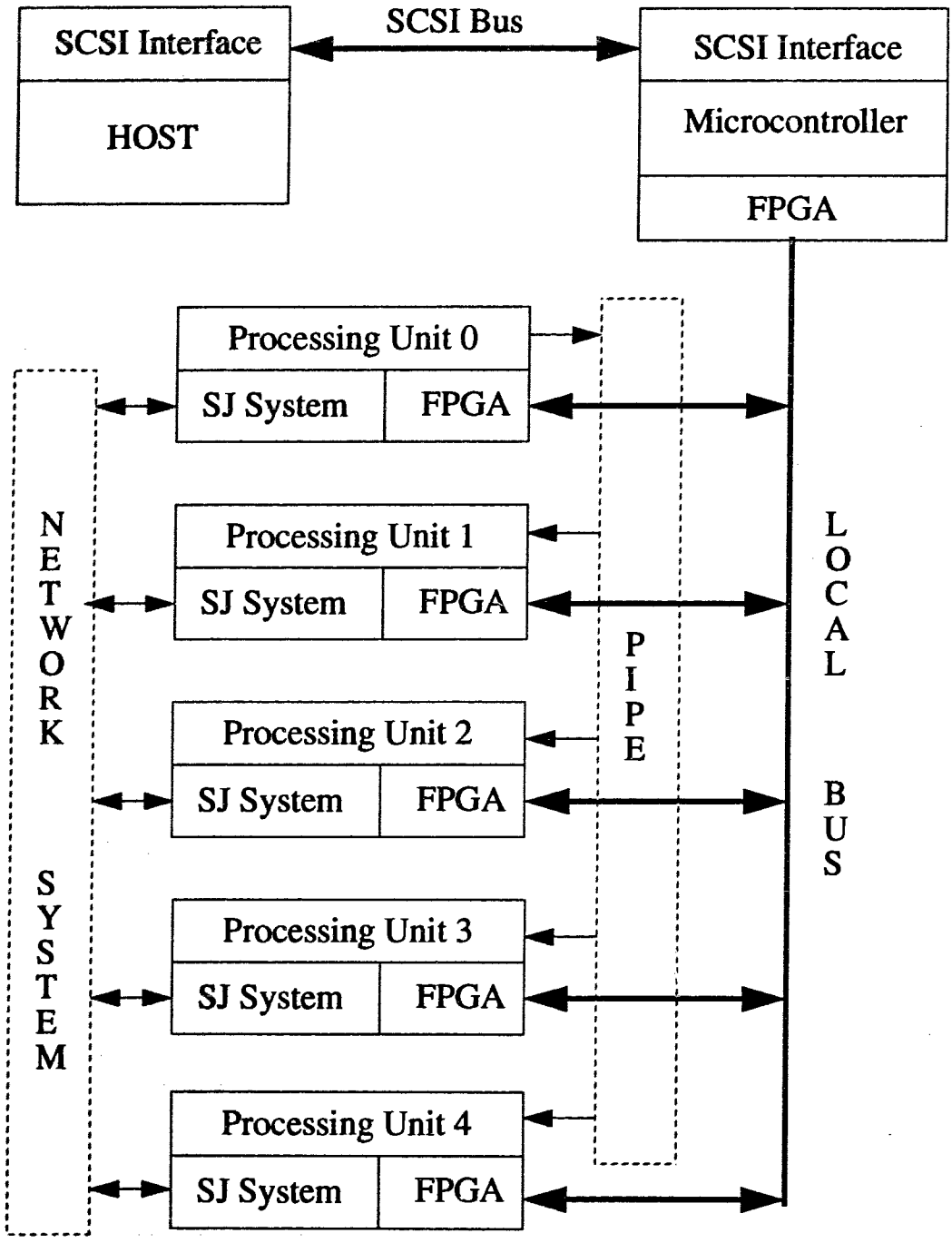


Figure 2.1: SAM-II Architecture

The embedded system is interfaced with the host computer through a fast SCSI interface. From the host's point of view, SAM-II is a conventional SCSI device hooked on the SCSI bus and all data transfers between the host and the embedded

system use the SCSI bus protocol. In this relationship the microcontroller serves two main purposes. First it is a dedicated SCSI driver handling the data exchange on the SCSI bus and second it performs the necessary SAM-II system management functions including system configuration, program management, and data management.

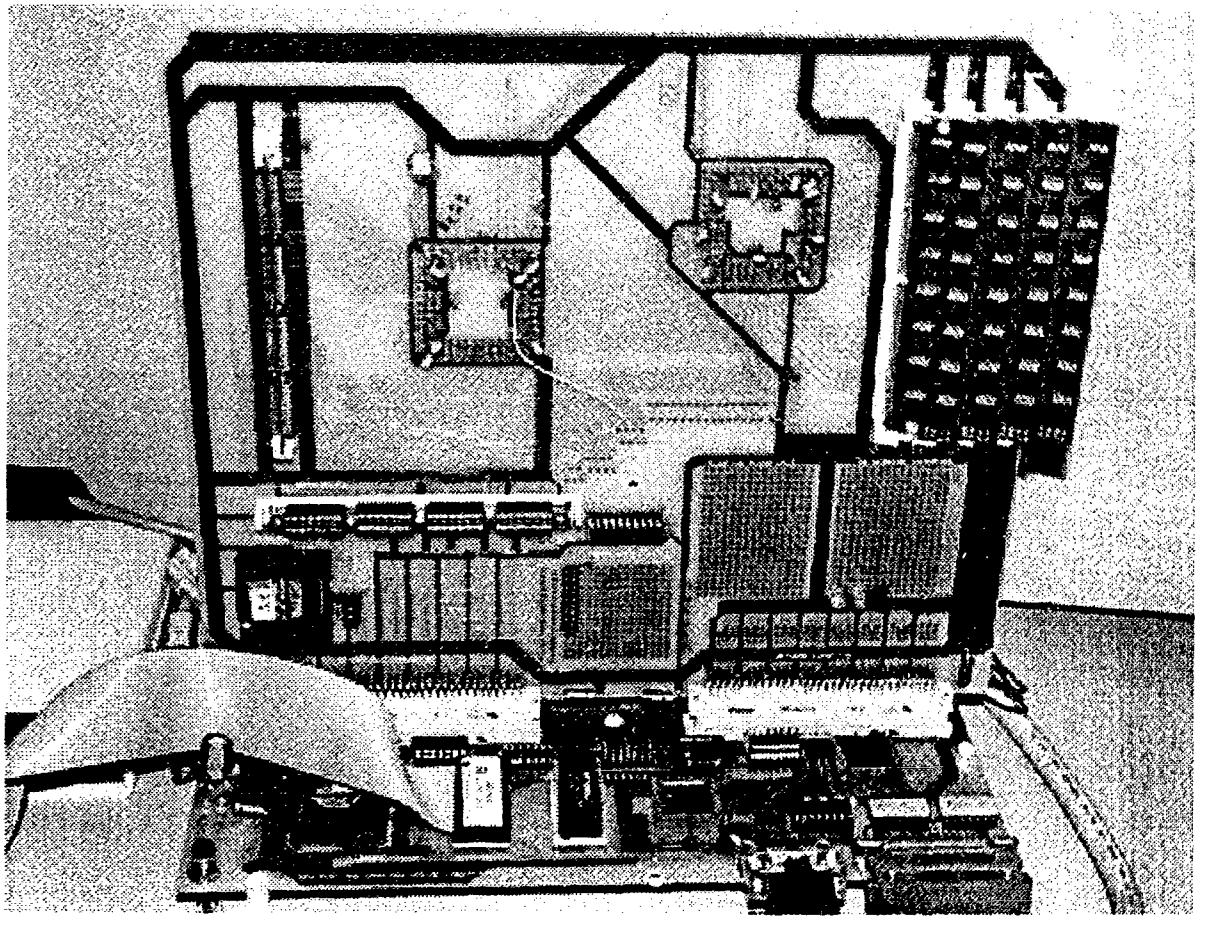


Figure 2.2: SAM-II Prototype

The host and the embedded system interact through a request-driven communication. The host sends a request to the embedded system which could be anything from performing a test to executing a program and the embedded system serves the request and sends the result back. The requests and results are transferred by means of SCSI transactions. The requests are classified in four areas - system test, system configuration, program management, and data management. With respect to the application program SAM-II could be seen as a vector-arithmetic coprocessor accessed through the operating system using conventional system calls.

2.2. Microcontroller

The embedded state of SAM-II multiprocessor imposes the necessity of a specialized microcontroller to service the embedded system-host interface and to control certain activities on the SJ-boards. From the host's point of view, SAM-II is a conventional SCSI device hooked-up to the SCSI bus and this determines pretty much a client-server relationship, the host sends the request, the microcontroller serves it and sends the result back, if any.

With respect to the host, the microcontroller should be able to serve requests in the following areas:

- System test - this would involve testing different system memories like the external microcode storage, external DRAM and on-chip memory and register files, testing separate functional units and inter-system-component interfaces.
- Program/Data loading and Verification - one of the major functions the microcontroller should perform is loading and verifying executables and data into the SJ boards.
- Program management - the microcontroller should be able to control the program execution full-speed, step-by-step, stop execution etc.

Microcontroller also has some monitoring functions concerning the overall system performance. In general, we might have several executables residing in the microcode storage at the same time, some of them system programs, some of them user applications. The monitor should be able to load executables at the proper locations, to start/stop one or another module and to be able to recognize when a certain module has finished. In other words, all the basic functions a conventional operating system would have.

The custom SJ chip set, designed at SFU's VLSI and Computer Design Laboratory, provides excellent hardware support for the implementation of the above functions. Some of these features are hardware-supported boundary-scan testing capabilities and quite straight-forward control interface. The SJ-boards are part of the microcontroller address space and a certain function on a certain board (Processing Unit) is triggered by addressing a specific location. The list of the functions is given in the Appendix.

2.2.1. System Design

The microcontroller is based on Dallas DS80C320 microprocessor, which is an enhanced version of Intel's 8051, capable of running at higher speeds with an

additional serial port, data pointer, and timer. This makes it very suitable for building embedded systems since, we have on-chip serial interface and we need few external resources to build a minimal-configuration working system. This is important when it comes to building prototypes, one needs a reliable connection to the external world to be able to monitor the state of the system in the process of development. The block diagram of the microcontroller is shown in Figure 2.1.

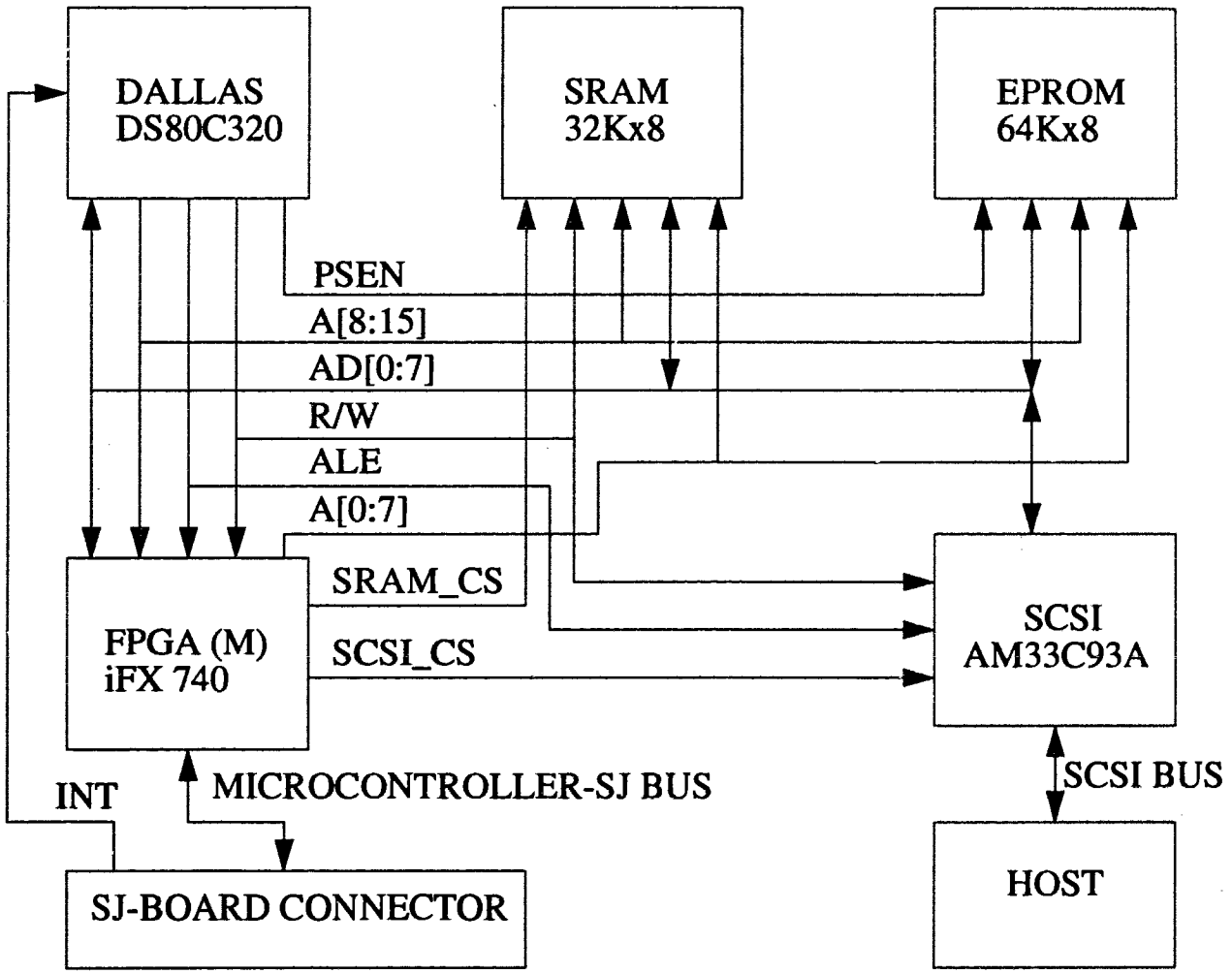


Figure 2.3: Microcontroller Block-Diagram

2.2.1.1. Address Space

The microcontroller has separate program and data address spaces. This is typical for many computer systems but as our research showed it could be a reason

for some undesired effects to take place. Addressing separate spaces generally results in changing high-order bits of the address when switching from one space to another, particularly bits which are used in decoding different components of the system. This could cause glitches on the decoding lines and respective malfunctioning. By changing the program location, one could get different system behavior. These kind of problems are very difficult to figure out since, they might appear only in combination with other events taking place on the boards.

Jumper-configurable, we could have 32k or 64k of program memory. The jumper turns on/off the A15 address line. We have been using PROMICE firmware development tool to emulate our program storage. The PROMICE ROM emulator could be connected to a PC through the standard serial or parallel interfaces and accessed and controlled through a command-line interface. Several PROMICE devices, each emulating 8-bit-wide memory, could be connected in parallel to emulate wider ROM storage of up to 512 kwords deep.

The data address space is divided among the system data SRAM, the SCSI protocol controller, and the SJ boards. The prototype could accommodate up to five SJ boards each corresponding to 256 bytes of addressable memory. The decoding scheme is shown in Table 1.

Table 1: Microcontroller Address Decoding Scheme

A15	A14	System Components
0	X	32k of system SRAM
1	0	SCSI protocol controller
1	1	SJ boards

Partial decoding is used. For our purposes 32k of SRAM is enough although, considering maximum-size SJ executable of 512k would complicate the loading protocol. But in any case we couldn't have more than 64k.

2.2.1.2. SCSI protocol controller

We are using the AMD Enhanced SCSI-Bus Interface Controller Am33C93A. It has 32 addressable registers and it is located directly on the CPU bus. The chip is configured to take advantage of the multiplexed address/data bus. The register address is latched internally at the falling edge of ALE and if the chip is selected a

read or write operation takes place.

DMA and interrupt handling are disabled and data transfer takes place by polling the status bits. This approach was adopted primarily because during building the prototype we wanted the software to run predictably and reliably. Even when something goes wrong it should keep running to report the state of the system. The choice also fits very well with the request-driven initiator-target relationship between the microcontroller and the host imposed by the SCSI bus protocol. In this way the system components' priorities are determined entirely by the software which is preferable during the trouble-shooting procedure.

Of course other solutions are possible. An interrupt handling approach would require a different software architecture. The request-driven communication still could be preserved but in general we will need interprocess communication mechanism between the different subroutines working on the completion of a single request. The software details are discussed in later sections.

2.2.1.3. SJ Boards

The prototype is designed to accommodate up to five boards. Each board has a unique 3-bit ID which is jumper-configurable.

In the first decoding scheme implementation, the address lines A11, A12 and A13 were used as SJ board ID bits, Table 2. They were passing directly to the SJ boards to be compared with the ID jumpers. The selection of a particular board was taking place dynamically within the timing of the current instruction. There are two special combinations, one selects all boards which means that the following activity read/write will take place on all boards and the other prohibits any activities on any of the boards. Sometimes though at procedure calls or during changing address spaces, we observed glitches on some of the lines. Usually the glitches were small but there was a possibility that sometimes they might get big enough to cause a trouble.

Table 2: SJ board Identification - Version 1

A13/D2	A12/D1	A11/D0	Function
0	0	0	No one board is selected
X	X	X	Unique SJ board ID
1	1	1	All boards are selected

In the second implementation the access to a particular board is divided in two parts. First the board is selected and after that its space is accessed in subsequent instructions. The board address is passed over the data bus using D0, D1, and D2 data lines. The result of the comparison with the ID jumpers is latched and used to enable the board for subsequent operations until the board is deselected. Once selected the board is accessible as a conventional memory location. The A13, A12 and A11 address lines are used to choose different selection options, Table 3. This scheme avoids eventual glitching problems, since the actual board access is separate in time from the board selection. Note the broadcast feature, which permits special subsets of the boards to be selected.

Table 3: SJ board identification - Version 2

A13	A12	A11	Selection Function
1	1	1	Select all unit; data bus ignored
1	1	0	Select upper half units; data bus ignored
1	0	1	Select lower half units; data bus ignored
1	0	0	Select single unit; D2, D1, D0 have the unit ID
0	0	1	Select FPGA location
0	0	0	NOP ; A7-A0 have a selected-board location

Each SJ board is mapped to 256 addressable locations of DS80C320 data memory. The first 128 addresses are used to access the SJCP scan chains and for control purposes. A complete list of the available functions is given in the Appendix. The second 128 addresses are used to access an SJCP internal SRAM block. The access to any location takes place within the timing of a single stretched data memory access instruction.

All SJ boards have a common reset line controlled by the system reset. Each SJ board has a dedicated interrupt line which is connected directly to one of the interrupt inputs of Dallas microprocessor. The interrupts are currently disabled and the handling is done by polling.

The physical interface between each board and the microcontroller is realized through pairs of FPGAs which perform decoding and bus control functions. The details are discussed in the following sections.

2.2.1.4. System Clocks

Clock generation is critical for proper system performance. We are using a Cypress CY7B992 Programmable Skew Clock Buffer (PSCB) to generate the necessary clock sequences. PSCB is capable of generating four clock sequences with possible relative shift of +/-180 degrees. One is used as clock input for DS80C320 and SCSI chips and two others are providing PHI1 and PHI2 clock sequences for the SJ boards. The skews are jumper-configurable, Table 4.

As we found out the relative timing between PHI1, PHI2 and the microcontroller clock is decisive for the proper system performance. PHI1 and PHI2 should be shifted at 90 degrees. This provides four equal time intervals timing different activities in the SJCP machine cycle. In reality though the situation is different, the time intervals could differ by as much as 40% with respect to the reference clock. The reasons for this are the PSCB resolution, ringing on the clock lines, propagation delays and different rising and falling times.

Table 4: System Clocks Configuration

Jumpers	Function
1F0, 1F1	Jumpers fixed (Open) Reference Clock
2F0, 2F1	DALLAS System Clock, respectively controlling ALE, READ and WRITE timing
3F0, 3F1	SJ System Clock, PHI1
4F0, 4F1	SJ System Clock, PHI2

At the same time PHI1 and PH2 should be synchronized with the DS80C320 bus control signals like READ, WRITE and particularly ALE. ALE triggers a finite state machine in SJCP which is going through 16 states (8 states for internal DPM access). Everything should be timed precisely in order for the data transfer to complete correctly.

The tuning of the system clocks goes through two stages. First, we tune the skew between PHI1 and PHI2 to be as close to 90 degrees as possible. Second, we tune ALE by shifting appropriately the microcontroller system clock.

The microcontroller system clock has no effect on the functionality of the SCSI interface.

2.2.2. Board Design

We are using four layer Printed Circuit Boards. The first and forth layers are signal layers while the second and third layers are power and ground layers respectively. The microcontroller board is shown in Figure 2.2. Table5 shows the function of each jumper.

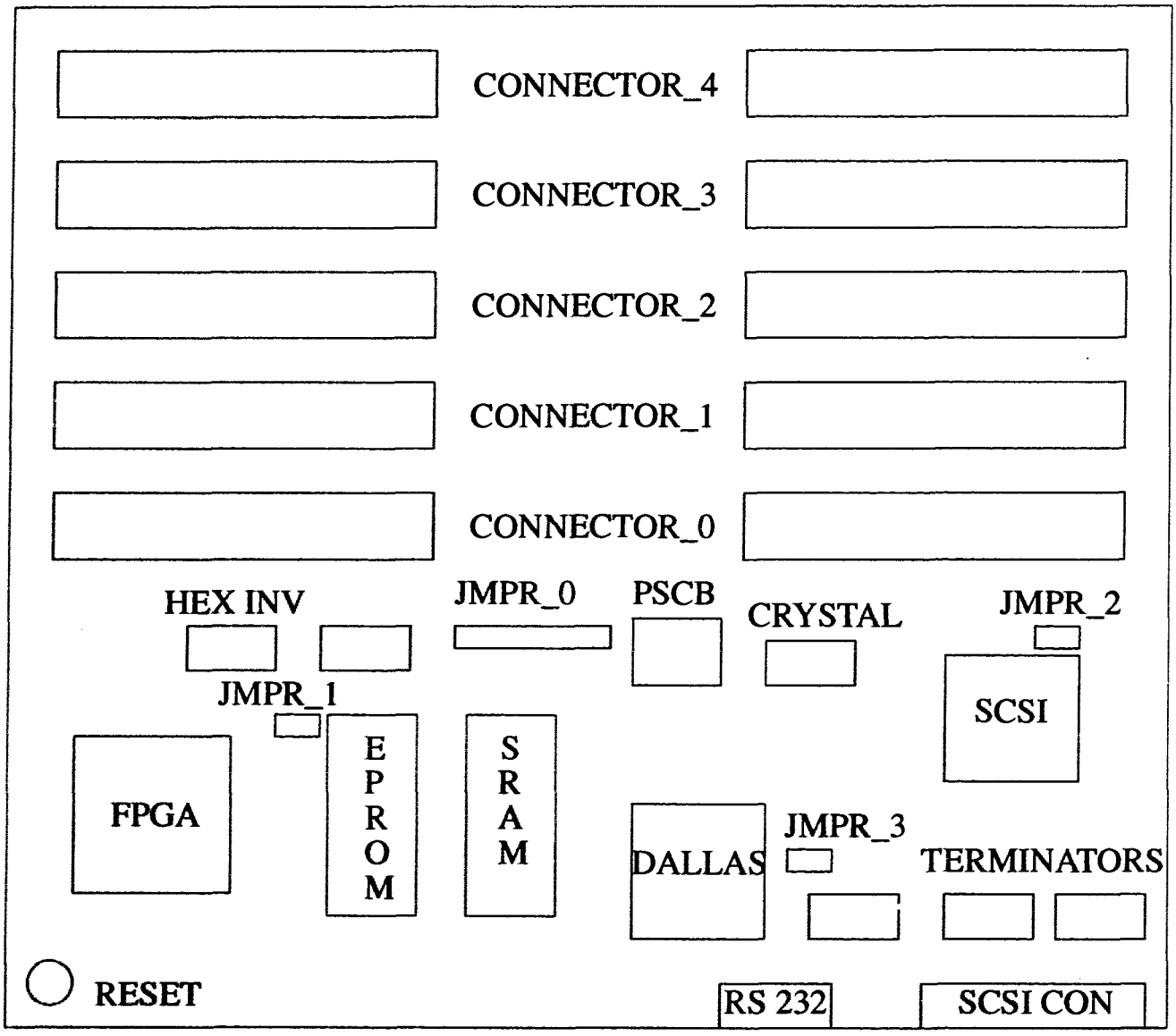


Figure 2.4: Microcontroller Board Layout

Dedicated POWER and GROUND layers help to reduce noise, crosstalk, and short circuit effects on the board. Also, it seems that one of the problems we encounter is associated with power distribution. That's why it is very important to

supply as much independent power to each component as possible.

Table 5: System Configuration Jumpers

Jumper	Function
JMPR_0	Clock Configuration Jumpers. See Table 3.
JMPR_1	Program Storage Configuration Jumper. OFF - 32k, ON - 64k
JMPR_2	SCSI Reset Line Configuration Jumper. OFF - reset from the SCSI bus, ON - system reset
JMPR_3	SCSI Interrupt Configuration Jumper. OFF - SCSI interrupt disabled, ON - SCSI interrupt enabled

The SCSI terminators are powered by the HOST, in our case by the PC through the cable. According to the SCSI specification, terminators should be powered by the SCSI bus but if necessary, local power could be supplied through a jumper.

The SJ boards attach to the microcontroller mother-board through pairs of 96-pin connectors. On the prototype we have five pairs of connectors. Each pair has a separate interrupt line. Since the interrupts are disabled, all boards have the same priority or the priority is determined by the order of polling. If the interrupts are enabled CONNECTOR_0 has the highest priority, this is where the Program Management Unit (PMU) will be plugged, and CONNECTOR_4 has the lowest priority.

2.3. SJ Processing Unit Board

SAM II Processing Unit (PU) is based on a custom 32 bit vector-oriented chip-set designed at SFU. The chip-set consists of:

- SJCP - 32-bit one-stage pipelined vector-oriented microprocessor employing 64-bit-wide high-level microcode. It is capable of executing up to 4 microoperations per clock cycle at peak performance.

- SJMI - 32-bit four-stage interleaved memory management unit. It can handle up to 64 Mbytes of interleaved memory organized in four banks.

- SJNI - 32-bit network controller, it is intended to handle direct data transfer between the Processing Units for efficient array manipulation.

- SJFP - Floating-Point Unit.

- SJIVU/SJOVU - instruction pipe control units.

Depending on its functions in the system each PU could be either a Program Management Unit (PMU) or a Data Management Unit (DMU). The PMU performs program-execution control functions, It broadcasts vector-algorithm instructions through an instruction-pipe interface. The DMUs are engaged only in data processing activities. In principle, the system could have one PMU and many DMUs. The prototype is designed to accommodate five boards altogether, one PMU and four DMUs.

2.3.1. SJ-board System Design

At the time we started building the prototype, only SJCP and SJMI chips were available so we were able to build a minimal configuration of SJ computing system consisting of the microprocessor module and data storage manager. The block diagram is shown in Figure 2.3. On the board we have reserved space for the rest of the system components and also we have intercomponent-interface pad arrays to connect the new components to the system.

2.3.1.1. Microprocessor Module

All the activities on the SJ-board are controlled by SJCP control processor. SJCP has 64k-word external microprogram storage. In future versions the microprogram storage might be on-chip. The microinstruction is 64 bits wide but actually SJCP is using only 56 bits. The other 8 bits (so called Y field) are specific for the Floating-Point Unit. The loading of the Microprogram memory takes place through the SJCP internal scan-chains. First, seven bytes are written consecutively into the instruction scan chain while the eighth byte is put on the Y BUS from the on-board FPGA. After that all eight bytes are strobed into the microprogram store at the address preloaded in the SJCP internal address scan-chain. In execution mode, the microprogram store is permanently enabled for reading, it is writable only during loading. We call this a read-mostly memory.

Access to SJCP resources is possible through an eight-bit address/data bus and a couple of control signals, CTR_0 and CTR_1, which specify the RD or WR

operation to take place. The operation could be a conventional data transfer or it could trigger a certain activity in SJCP. The lower eight address bits are latched in a on-chip address register during ALE and subsequently decoded to determine the selected function. The decoding of the control signals is shown in Table 6.

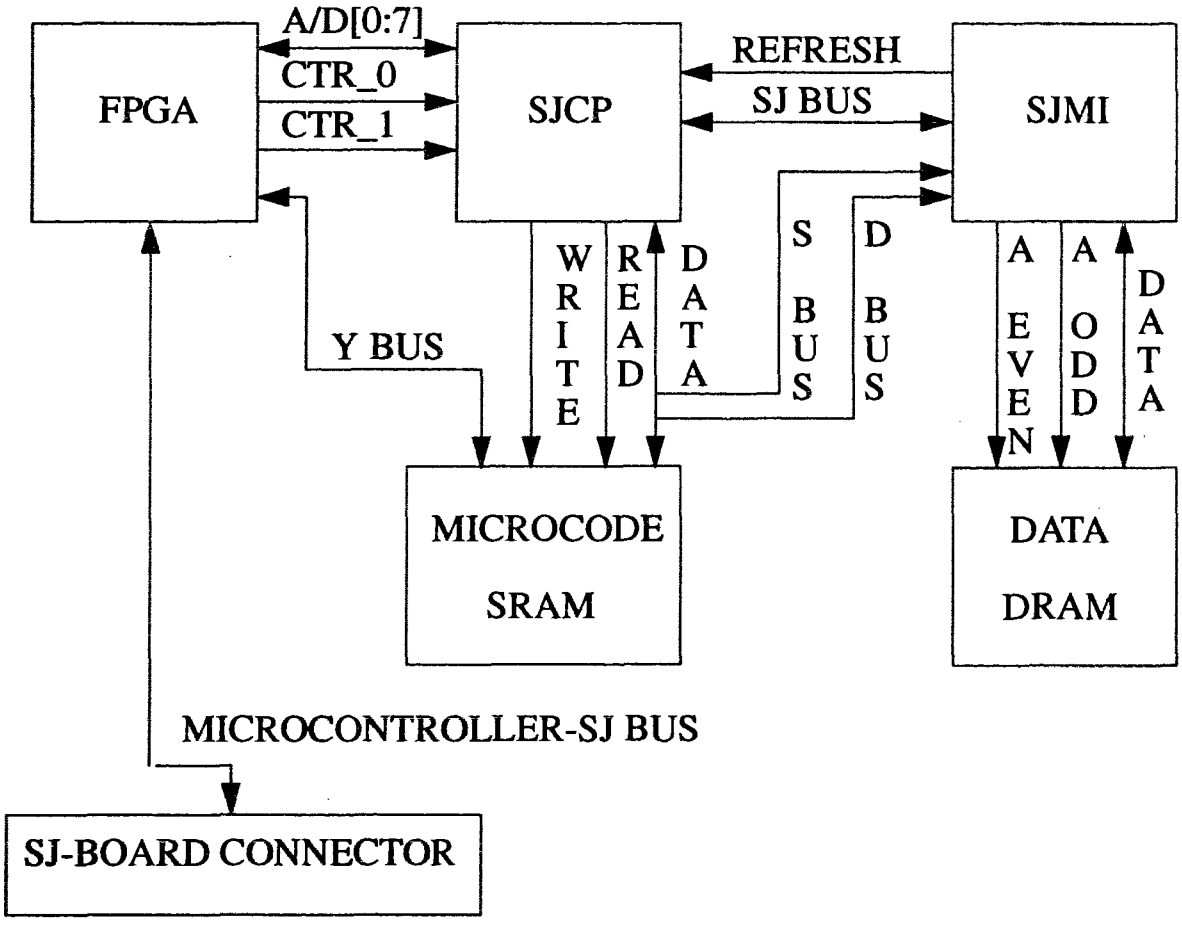


Figure 2.5: SJ-board Block Diagram

The generation of the control signals is not a trivial problem and it is discussed in detail in the next section. It is done through a pair of FPGAs which also perform bus control and decoding functions. Due to a shortage of I/O pins on SJCP, the microcontroller's three bus control signals were encoded into the two CTR1 and CTR0 signals.

Table 6: Control Signals Decoding

CTR_1	CTR_0	Operation
0	0	READ
1	0	NOP
0	1	WRITE
1	1	ALE

SJCP is connected with the Memory Manager (SJMI) by a 32-bit bidirectional bus (SJ BUS) which is used to transfer data between the two modules during program execution. SJMI is controlled by two instruction fields called source (S field) and destination (D field). SJCP stops when active REFRESH is detected, this happens when SJMI performs a DRAM refresh, and resumes when REFRESH goes away.

SJCP has 166 32-bit words of internal dual-port SRAM used as data storage or in data transfer operations between SJCP and SJMI, SJCP and the microcontroller, and SJMI and the microcontroller. This dual-port memory takes 128 addresses of DALLAS address space and a 3-bit bank register is used to access different 128-byte banks.

2.3.1.2. Memory Management

SJMI can handle up to 64 Mbytes of interleaved DRAM organized in four banks, two even and two odd, each 32 bits wide. It has on-chip refresh logic with a programmable refresh cycle. In the prototype we are using four 4 Mbyte SIMM modules for a total of 16 Mbytes.

During instruction execution, the source and destination fields of each microinstruction are passed from the microprogram storage to SJMI and other coprocessors along the S and D buses to specify the source and destination of the data transfer, if any.

2.3.2. Board Design

The board layout is shown in Figure 2.4. It is again a four-layer printed circuit

board. The first and forth layers are signal layers while the second and third layers are power and ground layers respectively.

The SJ-board connects to the microcontroller board by means of a pair of 96 pin connectors. CONNECTOR_0 is used to attach the board to the microcontroller bus while CONNECTOR_1 will be used for inter-process communications. It has three SJ ID configuration jumpers specifying the unique address of a particular board. The boxes drawn with a dashed line show the places reserved for the Instruction-Pipe Unit, Floating-Point Unit, and the Network Controller.

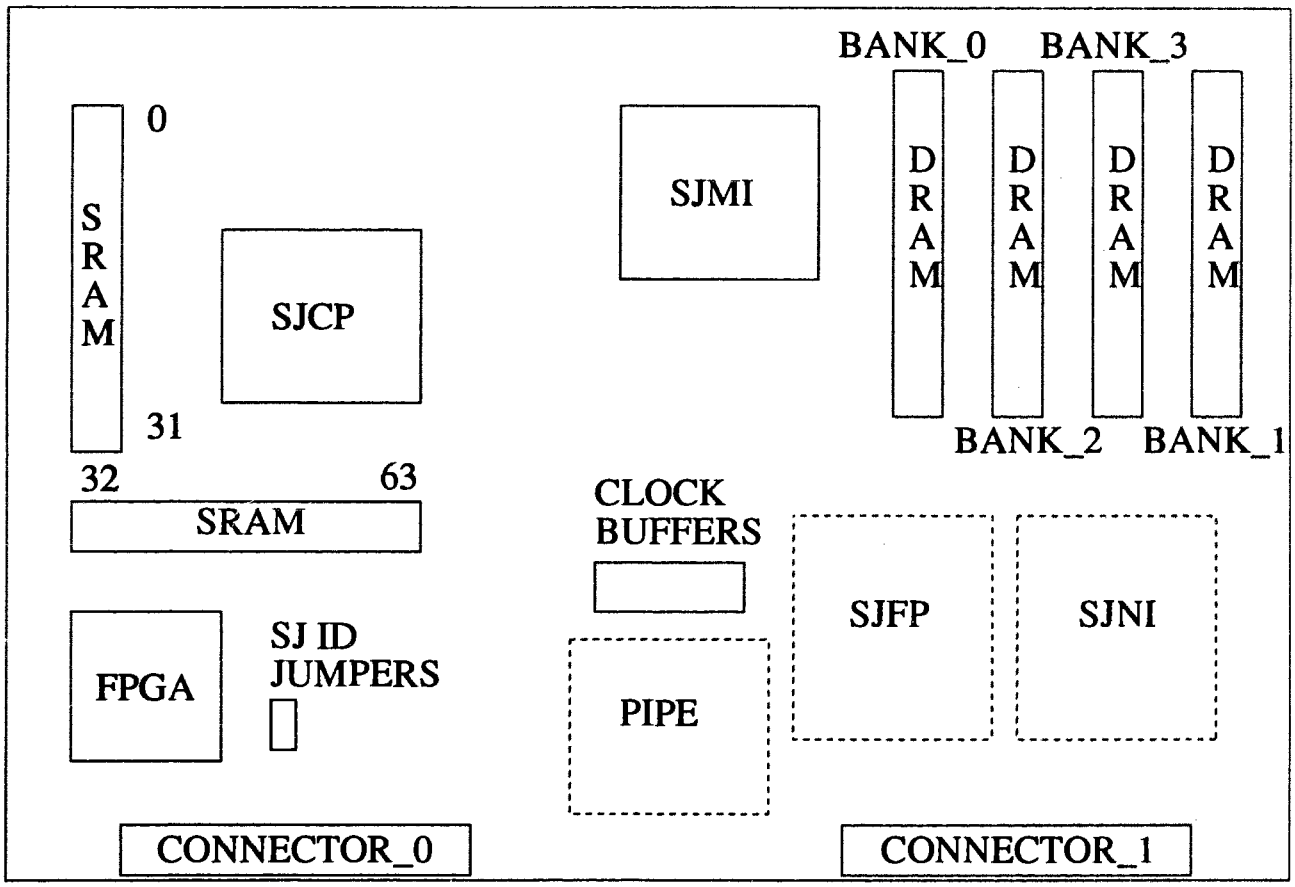


Figure 2.6: SJ-board Layout

The clock buffers are used to buffer and invert the PHI1 and PHI2 system clocks coming from the PSCB on the microcontroller board through CONNECTOR_0.

The board architecture could be improved along several lines. If we rotate the SJCP and SJMI modules in 180 degrees, this would shorten the address/data bus

traces between the FPGA and SJCP minimizing the ringing and crosstalk. We also can improve the system clock propagation delays, if we keep the SJCP clock inputs close to the bus connectors.

When we started designing the board, we used the SJCP and SJMI pin layout diagram given to us by the manufacturer. It was not until the board was ready for manufacturing, when we found out that actually we had the mirrored image of the pinout. At this point we had two alternatives: to redesign the board or to solder the SJCP and SJMI on the back side of the board. We chose the second one mainly for timing reasons. This will be fixed in later implementations.

2.4. Microcontroller - SJ Boards Interface

The microcontroller is facing the problem of how to interface efficiently multiple Processing Units. Some multiprocessor systems with large number of PUs have several levels of local buses even some custom networking which in any case involves buffering. In our case, each PU represents 256 bytes of directly addressable memory and all the transactions take place within one stretched microcontroller machine cycle. But even though the prototype has only five PUs, we still need to consider such factors like bus-lines overloading and signal propagation delays, and of course the timing.

In general, the basic functions the microcontroller-SJ-boards interface should be able to perform are:

- Address/Data Bus Control - The microcontroller and the SJ-boards are on the same physical bus. The interface logic should be able to control the direction of the bus depending on which device is driving it. Usually the bus is driven from the microcontroller towards the SJ boards. Only when we have a read from a particular PU, the bus direction is reversed. This is done to prevent the five SJ boards driving the bus at the same time, which would result in a short on the bus lines.

- Decoding - The SJ boards are in the microcontroller address space together with the SRAM and SCSI protocol chip. The interface logic should be able to identify each board uniquely in order to avoid bus contests and to generate the proper set of control signals. The decoding is done in two stages (see below please).

- Control Signals Generation - From one side we have the DS80C320 bus and on the other side the SJCP external interface. The interface logic should perform the

necessary decoding and conversions in order to generate the proper control sequences.

2.4.1. Physical Datapath

The interface logic is realized through a pair of FPGAs, one on the microcontroller board, which I will refer to as FPGA-M and one on each SJ board, FPGA-SJ. FPGA-M performs partial decoding and bus control functions and FPGA-SJ completes the decoding process and generates the necessary control signals. Since the FPGAs have limited internal resources and limited number of external inputs, the functions should be distributed evenly between them.

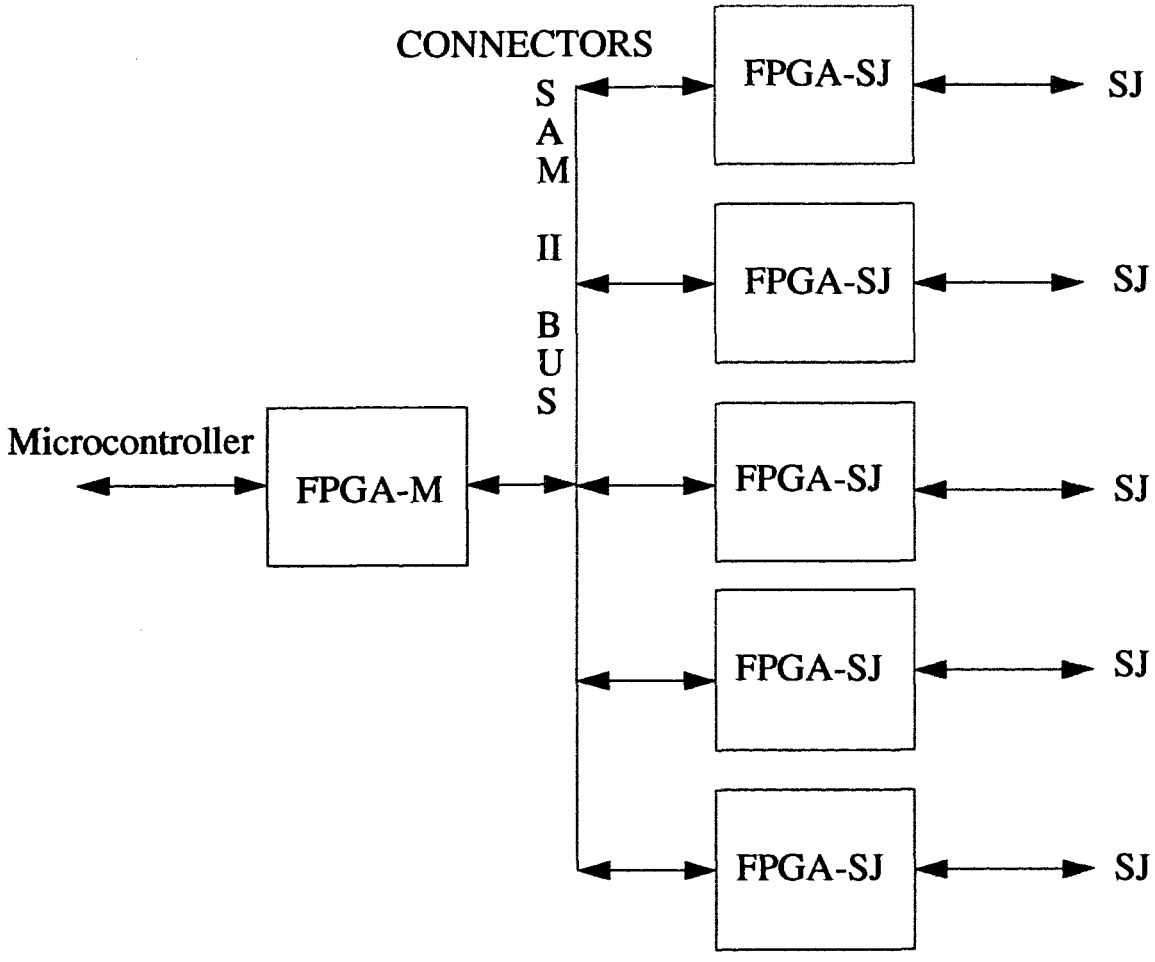


Figure 2.7: Physical Datapath

The DS80C320 bus goes to FPGA-M and after that it directly connects to

as many as five FPGA-SJ units, which expands the bus into five separate buses, Figure 2.5.

As far as the dataflow is concerned, the FPGAs are playing simply a buffering role. The bus is driven normally towards the SJ boards and only during a decoded read it reverses its direction.

2.4.2. Microcontroller FPGA Design

We have been using Intel's and Altera's FX740 coming in 68 pin PLCC package having four Configurable Function Blocks (CFB) each containing ten macrocells and having a pin-to-pin delay of 10ns. In the first implementation, FPGA-M performed some partial decoding of the address while the DS80C320 bus control signals passed through in the original timing. In the second version, improved design by Dr. Rick Hobson, we generated our own bus control signals under state-machine control. Both designs will be discussed and compared where relevant. The logic diagram of the first version is shown in Figure 2.8. The PLDSShell file and the simulations are given in the Appendix.

2.4.2.1. LSB Address Latch

DALLAS has multiplexed address/data bus. First the LSB of the address is put on the bus and after that the same bus is used for data transfer. For the proper system functionality we need to store the LSB of the address for a subsequent use in data transfer operations which could be an instruction fetch from the microcontroller program storage or data transfer to/from the microcontroller data storage or the SCSI chip (LSB is also latched in FPGA-SJ and internally in SJCP for addressing and operation decoding purposes). Eight macrocells in the FPGA-M are organized as an eight-bit register. According to the DS80C320 specification, the LSB of the address should be latched at falling edge of ALE (active high). Since the data is getting strobed into the macrocell-D-flip-flops at the rising edge of the D-flip-flop-ACLK-input, ALE should be inverted. The macrocell-drivers are permanently on, the LSB-address-lines are never floating and they are stable by the time of the data transfer.

2.4.2.2. SCSI Protocol Chip Selection

We are using A15 and A14 address lines to divide the data address space among 32k of SRAM, the SJ boards and the SCSI Protocol Chip (see Table 1). In order to enable the SCSI chip data bus drivers, both CS_SCSI and RD/WR should be active

at the same time (see Figure 2.6 for CS_SCSI logic).

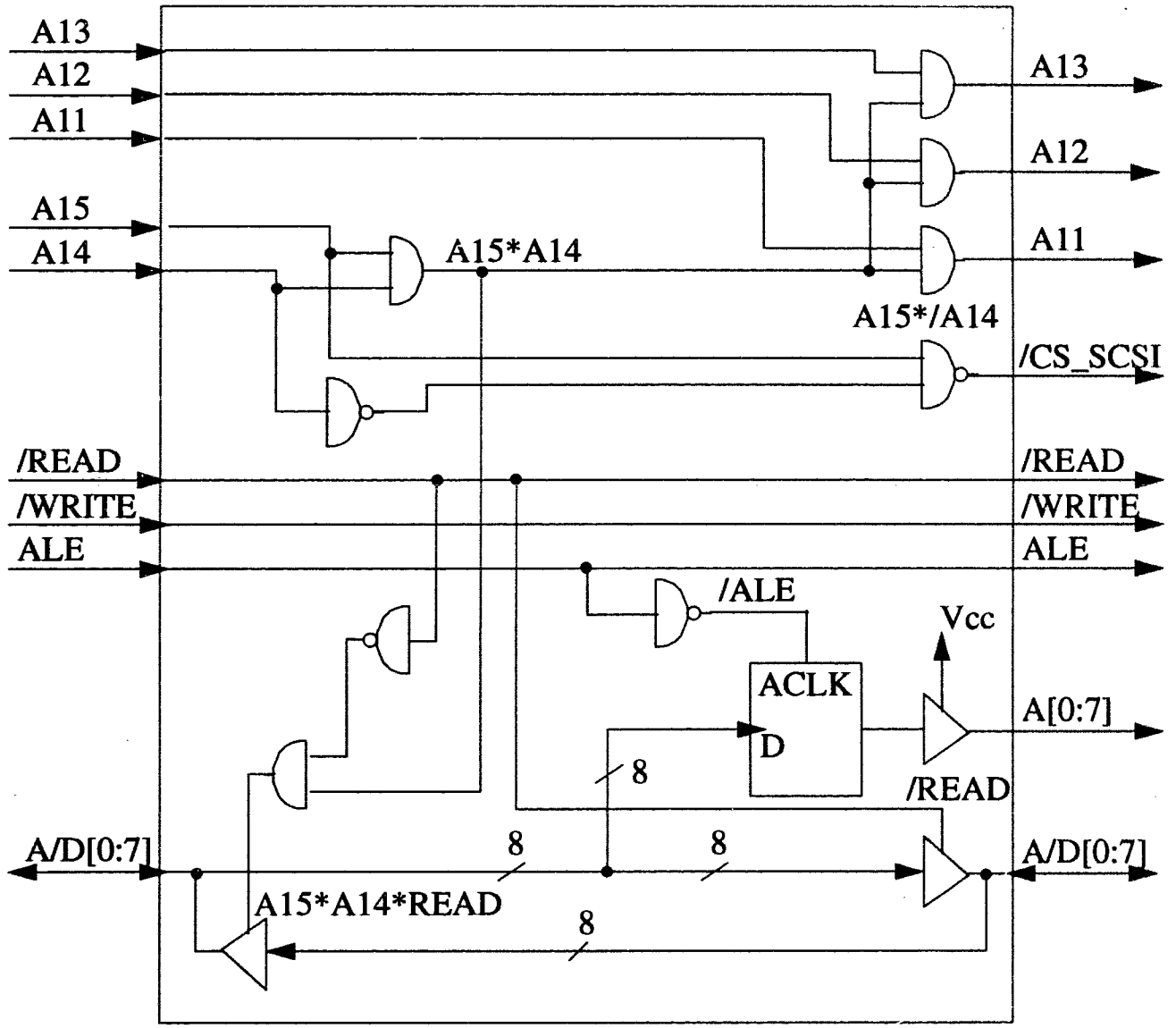


Figure 2.8. Microcontroller FPGA-M Version-1 Logic Design

2.4.2.3. READ, WRITE and ALE

In the first implementation the control signals READ, WRITE and ALE are passed through FPGA-M directly to the FPGA-SJs on the SJ boards where they are used to generate the corresponding control sequences on the selected board(s). In this case FPGA-M serves only as a buffer. The control signals reach the SJCP I/O interface in the original timing. We had some difficulties tuning the relative timing

between the microcontroller bus control signals and the SJCP clocks. Most of the time the interface was working fine but sometimes we had to retransfer the data several times. We were not sure what the problem was but to avoid potential problems the control signals logic was redesigned.

In the second implementation, we take the original bus control signals and one of the system clocks and use a state-machine control logic to generate a new set of control signals timed better with the SJCP clocks, Figure 2.9.

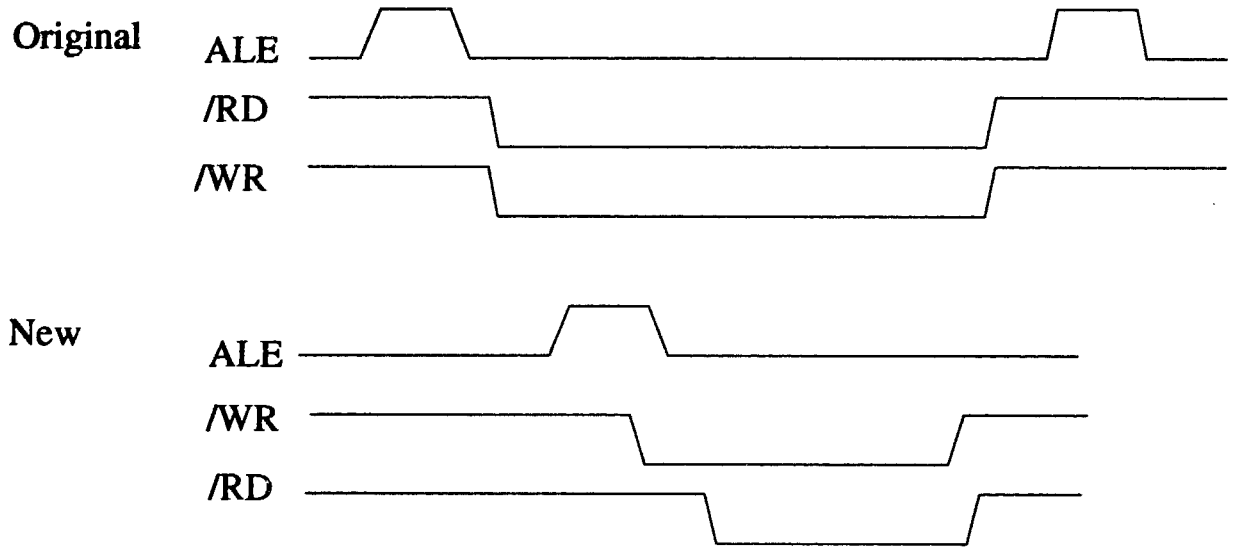


Figure 2.9: Bus Control Signals Timing

In order to be able to accommodate the new control signals in one machine cycle, the machine cycle is stretched by 8 clocks. Two goals are pursued here. First ALE is delayed to align it better with the SJCP interface logic clock. This would guarantee correct data transfer. Second, the two original signal sequences we have are NOP-ALE-RD-NOP and NOP-ALE-WR-NOP, see Table 6 for the coding scheme. All the transitions involve only one bit change, except the transition from WR to NOP where we have to change both bits. This might cause an ALE glitch and misfiring of the SJCP interface state-machine. That's why RD has been added momentarily between WR and NOP. Instead of having WR-NOP we have WR-RD-NOP. The intermediate RD state does not cause any problems.

2.4.2.4. SJ Boards Selection

High A15 and A14 select the SJ boards. In order to be able to identify uniquely five PUs, we need three more bits. In the first implementation we are passing through A13, A12 and A11 gated with the product of A15*A14 to the FPGA-SJ and use them to ID a particular board, see Table 2.

Because of timing, propagation delays, and address space change reasons, we were getting glitches on some lines. In order to avoid this, in the second implementation, we use A13, A12 and A11 only to encode the type of selection (Table 3), broadcasting, single board selection etc., and the actual ID code is passed on the data bus. D0, D1, and D2 are compared with the jumpers and the result is latched. In subsequent board access operations the high-order address bits. (A13, A12 and A11) are set to zero.

2.4.2.5. SAM II Bus

As an output from FPGA-M, we get two groups of signals. The first group consists of A[0:7] and CS_SCSI used on the microcontroller board and second includes A13, A12, A11, A/D[0:7}, /READ, /WRITE and ALE.

The second group together with the system reset line (RESET), the interrupt line (INTR) and the two clock signals PHI_1 and PHI_2 constitute a bus going to each SJ board, which we called SAM II Bus, see, Table 7.

The RESET signal is generated by the reset logic and is driven towards the SJ boards.

Table 7: SAM II Bus Signals

Signals	Direction	Function
A13, A12, A11	towards SJ board	SJ board ID/ID function select
A/D[0:7]	bidirectional	bidirectional address/data bus
READ, WRITE, ALE	towards SJ board	control signals
RESET	towards SJ board	system reset signal
INTR	towards DALLAS	SJ board interrupt signal
PHI_1, PHI_2	towards SJ board	SJ board clock signals

The INTR is coming directly from SJCP to the corresponding connector and from the connectors all interrupt signals are going to dedicated DS80C320 interrupt inputs.

The clock signals, PHI_1 and PHI_2, are coming directly from the Programmable-Skew-Clock-Buffer.

2.4.3. SJ board FPGA Design

FPGA-SJ completes the SJ-board decoding procedure and generates the bus and SJCP I/O interface control signals.

2.4.3.1. Decoding

In the first implementation the A13, A12, and A11 address lines are compared dynamically within the timing of the current instruction with the ID jumpers, Figure 2.10. The result of the comparison is used as a global enable signal to trigger different activities in FPGA-SJ. When the result of the comparison is negative the address/data bus is driven towards SJCP and the control signals are in NOP state.

In the second version, improvrd by Dr. Rick Hobson, the A13, A12, and A11 address lines are still used but for decoding the selection function, which might be broadcasting, select lower half, select upper half etc. The actual ID code is transferred over the data bus. The D2, D1, and D0 data lines are compared with ID jumpers and the result is latched in a flip-flop. On selection the board stays selected until it is deselected. This approach gives a little bit more addressing flexibility, eliminates problems due to glitching and saves a little bit of address space if we need it.

2.4.3.2. Y-Register

The external microprogram storage is 64 bits wide. The microprogram is loaded one instruction at a time. Part of the instruction, 56 bits, is loaded through the SJCP internal instruction scan-chain, seven bytes are written sequentially into the scan-chain and strobed later into the microprogram storage. For the eight byte, we need an external register accessible by the microcontroller. Since this byte belongs to the so called Y field in the microinstruction, we call this register the Y-register. Eight macrocells in FPGA-SJ are organized as the eight-bit Y-register.

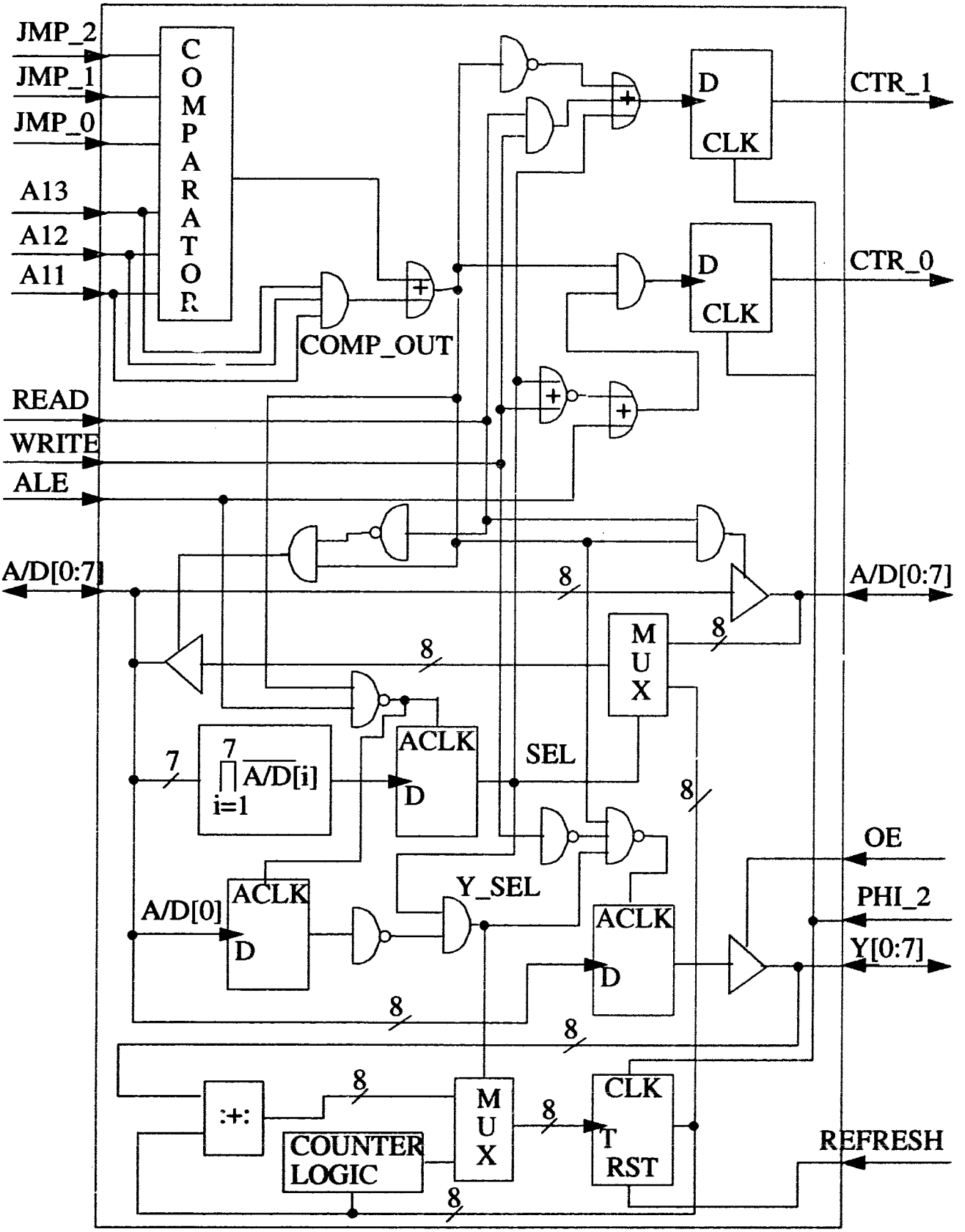


Figure 2.10: FPGA-SJ Logic Diagram

Each SJ board takes 256 bytes of DALLAS address space. The LSB of the address is used to access the SJCP resources or the Y register. One of these 256 addresses, LSB = 00H, is used to access the Y register (not all 256 values are used internally in SJCP).

2.4.3.3. Counter

In the first version, this was a real-time solution to a timing problem we might have. During program step-by-step execution, we need to know how many clock cycles we have before DRAM refresh in order to figure out if SJCP will be in condition to execute the next step. During REFRESH, SJCP stops and will not respond to a step request. As a matter of fact running at 16MHz, the probability of stepping SJCP during a refresh is less than 1%. Eight macrocells are organized as an eight-bit counter. The counter is reset by REFRESH and clocked by PHI_2. Knowing how many cycles we have between two REFRESH signals and the current value of the counter we could figure out how many clock cycles we have before the next REFRESH. The counter is also used for testing purposes as an intermediate buffer to read the Y register or the Y field of the microinstruction.

In the second implementation the counter is removed. The REFRESH signal is connected to an DS80C320 input and used to reset a programmable on-chip timer. The microcontroller checks the current value of the timer before doing a step. If there are enough clock cycles left to do a step before the next REFRESH it goes ahead, otherwise it waits. The number of clock cycles for a step can be calculated from the microcontroller step instruction sequence.

2.4.3.4. Bidirectional Address/Data Bus

Sixteen macrocells are organized as an eight-bit bidirectional tristate buffer Figure 2.8. The bus direction is controlled by COMP_OUT and READ. Normally the bus is driven towards SJCP. Only on a read, it reverses its direction. On a read the buses coming from SJCP and the counter are multiplexed. The multiplexer is controlled by SEL.

2.4.3.5. Control Signals Generation, CTR_0 and CTR_1

The external SJCP control interface consists of two control signals CTR_0 and CTR_1. These two control signals determine the type of operation we have, see Table 6. The control signals are function of several variables. They depend on COMP_OUT, SEL, READ, WRITE and ALE. In more formal way we could write:

CTR_1 = F1(COMP_OUT, SEL, READ, WRITE, ALE)
CTR_0 = F0(COMP_OUT, SEL, READ, WRITE, ALE)

After performing the necessary operations, for the final set of equations we get:

CTR_1 = /COMP_OUT + READ* WRITE + SEL
CTR_0 = COMP_OUT*(ALE + /WRITE*/SEL)

The simulations and practical measurements showed that, the generation of the control signals is correct. But in order for SJCP to function properly it should receive the right set of control signals at the right time.

Because of a signal propagation delay we were observing a glitch on CTR_0 line. The problem was that there is delay between the time the address goes away and the time COMP_OUT goes away. Ideally they should go away at the same time. But, in practice COMP_OUT goes low later and for some time it is active during the next ALE. That is why we were receiving a glitch on CTR_0 at the beginning of the next ALE.

After observing the timing of the system clocks, we used PHI_2 to strobe the control signals into D flip-flops. In this way we avoided the glitching without affecting the relative timing.

Another problem we had was related with the relative timing between the control signals and SJCP internal clocking. The control signals timing is determined by the DALLAS CPU clock which doesn't necessarily have to agree with the SJCP timing requirements. We had to shift the DALLAS clock in order to tune the control signals generation timing and particularly ALE.

2.5. Summary

The board-level hardware implementation of SAM-II took a little bit more than a year. It should be noted that the SJ chip set designed at the VLSI Laboratory under the leadership of Dr. Rick Hobson provides excellent hardware supported testing and debugging features. I designed and populated the microcontroller and processing unit printed circuit boards, debugged fully the microcontroller and partially the processing unit, debugged the SCSI interface hardware and firmware, designed the first version of the FPGA-based interface and wrote a number of testing and debugging routines. Dr. Rick Hobson designed the second version of the FPGA-based interface logic, partially debugged the processing unit board and wrote a number of processing unit testing routines.

Chapter 3: Software Design

3.1. Microcontroller-Host Interface

Using an embedded system as a vector arithmetic coprocessor imposes the necessity of having a fast way to transfer data between the host and the embedded system in order to achieve a reasonable system performance. From application program point of view, the overall data processing time consists of several parts:

- time to transfer and load SAM II executable
- time to transfer the data from the host to the embedded system
- time to process the data
- time to transfer the result from the embedded system to the host.

One can see that there is a substantial data traffic going on between the host and the embedded system, particularly when we are talking about processing big arrays of data. Sometimes we don't really need to process any data but rather we perform some system configuration, control or testing activities which involve data transfer in both directions. In any case in order to create the feeling of real-time data processing and control, we need a fast way to interface the embedded system. The ideal case would be if we can transfer data at the speed of the local bus but with increasing the number of processors, it would be difficult to accommodate the system in a standard PC box.

For interfacing purposes, we are using eight-bit-wide fast SCSI 2 interface. In this case the embedded system is treated as a conventional SCSI device hooked up on the SCSI bus, pretty much like a hard disk drive. SCSI 2 allows transfer speeds of up to 5Mbytes/sec. The limitation on the transfer rates in our case is coming from the microcontroller clock rate which determines instruction execution times and device driver timing respectively. Running at 16MHz we can get a transfer rate of about 0.3Mbytes/sec. Eventually if the code is optimized, we could get about 0.5-0.6 Mbytes/sec.

3.1.1. SCSI Interface Principles

SCSI stands for Small Computer Systems Interface and was originally designed to interface block-oriented hard disk drives. Nowadays, SCSI is getting more and more popular because of its reliability and speed and it is used to interface all kinds of peripherals. Detailed discussion of SCSI specification can be found in the

specialized literature [1, 2, 3]. Here, we will focus on points which are relevant to interfacing an embedded system.

3.1.1.1. SCSI Bus Configuration

A typical SCSI bus configuration is shown in Figure 3.1. Up to eight devices could be connected on the SCSI bus. We could have one or more host computer system adapters and one or more peripheral controllers on the same SCSI bus.

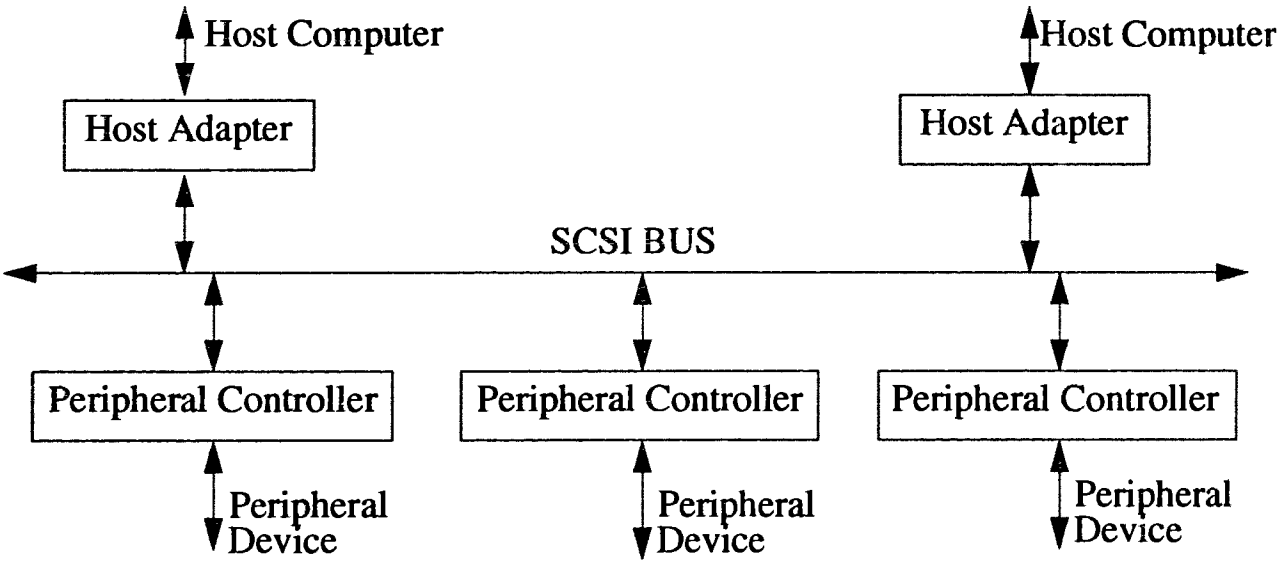


Figure 3.1: SCSI Bus Configuration

Any two devices connected to the SCSI bus can communicate. A peripheral could communicate with a peripheral, host could communicate with another host and a host could communicate with a peripheral. The devices capable of initiating a transaction are called initiators. Usually the host adapters are initiators and the peripheral controllers are targets. The initiator initiates the transaction by sending a request to the target and after that the target controls all the activities and the completion of the transaction.

3.1.1.2. SCSI Bus Phases and Phase Sequences

During a SCSI transaction the SCSI bus goes through several phases (states). The phases follow a certain order depending on the configuration and type of transaction.

Bus Free Phase - This phase indicates that no SCSI device is currently using the

bus.

Arbitration Phase - This is an optional phase which permits a device to gain control of the SCSI bus as an initiator or a target. This phase is necessary when we have more than one initiator on the bus.

Selection Phase - This phase permits an initiator to select a target to perform a certain function.

Reselection Phase - This is an optional phase that permits a target to reconnect to an initiator after the target has disconnected from the SCSI bus.

Command Phase - This phase allows the target to request the command information from the initiator.

Data Phase - This is the data transfer phase.

Status Phase - During this phase the target requests that status information be sent from the target to the host.

Message Phase - This phase allows multiple messages to be sent in both directions during any other phase.

An example of a SCSI transaction phase sequence is shown in Figure 3.2.

Bus Free	Arbitration	Selection	Command	Data	Status	Command Complete	Bus Free

Figure 3.2. SCSI transaction phase sequence

After the Bus Free phase all the initiators which want to take control of the SCSI bus arbitrate for the bus. The initiator with the highest priority takes the bus and gets into Selection phase to select the corresponding target. After the selection, the target requests the command information during Command phase. The data is transferred during the Data phase. After the data transfer, the target sends status information and Command Completion message to the initiator. The Command

Complete phase completes the transaction and the bus gets into Bus Free phase.

3.1.1.3. Command Description Block (CDB)

After the selection, the initiator transfers to the target several bytes of control information in a Command Descriptor Block (CDB), specifying the operation to be performed. The CDB could contain up to 12 bytes. We are using Group 0 CDB of 6 bytes, this is determined by the host adapter, see Table 8.

Table 8: CDB format for SEND

byte	Function
0	SEND operation code (0Ah)
1	LUN (unused = 00), Reserved (00)
2	Transfer Length (MSB)
3	Transfer Length
4	Transfer Length (LSB)
5	Reserved (00), Flag, Link

The first byte specifies the operation. In this case the operation is SEND and the data is supposed to be transferred from the initiator to the target. LUN is a three-bit field specifying the Logical Unit Number of the target. Bytes 2, 3 and 4 specify the length of the transfer. The last byte is reserved and is only used when we have linked commands.

The format of the CDB has been designed to serve the purposes of hard disk block data transfer. What we are interested in is the speed and reliability of SCSI interface and we don't really need to stick with the conventions of hard disk block data transfer protocols. We are using the fields in the CDB to transfer control information relevant to SAM II functionality and activities. One should be careful though, since some chips have embedded intelligence and they decode automatically certain fields in the CDB on the receive.

After receiving the CDB, the target decodes the operation, determines the data transfer length and takes over the SCSI bus controlling all the activities around the data transfer and transaction completion.

3.1.2. SCSI Interface Drivers

On the SCSI bus we have only two devices - the PC host adapter and SAM II. By default, the host adapter has the highest priority 7 and is configured as an initiator. SAM II has the lowest priority 0 and could be configured as an initiator or as a target but since the host adapter can only be an initiator, it is hooked to the bus as a target.

3.1.2.1. Host SCSI Driver

On the PC side we have ADAPTEC AHA-1540 intelligent host adapter card based on the Intel 8085 microprocessor [4, 5]. The AHA-1540 provides a multitasking interface between PC/AT bus and SCSI bus supporting maximum asynchronous SCSI rate of 2.0 Mbytes/sec and synchronous transfer rate of 5 Mbytes/sec. The AHA-1540 is configured to use interrupt channel 11 and DMA channel 5 with 10 Mbytes/sec burst data rate.

We are using the ASPI (Advanced SCSI Protocol Interface) [6] to access the resources of AHA-1540 and to control the activities on the SCSI bus. The ASPI provides a protocol to submit I/O requests to the host adapter specific ASPI manager. Usually, there is a separate ASPI manager written for each host adapter which is hiding the hardware from the application programs and SCSI drivers. Once the ASPI manager is loaded it becomes a part of the operating system by intercepting certain system calls (including DOS interupt 21H) and the SCSI drivers integrate each type of SCSI device into the operating system through ASPI independent of the installed hardware.

Special data structures called SCSI Request Blocks (SRB) are constructed by drivers and application programs to access the services of the SCSI driver layer. First the SRB is constructed in the application program address space and after that a pointer to the SRB is passed as an argument to a subroutine calling ASPI. All control information necessary to perform a certain SCSI operation correctly is put together in the SRB. Different SCSI services (Commands) have different SRB formats.

The SRB format for the SCSI I/O Command is given in Table 9. The first byte in the SRB is the Command Code, for performing an I/O operation it is 02h.

The Status byte shows the status of the current SCSI transaction.

Table 9: SCSI I/O Request SRB

Offset	# of bytes	Description	R/W
00	01	Command Code = 02h	W
01	01	Status	R
02	01	Host Adapter Number 07h	W
03	01	SCSI Request Flags	W
04	04	Reserved For Expansion	
08	01	Target ID 00h	W
09	01	LUN	W
10	04	Data Allocation Length	W
14	01	Sense Allocation Length (N)	W
15	02	Data Buffer Pointer (Offset)	W
17	02	Data Buffer Pointer (Segment)	W
19	02	SRB Link Pointer (Offset)	W
21	02	SRB Link Pointer (Segment)	W
23	01	SCSI CDB Length (M)	W
24	01	Host Adapter Status	R
25	01	Target Status	R
26	02	Post Routine Address (Offset)	W
28	02	Post Routine Address (Segment)	W
30	34	Reserved for ASPI Workspace	
64	M	SCSI Command Descriptor Block (CDB)	W
64 + M	N	Sense Allocation Area	R

The SCSI Request Flags specify the direction of the data transfer also, they determine if the length of the data transfer is to be checked.

The Data Allocation Length is a four-byte field and gives the length of the data transfer.

The place where the data is to be found for sending or to be stored on receiving is given by the Data Buffer Pointer fields.

The Host and Target status bytes show the status of the Host Adapter and the Target. The Target status byte is the byte sent by the Target at the end of the SCSI transaction.

At the end of the SRB, we construct the SCSI Command Descriptor Block, in our case it is a six-byte field.

The currently supported SCSI bus commands (SCSI driver services) are given in Table 10.

Table 10: ASPI Command Codes

Command Code	Description
00h	Host Adapter Inquiry
01h	Get Device Type
02h	Execute SCSI I/O Command
03h	Abort SCSI I/O Command
04h	Reset SCSI Device
05h	Set Host Adapter Parameters
06h	Reserved For Target Mode
07h - 7fh	Reserved For Future Expansion
80h - ffh	Reserved For Vendor Unique

ASPI allows linking SCSI requests by constructing several SRBs, each SRB has a pointer to the SRB corresponding to the next command to be executed.

An example of calling the ASPI manager in order to execute a SCSI command is given in Figure 3.3.

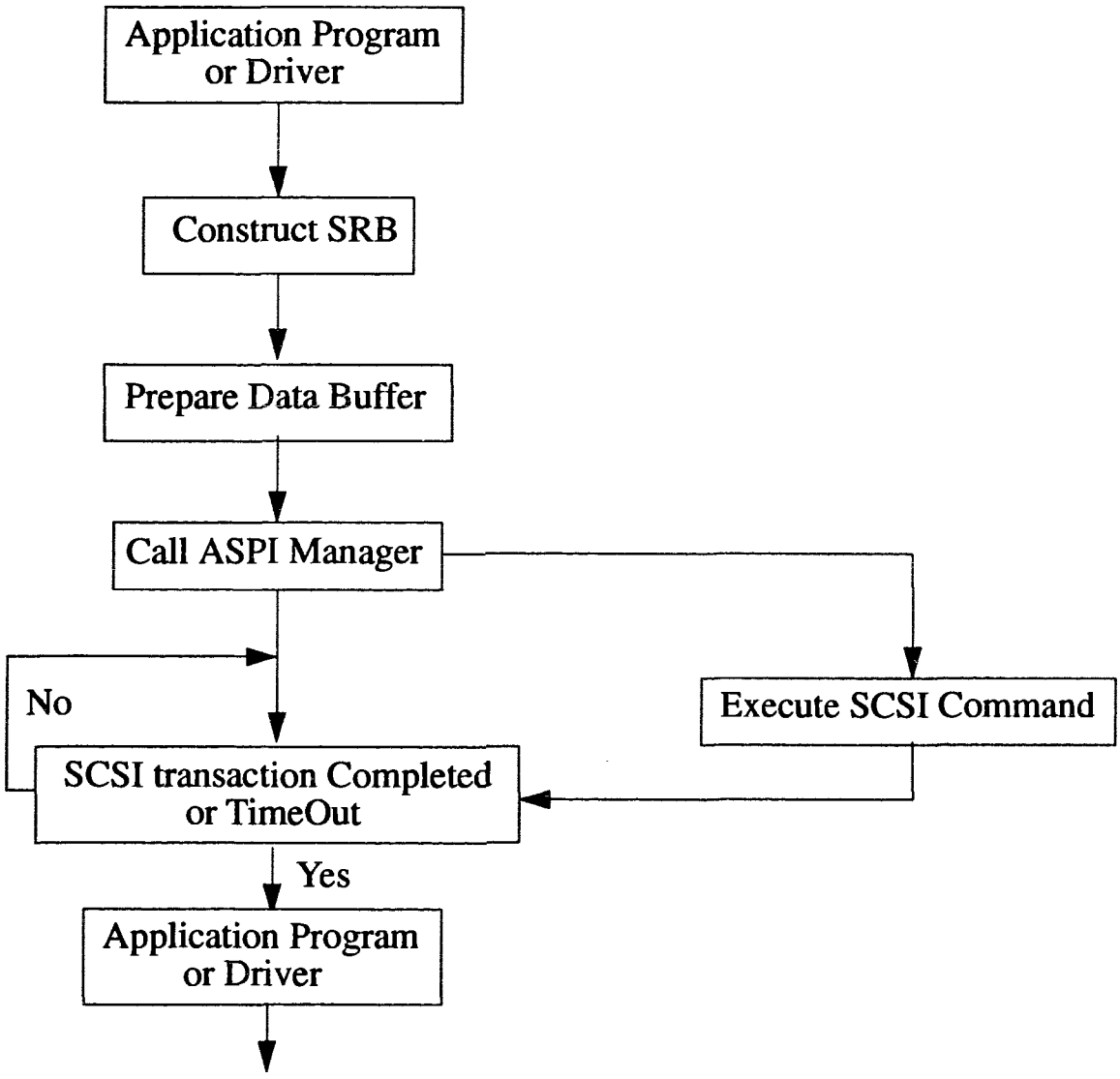


Figure 3.3: Calling ASPI Manager

First the Application Program constructs SRB and prepares the Data Buffer which could be reserving a space for the data to be received or filling the buffer with data to be sent. After that it is calling the ASPI Manager which sends the request to the Intelligent Host Adapter. From this point, the SCSI transaction is processed completely independent from the main CPU which continues with executing the Application Program code. On the SCSI Command Completion, the Host Adapter will transfer the data directly into the SRB and Data Buffer allocated address spaces through DMA channel 5. Meanwhile the Application Program keeps checking the status bytes in the SRB until the transaction completes or times out.

3.1.2.2. Microcontroller SCSI Driver

The SCSI interface is built around the Advanced Micro Devices Enhanced SCSI-Bus Interface Controller Am33C93A [7]. The SCSI Controller takes 32 locations of DALLAS addressable memory and the SCSI Bus operation is controlled by writing commands and reading data and status information to/from the SCSI-Bus Controller registers. With respect to the instruction set complexity, the SCSI-Bus Controller can execute two types of commands: level I commands performing low-level SCSI bus control, and level II or combination commands realizing high level control. The combination commands radically reduce interrupt handling responsibilities of the main processor.

One complete SCSI transaction consists of several stages. Usually, after each stage an interrupt is generated to indicate the completion of the stage. During one SCSI transaction, the microprocessor must serve several interrupts before the transaction is completed successfully. Depending on how the SCSI transaction is managed - by handling interrupts or by polling status registers, the transaction is atomic or it is interrupted by side activities, there are several approaches to design a driver.

3.1.2.2.1. Conventional Hard-Disk Drives Architecture

Originally, SCSI was designed to serve the purposes of a high-speed Hard-Disk data block transfer. In this particular case, the driver is managing a predictable and passive device like the hard disk, everything is defined from the very beginning and the driver is written as a devoted SCSI bus server. Also, on the SCSI bus we could have several SCSI devices and sometimes a given transaction needs to be interrupted for a while and resumed later. Under these conditions the implementation of the drivers employing interupt handling is pretty straightforward and resembles pretty much a big "CASE" statement. Everything a driver has to do when an interrupt comes is to recognize where in the SCSI transaction it is right now and this will determine what to do next. Depending on the level of implementation of the SCSI driver hardware and the chip set used we might have up to a couple of hundred possible states. For example, if we want to write an interupt driven driver for Am33C93A (this is an intelligent SCSI bus protocol chip), we need to be able to check for about 150 different states.

The block structure of an interupt driven SCSI driver is given in Figure 3.4. When an interupt comes the processor reads the Interupt Register and gets into the Interupt CASE statement. After that it reads the Phase Register to determine the

phase where the current command has stopped and executes the corresponding function.

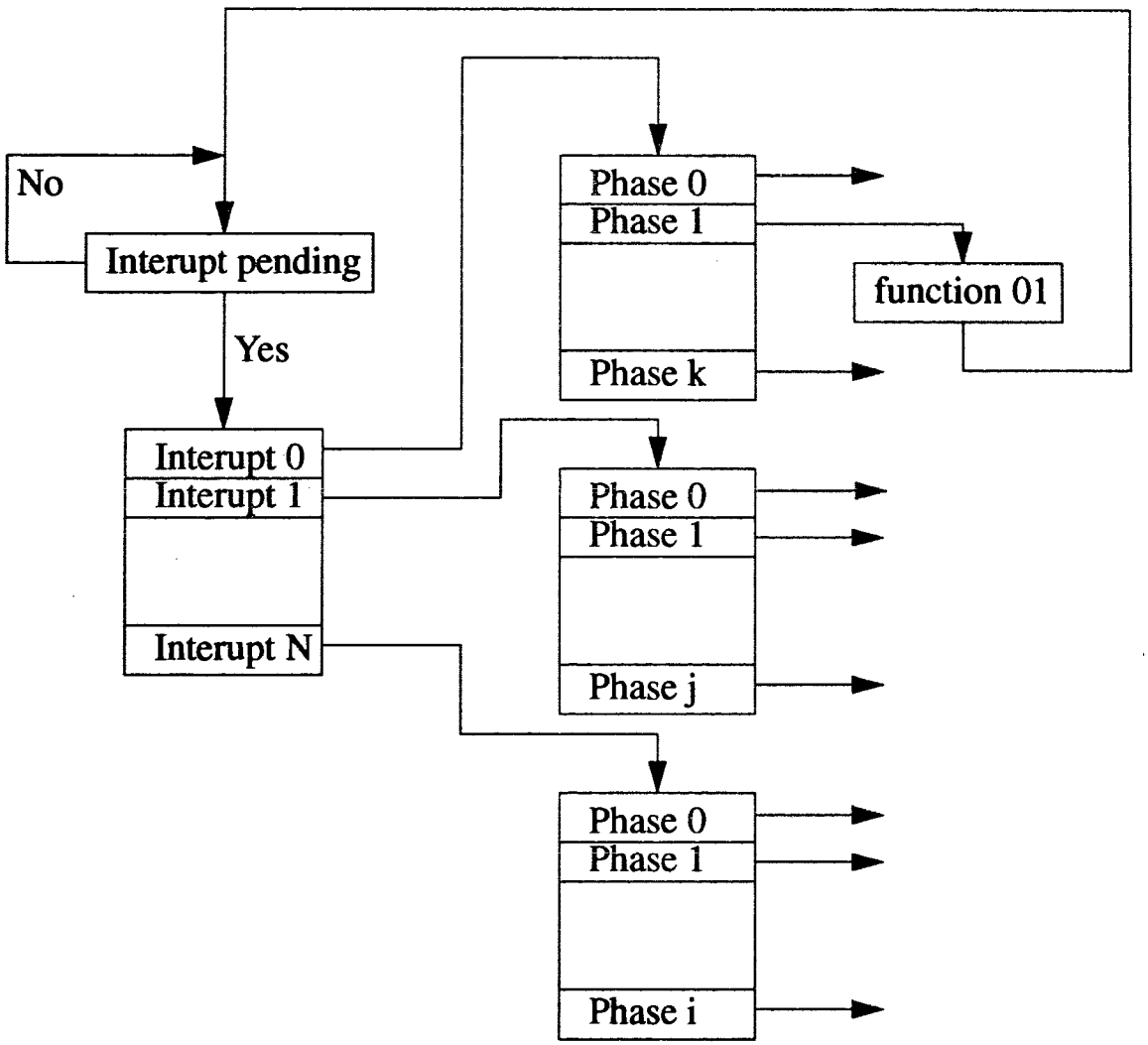


Figure 3.4: Interupt Driven SCSI Bus Driver

Obviously, the implementation is straightforward but this architecture has some shortcomings. It is relatively slow since we have to do a certain amount of checking even in the case when the interrupts are coming from the same transaction. Well, this is not a big problem considering that the main delay is coming from the hard disk mechanics. Also, code repetition is possible. Some optimization is possible considering the requirements of the specific application. The main complications are coming from handling exceptions and bus error states. In general this architecture gives relatively good performance and it doesn't represent any challenge from software design point of view.

3.1.2.2.2. SAM II SCSI-Bus Driver Architecture

In designing the SCSI driver we have taken into account several features specific for our application. First we have only two devices on the SCSI bus, the host adapter connected as an initiator and SAM II connected as a target. This eliminates the necessity of having Arbitration at the beginning of the SCSI transaction since, we have only one device connected as an initiator and it can take the bus any time it wants.

What we want from the SCSI interface is to transfer data at high speed. We don't need all the features of the SCSI protocol related to interfacing hard disk devices. We are interested in transferring data in both directions and for this purpose we need only two commands Send data and Receive data. We will be using the fields in the Command Descriptor Blocks to encode our own commands and to define our own protocol as much as possible. We can not completely customize the CDB because the host adapter and the AMD SCSI chip react automatically to certain codes which is critical for the proper system performance. For example the AMD SCSI chip might disconnect automatically from the SCSI bus on a Read after the Selection.

Since we have only two devices on the SCSI bus, one initiator and one target, there is no contest for the bus and the target doesn't need to disconnect from the bus. Some of the hard disk operations take relatively long time and the target after receiving the CDB disconnects from the bus allowing other devices to use it, while it is executing the required command independently of the host adapter. After the command is executed, it reconnects to the bus and completes the transaction, Figure 3.5.

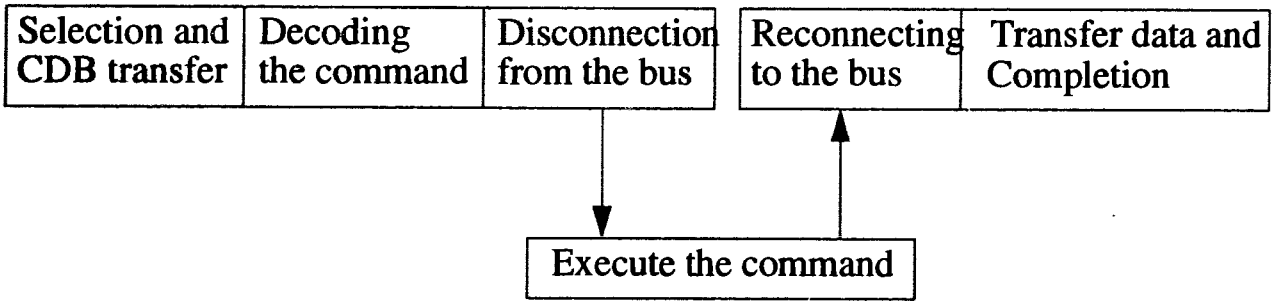


Figure 3.5: SCSI Command Execution Through Disconnection

In our case we don't need to disconnect from the bus and we can complete the transaction in one shot. Besides that, since the SCSI protocol is using hand-shaking

we can accommodate all SAM II activities within the SCSI transaction without any

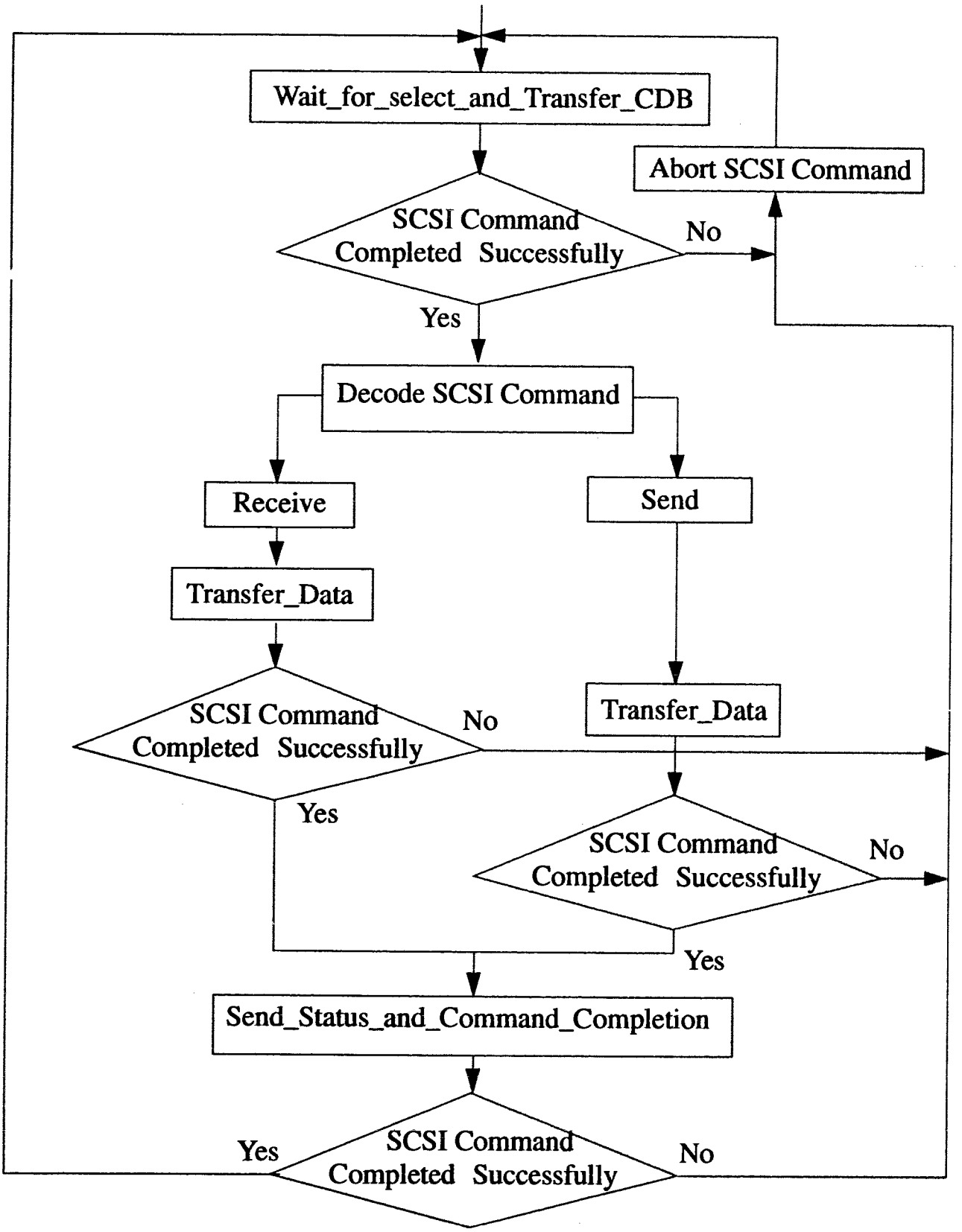


Figure 3.6: SAM II SCSI Driver Architecture

complications or problems.

On Power-up the SCSI Protocol Controller is configured for asynchronous data transfer, DMA channel signals are negated and the internal register address is latched at falling edge of ALE. There is no explicit Target configuration, its status on the SCSI bus depends entirely on the commands it executes.

The SCSI transaction goes through three major stages: Selection, Data_Transfer and Command_Completion. First the SCSI Protocol Controller is set to wait to be selected by the host adapter. If the Selection is successful, the driver decodes the SCSI command to be executed, Send or Receive, and the SAM II operation to be performed. Actually, the term SAM II operation stands for a pretty complex piece of software and will be discussed later. For now, with respect to the SCSI bus functionality, it is important only to point out that, while the microcontroller performs certain activities the host adapter is hanging waiting for a reply by the target.

The SCSI command to be executed specifies the direction of the data transfer. Send for the microcontroller means that, the microcontroller should send data to the host adapter. Usually, the microcontroller first executes a SAM II operation and after that sends the result back to the host adapter within the timing of the current SCSI transaction. In case of Receive, the microcontroller first receives the data and after that does any data processing.

If the Transfer_Data stage has completed successfully, the microcontroller completes the SCSI transaction by sending Status and Command_Complete messages.

The successful completion of each stage is monitored by polling the SCSI Protocol Controller status registers. If a certain stage fails, the transaction is aborted and eventually repeated by the host adapter later.

The polling and the fact that we need to use a small arsenal of SCSI bus commands allows us to construct the driver to follow the natural timing of the SCSI transaction. This makes it faster and radically simplifies its architecture. At the same time it allows us to eliminate all time constraints and to accommodate all microcontroller activities within the timing of the current SCSI transaction.

3.2. System Software

The system software allows the end-user to get access to SAM II resources to perform different configuration and testing functions or to do certain data-processing operations. It consists of two major components - the Host Control Interface and the Microcontroller Monitor Program, Figure 3.7.

The Host Control Interface is running on the host computer and its task is to accept the user commands, to encode them and to send them over the SCSI bus to the Microcontroller for execution. After that it waits for the Command Completion and reports the result.

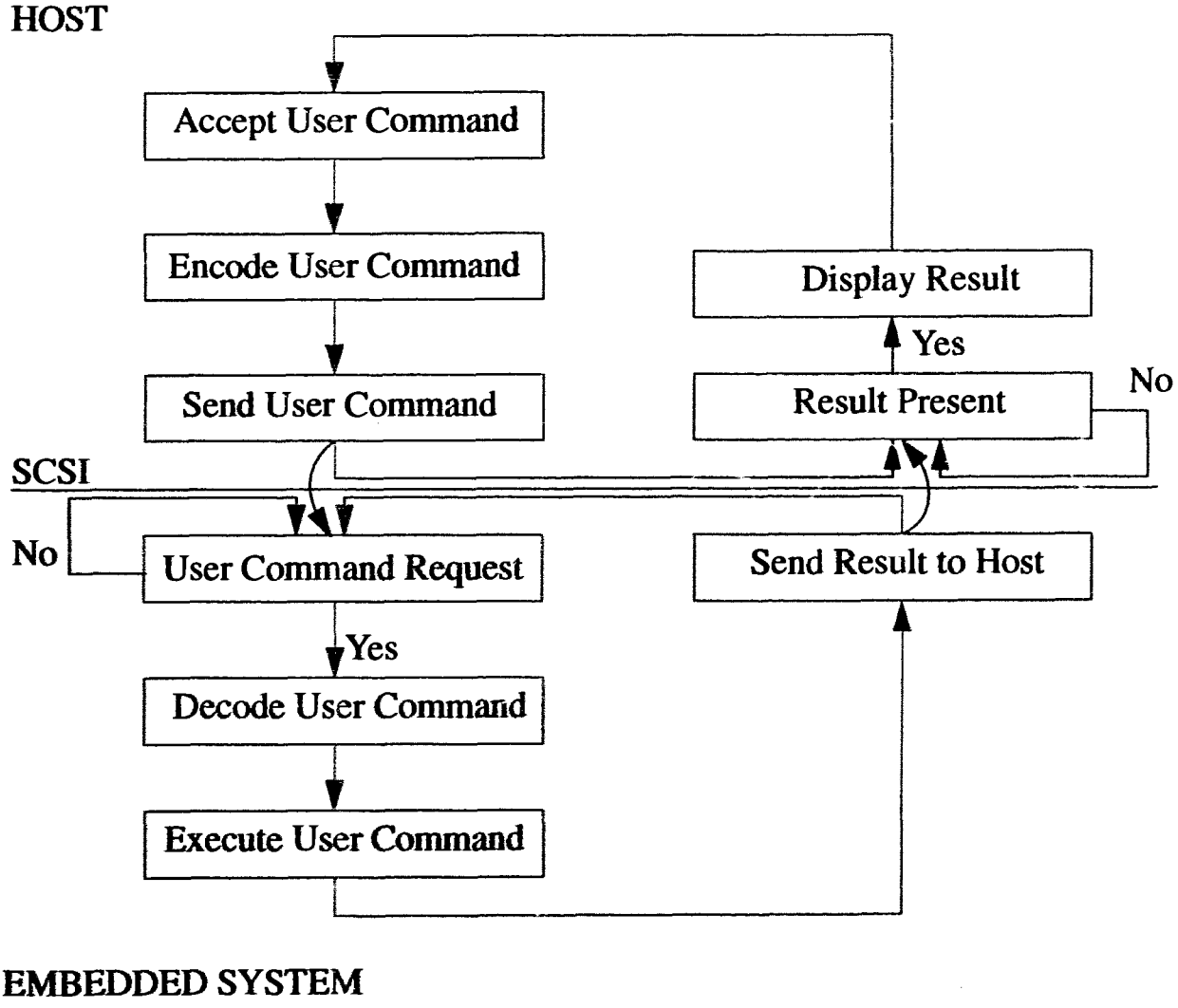


Figure 3.7: Main Command Execution Loop

The Microcontroller Monitor Program receives commands over the SCSI interface, decodes them, executes them and sends results back to the host computer. The command execution takes place within the current SCSI transaction.

The Host-Embedded System communication is command driven and the concept is very close to the idea of the Remote Procedure Call. The system should serve commands in the areas of System Configuration, System Testing, Program and Data management and Data Processing.

3.2.1. Host Control Interface

The Host Control Interface performs two basic functions - User Interface and

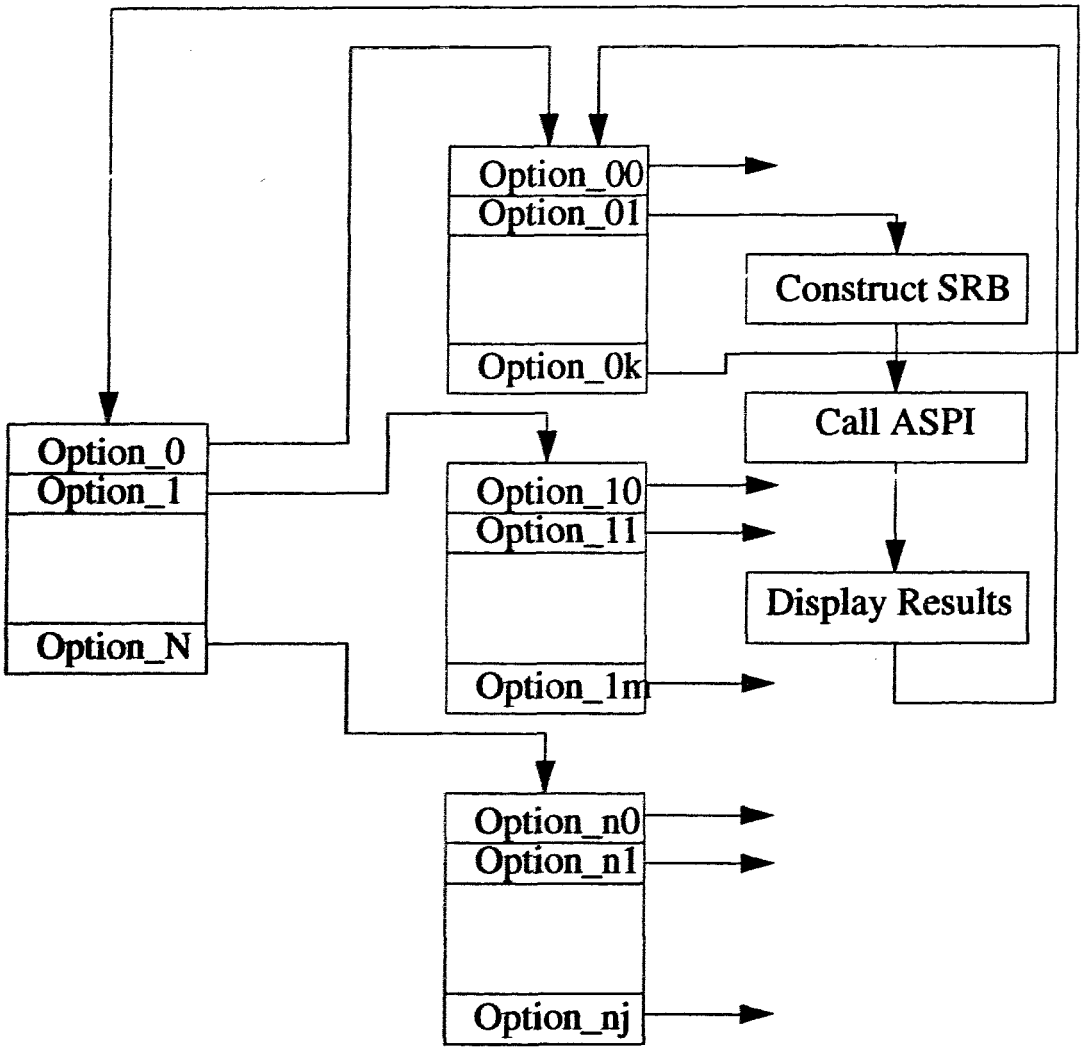


Figure 3.8: Host Control Interface Main Loop

High-level Control. It is a menu-driven hierarchically organized system, Figure 3.8. Each menu could have several options and some options could be links to corresponding submenus.

After an option has been selected the request processing goes through the same routine. The program constructs the corresponding SRB (SCSI Request Block) and calls ASPI which on its behalf calls the low-level SCSI drivers and the request is sent for processing over the SCSI bus to the microcontroller. After the request has been served, the control flow goes to the same menu or to the main menu one level up. The last option in each menu is Exit or Up.

Each option in the menu system has a unique binary code. This code is encoded in the CDB and is transferred over the SCSI bus along with other relevant control information to the Microcontroller. The Microcontroller is using it to fire up certain function(s). It is important to note that this code system is different and independent from the SCSI Standard Specification code system.

In the SCSI Bus Protocol, the operation code is the first byte in the CDB, the rest of the bytes are used to transfer other control information. In our case, we are using two standard SCSI operations - Send and Receive. We preserve the first byte in the CDB for the SCSI operation code, this is important for the ASPI software and the SCSI bus hardware, but we use the rest of the bytes to implement our own control protocol. An example format of a CDB used to transfer files into the SJCP external program storage is given in Table 11.

Table 11: Load Executable CDB Format

Byte	Code	Function	SCSI Status
0	0Ah	SEND opcode	reserved
1	XXX00000B	LUN specification	reserved
2	06h	Load Executable opcode	user definable
3	X	Total number of packets	user definable
4	X	Current number of packet	user definable
5	00h	System field	reserved

Bytes 0,1 and 5 are preserved for the Standard SCSI Specification applications. Byte 2 contains the Load Executable opcode which tells the Microcontroller that

what is coming is a part of a file to be loaded into the SJCP external program storage. Bytes 3 and 4 contain the total number of packets to be transferred and the number of the current packet respectively. For this particular operation the length of the packet is fixed and we don't need to transfer the number of bytes in a packet. The different commands might have different CDB formats. This approach allows us to accommodate our control protocol into the Standard SCSI Bus Protocol without affecting the system performance.

Table 12: Option Codes Description

Option Code	Function Description
00h	Select all units. The microcontroller selects all processing units for subsequent operations.
01h	System Test. The microcontroller tests the system memories and sends the result to the Host.
02h	Executable verification. The microcontroller keeps an image of the loaded executable in its data SRAM. It verifies the image in its SRAM with the executable in the SJCP external program storage and sends the result to the Host. Basically it sends back the number of mismatches. Currently this option is disabled.
03h	Step-by-Step Program Execution. The Microcontroller controls the execution of the instruction pointed by the SJCP Program Counter and returns the address of the next instruction to be executed and the instruction itself. If the program is running full-speed it stops the Microprocessor and executes the first instruction in the program.
04h	Full-speed Program Execution. The Microcontroller sets SJCP into full-speed program execution mode.
05h	Read 128 bytes (1 bank) of SJCP Internal Dual-Port Memory. The Microcontroller reads the specified bank in the SJCP internal Dual-Port Memory and sends it to the Host.
06h	Load SJCP Executable. An SJCP executable file is loaded into the SJCP external microprogram storage. The file is transferred in packets of 128 bytes. First the whole file is transferred into the DALLAS data storage and after that the Microcontroller is loading the image into the SJCP external microprogram storage. The address scan-chain is loaded with 0 to point the first instruction in the program. Currently this option is disabled.
07h	Switch to terminal control. The microcontroller exits the SCSI driver and switches to a terminal software connecting the embedded system to an external ASCII terminal via the serial interface. SCSI is ignored.

Table 12: Option Codes Description

Option Code	Function Description
08h	Select upper half units. The microcontroller select the upper half of the processing units. Currently there could be only one PMU in the upper half
09h	Select lower half units. The microcontroller selects the lower half of the units. Currently there could be four DMUs in the lower half.
0Ah	Select a single unit.
0Bh	Deselect all selected units.
0Ch	Load HEX record. The executable program is processed transferred over the SCSI one HEX record at a time. The executable record is transferred and loaded into the SJCP external program memory at the specified location. This allows to load programs anywhere in the memory and there is no need to keep the whole image in the microcontroller SRAM.
0Dh	Initialize a new program. The microcontroller initializes a new program for execution. This is necessary because we could have several programs in the SJCP external program storage. It initializes the starting address and clears the execution flag.
0Eh	Select a DPM bank. The microcontroller selects the DPM bank specified by the user in the currently selected processing units. The bank will be used in all subsequent operations until it is changed explicitly.
0Fh	Write a DPM bank. This The microcontroller writes the received data into the currently selected DPM bank starting from the address specified by the user.

The currently supported options are described in Table 12. The addition of new options is pretty straightforward, one just needs to keep track of the uniqueness of the option codes.

3.2.2. Microcontroller Monitor Program

3.2.2.1. Architecture

Usually, the monitor programs represent a set of routines dealing with memory management, interface communication and the servicing of different system state exceptions. It is also possible for the monitors to have a hierarchical architecture where at the bottom we have a set of basic routines dealing directly with the hardware, organized in a kernel, and on the top of the kernel some more

sophisticated system management or communication tools are developed. Regardless of the differences, the conventional monitors have two common features - they are kernel based and they use interupt handling.

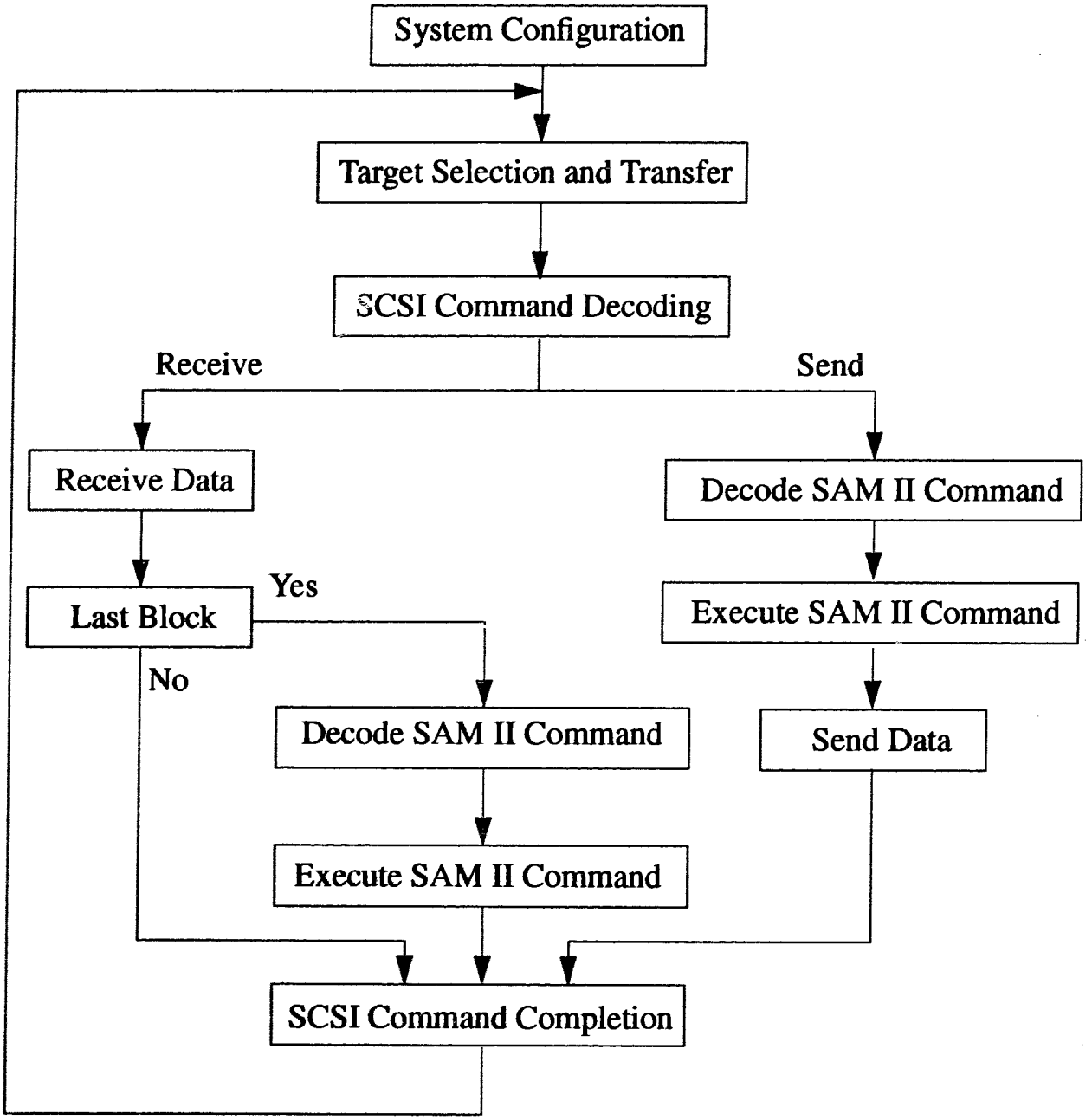


Figure 3.9: Microcontroller Monitor Architecture

It should be noted that for systems with certain level of complexity this is the only approach. But sometimes considering the specific characteristics of the system,

one might be able to simplify the architecture and to build more efficient and fast monitoring software.

During the development of SAM II Monitor Program we have considered several factors. First the Embedded System is connected to the Host by dedicated SCSI Bus Interface. This means that the priorities are fixed from the very beginning and we can communicate through atomic SCSI transactions without having any device disconnected from the bus at any particular moment during the transaction.

Second, the explicit initiator-target relationship between the Host and the Microcontroller determined by the SCSI bus Protocol imposes a certain pattern in the Host-Microcontroller communication. The presence of a pattern allows us to predict the activities on the SCSI Bus.

The decision to employ status-bytes polling instead of interupt handling made possible to fit the SAM II command execution within the timing of the current SCSI transaction, using interupt handling this would be pretty difficult to do, and it was pretty convenient during the system debugging.

Finally, the idea of a command driven Microcontroller-Host communication fits very well with the principles of the SCSI Bus Protocol.

As a result of this we managed to build the Microcontroller Monitor around the SCSI Interface driver. The Monitor is looping infinitely in the relevant chronological order through the stages of the SCSI transaction and accommodates the execution of a particular SAM II command within the timing of the current SCSI transaction Figure 3.9.

On Power-Up the Monitor performs system configuration activities. First, it initializes the SCSI and serial interfaces and after that it performs system test, testing the microcontroller memory space, FPGAs and SJCP external and internal memories. The results from the tests are stored at the beginning of the data memory and they can be requested at any time. Also the memory tests can be repeated at any time on request by the Host.

After the System Configuration, the Monitor is looping infinitely through the three basic SCSI Bus stages - Selection, Data Transfer and Command Completion. From SCSI Bus Protocol point of view, we are using two standard commands - Send and Receive. Usually, Receive is used to transfer executable and data files from the Host to the Microcontroller and Send is used to get the results or system

parameters back or simply to fire certain activities in SAM-II.

During the Selection stage SAM-II is selected by the Host and the six CDB bytes are transferred from the Host to the SCSI Protocol Controller on the Microcontroller board. After the stage is completed the Monitor decodes the SCSI command by testing the first byte in the CDB.

If the command is Send, the Monitor decodes the SAM-II command by testing the command code in the third byte in the CDB, which can be one of the Option Codes from Table 12 and executes the corresponding command. After the command is executed it transfers the result back to the Host during the Data Transfer stage. On Receive, first it transfers the data and after that executes the SAM-II command. In any case a decoding is taking place to determine what SAM-II activity is supposed to be performed, followed by an execution.

When the corresponding SAM-II command has been executed, the Monitor completes the current SCSI transaction and goes back to Selection stage waiting for Target Selection.

The currently supported SAM-II commands are given in Table 12. In order to illustrate the nature of the activities going on in SAM-II we will describe the execution of one of the commands.

3.2.2.2. An Example of SAM-II Command Execution

As an example of a SAM-II command execution, we will discuss the first variation of loading of an executable image into the SJCP External Program Storage. This was used at the initial stages of system debugging. The loading goes through two stages. First, the executable image is transferred from the Host hard-disk into the Microcontroller data memory. The image is transferred in packets of 128 bytes each. When the last packet is transferred the image is loaded into the SJCP External Program Storage. The Microcontroller data memory is byte oriented while the SJCP Program Storage is 8-byte word oriented. One SJCP microprogram word is stored in eight consecutive bytes in the Microcontroller Memory. The transfer from the Microcontroller memory into the SJCP Program Storage takes place through the SJCP Internal Scan-Chains.

The Monitor keeps receiving and storing packets into the Microcontroller data memory until it detects that the last packet has been transferred. When the last packet has been received, it means that we have the whole executable image in the

Microcontroller data memory and the loading into the SJCP Program Storage could

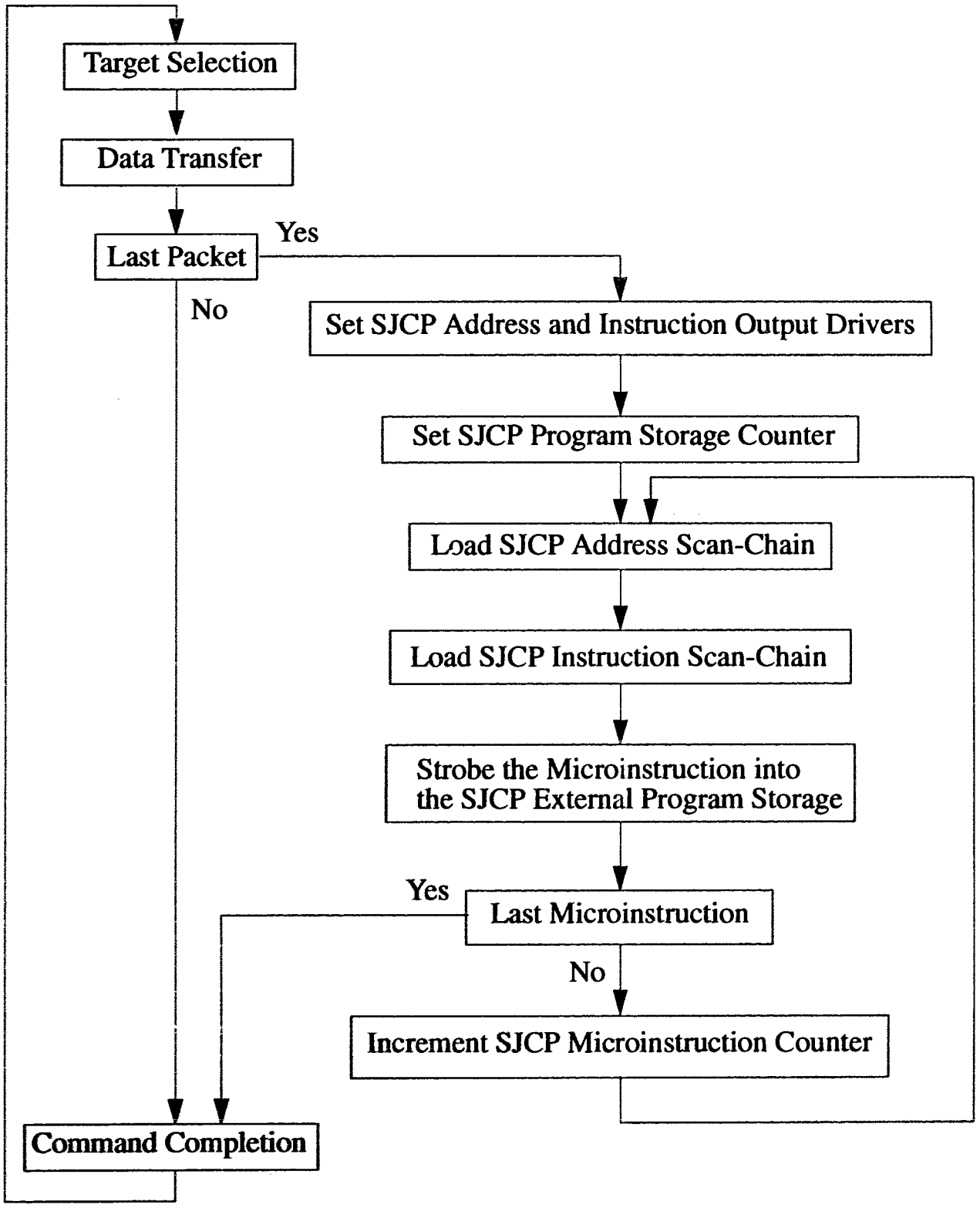


Figure 3.10: Loading Executable Image into the SJCP External SRAM

begin. The concept of having the whole executable image in the Microcontroller SRAM before loading limits the length of the Image to 32kbytes or 4k8-byte-words but it simplifies the loading and executable verification protocols. Also, in the process of building the prototype it is good to have the image permanently in the microcontroller SRAM for testing, reloading and verification purposes, it eliminates the SCSI traffic.

The loading takes place through the SJCP internal scan-chains. During the loading procedure we are using two of them - the address scan-chain, two bytes long and the instruction scan-chain eight bytes long. The address scan-chain is loaded with the address of the next microinstruction to be loaded by writing consecutively two bytes, MSB first, into an address corresponding to the address scan-chain. In the same fashion we are loading the next microinstruction to be loaded by writing consecutively eight bytes into an address corresponding to the instruction scan-chain. The scan-chain output drivers are set driving towards the external SRAM permanently. The microinstruction is strobed into the external SRAM by writing to a specific address which triggers the generation of a strobe pulse. After that, we check if this was the last instruction to be loaded. If it is not we increment the microinstruction counter and the procedure repeats, otherwise we complete the transaction and message for successful loading is sent to the Host.

When the loading procedure is completed, the Monitor is ready to accept new commands.

3.2.3. Software Development Tools and Program Debugging

In order to take advantage of SAM II computational resources, we need to be able to develop and execute SAM II programs. In other words we need tools to allow us to write, to debug, to execute and to do some program and data management operations in SAM II environment. We have a package which outputs SAM II HEX file (the source code is written in microAPL). From this point we need to convert the HEX file into a SAM II executable, to load the executable, to manage its execution, start and termination, and of course we need to be able to debug it. Right now the program management and debugging tools are tightly coupled but later with the expansion of the system most probably they will separate and become more autonomous.

3.2.3.1. SAM II Program Management

The program management utilities include loading and verification of the

executable and program start and termination. These utilities allow us to manage the program execution from the beginning to successful completion and to proceed reliably after that with other activities if any.

3.2.3.1.1. SAM-II Executable Loading

We discussed the first implementation of our loading procedure in the previous section. Currently, the executables are loaded one HEX record at a time. The HEX records are independent units. They have all the necessary information for the record to be processed correctly, this includes the starting address and the number of instructions to be processed.

3.2.3.1.2. SAM-II Executable Verification

This utility allows us to verify the loaded executable with the image in the Microcontroller Data Memory. This is necessary to make sure that we have the executable file loaded correctly into the SJCP Program Storage. The reading of the SJCP external SRAM takes place through the SJCP internal scan-chains. The 64-bit microinstruction is read by reading consecutively 8 bytes from the address corresponding to the SJCP instruction scan-chain. These 8 bytes are compared with the corresponding 8 bytes from the microcontroller image. The mismatch counter is incremented on a byte mismatch. When all microinstructions are processed, the counter is sent over the SCSI to the Host. If the counter is zero, the program execution can start otherwise the program should be reloaded. This option was very useful during initial system debugging.

3.2.3.1.3. SAM-II Program Initialization

The conventional way to start a program execution is to load the Program Counter with the address of the desired routine and after that the execution can start. The differences are coming from the way you load the Program Counter and the way you interface the Program Storage. The SJCP Program Storage is Interfaced (Figure 3.12) through the SJCP internal scan paths.

The Program Counter is not directly accessible and the address of the first instruction to be executed is loaded into the Address Scan-Chain. The Program Counter and the Address Scan-Chain are multiplexed, the multiplexer is controlled by a Trap signal. The Program Counter is 16 bits wide but only 13 bits are coming from the Instruction Scan-Chain to be used eventually in the construction of the next address, on Jump or Call the lower three bits are cleared.

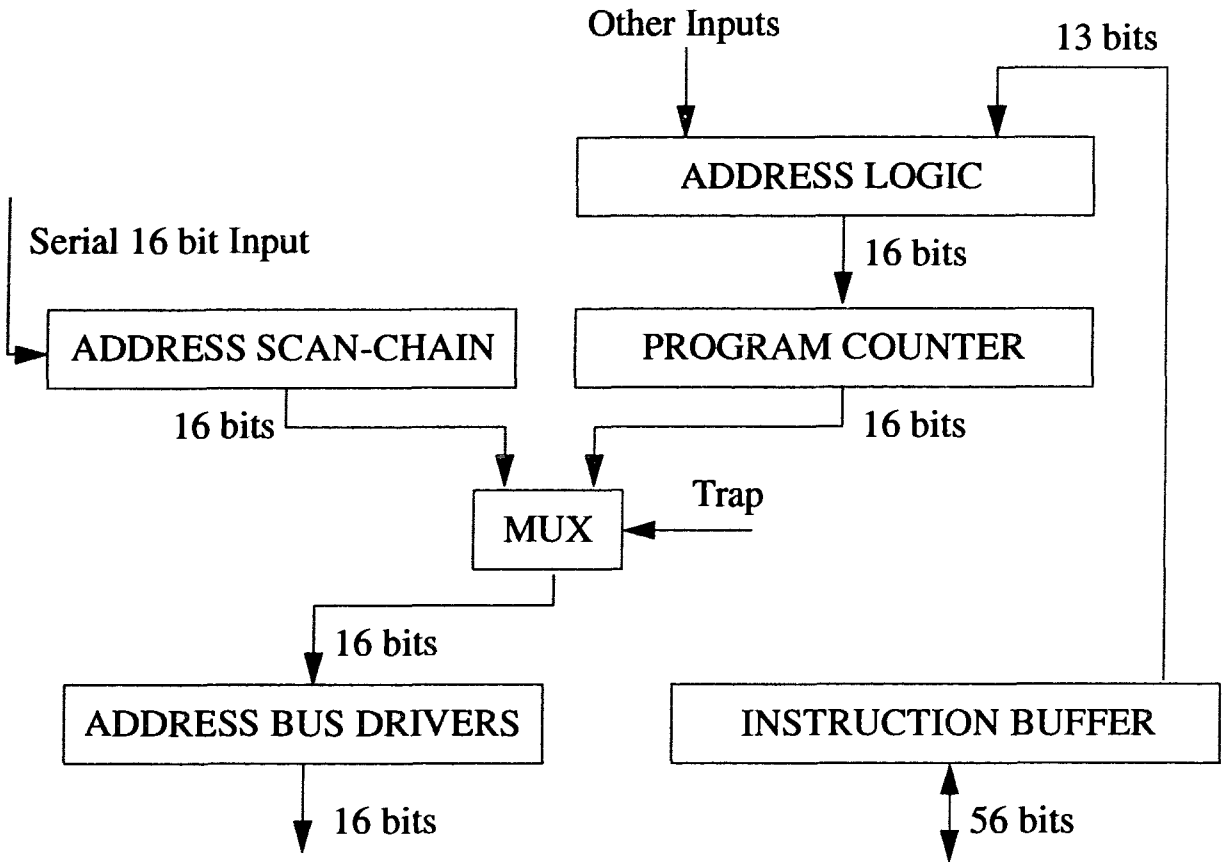


Figure 3.11: SJCP Program Storage Interface

After the SJCP is stopped, the Address Scan-Chain is loaded with the address of the first instruction to be executed, Figure 3.13 (the Trap is set and the multiplexer selects the Address Scan-Chain). The first instruction should be either Jump or Call in order to initialize the Program Counter. After executing one step, we set the Execution Flag, a byte in the microcontroller memory, to show that the first instruction has been executed and the Programming Counter has been initialized. The Execution Flag is used also by Step-by-Step, Full-Speed and Loading routines. We clear the Trap to select the Program Counter and with this the execution can proceed Step-by-Step or Full-Speed.

Using the start-up algorithm from Figure 3.13 imposes certain limitations. The first instruction to be executed should be a Jump. The new address is constructed by using only 13 bits from it as the most significant bits and clearing the three least significant bits. This means that, the second instruction should be at address

location with the three least significant bits zero.

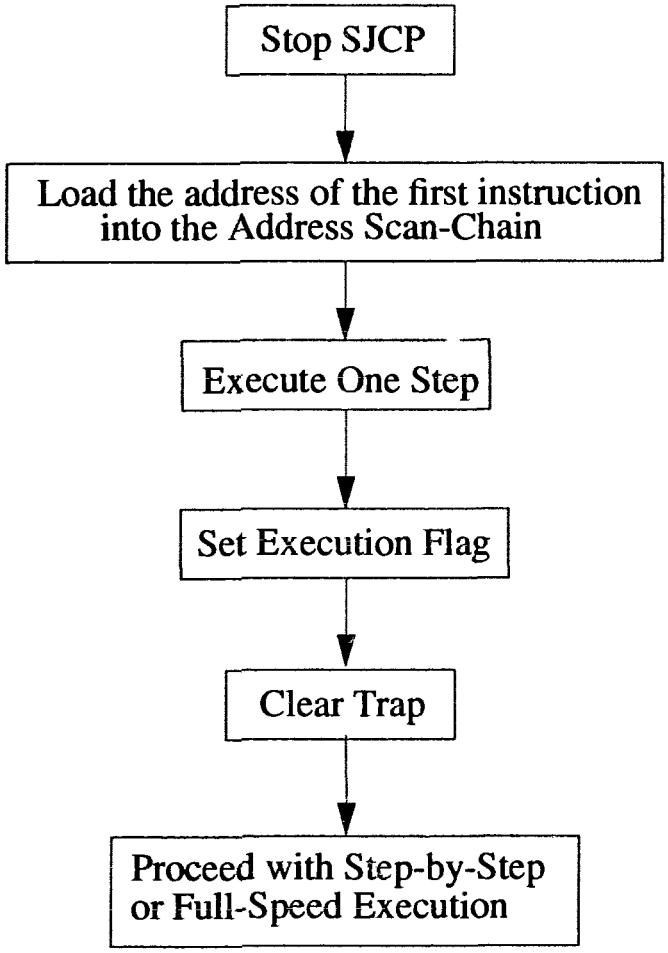


Figure 3.12: Program Initialization Routine

A more general way to start a program is by using the stack. The microcontroller writes the first address into a system area in the SJCP internal SRAM (the Dual-Port Memory) and after that it starts a start-up routine running on SJCP using the algorithm from Figure 3.13. The routine takes the address from the Dual-Port Memory, puts it onto the stack and executes a RETURN instruction. The RETURN instruction loads the Program Counter with the stack value which effectively starts execution of the program at the specified address.

3.2.3.1.4. SAM II Program Termination

We need some way to signal the system that the program has completed and to show the state of completion. For this purpose, we can use a system interupt.

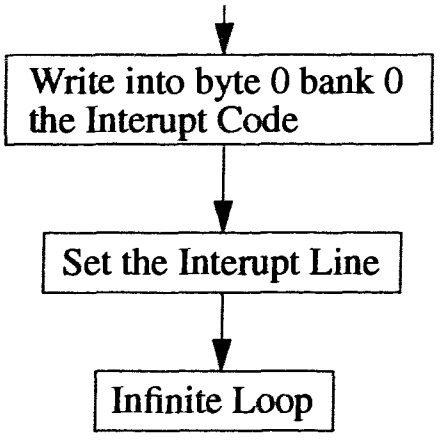


Figure 3.13: SJCP Program Termination Routine

At the end of each program, we attach a program termination routine, Figure 3.14. This routine writes into byte 0, bank 0 of the SJCP Dual-Port Memory the Interupt Code which shows the program has completed. It also shows the state of completion if we have more than one. After that, it sets the Interupt Line and goes to an infinite loop. Whenever appropriate, the Microcontroller will poll the Interrupt Line, will detect there is an interrupt pending, will read byte 0 bank 0 and will take the appropriate actions, stop SJCP etc.

3.2.3.2. SAM II Debugger Concepts

Debugging tools are necessary to write efficient and correct programs. With respect to the level of programming we have different types of debuggers, some debuggers work on assembly code interfacing directly to the hardware, others work on high-level language source codes. and some have both features. As an example, we will have look at "CodeWatch" [29] an interactive source-level debugger features:

- Controlling Program Execution
  - Breakpointing
  - Stepping
  - Tracing
- Examining the Source Program
- Environment Control
- Symbolic Access

- Action Lists
- Macro-Facility
- Command Files

"CodeWatch" is a pretty complex debugger supplying its own command language but the functions it performs can be grouped in two major areas - Program Execution Control and Environment Management.

In designing a debugging tool, one should consider the underlying hardware, the hardware-supported debugging capabilities, the control interface and the programming language. There are several features specific for SAM-II system. First, SAM II is a multiprocessor and we need to be able to debug a program running in multiprocessor environment. Second, SAM II is an embedded system and this raises the question of interfacing the system during debugging. Third, the SJCP internal register files are not directly accessible and the interface should take place through the Dual-Port Memory. This imposes the necessity of incorporating code into the executable program which copies the register files into the Dual-Port Memory for subsequent reading by the Microcontroller. It should be noted that, SJCP provides excellent hardware-supported program-control capabilities.

The SAM-II debugging system although far from being sophisticated has most of the general features of a conventional debugger. Of course the implementation decisions reflect the machine architecture and machine code organization. The debugger has the following program control options:

- Load and Verify - these options allow to load a new program and verify the image. On reloading, the program environment is initialized. The verification of the image was necessary because at certain point we had problems with reliably accessing the SJCP program storage and/or external interface. Verifying the image makes sure that the program is correctly loaded and it is also used for testing purposes. Right now Load and Verify are independent but they can be combined.

- Step and Breakpoints - these options allow to step one or several instructions at a time. The global control is provided by the front-end interface but depending on the responsibility distribution between the microcontroller and the host two implementations are possible. The first one is when execution control is provided by the host in which case the host sends RPC-like requests through the SCSI to the microcontroller for each single instruction. The microcontroller still maintains the program environment but the execution pointer is handled by the host. The second way is when the microcontroller performs low-level execution control, handling the

execution pointer, and it just sends the result to the host at the breakpoints. The second variation is faster but the first one is more suitable for tracing.

- Tracing - enables recording the execution steps into a file for later examination.

- Disassemble - during stepping or breakpointing the next instruction to be executed is disassembled and displayed on the screen.

SAM-II is an embedded system and this affects the way the program environment is handled. Some of the functions are performed by the microcontroller and some by the host. The microcontroller maintains a set of functional flags reflecting the program execution status. This information is necessary when the program is switching execution mode between step, breakpointing and full-speed.

3.2.3.3.1. Step-by-Step Execution

This utility allows us to step through a program one instruction at a time. In this way we can trace the program to monitor how the Program Counter is changing and also to observe the next instruction to be executed, hence partial verification.

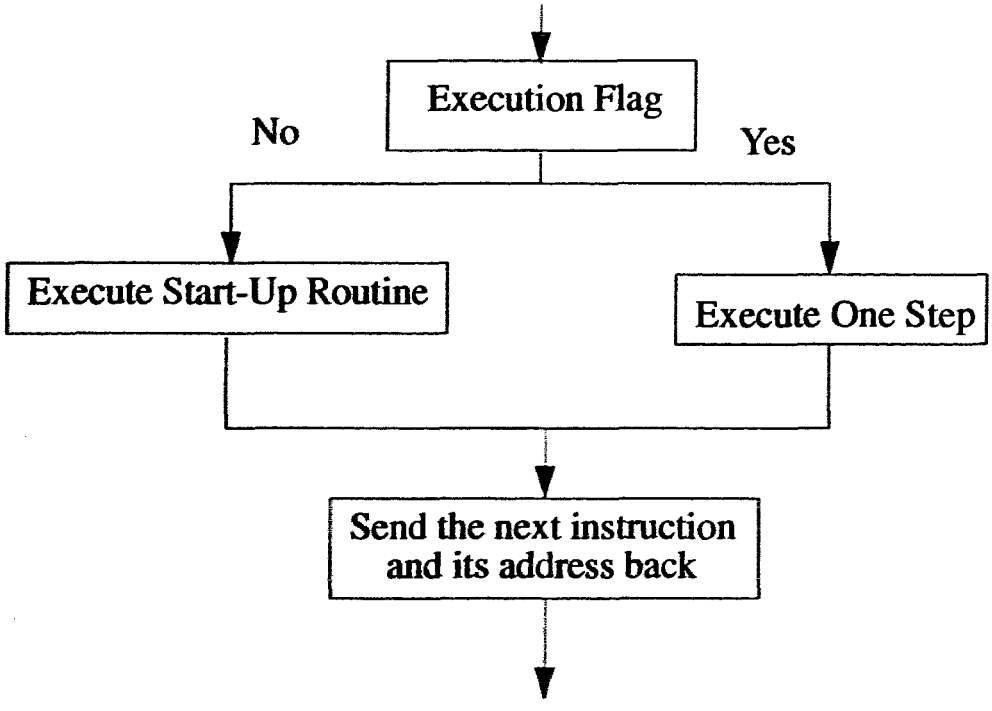


Figure 3.14: Step Execution Command Algorithm

SJCP has hardware support for step execution. In step mode, the next instruction is executed by writing to a specific address location within the microcontroller address space. The step execution starts always with the first instruction of the program. The step-by-step execution routine first checks the execution flag to get the current state of the program, Figure 3.15. If the program has just been loaded it executes the start-up routine, if the program is already in step-by-step execution it executes one step and returns the Program Counter and next instruction to be executed. At any point the step execution mode can be switched into full-speed.

In both execution modes, full speed and step-by-step, the program execution begins with the start-up routine. This is because the Program Counter is not directly accessible and it should be initialized under program control.

3.2.3.2.2. Full-Speed Execution

This utility sets full-speed execution mode, Figure 3.16. The mode is set by writing to a specific address. The full-speed execution mode can be set at any time during step-by-step mode or at the initial program start up.

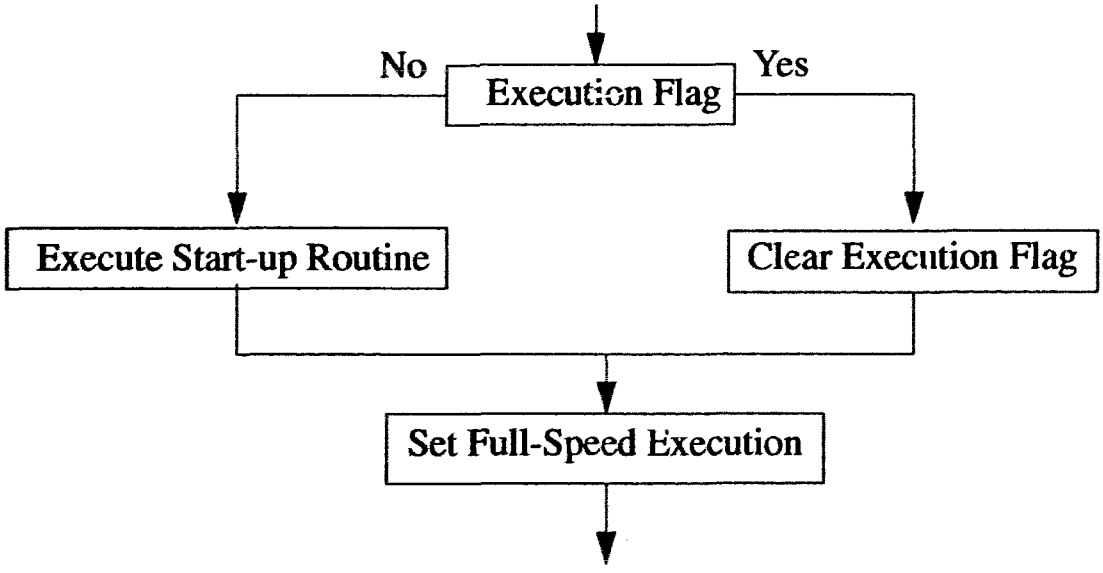


Figure 3.15: Setting Full-Speed Execution Mode

If the Execution Flag is set, the program is already in Step mode, the Execution Flag is cleared and then Full-Speed Execution mode is set. If the Execution Flag is

cleared, the program has just been loaded or it is running full-speed, we execute the start-up routine and after that we set Full-Speed Execution mode. This is done to guarantee system stability during switching between the two modes.

3.2.3.2.3. Dual-Port Memory Monitoring

The SJCP internal register files are not directly accessible by the microcontroller. The SJCP internal Dual-Port Memory is used as an intermediate buffer to get an access to the data in the SJCP internal register files or the external DRAM. The SJCP executables should contain routines for transferring data into the Dual-Port Memory from the SJCP internal register files or DRAM and the timing synchronization with the Microcontroller can be achieved using the SJCP external interupt lines. The SJCP's Dual-Port Memory is in the microcontroller data address space and it is accessed as a conventional memory.

3.2.3.2.4. Breakpointing

Breakpointing allows to interrupt the program execution at certain check points in order to perform execution flow control. There are two ways to realize breakpointing. One can use step mode with address check before each step. When a breakpoint address is reached, the Microcontroller stops program execution and performs the necessary actions. The other way is to replace the breakpoint instruction with an SJCP executable routine to service the breakpoint. This routine informs the Microcontroller that a breakpoint is reached and it can take the appropriate actions. In this way the breakpointed program can run full-speed between the breakpoints.

Depending how the responsibilities between the microcontroller and the host are distributed with respect to program execution control, we have two alternatives. The first one is when the microcontroller handles the breakpoint table and controls the program execution between the breakpoints. The host only sends a request for a breakpoint execution and waits for the result. The other approach is to let the host handle the breakpoint table and program execution. In this case the host sends to the microcontroller Step-by-Step execution requests, where every request requires a separate SCSI transaction. The host is responsible for the execution pointer. The first approach is faster since it does not need that intensive SCSI communication between the breakpoints, but the second gives a little bit more flexibility and removes some complexity from the microcontroller software.

3.2.3.2.5. Instruction Disassembling

In step or breakpoint mode, the next instruction to be executed or the breakpoint instruction respectively is disassembled allowing the user to verify the program control flow. The instruction disassembly is done on the host. It makes it easier to trace program execution and it is useful in verifying the compiler correctness (the software development tools have not been tested completely yet).

3.2.3.2.6. Parallel Execution and Debugging

The different activities on the processing units are triggered by writing into specific addresses. If we have more than one unit selected at certain moment and execute some function, this function will be executed on all selected units simultaneously. The programs in different processing units are completely independent as far as their status is concerned. The status of the programs stays the same regardless of any changes on the selection mode of the processing unit, until it is changed explicitly.

The functions executed on the processing units are executed in atomic manner in their entirety and they are completely independent from each other. The parallel execution and debugging of programs residing on different processing units is done by selecting the corresponding units and executing the corresponding program execution commands.

For example, if we want to execute programs in step mode in the lower four processing units, first, we select the lower four units and after that we start executing step-execution commands. The commands will be applied to all four units simultaneously. If at some point we want to see the status information of a particular unit we select this unit and read the relevant information.

The front-end interface is running on Windows 3.1. The commands are transferred over the SCSI in atomic fashion and they are completely independent regardless if they are going to a single unit or to several processing units running in parallel. We could have an open window for each unit and control the processing units from different windows. This allows us to trace and display all the relevant information for all processing units at the same time. We also can step the processing units one after the other. This feature would be pretty convenient in debugging programs with data exchange among the processing units during program execution for detecting deadlock conditions or to time the processing units in the data exchange.

Parallel programs can be run in all three modes: step-by-step, breakpoint and full-speed execution. It is also possible to run different processing units in different mode, for example one unit in step mode and another in breakpoint or full-speed mode.

3.3. Test Software

The testing software is a collection of routines which perform tests on different system components like system memories, interfaces and component resources. The testing process can be divided into two stages - system components testing in the process of system development and routine system check performed at power-up or at any time a user wants to make sure that all system resources are functional and accessible.

In the process of system integration, each component is tested separately and in conjunction with the other system components. The testing has two goals, first it verifies the hardware functionality, signal levels and timing and second it performs high-level logic test. During this type of testing one component could require several routines for testing different features and properties of the component. The number of routines could be quite big, for example the complete testing collection for SAM II consists of a couple of hundred routines.

Once the system is built, a system resource testing is performed at power-up or at any time a user thinks it is appropriate to check the system functionality. This is a high-level logic test, testing system resource functionality and accessibility and it doesn't require human interaction.

At board level, SAM II consists of two major hardware components - the Microcontroller and the SJ Processing Unit. In the following sections we discuss the approaches and strategies in component and board-level testing of these components and the system in its entirety.

3.3.1. Testing The Microcontroller

The Microcontroller has three major subsystems to be tested - the microprocessor module including the microprocessor and system-clock circuit, the memory system and the interface system.

The Microcontroller integration starts with building the system clock circuit. Once we have the clock rate tuned, we can plug in the program memory emulator and the microprocessor. As a result, we have a minimal configuration running system. The functionality of the system is checked by measuring and observing the behavior of the components.

Once able to run programs, we can keep building up and testing the system gradually component by component.

3.3.1.1. Testing the Serial Interface

The microcontroller has on chip serial interface and it takes few external resources to build an RS 232 compatible serial interface. The hardware functionality is tested by generating 2400 Hz output signal. This is achieved by configuring the serial interface for 8 bit data transfer plus one start (active high) and one stop (active low) bits and writing continuously 55H into the output port.

Once we are convinced that the serial interface is operational we write several routines to perform data transfer between the Microcontroller and the Host acting as a Terminal. At 2400 bits/sec it takes about 4ms to transfer one byte. If the Terminal isn't fast enough, a hand-shaking protocol might be necessary to prevent loss of information. This is the case when you use a 16 MHz 286 machine as a terminal and try to write a server in C using printf() system calls.

3.3.1.2. Testing the SCSI Interface

The SCSI is the major Host-Microcontroller Interface used for fast data transfer in both directions. There are two types of testing routines involved. First the SCSI is debugged and tested in the course of system development. As a multiphase bus protocol it requires additional end-user interface with high reliability to monitor the SCSI status at every phase of completion. The serial interface is used for this purpose. Once the SCSI has been debugged, a power-up routine tests to check the embedded system functionality. This check can be done at any time the user wants. In general once debugged SCSI is a pretty reliable interface.

The SCSI Protocol Controller represents 32 bytes of the microcontroller addressable memory. SCSI transactions are performed by writing SCSI commands into the SCSI Protocol Controller command registers. Each command could start and complete one or more SCSI bus phases. The SCSI bus status before and after

each command is sent over the serial port to the Terminal Unit.

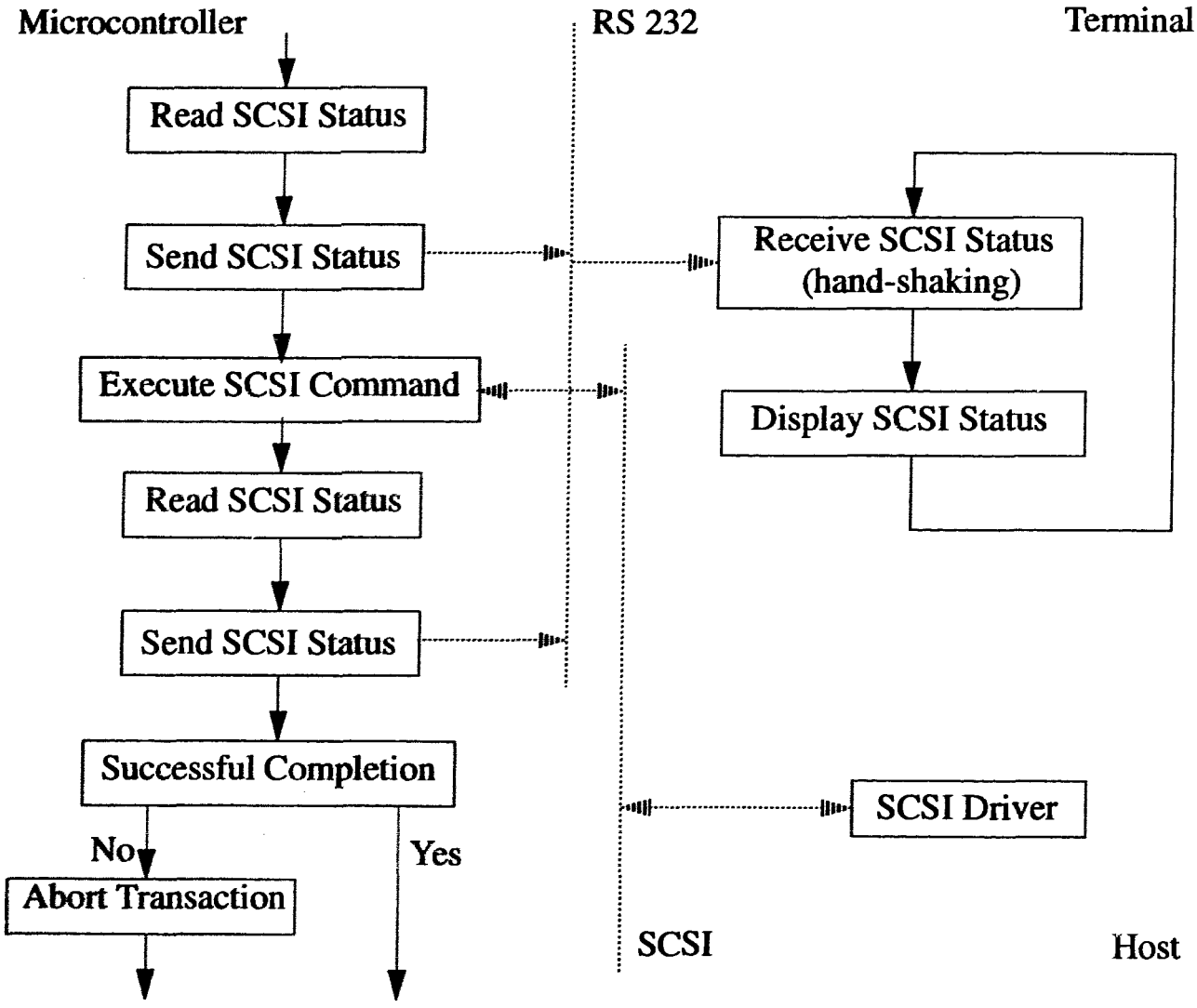


Figure 3.16: SCSI Bus Single Phase Test Algorithm

The SCSI Bus transaction cycle is being debugged phase by phase (Figure 3.17) in the right phase sequence until a complete transaction is executed successfully. The SCSI transaction takes place between the Microcontroller and the Host and, an independent third party, the Terminal, is used to monitor the SCSI Bus status at different control points in the SCSI transaction. The same procedure is applied to trouble-shoot a data transfer in both directions.

On power-up the Host is performing an initialization procedure to check the functionality of the SCSI devices on the SCSI Bus. The Host sends a request to each SCSI device to get its configuration parameters used in the subsequent transactions with this device. The purpose of this test is to make sure that the SCSI interface is working properly. The test involves data transfer in both directions. The test is successful if the data has been transferred successfully. If the transaction fails, the interface is not operational and it should be debugged in the way described in the previous paragraphs.

SCSI Bus functionality can be tested at any time by the user by resetting the system. Sometimes, reset is necessary also when a transaction fails. If a transaction fails, because of the hand-shaking nature of the protocol, at least one of the sides will keep hanging on the bus infinitely and the bus should be reset to make it usable again. Assuming working software the transaction would fail only on a major hardware failure.

3.3.1.3. Testing The Microcontroller - SJ boards Interface

The Microcontroller - SJ board Interface is realized through a pair of FPGAs. It has two major goals - generation of the right control signals at the right time and providing the data at the right place and time (bus control).

The interface is tested by writing and reading in an infinite loop to/from an address belonging to a corresponding SJ board. The Microcontroller FPGA should generate the right ID bits, should repeat RD, WR and ALE control signals and should repeat the data bus as well during ALE and WR.

The FPGA-SJ generates the control signals and buffers the data bus at its input and output. The accessibility of the Y register in the FPGA-SJ is tested by writing/ reading to/from it and also by observing the output pins. This is primarily hardware test by observation.

In case, we want to check a point which is inside the FPGA, an external pin is assigned to it to make it accessible. If the FPGA resources are not enough to be reassigned, it is tested in parts.

3.3.2. Testing The SJ boards

3.3.2.1. SAM-II Boundary Scan Testing Concept

The SJCP incorporates a variation of the boundary scan testing concept to facilitate the chip- and board-level testing.

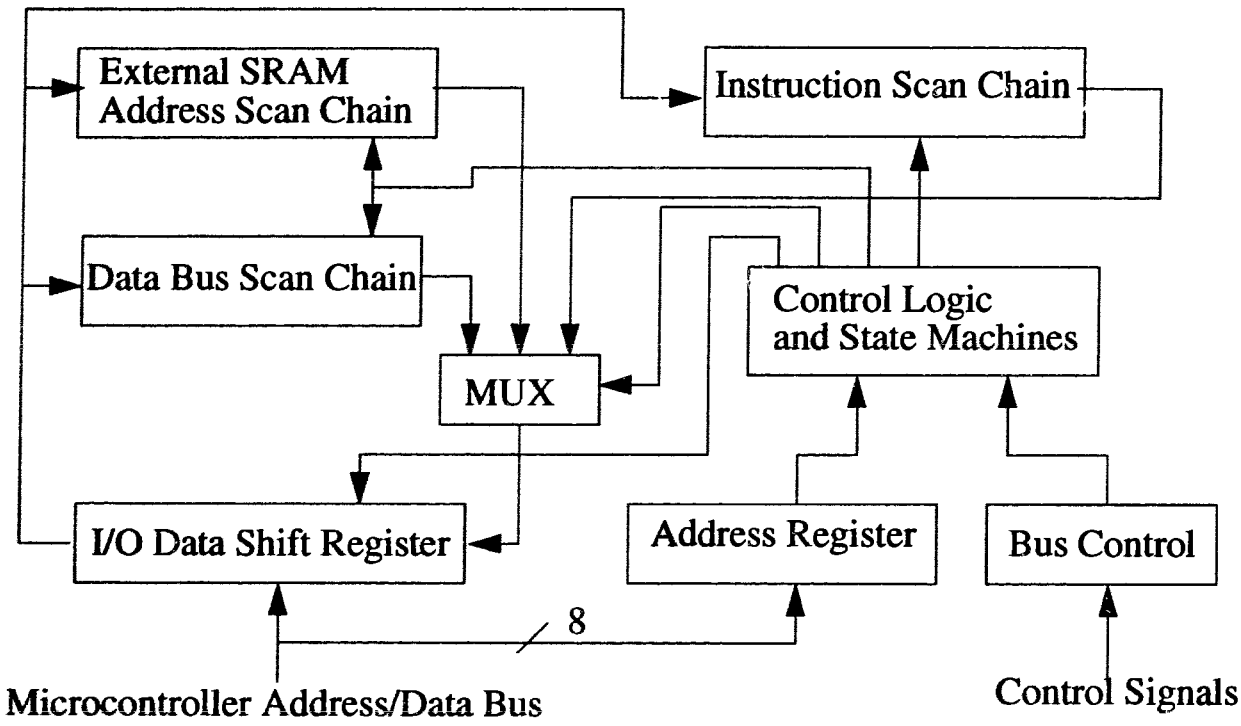


Figure 3.17: SJCP Boundary Scan-Chain Architecture

The SJCP scan-chain architecture [23] consists of three scan chains - a 56 bit external SRAM (SJCP program storage) instruction scan chain, a 16 bit external SRAM address scan chain and a 32 bit data bus scan chain, Figure 3.18.

Each scan chain corresponds to a certain address and is interfaced through an 8-bit I/O data shift register under state machine control. On a write into a particular scan chain one writes to the corresponding address as many bytes as the length of the scan chain. On a read, there is a starting read from the corresponding address which causes the scan chain to advance 8 bits and move the first byte into the I/O data shift register and after that the actual reads are following. A single read or write access takes 16 clock cycles and can be fit within a single stretched DS80C320

memory access instruction. The scan chains provide a convenient way to test the boundaries of the chip as well as the interconnections between SJCP and external program storage.

Most of the Boundary Scan testing implementations are using external hardware control and interface support according to the IEEE standard. They have dedicated pins for control and access. In SJCP the scan chains are part of the chip interface circuitry and they are accessed through the SJCP I/O interface without any additional pins. This is saving hardware resources and pins in the package and also makes scan chains manipulation fast and pretty straightforward.

In the current implementation the scan chains are used and controlled externally by the microcontroller during the test or interface operations. Since the scan chain control logic is on-chip though, it is possible in later versions to design and include on-chip self-test control capabilities. The on-chip BIST circuitry would test the local resources and store the result in a place accessible by the microcontroller. Everything the microcontroller should do is to read the results of the tests from each processing unit and report them to the host. This idea is particularly attractive when the number of the processing units increases.

3.3.2.2. Testing the Dual-Port Memory

The Dual-Port Memory takes 128 bytes of the microcontroller directly addressable address space. Since we are using custom chips directly coming from the foundry which have not been tested completely, this test has two purposes - testing the memory functionality and the size of the accessible memory.

The routine is testing the memory byte by byte and in blocks of 128 bytes. A given block is selected by preloading the block base address into the SJCP block base address register and a byte within a block is accessed as a conventional memory location within the corresponding 128 byte frame. The test writes the LSB into the current memory location, writes a constant value different from any of the test vectors into the next location in order to clear the value on the data bus and reads and verifies the content of the current memory location. The random write is necessary since we test the memory byte by byte and if we do read immediately after write we might actually read what is on the data bus instead of the memory cell. The test stops when a verification fails. The result of the test is the number of blocks and the number of bytes in the last block of sequentially accessible memory. The software can be dynamically reconfigured in such a way so that it can work with any size (at least one block long) of operational Dual-Port Memory.

3.3.2.3. Testing the Event Interface

The event interface consists of SJCP interrupt line (output), flags which are triggered by writing to specific locations and M-bus, a message input lines which can be monitored by SJCP executables.

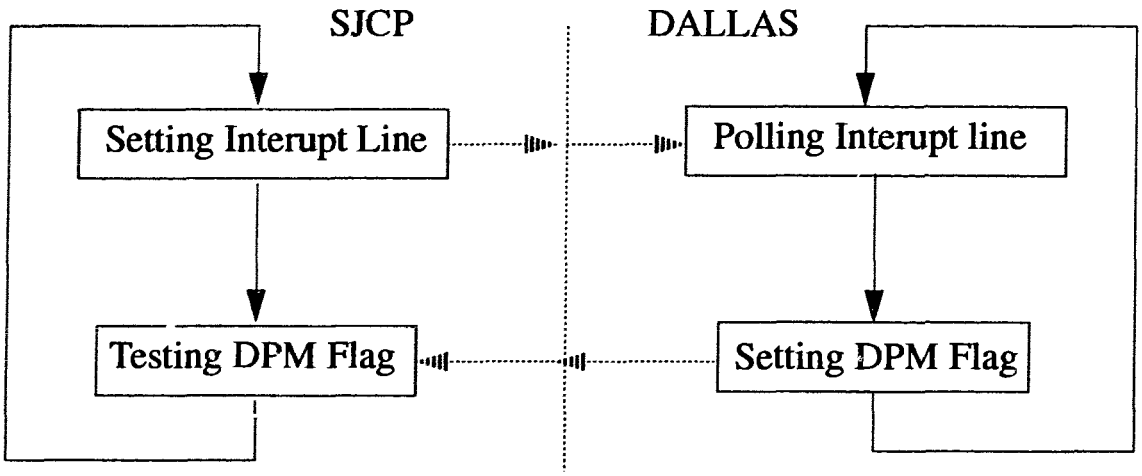


Figure 3.18: Testing the Event Interface

The interface is tested by infinite hand-shaking so the changes on the signal lines can be seen on the scope, Figure 3.19.

3.3.2.4. Testing the SJCP External Program SRAM

The external program storage is tested in two stages. First we test the SJCP external program SRAM boundary. The SJCP Boundary is tested by loading different test vectors into the instruction and address scan chains. After that the chip boundary is tested and verified using a digital scope for each vector. The scan chains themselves are tested by writing and reading to/from them. This test does not require any hardware approaches and is performed entirely by software.

Once the boundary is tested, we test the program storage. In previous sections, we discussed loading an SJCP executable into the SJCP external Program Storage and also the verification of an already loaded program. The testing of the Program Storage (external SRAM) involves these two procedures. First the Program Storage is loaded with test vectors and after that it is read to verify that the corresponding

memory locations contain the correct values.

3.3.2.5. Testing External DRAM

Since the microcontroller does not have access to the data-path hardware, the trouble-shooting process is entirely under SJCP program control. The microcontroller is used to trigger one or another preloaded testing routine.

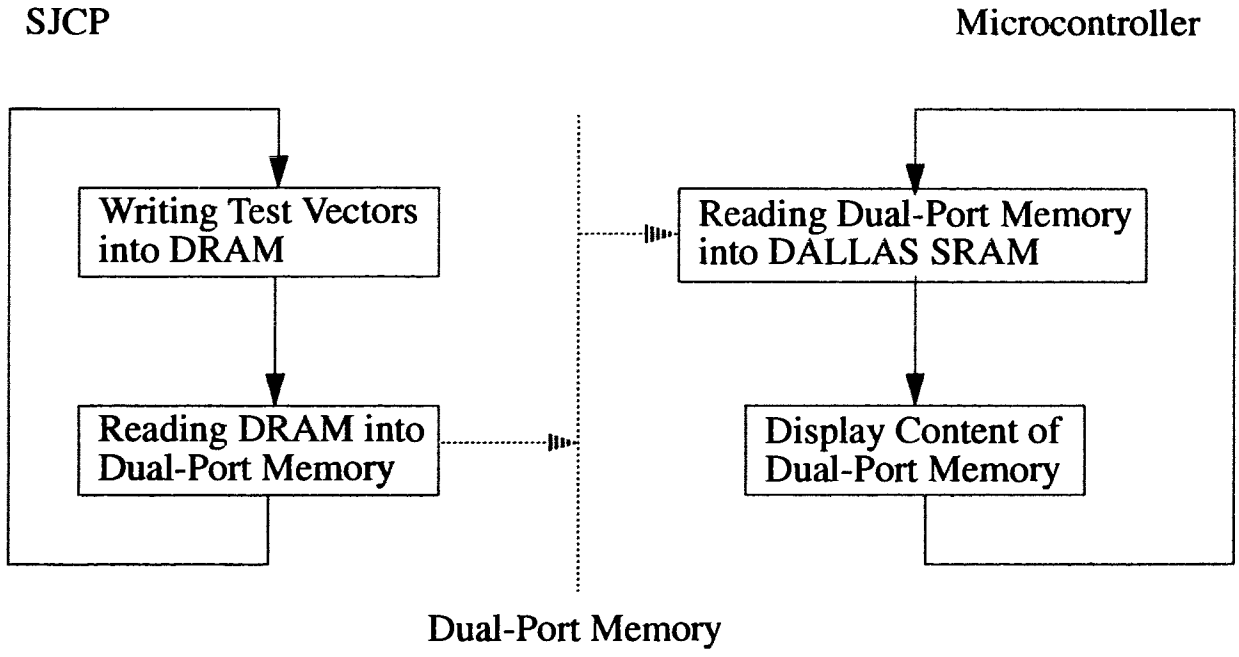


Figure 3.19: DRAM Testing Algorithm

The first thing to be tested is the 32 bit SJ bus between SJCP and SJMI used to transfer commands and data in both directions. This is done by writing a program involving communication between SJCP and SJMI and observing the behavior of the SJ bus. Once we are convinced that the SJ interface is operational, we execute a test routine which actually outputs valid DRAM data and addresses. It allows us to check the logic levels on the address and data buses for different test vectors.

During the DRAM test, the microcontroller plays primarily a supervisory role. The microcontroller is loading a DRAM testing procedure into the SJCP program storage. It might also provide some other information, like test vectors for example, using the dual-port memory and starts the test routine. The communication between the microcontroller and the DRAM test routine is realized through the dual-port memory.

The process running on SJCP could simply read the DRAM into the DPM but it also could do some preprocessing involving other hardware resources. For example one of the tests we are running outputs the check-sum of the data in the DRAM. This is an example of test response compaction, where instead of verifying every single test vector, the test vectors are preprocessed and a single result vector is output.

In order to avoid eventual timing problems, the two processes might need additional synchronization. For example, when the microcontroller is reading the dual-port memory it might have to stop SJCP. This shouldn't really matter since SJCP has synchronization logic, but we found out that sometimes SJCP behaves abnormally when the microcontroller is reading the DPM and SJCP is running full-speed. Problem-avoiding strategy is a good idea to apply anytime it is possible, until the system is fully tested.

3.4. Summary

The development of the system software started from the very begining along with the debugging of the hardware. Many of the hardware debugging and testing routines, with small modifications, were used in the implementation of the system software. I developed the SCSI interface embedded system and host's drivers, the microcontroller monitor built around the SCSI interface driver, the menu-driven front-end interface, the SAM-II debugging system and wrote a number of testing and initialization procedures. Dr. Rick Hobson designed his own monitor allowing him to access the system through a serial interface using a standard ASCII terminal. He also wrote a number of processing unit testing routine and was helping with whatever was necessary throughout the whole project.

Chapter 4: General Discussion

4.1 Architectural Issues

4.1.1. Next Generation

SAM-II develops further the concepts of the first implementation of the Structured Architecture Machine, SAM-I, and is a stepping stone towards the development of a massively parallel computer system. Our main objective was to build an embedded multiprocessor and get practical knowledge and experience of how to design a multiprocessor system, how to interface efficiently an embedded system, how to design an embedded system software architecture etc.

The solutions to the above problems will help with further component integration and development of the next generation system. With the current chip set, each processing unit takes one board and if we want to put together for example 64 units, we have to interconnect and interface 64 boards. This would take a lot space for the eventual performance gain, interfacing 64 units through an external interface would obviously increase system access times affecting the overall system performance and finally signal propagation delays particularly system clock synchronization will require special attention. The way to go is to further integrate the components.

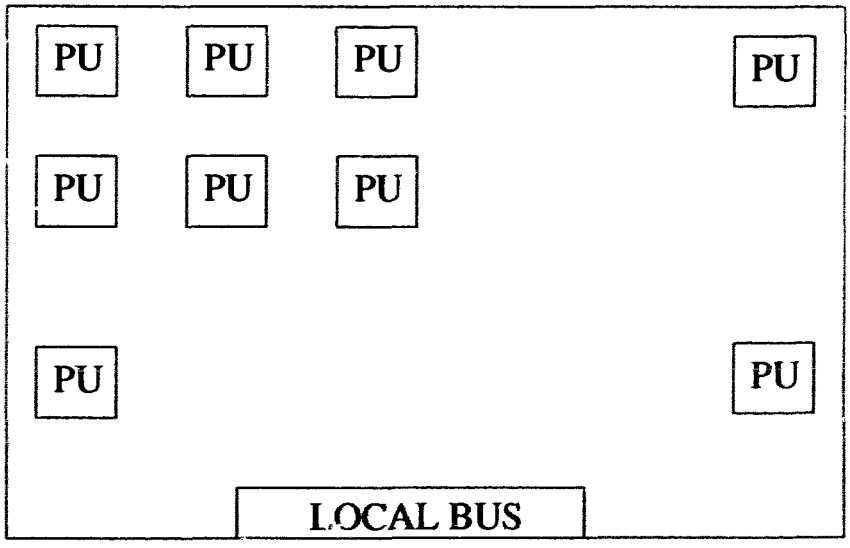


Figure 4.1: SAM-III Architecture

In the next generation machine, all SJ components SJCP, SJMI, SJNI, FPU and the Instruction Pipe will be integrated into a single chip. This would make possible to accommodate an array of up to four or eight processors on a single board Figure 4.1. The board can be plugged directly on the CPU local bus minimizing the interfacing delays. In this case special logic will be necessary to handle interrupts and DMA data transfer. In order to make the board accessible by all system applications, it should be interfaced through the operating system. If we want to increase the number of processors, we have to build the system in a separate box and interface it as an embedded system.

4.1.2. System Interfacing

Currently, SAM-II is interfaced through an 8-bit-wide SCSI 2 interface with certain limitations on the data transfer speeds coming primarily from the microcontroller clock rate affecting directly the speed of the bus and also from the Advanced SCSI Protocol Interface (ASPI) used by the application software to interface the system. One way to improve the situation is to use 32-bit SCSI 3 interface and use 32-bit microcontroller to interface the multiprocessor environment.

A good question would be how to interface a high-number of processors, for example 64 or 128. With all components integrated and using surface-mount technology, it would be possible to build boards with 8 even 16 processors running under the same clock and timing. Each board could be designed as an independent SCSI device. We can attach seven of these boards to the host's SCSI interface and get a system with 118 processors.

An interesting idea is to use the system as a vector arithmetic server. If the SCSI is replaced with Ethernet or ATM interface, the system could be connected to a network and used as a remote server. A special software accessing directly the Ethernet cards on the hosts should be developed to interface the system efficiently. One way to avoid the development of a complex software interface tools would be to connect the system to a standard SCSI interface of a workstation connected to a network. Then everything we need is a daemon intercepting the requests to the particular SCSI interface. The system could be accessed through the operating system using either remote procedure calls by applications running on remote machines or by remote logon to the server workstation.

4.2. Software Issues

The currently running system software provides the basic tools to interface and manage the system resources. The architectural principles were adopted during and in accordance with the requirements of the prototype development process. We wanted to have reliable and flexible tools which would allow us to work in a not very stable environment. In the next versions the software will be refined and in the following sections, we will discuss the principles and some possible ways to go in future implementations.

4.2.1. Alternative System Applications

Originally, SAM system was designed for fast hardware interpretation of APL language. APL (A Programming Language) is a language specially designed to reflect the requirements of array processing applications. A multiprocessor hardware interpreter built around APL could be expected to outperform general purpose computers as far as array processing applications are concerned.

An alternative approach, using the system as a vector arithmetic server, would take advantage of the fast hardware interpretation and also give interfacing flexibility making the system accessible by any general purpose language.

In any application involving array processing, the array processing part comes at the end to performing generic or combination of generic operations like addition, multiplication, division etc. on the arrays of data. The idea is to develop a set of generic array processing routines loaded into the SAM processing units microprogram storages at power up time. After that, an application instead of running the array processing part on the main CPU would do a system call to the operating system providing the data and the operation code. In this way the system can be accessed by any general purpose language through the operating system.

The set of generic routines could still be written in the assembler language, which we have been using so far to develop programs. Once the routines are developed, they can be put in a ROM replacing the current microprogram SRAM, or in the microcontroller EPROM loaded into the microprogram SRAM on power-up or they can be stored on the host hard disk and load them through the SCSI at system power up.

In this scenario, the microcontroller should assume a little more responsibilities concerning program and data management and particularly in array processing

commands execution. The generic routines could be organized also in macro operators and/or the microcontroller could perform some parsing on more complex array processing commands.

From an application's point of view the system is interfaced through the operating system. A set of resident routines intercepting one or more interrupts is loaded at power-up. The routines are in the role of custom SCSI drivers mediating the data transfer between the application and the microcontroller through the SCSI interface. The application puts all source and destination data and command information into a data structure and does the corresponding system call. The pointer to the structure can be stored into the CPU registers before the system call or into a system shared memory if the resident routines are fully integrated with the operating system.

This concept is applicable for SAM-III as well as when the system is directly plugged on the bus. This time the control information can be written into memory locations occupied by SAM-III which does not require the interupt handlers to be fully integrated with operating system.

4.2.2 Program and Data Management

Besides servicing the SCSI interface, the microcontroller has to assume certain SAM-II resources management responsibilities and particularly program and data management during system operation.

In the SJCP microprogram storage we might have several active programs and routines, some of them application-oriented, some of them system-oriented handling I/O data transfers for example. The microcontroller should keep track of the state of any one of these routines in every board and it should be able to perform real-time task-switching whenever necessary without affecting the system performance. This is not really a multitasking operating system paradigm, the task-switching is caused either by the application program in case of data exchange or by the host. The return from a task would be like return from a procedure call. From a user point of view the system is unithreaded and the main thread is the application program. Some of the questions to be researched are: How the task will be switched? In what timing and sequence and would it be possible to put all the control information in the application program or the microcontroller would need to have some high priority control capabilities?

Some of the routines in the microprogram storage will be dealing with data and

memory management. Supposed that at certain point the application program needs to get rid of some data or to input or output some partial results. In this case the application program should be stopped and system routines should be called to perform the requested operations. For this purpose, we will need two processes, one running on SJCP and the other on the microcontroller.

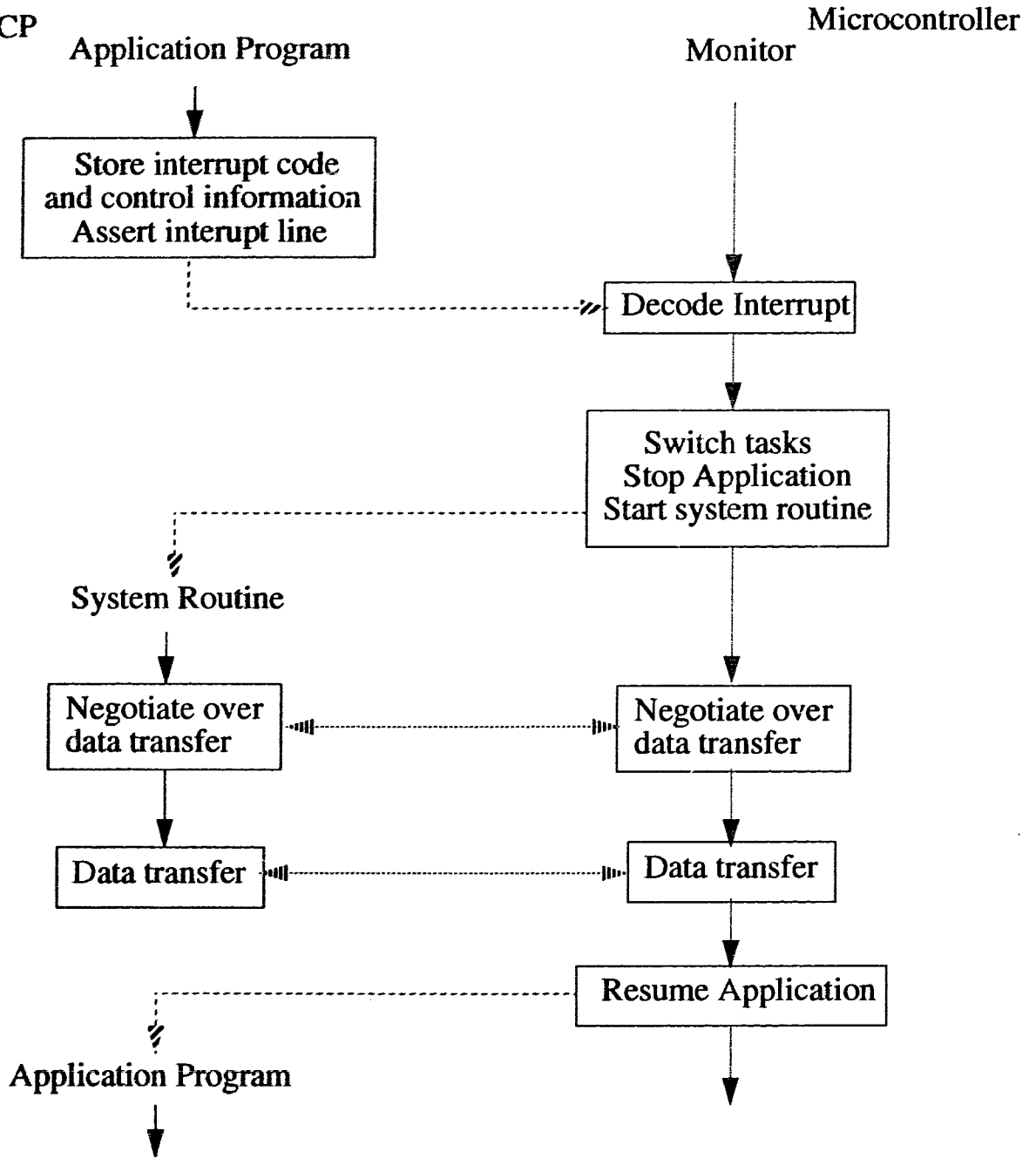


Figure 4.2: Task-Switching and Data Transfer

Let's consider the case when the application has some partial results to output. One way is to use hand-shaking on the interupt lines to synchronize the processes and negotiate over the data transfer, Figure 4.2. In this particular case the microcontroller switches the tasks on request by the application. It starts the system routine, negotiate the data transfer parameters and performs the operation.

The Microcontroller could read and store the application program pointer at the interupt point and resume the application by it restoring the return point but this would require a termination routine at the end of the system routine. Another way is for the application to put its pointer on the stack and the return instruction of the system routine will resume the application.

Note that in this case the application has all the information necessary to initiate the data transfer. But it might happen that resource management operations be performed without the knowledge of the application in case of system resource limitations for example. More sophisticated approaches for resource management in embedded systems is definitely one of the hot topics for future research.

4.2.3. Parallel Debugging

A parallel debugger should be able to perform all the basic functions a sequential debugger can perform plus functions relevant to multiprocessing program execution and particularly with it should cope with shared data consistency and exchange and process race and block conditions. The particular design and implementation depends on the two generic aspects of the system - system software and hardware.

4.2.3.1. Debugger-Operating System Relationship

A very important point is the relationship between the debugger and the operating system. In some systems the debugger is completely integrated with the operating system as it is the case with most of the "C" debuggers running on UNIX workstations. This is a good approach when the operating system is debugged and stable. This approach allows direct access to the operating system data structures and the ability to monitor and control the state of the program.

The researchers working on Amoeba adopted a different approach [36]. The debugger is separate from the kernel which allows modification of the operating

system while using the same debugging tools, assuming the interface with the kernel is preserved. In this case the debugger is more independent but most probably some performance trade-off should be made since the program state is still monitored through the operating system.

An interesting concept is realized in Parasight - a high-level debugger [38]. Special programs called "parasites" are linked dynamically with the source at dynamically created stub places. All debugging functions like dumping a trace of the stack or program state etc. are performed by these parasite programs. This makes the debugging completely independent of the operating system but it affects the performance as well. Anyway, during debugging, we don't care much about performance.

4.2.3.2. State-Driven and Event-Driven Debuggers

Depending on the way the program execution is monitored we could have event-driven and program-state-driven debuggers. The events can be anything like sending a message to another process, creating a task, reaching a breakpoint, divide by zero etc. The conventional procedure-call stubs are replaced and any time a procedure call is made, a message is sent to a higher-level client process. The program execution resumes after the reply by the client process. The event records are saved in log files and used for program execution replay. High-level processes are monitoring the events and control the debugging.

The debugger for Warp is a typical state-driven one [39]. There is a separate process keeping track of the state and execution point of each processing unit process. The process state is used to control the debugging procedures. The state debuggers require access to the program state areas through the operating system or directly accessing the hardware if possible.

Most of the time though, just keeping track of the events or program state might not be enough and the debuggers should have capabilities to control the program execution as well as to monitor the events in the system.

The current SAM-II debugger is a state driven one. We keep track of the program execution point by reading directly the program counter and the microinstruction from the corresponding processing unit. Next versions will have to include some event tracing capabilities to be able to monitor more complex programs.

4.2.3.3. SAM-II Debugging Concepts

When thinking about the SAM-II debugger design, we have to consider two factors - the system architecture and the type of the application programs to be executed on the system. For now, the program and data management tools are separate from the debugger and we could assume that this trend will be preserved.

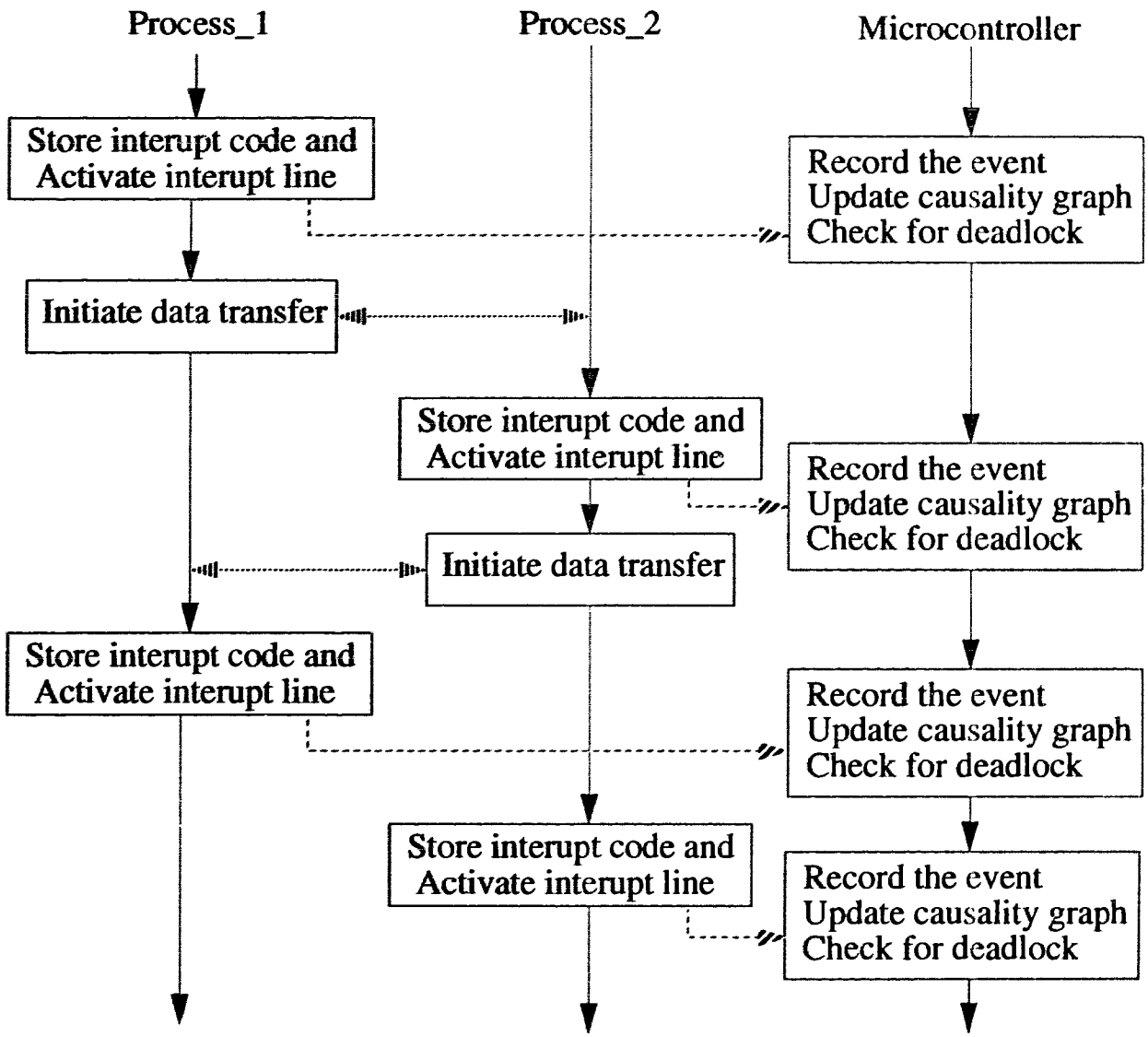


Figure 4.3: Deadlock and Race Conditions Detector Algorithm

The system will be used to run array processing applications. We could expect that there will be intensive data exchange among the processing units passing

partial results and data to each other. In general we can assume that one processing unit would want to send data to another processing unit and receive data from a third etc. The debugger will have to keep track of the execution points of all three processes and the events of data transfer. The possible complication here is an eventual deadlock condition if each process is waiting for another one.

The debugger should be able to keep track of the execution points and states of all three processes and also to create a truthful picture of the events. The execution points and program states can be obtained from the processing units CPU registers and microcontroller system areas respectively.

The processing units should inform the microcontrolier about the pending data transfers. One way is to use the interrupt mechanism or just to store certain control data into the system areas in the dual-port memory to be read by the microcontroller. The Microcontroller uses the information to build a causality graph for detecting deadlock conditions Figure 4.3.

The debugger controls the program execution in step or breakpoint mode. At the same time the processes running on the processing units are passing messages to the microcontroller through the SJCP dual-port memory informing the debugger about upcoming events. The debugger is using the information to construct a causality graph for detecting deadlocks and race conditions. Note that in this implementation, the SJCP application executables should include code performing the message passing. This code could be inserted at compile time or by the debugger during loading.

The algorithm illustrates a concept which could be expanded to handle shared data consistency as well.

4.3. Performance Issues

4.3.1. Technology

The technology is one of the factors which affects directly the overall system performance by defining the maximum possible program execution rate. The current chip set is designed in 1.2 u technology running at 15 MHz. The CPU is capable of executing one microinstruction per clock cycle which means 15 MIPS assuming no memory access penalties. Using better technology would make possible to redesign the chip set to run at 40 even 60 MHz which would increase

several times the performance.

It should be noted that, a single SAM processor module is not intended to compete one-on-one with commercial superscaler processors. But, it is expected that, an array of these relatively cheap processors could be competitive.

4.3.2. Component Integration

The current system consists of several separate components - the microprocessor, the memory manager, the floating point unit and the network interface. All these components are connected through an external board-level bus. An intercomponent data transfer will have to cross chip boundaries which basically would require an extra clock cycle, or several clock cycles if you want to increase the clock rate, for intercomponent data exchange. Integrating the components in a single chip would eliminate the boundaries-crossing penalty and make possible to increase the clock rate.

4.3.3. System interface

The overall data processing time is equal to the time to transfer and load the program and data into the processing units plus the time to execute the program and plus the time to get the results back. These times could be in different proportions but in any case in order to take full advantage of the optimized embedded coprocessor one needs a fast way to transfer data in both directions.

In the current implementation, we are using fast SCSI 2 interface. We managed to get a transfer rate of 0.3 Mbytes/sec (about a 80kwords/sec). Let's suppose, we want to multiply two arrays of 100x100 words. This means we have to transfer 20000 plus 10000 result words which would take approximately 0.4 sec. For now let's ignore the loading time. It is going to take 10^6 multiplications to multiply two 100x100 arrays. Assuming three clocks plus one per multiply, we get $4*10^6$ clocks. Further assuming four processing units with 20 MHz clock, no interprocess communication and no memory delays we get an execution time of 0.05 sec. As one can see the interface overhead is an order of magnitude bigger than the actual execution time!? This simple example illustrates the importance of the system interface in getting performance results.

The situation can be improved in several ways. Using SCSI 3, 32 bit 20 MHz synchronous data transfer with DMA, the transfer rate can be increased up to 40

times which would decrease the interface overhead from 0.4 sec to 0.01 sec. The transfer rate can be improved further, if the data is transferred directly from the host hard drive to the embedded system.

There is an idea to connect local hard drives to the system but one still would need to transfer data from the host unless there is a way to generate or provide data locally.

4.4. Conclusion

We implemented the SAM-II prototype system capable of accommodating up to five processing units and interfaced through a standard SCSI 2 interface. We built and debugged the motherboard and processing unit hardware and developed the basic system resource management and program debugging software tools.

We met the objectives stated at the beginning of the project and here is the summary of our results:

* built and debugged a microcontroller printed circuit board performing basic system management and interfacing functions

* built and debugged a processing unit printed circuit board performing program management and data processing functions

* fully tested and verified the functionality of SJCP and SJMI and that they can work together

* implemented an FPGA-based motherboard-parallel processing unit interface logic

* built and debugged the microcontroller SCSI hardware and firmware

* developed a system software allowing the user to efficiently interface and work with the system

* developed a debugging tool with basic program management and debugging capabilities

Valuable practical knowledge and experience has been gathered to support a possible next generation SAM III prototype implementation.

We obtained experience on how to interface an embedded system through high-performance interfaces like SCSI, how to design the interface and interface drivers for efficient data transfer and how to approach the interface design for efficient debugging and testing.

We learned how to design and interface a multiprocessor environment, how to design boards capable of accommodating several processing units. This will be very helpful in possible later implementations with high number of processors.

We gathered practical experience on how to handle clock distribution, signal propagation delay as well as power distribution at chip and board-level problems. This will be very useful particularly in the further component integration and next board-level implementations.

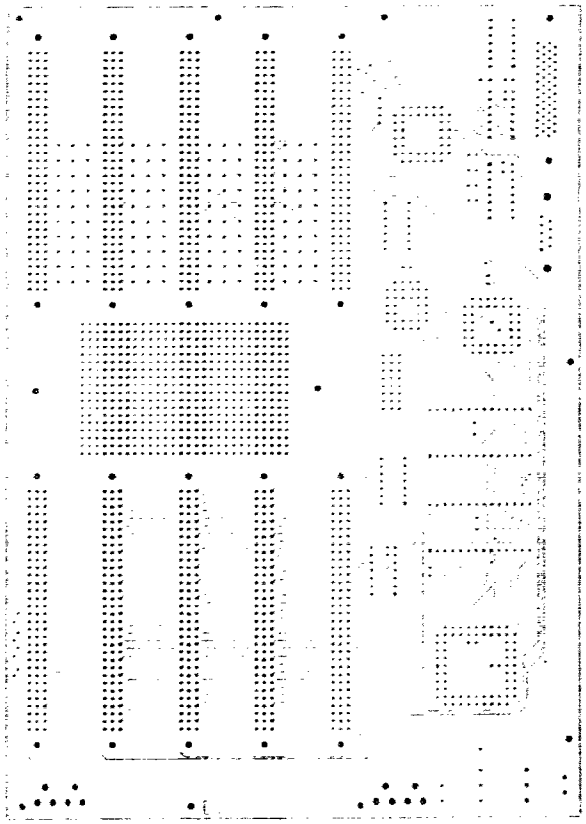
Very valuable experience has been gathered on how to approach the design of and how to develop embedded system and front-end software, how to integrate the different software components, and how to design the software to be easy to expand.

Finally, we learned how to design an embedded system state-driven program debugging tools allowing real-time program debugging, how to distribute the functions between the host and the embedded system for optimal system performance.

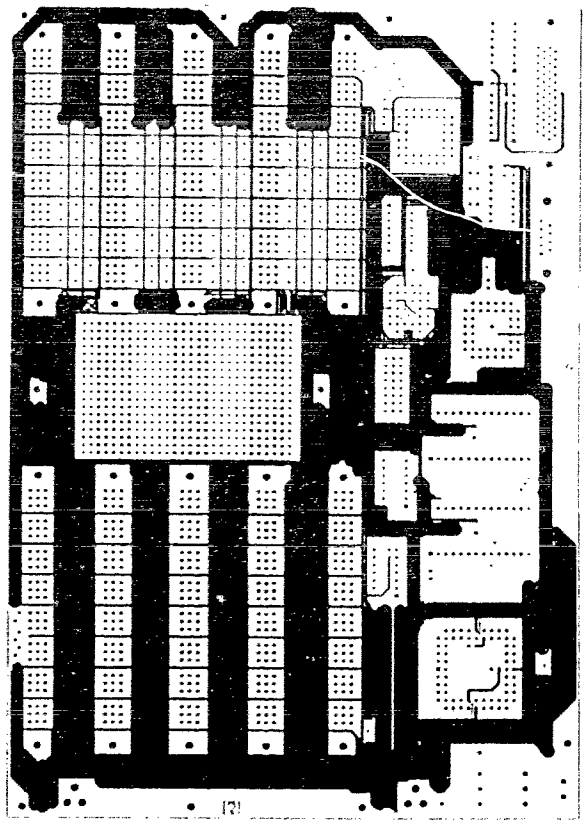
Appendix A: Microcontroller Printed Circuit Board

This Appendix contains the scaled layouts of the Microcontroller Printed Circuit Board (PCB) layers. The board has four layers altogether.

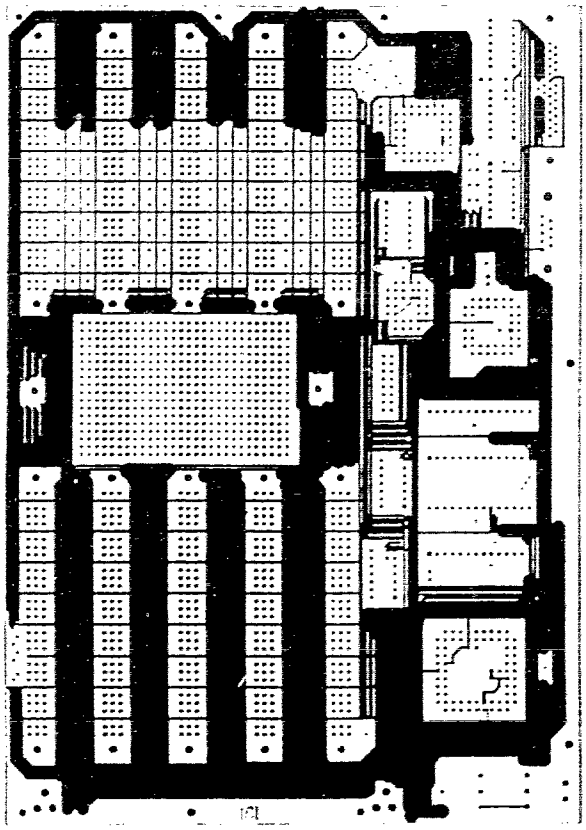
The Microcontroller PCB was designed using EZ-Board PCB design package and was manufactured by OMNI GRAPHICS Inc. The Appendix contains the original layouts. Some minor changes have been made to the board during the trouble-shooting process.



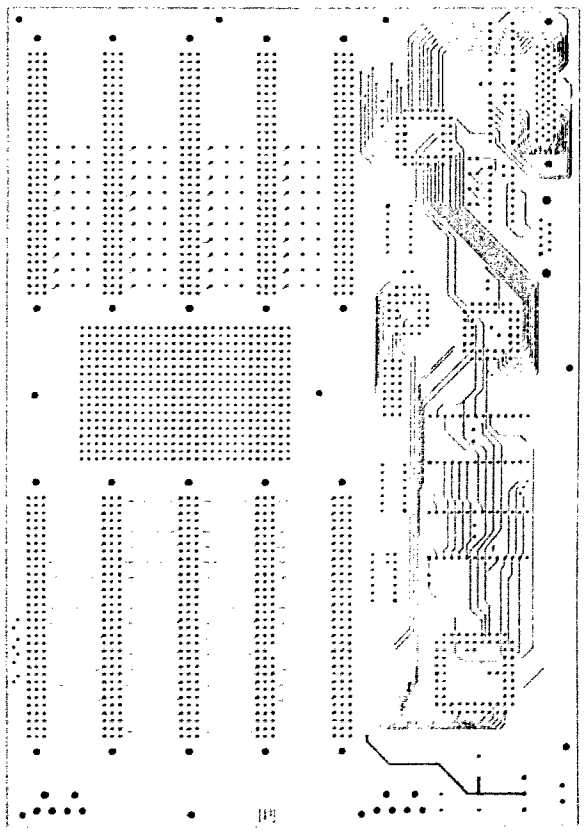
A.1 Layer 1/Signal



A.2 Layer 2/Power



A.3 Layer 3/Ground

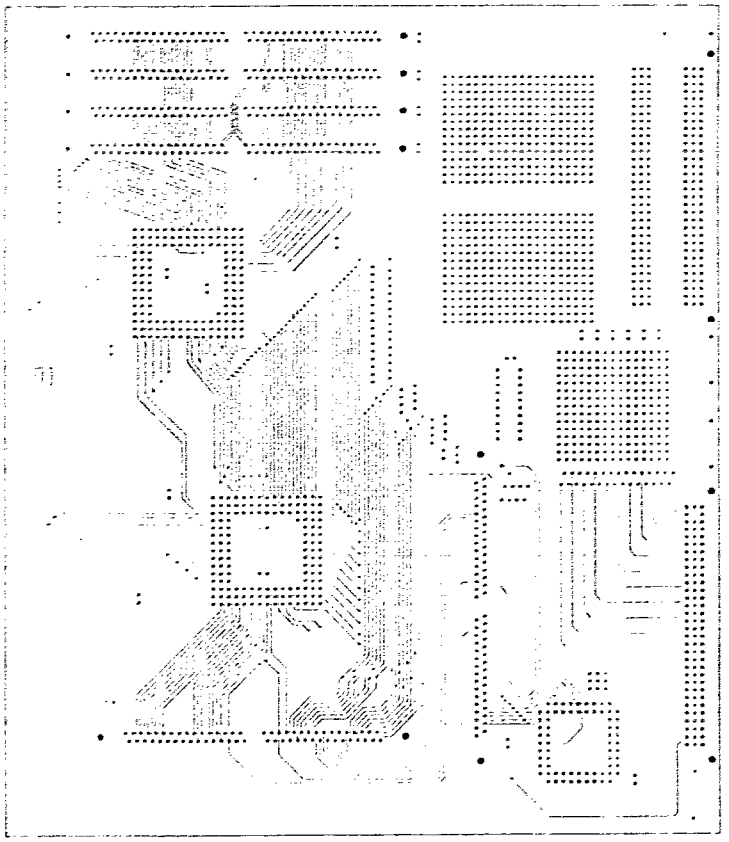


A.4 Layer 4/Signal

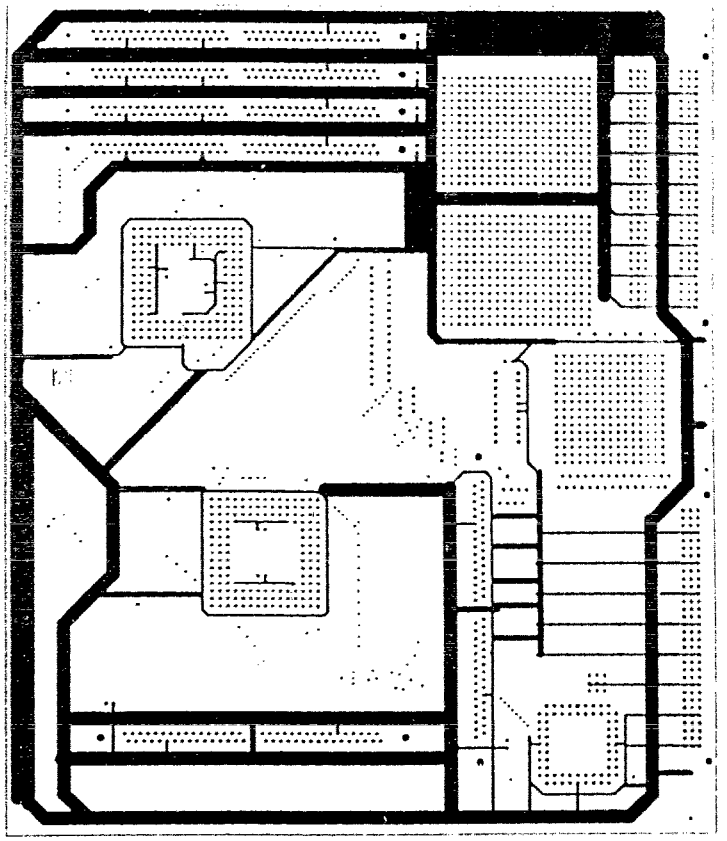
Appendix B: SJ Processing Unit Printed Circuit Board

This Appendix contains the scaled layouts of the SAM Processing Unit PCB layers. The board has four layers altogether.

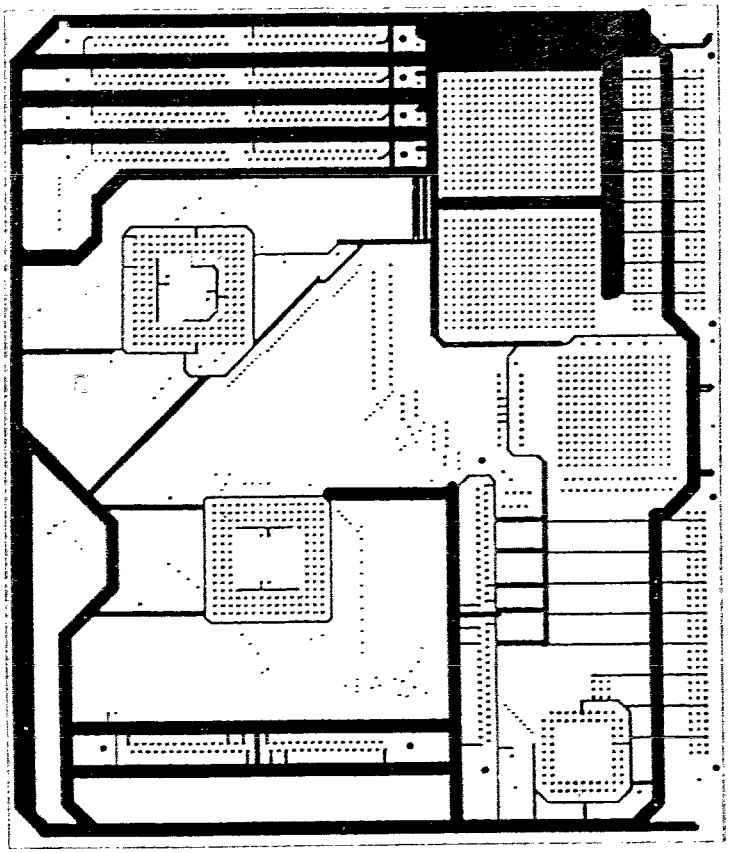
The Processing Unit PCB was designed using EZ-Board PCB design package and was manufactured by OMNI GRAPHICS Inc. The Appendix contains the original layouts. Some minor changes have been made to the board during the trouble-shooting process.



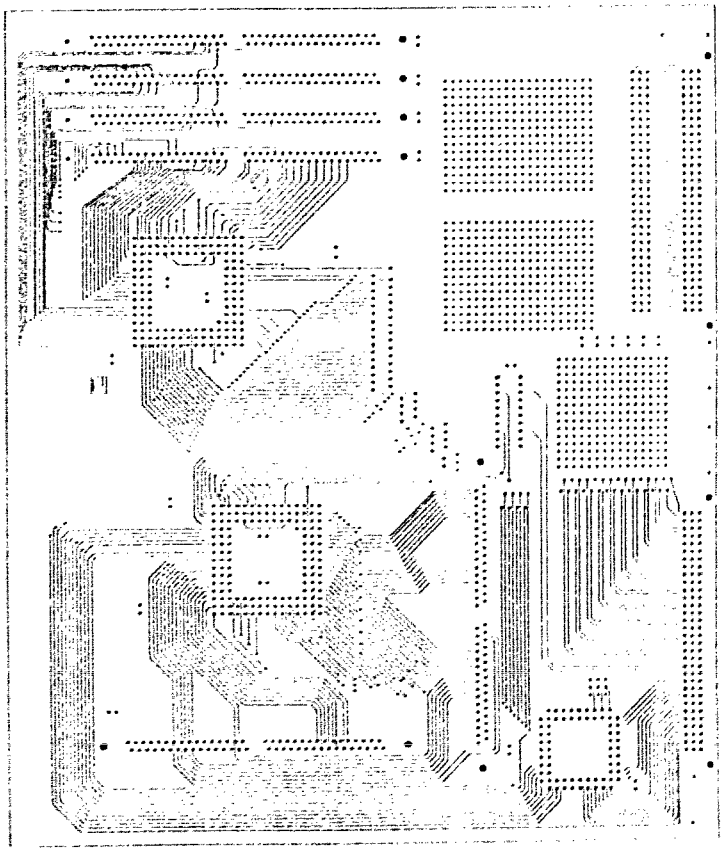
B.1 Layer 1/Signal



B.2 Layer 2/Power



B.3 Layer 3/Ground



B.4 Layer 4/Signal

Appendix C: Microcontroller FPGA Design Files

This appendix contains the microcontroller FPGA PLDShell Plus (.PDS) design files.

The first and second improved versions of the FPGA-based interface logic design files are given along with the simulation signal wave forms. The logic was designed and simulated using PLDShell Plus FPGA design software.

Appendix C.1.1.: PDS Design File/Version I

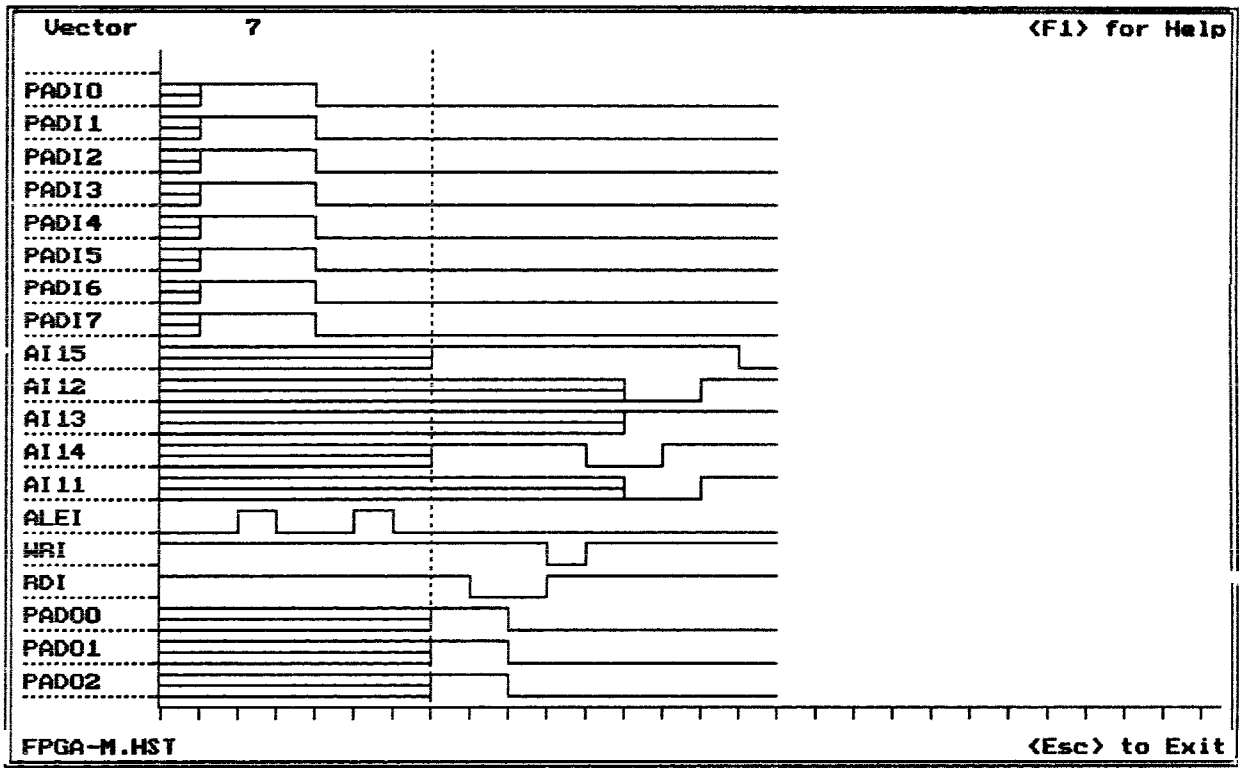
```
;Chip specification: iFX740 68-pin PLCC
CHIP FPGAM iFX740_68

;PIN AND MACROCELL ASSIGNMENTS

;Bidirectional address/data bus buffer
;16 macrocells are organized as an 8-bit bidirectional buffer
;---Microcontroller side
;------Output address/data macrocells
;------The macrocells are driving the bus during a READ
PIN 40 ADI0
PIN 41 ADI1
PIN 42 ADI2
PIN 43 ADI3
PIN 44 ADI4
PIN 45 ADI5
PIN 46 ADI6
PIN 47 ADI7

;------Input address/data pins
;------The macrocells' pins are used as an input during a WRITE or ALE
PIN 40 PADI0 PINFBK
PIN 41 PADI1 PINFBK
PIN 42 PADI2 PINFBK
PIN 43 PADI3 PINFBK
PIN 44 PADI4 PINFBK
PIN 45 PADI5 PINFBK
PIN 46 PADI6 PINFBK
PIN 47 PADI7 PINFBK

;---SJ Processing Unit side
;-----Output address/data macrocells
;-----The macrocells are driving the SJ bus during a WRITE or ALE
PIN 23 ADO0
PIN 22 ADO1
PIN 14 ADO2
PIN 12 ADO3
PIN 13 ADO4
```

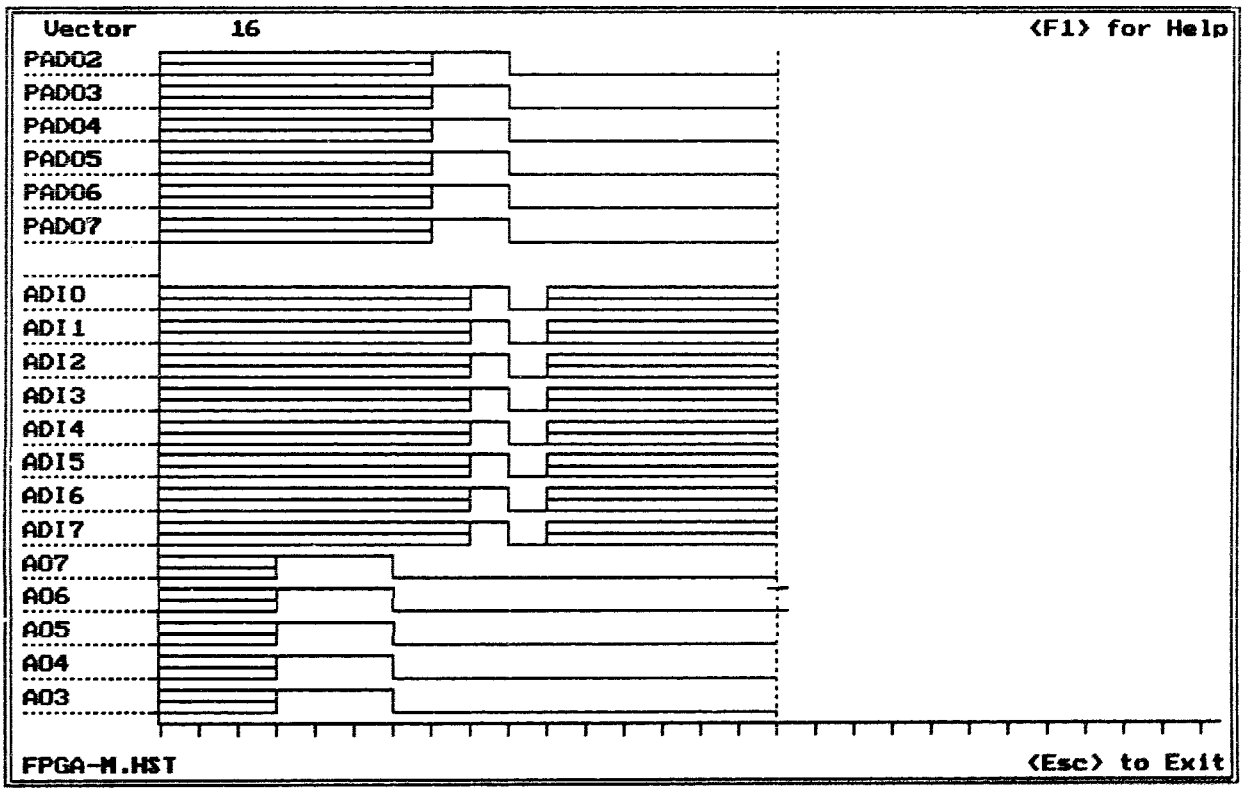
```
PIN 10 ADO5
PIN 11 ADO6
PIN 9 ADO7

;------Input address/data pins
;------The macrocells' pins are used as an input during a READ
PIN 23 PADO0 PINFBK
PIN 22 PADO1 PINFBK
PIN 14 PADO2 PINFBK
PIN 12 PADO3 PINFBK
PIN 13 PADO4 PINFBK
PIN 10 PADO5 PINFBK
PIN 11 PADO6 PINFBK
PIN 9 PADO7 PINFBK


;LSB address register
;8 macrocells are organized as 8-bit register to store
;the LSB of the address at ALE
PIN 63 AO7
PIN 61 AO6
PIN 64 AO5
PIN 60 AO4
PIN 62 AO3
PIN 58 AO2
PIN 59 AO1
PIN 57 AO0


;5 macrocells are used to input the five most significant bits of the address
PIN 39 AI15
PIN 2 AI12
PIN 3 AI13
PIN 4 AI14
PIN 5 AI11

;3 macrocells are used to output the three SJ ID bits
PIN 8 AO11
PIN 7 AO12
PIN 6 AO13
```

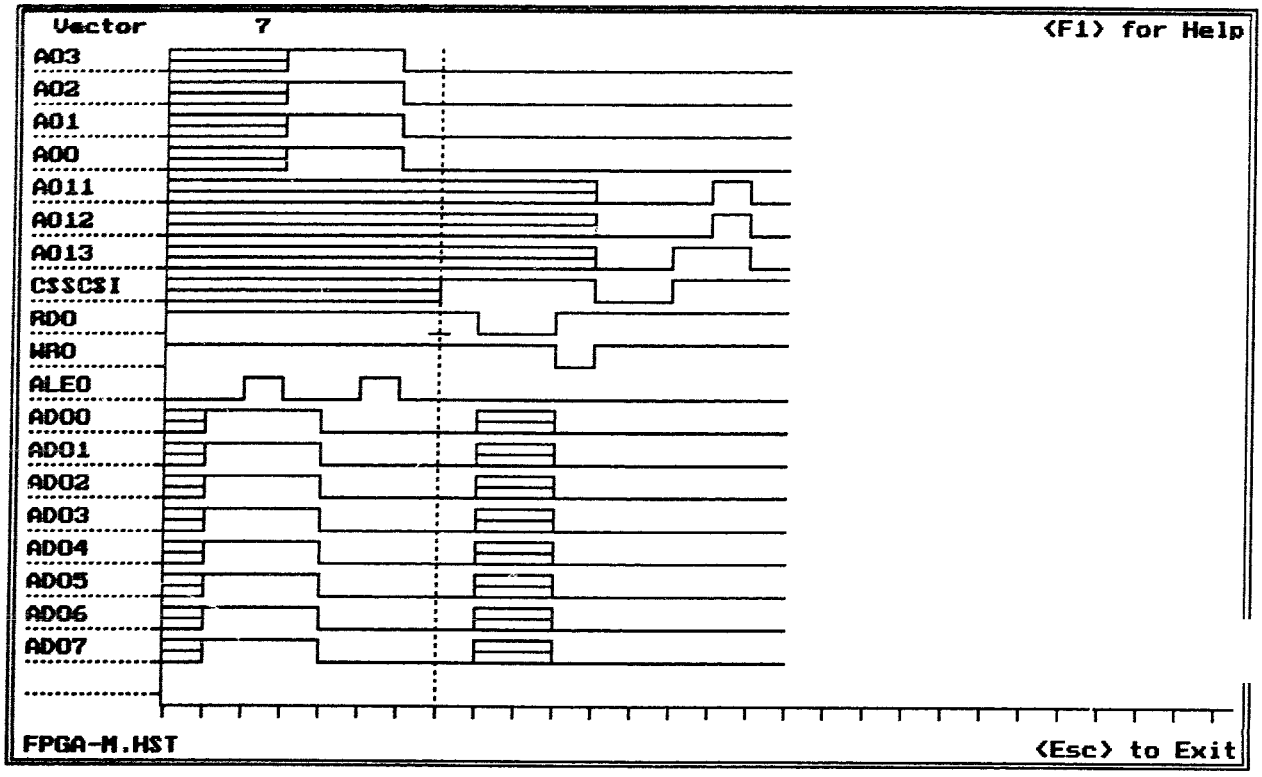
```
;3 macrocells are used to input the bus control signals READ, WRITE and ALE
PIN 48 ALEI
PIN 50 /WRI
PIN 49 /RDI

;SCSI chip-select output
PIN 31 /CSSCSI

;3 macrocells are used to output the three bus control signals
;towards the processing units
PIN 26 /RDO
PIN 25 /WRO
PIN 24 ALEO


EQUATIONS

;Bus control signals buffering
RDO = RDI
WRO = WRI
ALEO = ALEI

;8-bit bidirectional buffer
;---address/data transfer direction control
ADO[0:7].TRST = /RDI
ADI[0:7].TRST = RDI*AI15*AI14

;---Towards the processing unit
ADO[0:7] = PADI[0:7]

;---Towards the microcontroller
ADI[0:7] = PADO[0:7]

;LSB of the address latching
AO[0:7].ACLK = /ALEI
AO[0:7].TRST = VCC
AO[0:7].D := PADI[0:7]

;SCSI chip-select signal generation
CSSCSI = AI15*/AI14
```

```
;Processing units ID bits generation
AO11 = AI15*AI14*AI11
AO12 = AI15*AI14*AI12
AO13 = AI15*AI14*AI13
```

Appendix C.1.2: PDS Design File/Version I : Simulations






Vector 7
<F1> for Help
A03
A02
A01
A00
A011
A012
A013
CSSCSI
RDO
WRO
ALEO
AD00
AD01
AD02
AD03
AD04
AD05
AD06
AD07
FPGA-M.HST
<Esc> to Exit

Appendix C.2.1: PDS Design File/Version II

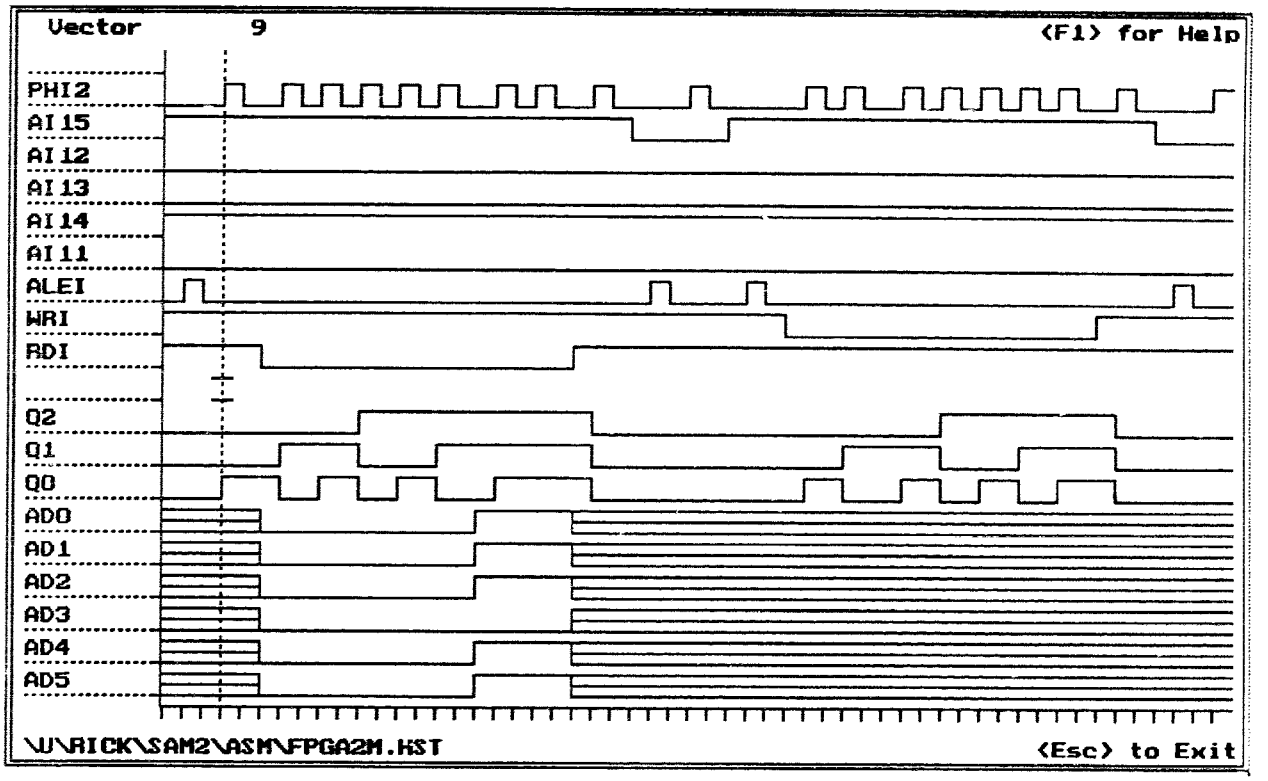
```
;Chip specification: iFX740 68-pin PLCC
CHIP FPGAM iFX740_68

;PIN AND MACROCELL ASSIGNMENTS

;PHI2 has a good phase for falling edge of ale, and rd, wr.
PIN 19 PHI2 input ; use for synchronous timing.

NODE Q[2:0] reg ; for sj state machine.
NODE ADSEL cmbfbk

;I/O address/data macrocells
PIN [40:47] AD[0:7] pinfbk

;Latch for the LSB of the address
PIN 63 AO7 regfbk
PIN 61 AO6 regfbk
PIN 64 AO5 regfbk
PIN 60 AO4 regfbk
PIN 62 AO3 regfbk
PIN 58 AO2 regfbk
PIN 59 AO1 regfbk
PIN 57 AO0 regfbk

;Input the MSB of the address
PIN 39 AI15
PIN 2 AI12
PIN 3 AI13
PIN 4 AI14
PIN 5 AI11


;SJ ID bus output
PIN 8 AO11 reg
PIN 7 AO12 reg
PIN 6 AO13 reg
```

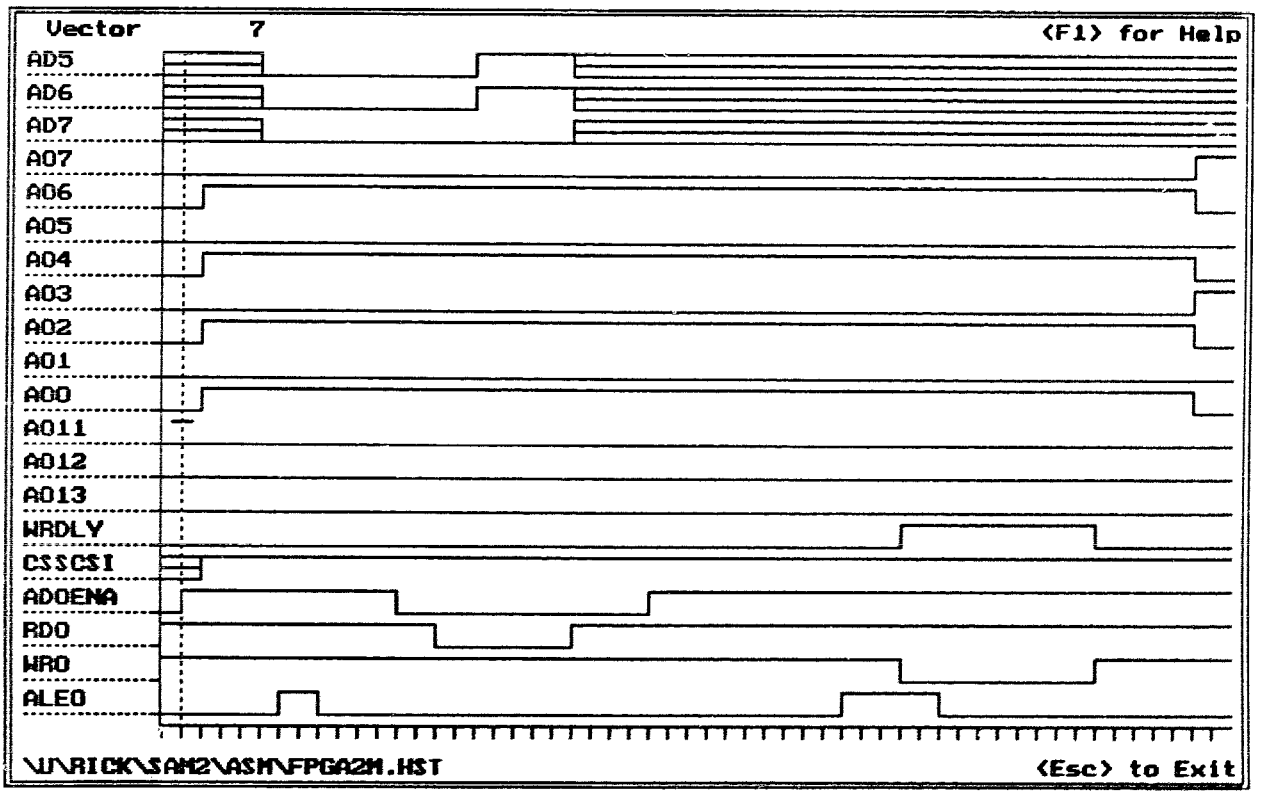
```
;Input bus control signals
PIN 48 ALEI
PIN 50 /WRI
PIN 49 /RDI
;This is to delay the write long enough for the data bus to change from addr to data.
NODE WRDLY

;SCSI chip-select output
PIN 31 /CSSCSI

;Bus control signals output
PIN 26 /RDO
PIN 25 /WRO
PIN 24 ALEO
;Read turnaround control
PIN 27 ADOENA reg

;Address/data SJ bus macrocells
PIN 23 ADsj0 pinfbk
PIN 22 ADsj1 pinfbk
PIN 14 ADsj2 pinfbk
PIN 12 ADsj3 pinfbk
PIN 13 ADsj4 pinfbk
PIN 10 ADsj5 pinfbk
PIN 11 ADsj6 pinfbk
PIN 9 ADsj7 pinfbk

STRING SJ 'ai15*ai14'
STRING SCSI 'ai15*/ai14'
STRING MEM '(rdi+wri)*sj'
STRING SJHLD 'sj*/rdi*/wri'
;string adsel 'q2+/q1*/q0+q1*q0'

;Moore state machine used to retime the bus control signals
;---state assignments:
IDLE = /Q2*/Q1*/Q0
B6 = /Q2*/Q1* Q0
B0 = /Q2* Q1*/Q0
B1 = /Q2* Q1* Q0
B2 = Q2*/Q1*/Q0
```

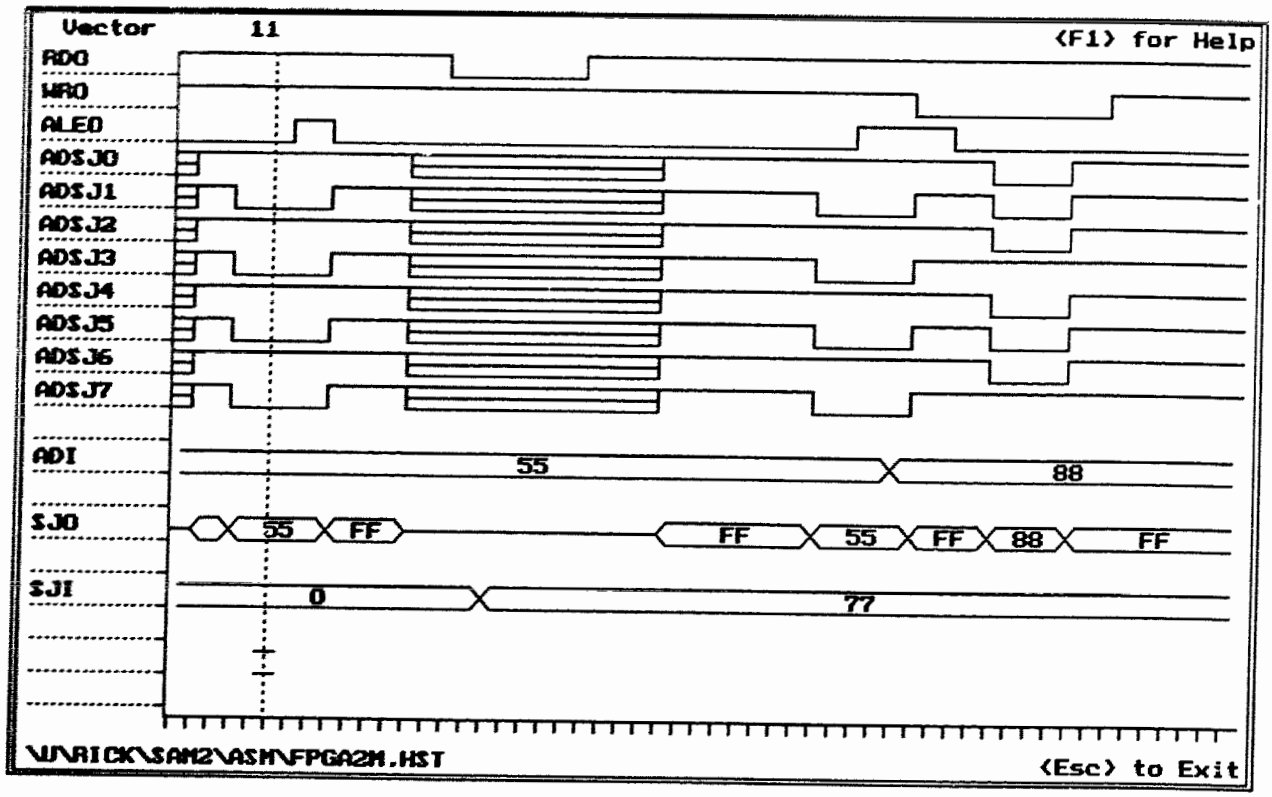
```
B3 = Q2*/Q1* Q0
B4 = Q2* Q1*/Q0
B5 = Q2* Q1* Q0

;---state transitions.
IDLE := MEM -> B0
;SJSEL := MEM -> B0
;+ SJHLD -> SJSEL
B0 := VCC -> B1
B1 := VCC -> B2
B2 := VCC -> B3
B3 := VCC -> B4
B4 := VCC -> B5
B5 := RDI*/WRI -> B5 + WRI*/RDI -> B6
B6 := WRI -> B6 ; stay here to avoid a second ale pulse.


EQUATIONS

Q[2:0].CLKF = /PHI2 ; state machine clock.

;Bus control signals output
RDO = (B4+B5)*RDI
WRDLY = WRI*(B1+B2+B3+B4+B5+B6)
WRO = WRDLY
ALEO = B0+B1*WRI ;extend ale for write but assert wr half way.

;Bidirectional address/data bus
;---control
ADOENA := VCC
ADOENA.RSTF = RDI*B3 ; shut off just before fpga_sj starts to drive.
ADOENA.ACLK = ALEI
ADsj[0:7].TRST = ADOENA
AD[0:7].TRST = RDI*SJ

;---towards the processing unit
;-----output the address in sjsel so it will be stable when aleo goes positive.
;-----then we don't have to delay ale in fpgaxs.
adsel = b0+mem*idle
ADsj[0:7] = ad[0:7].io*/adsel+ ao[0:7].fb*adsel
```

```
;---towards the microcontroller
AD[0:7] = ADsj[0:7].io


;Latching the LSB of the address
AO[0:7].ACLK = /ALEI
AO[0:7].TRST = VCC
AO[0:7].D := AD[0:7].io

;Selecting the SCSI
CSSCSI := SCSI
CSSCSI.ACLK = /ALEI ; clock scsi chip sel to avoid addr transients.

;PMU/DMU ID bits generation
AO11 := AI11*sj
AO12 := AI12*sj
AO13 := AI13*sj
AO[11:13].ACLK = /ALEI
```

Appendix C.2.2: PDS Design File/Version II: Simulations






Vector
11
<F1> for Help
RDO
WRO
ALEO
ADSJ0
ADSJ1
ADSJ2
ADSJ3
ADSJ4
ADSJ5
ADSJ6
ADSJ7
ADI
55
88
SJO
55
FF
FF
55
FF
88
FF
SJI
0
77
\U\RICK\SAM2\ASM\FPGA2M.HST
<Esc> to Exit

Appendix D: SJ Processing Unit FPGA Design Files

This appendix contains the processing unit FPGA PLDShell Plus (.PDS) design files.

The first and second improved versions of the FPGA-based interface and control logic design files are given along with the simulation signal wave forms. The logic was designed and simulated using PLDShell Plus FPGA design software.

Appendix D.1.1: PDS Design File/Version I

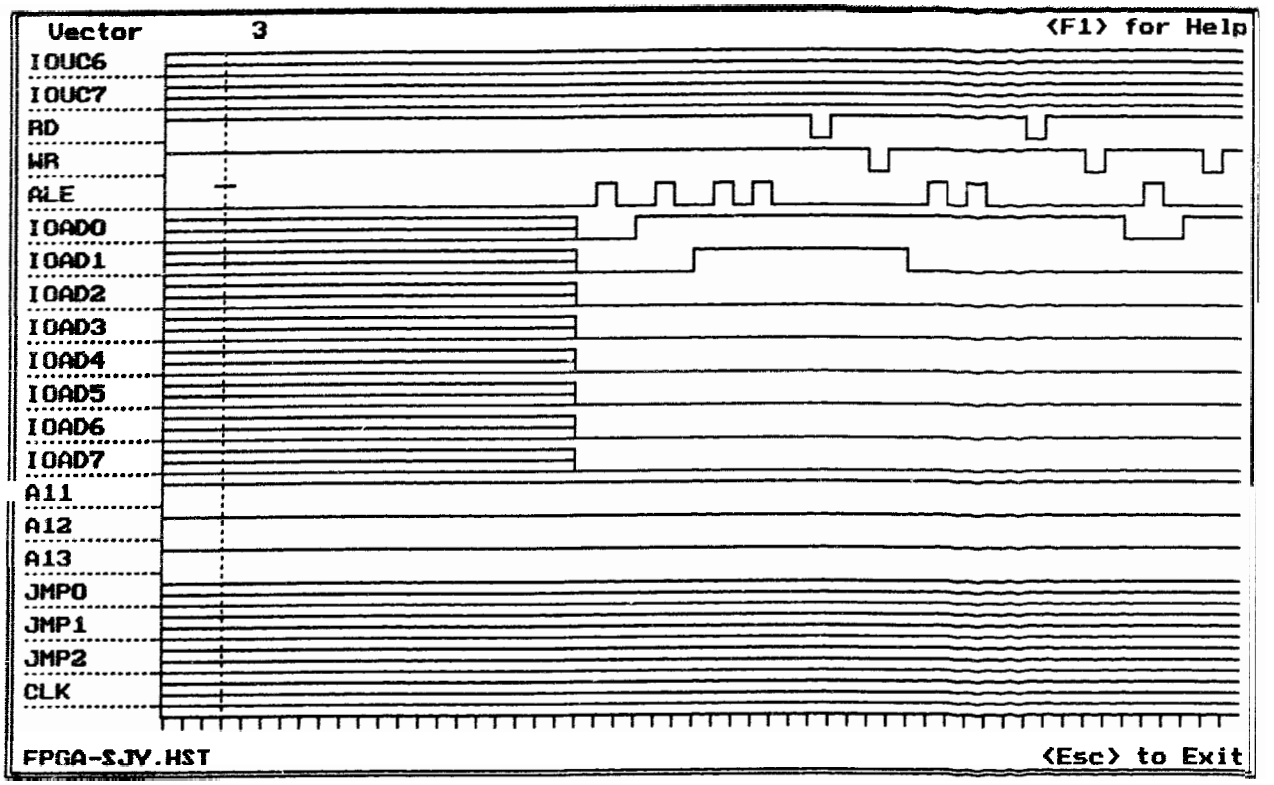
```
;Chip specofocation: iFX740 68-pin PLCC
CHIP FPGAM iFX740_68

;PIN AND MACROCELL ASSIGNMENTS

;Y Bus
;---Y register output enable
PIN 4 OE
;---Y register macrocells
PIN 6 Y0
PIN 7 Y1
PIN 8 Y2
PIN 9 Y3
PIN 10 Y4
PIN 11 Y5
PIN 12 Y6
PIN 13 Y7
;---Y register I/O pins
PIN 6 INY0 PINFBK
PIN 7 INY1 PINFBK
PIN 8 INY2 PINFBK
PIN 9 INY3 PINFBK
PIN 10 INY4 PINFBK
PIN 11 INY5 PINFBK
PIN 12 INY6 PINFBK
PIN 13 INY7 PINFBK


;SJCP control and address/data bus
;---SJCP control signals: CTR0 and CTR1
PIN 14 CTR0 OUTPUT
PIN 22 CTR1 OUTPUT
;---address/data bus macrocells
PIN 23 UC0
PIN 24 UC1
PIN 25 UC2
PIN 26 UC3
PIN 27 UC4
```

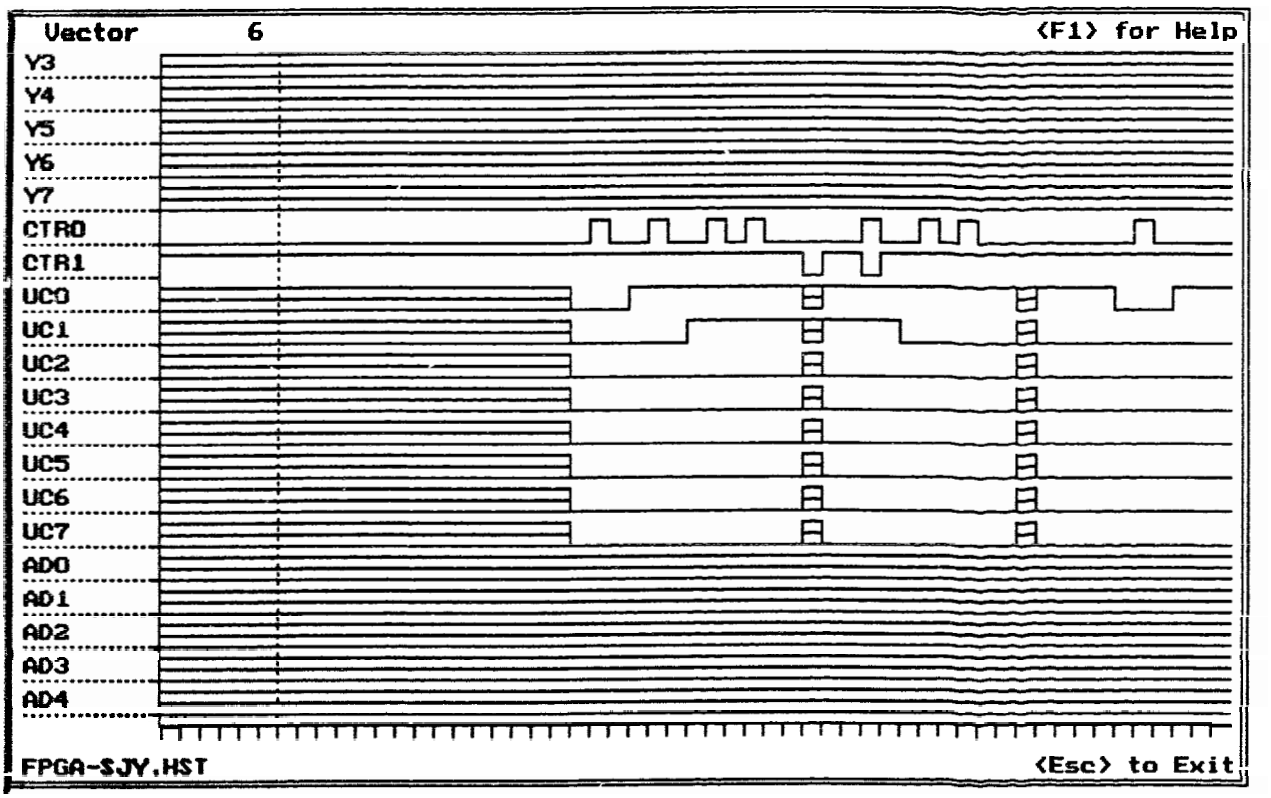
```
PIN 28 UC5
PIN 29 UC6
PIN 30 UC7
;---address/data bus I/O pins
PIN 23 IOUC0 PINFBK
PIN 24 IOUC1 PINFBK
PIN 25 IOUC2 PINFBK
PIN 26 IOUC3 PINFBK
PIN 27 IOUC4 PINFBK
PIN 28 IOUC5 PINFBK
PIN 29 IOUC6 PINFBK
PIN 30 IOUC7 PINFBK


;Microcontroller bus
;---microcontroller bus control signals
PIN 36 RD INPUT
PIN 37 WR INPUT
PIN 38 ALE INPUT
;---microcontroller bus macrocells
PIN 39 AD0
PIN 40 AD1
PIN 41 AD2
PIN 42 AD3
PIN 43 AD4
PIN 44 AD5
PIN 45 AD6
PIN 46 AD7
;---microcontroller bus I/O pins
PIN 39 IOAD0 PINFBK
PIN 40 IOAD1 PINFBK
PIN 41 IOAD2 PINFBK
PIN 42 IOAD3 PINFBK
PIN 43 IOAD4 PINFBK
PIN 44 IOAD5 PINFBK
PIN 45 IOAD6 PINFBK
PIN 46 IOAD7 PINFBK
```

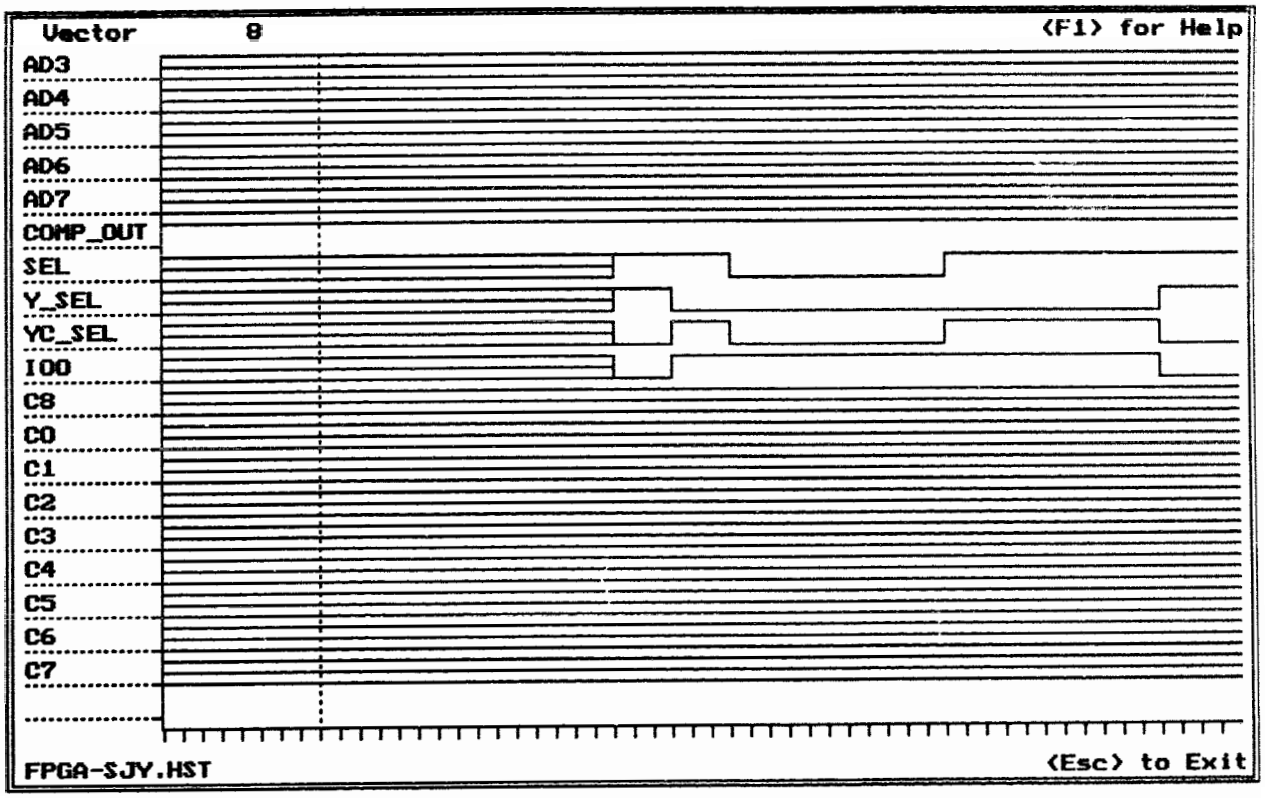
```
;ID address lines input pins
PIN 58 A11 PINFBK
PIN 59 A12 PINFBK
PIN 60 A13 PINFBK

;Comparator and selection logic
PIN 5 COMP_OUT
PIN 31 SEL
PIN Y_SEL
PIN YC_SEL
PIN 48 IO0

;ID jumpers input pins
PIN 61 JMP0 PINFBK
PIN 63 JMP1 PINFBK
PIN 65 JMP2 PINFBK

;Counter
;---control
PIN 19 CLK
PIN 2 REFRESH
;---counter macrocells
PIN 57 C8 REGFBK
PIN 58 C0 REGFBK
PIN 59 C1 REGFBK
PIN 60 C2 REGFBK
PIN 61 C3 REGFBK
PIN 62 C4 REGFBK
PIN 63 C5 REGFBK
PIN 64 C6 REGFBK
PIN 65 C7 REGFBK

EQUATIONS

;ID check
COMP_OUT.CMP = [JMP2,JMP1,JMP0] == [A13,A12,A11]
COMP_OUT = A13*A12*A11

;Y register selection at address 00H
SEL.ACLK = /(COMP_OUT*ALE)
```

```
SEL.D := /IOAD1*/IOAD2*/IOAD3*/IOAD4*/IOAD5*/IOAD6*/IOAD7
IO0.ACLK = /(COMP_OUT*ALE)
IO0.D := IOAD0
Y_SEL = SEL*/IO0
YC_SEL = SEL*IO0

;Control signals generation
CTR0 = COMP_OUT*(ALE + /WR*/SEL)
CTR1 = /COMP_OUT + RD*WR + SEL

;Bidirectional control/address/data bus control
UC[0:7] = IOAD[0:7]
UC[0:7].TRST = COMP_OUT*RD

AD[0:7] = IOUC[0:7]*/Y_SEL*/YC_SEL + C[0:7]*(Y_SEL+YC_SEL)
AD[0:7].TRST = COMP_OUT*/RD

;Latching the data in Y register
Y[0:7].D := IOAD[0:7]
Y[0:7].ACLK = /(COMP_OUT*Y_SEL*/WR)
Y[0:7].TRST = OE

;Counter logic
C[0:8].CLKF = CLK
C[0:8].RSTF = REFRESH
C8.T := VCC
C0.T := (C0:+:INY0)*Y_SEL + C8*/Y_SEL
C1.T := (C1:+:INY1)*Y_SEL + C8*C0*/Y_SEL
C2.T := (C2:+:INY2)*Y_SEL + C8*C0*C1*/Y_SEL
C3.T := (C3:+:INY3)*Y_SEL + C8*C0*C1*C2*/Y_SEL
C4.T := (C4:+:INY4)*Y_SEL + C8*C0*C1*C2*C3*/Y_SEL
C5.T := (C5:+:INY5)*Y_SEL + C8*C0*C1*C2*C3*C4*/Y_SEL
C6.T := (C6:+:INY6)*Y_SEL + C8*C0*C1*C2*C3*C4*C5*/Y_SEL
C7.T := (C7:+:INY7)*Y_SEL + C8*C0*C1*C2*C3*C4*C5*C6*/Y_SEL
C[0:8].TRST = GND
```

Appendix D.1.2: PDS Design File/Version I: Simulation






Vector 8
<F1> for Help
AD3
AD4
AD5
AD6
AD7
COMP_OUT
SEL
Y_SEL
YC_SEL
100
C8
C0
C1
C2
C3
C4
C5
C6
C7
FPGA-SJY.HST
<Esc> to Exit

Appendix D.2.1: PDS Design File/Version II

```
;Chip specification: iFX740 68-pin PLCC
CHIP FPGAM iFX740_68

;PIN AND MACROCELL ASSIGNMENTS

;FPGA3S uses a local unit select bit for SJI/O.
;commands needed: set u (single addr or no addr)
;broadcast High and Low (need to single out pmu)
;broadcast all; local rd/wr, based upon unit #
;broadcast should work with local Y reg.

;Added new timing scheme, 95.04.17, rfh.
;bus turnaround is handled by FPGA2M (shuts off its own drive with a
;margin around the sjcp read.
;this design gates ale with clk2 to catch the low addr before
;the write line falls, which is too late.

;Y bus
;---control
PIN 4 OE input
;---macrocells
PIN [6:13] Y[0:7] regfbk ; PINFBK

;SJCP control/address/data bus
;---control signals macrocells
PIN 14 CTR0 OUTPUT
PIN 22 CTR1 OUTPUT
;---delay output to avoid driving ucbus when the address is coming out (t1).
PIN 31 CTR1Z CMBFBK
NODE CTR1P1 CMBFBK
NODE CTR1P2 CMBFBK
NODE CTR1P3 CMBFBK
NODE CTR1P4 CMBFBK
NODE CTR0P1 CMBFBK
NODE CTR0P2 CMBFBK
NODE CTR0P3 CMBFBK
;NODE CTR0P4 CMBFBK
NODE TRSTCTL CMBFBK
```

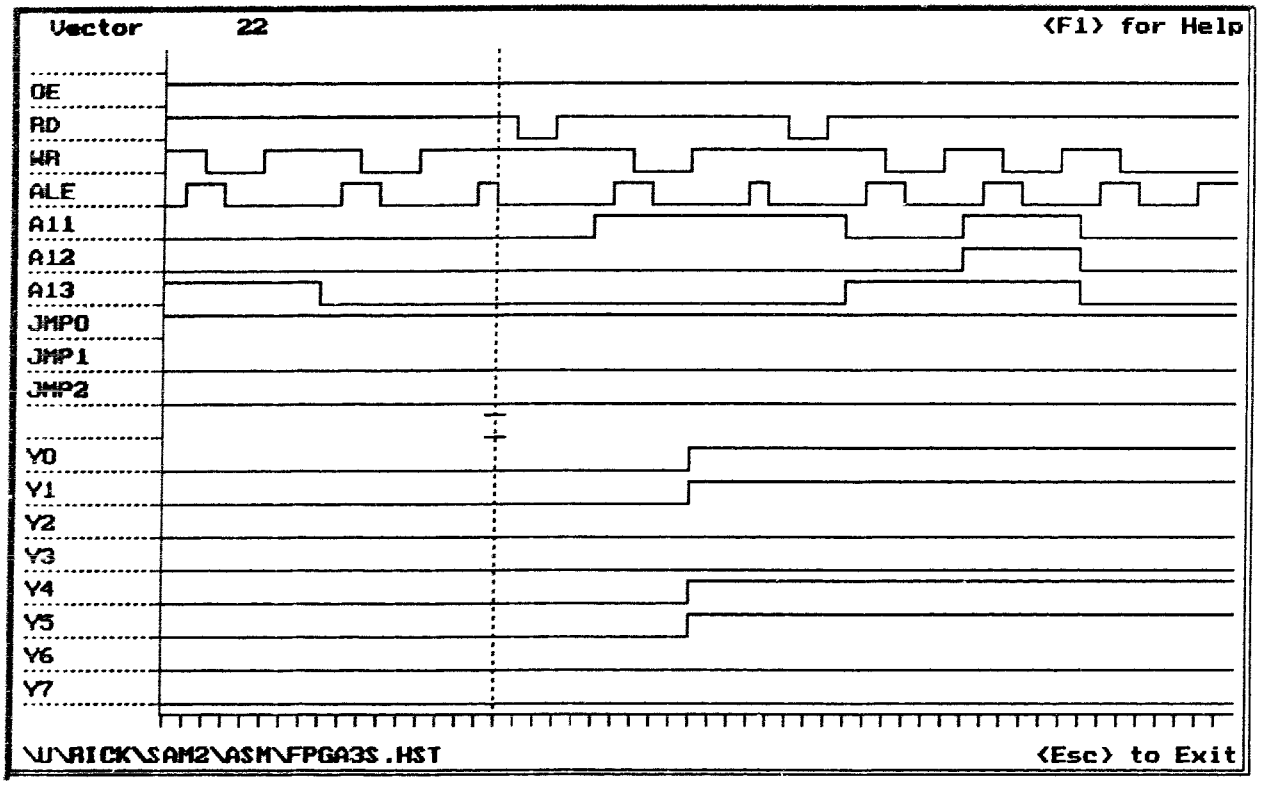
```
;---SJCP control bus macrocells
PIN [23:30] UC[0:7] PINFBK

;Microcontroller bus
;---these have delayed timing from FPGA2M.
;---address should be stable when ale rising edge comes along.
PIN 36 /RD INPUT
PIN 37 /WR INPUT
PIN 38 ALE INPUT

;---Microcontroller bus macrocells -- ad goes to dallas.
PIN [39:46] AD[0:7] PINFBK

;ID address lines input
PIN [58:60] A[11:13] INPUT

;Comparator
PIN 56 UNIT REG OUTPUT
PIN 57 Y_SEL REG OUTPUT

;ID jumpers input
PIN 61 JMP0 INPUT
PIN 63 JMP1 INPUT
PIN 65 JMP2 INPUT

PIN 62 COMP_OUT CMBFBK
PIN 64 RUNNING REG OUTPUT
PIN 5 BRDCST REGFBK OUTPUT

;Selection string definitions
STRING USEL 'A13'
STRING ALL 'A13*A12*A11'
STRING UPPER 'A13*A12*/A11'
STRING LOWER 'A13*/A12*A11'
STRING SINGLE 'A13*/A12*/A11'
STRING LOCAL '/A13*/A12*A11'
STRING SJ '/A13*/A12*/A11'
STRING YSEL '/AD0' ; MUST BE ADDRESS PHASE OF DATA BUS.
STRING RUN '/AD7*AD6*/AD5*/AD4*/AD3*/AD2*AD1*AD0'
STRING STOP '/AD7*AD6*/AD5*/AD4*/AD3*/AD2*/AD1*AD0'
```

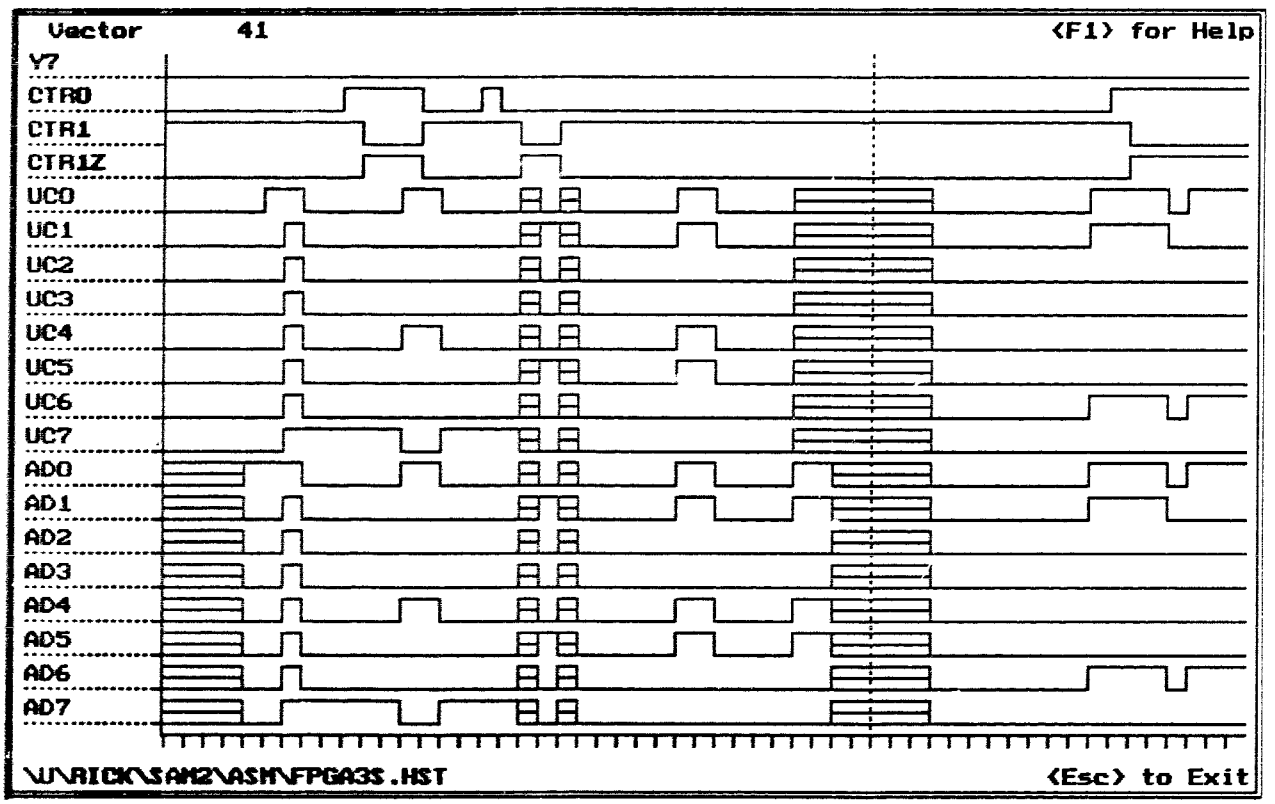
EQUATIONS

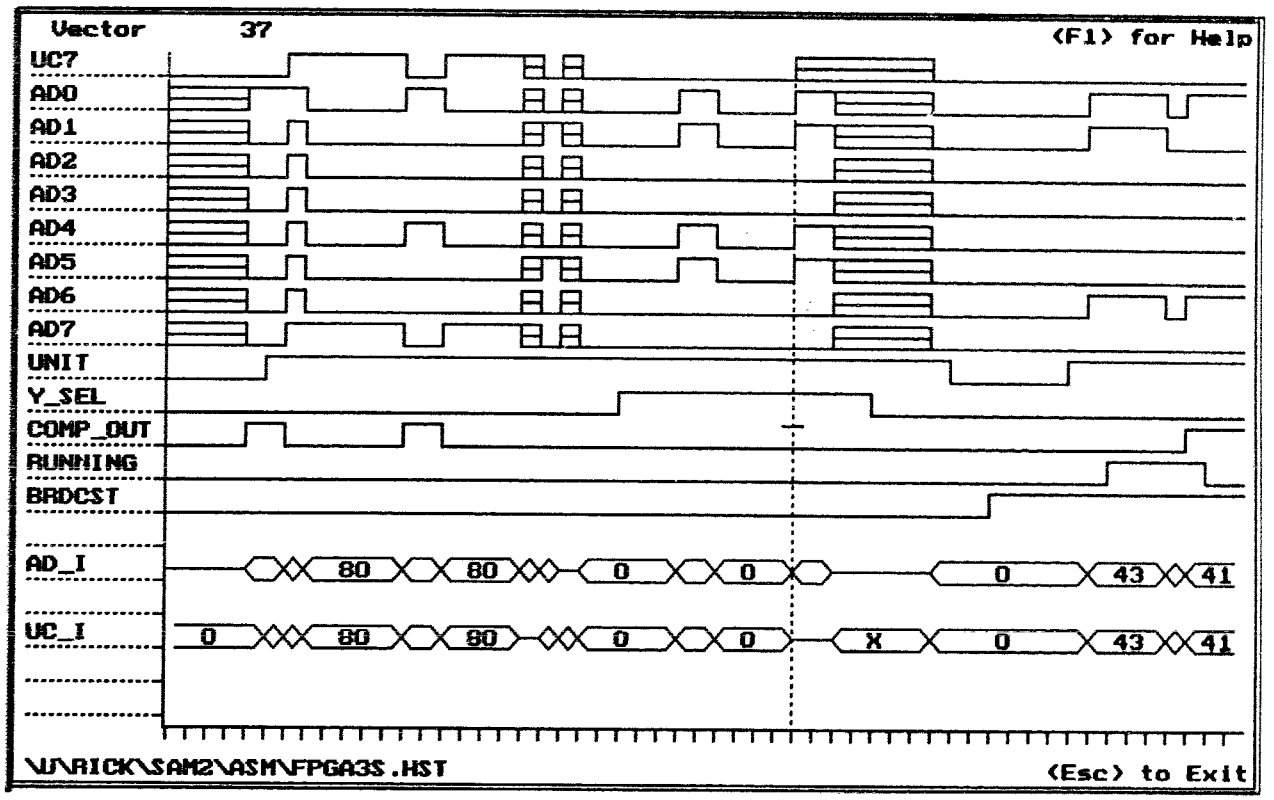
```
;Unit selection equations
;---broadcast selection
BRDCST := ALL+UPPER+LOWER ; use to prevent rd shorts.
BRDCST.ACLK = ALE*USEL ; don't wait for data, any broadcast address works.

;---comparison and unit selection
COMP_OUT.CMP = [AD[2:0]] == [JMP[2:0]]
UNIT := ALL+UPPER*JMP2+LOWER*/JMP2+SINGLE*COMP_OUT
UNIT.ACLK = /(USEL*WR)

;---Y register selection
Y_SEL.ACLK = ALE
Y_SEL := LOCAL*UNIT*YSEL

;---running mode
RUNNING.T := SJ*RUN*/RUNNING*UNIT + SJ*STOP*RUNNING*UNIT
RUNNING.ACLK = ALE


;Control signals generation
;---nop:10, ale:11, rd:00, wr:01
CTR0P1 = (ALE+WR)*SJ*UNIT
CTR0P2 = CTR0P1
CTR0P3 = CTR0P2
; CTR0P4 = CTR0P3
CTR0 = CTR0P1
CTR1Z = (RD*/BRDCST+WR)*SJ*UNIT ; user uses Unit to select for broadcast.
CTR1P1 = /CTR1Z ; delayed to avoid ale glitch passing from wr to nop.
CTR1P2 = CTR1P1
CTR1P3 = CTR1P2
CTR1P4 = CTR1P3
; CTR1P5 = CTR1P4
CTR1 = CTR1P1
```

```
;Bidirectional address/data bus control
;---towards SJCP
UC[0:7] = AD[0:7]*UNIT
TRSTCTL = (WR+ALE)*SJ*UNIT
UC[0:7].TRST = TRSTCTL ; shut off except for ale+wr.

;---towards the microcontroller
AD[0:7] = UC[0:7]*/Y_sel + Y[0:7]*Y_sel
AD[0:7].TRST = RD*UNIT*/BRDCST ; activate for board selected read.


;Latching the data in the Y register
Y[0:7] := AD[0:7]
Y[0:7].ACLK = /(Y_SEL*WR)
Y[0:7].TRST = OE ; OE COMES FROM SJCP.
```

Appendix D.2.2: PDS Design File/Version II: Simulation






Vector 37
<F1> for Help
UC7
AD0
AD1
AD2
AD3
AD4
AD5
AD6
AD7
UNIT
Y_SEL
COMP_OUT
RUNNING
BRDCST
AD_I
80
80
0
0
0
43
41
UC_I
0
80
80
0
0
X
0
43
41
\U\RICK\SAM2\ASM\FPGA3S.HST
<Esc> to Exit

Appendix E: Memory Map of SJCP special function memory

Table 13: Memory Map of SJCP data and special function memory

Address Range	Feature	X-Function
8x - Fx	Dual-Port Memory (DPM) block access	
0x	Not used in SJCP. Used to access FPGA registers.	x = 0 - selects the Y register
1x	Scan chains selection. Strobes the data inputs into the selected scan chain.	x = 0 - selects address scan-chain. x = 1 - selects data scan-chain. x = 2 - selects instruction scan-chain.
2x	Scan chains I/O.	x = 0 - selects address scan-chain. x = 1 - selects data scan-chain. x = 2 - selects instruction scan-chain.
3x	SJCP external SRAM bus control.	x = 0 - OE on x = 1 - OE off x = 2 - WE strobe
4x	Program execution control.	x = 0 - executes a step. x = 1 - Stop^trap SJCP. x = 2 - Clear trap. x = 3 - Run full-speed.
5x	Event interface control.	x = 0 - sets DPM data flag. x = 3 - Clears IRQ.
6x	Not used.	
7x	DPM block base address register select.	x = 3 - Loads DPM base address register.

Glossary

ASPI	Advanced SCSI Protocol Interface
BIST	Built-In Self-Test
CDB	Command Descriptor Block
CFB	Configurable Functional Block
MIMD	Multiple Instruction Multiple Data
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PDS	Extension of the PLDShell Plus design file - .pds
PLDShell Plus	FPGA design package provided by Intel
PROMICE	EPROM Emulator
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
SAM-II	Structured Architecture Machine - II
SCSI	Small Computer System Interface
SIMD	Single Instruction Multiple Data
SJ	SAM Junior
SJ BUS	SAM Junior Bus
SJCP	SAM Junior Control Processor
SJMI	SAM Junior Memory Interface
SJNI	SAM Junior Network Interface

Bibliography

[1] Fujitsu Microelectronics Inc., "Fast track to SCSI"
Prentice Hall, Englewood Cliffs, NJ, 1991

[2] NCR Corporation, "Understanding the Small Computer Systems Interface"
Prentice Hall, Englewood Cliffs, NJ, 1990

[3] American National Standards Institute Inc., Small Computer System Interface (SCSI)",
American National Standards Institute, New York, 1986

[4] Adaptec Inc., "AHA-154X Host Adapters Programming Guide"
Adaptec Inc., Milpitas, CA, 1990

[5] Adaptec Inc., "AHA-154X Technical Reference Manual"
Adaptec Inc., Milpitas, CA, 1990

[6] Adaptec Inc., "Adavanced SCSI Protocol Interface (ASPI)"
Adaptec Inc., Milpitas, CA, 1990

[7] Advanced Micro Devices, "Enhanced SCSI-Bus Interface Controller"
Advanced Micro Devices, Sunnyvale, CA, 1989

[8] Intel Corporation, "PLDshell Plus and PLDasm" User's Guide V3.1
Intel Corporation, Santa Clara, CA, 1993

[9] Dallas Semiconductor, "High-Speed Micro" V1.0 Draft Copy
Dallas Semiconductor, Dallas, TX, 1993

[10] Grammar Engine Inc., "Promice Userman"
Grammar Engine Inc., Columbus, OH, 1991

[11] Intel Corporation, "Advanced Information iFX740"
Intel Corporation, Santa Clara, CA, 1993

[12] Advanced Micro Devices, "Microcontrollers Data Book/Handbook"
Advanced Micro Devices, Sunnyvale, CA, 1988

[13] Dallas Semiconductor, "DS80C320, High-Speed Micro"
Dallas Semiconductor, Dallas, TX, 1993

[14] Advanced Micro Devices, "EPROM Products Data Book/Handbook"
Advanced Micro Devices, Sunnyvale, CA, 1994

[15] Micron Technology Inc., "DRAM Data Book"
Micron Technology Inc., Boise, Idaho, 1992

[16] Bogdan Lent, "Dataflow Architecture for Machine Control"
Research Studies Press Ltd., Taunton, Somerset, England 1989

[17] A.T. Kundig, "A Note on the meaning of Embedded Systems"
Embedded Systems: A New Approach to Their Formal Description and Design. An Advanced Course. Zurich, Switzerland, Springer 1987

[18] D. Gajski, F. Vahid, S. Narayan, J. Gong, "Specification and Design of Embedded Systems",
Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1994

[19] J. Black, "The System Engineer's Handbook",
Academic Press Inc., San Diego, California, 1992

[20] Phoenix Technologies Ltd., "System BIOS for IBM PC/XT/AT Computers and Compatibles"
Addison-Wesley Publishing Company Inc., 1988

[21] Leo J. Scanlon, "IBM PC XT/AT Assembly Language"
Simon&Schuster Publishing Company, NY, 1985

[22] Borland International, "Turbo C"
Borland International, Scotts Valley, CA, 1988

[23] Richard F. Hobson, "Combining Boundary Scan with I/O and other System Functions to Reduce System Complexity",
Microelectronics Journal, 23 (1992) 179-184

[24] Wang, Laung-Terug, Marhoefer, Michael, and McCluskey, Edward J., "A Self-Test and Self-Diagnosis Architecture for Boards Using Boundary Scans", Proceedings 1st European Test Conference,
IEEE Press, April 1989, pp.119-126

[25] van Riessen, R.P., Kerkhoff, H.G., and Kloppenburg, A., "Design and Implementation of a Hierarchical Testable Architecture Using the Boundary Scan Standard", Proceedings 1st European Test Conference,
IEEE Press, April 1989, pp.112-118

[26] Tulloss, R.E., Yau, C.W., "BIST & Boundary Scan For Board Level Test: The Program Pseudocode", Proceedings 1st European Test Conference,
IEEE Press, April 1989, pp.106-111

[27] van de Lagemaat, Dick, "Testing Multiple Power Connections with Boundary Scan", Proceedings 1st European Test Conference,
IEEE Press, April 1989, pp.127-130

[28] Texas Instruments, "Scope Scan Path Support Devices", Product Bulletin Texas Instruments, Dallas, TX, 1990

[29] Language Processor Inc., "CodeWatch, An Interactive Source-Level Debugger"
Prentice Hall,Englewood Cliffs, New Jersey, 1989

[30] Ehud Shapiro, "Algorithmic Program Debugging"
The MIT Press, PhD Thesis, 1982

[31] Borland International Inc. "Turbo Debugger"
Borland International Inc., Scotts Valley, CA, 1990

[32] Robwrt H. Netzer, "Optimal Tracing and Replay for Debugging Shared Memory Parallel Programs", Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging,
San Diego, California, 1993

[33] M.Timmerman, F. Gielen, P. Lambrix, "High-Level Tools for debugging Real-Time Multiprocessor Systems", Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging,
San Diego, California, 1993

[34] J. May, F. Berman, "Panorama: A Portable Extensible Parallel Debugger" Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, California, 1993

[35] C. Valot, "Characterizing the Accuracy of Distributed Timestamps", Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging, San Diego, California, 1993

[36] I. Elshoff, "A Distributed Debugger for Amoeba",
Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging,
Madison, Wisconsin, 1988, pp. 1 - 10

[37] C. Lin, R. Leblanc, "Event-Based Debugging of Object/Action Programs",
Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging,
Madison, Wisconsin, 1988, pp. 23 - 34

[38] Z. Aral, I. Gertner, "High-Level Debugging in Parasight"
Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging,
Madison, Wisconsin, 1988, pp. 151 - 162

[39] B. Bruegge, T. Gross, "A Program Debugger for a Systolic Array"
Proceedings ACM/ONR Workshop on Parallel and Distributed Debugging,
Madison, Wisconsin, 1988, pp. 174 - 182

[40] R. Hobson, "High-Level Microprogramming Support Embedded in Silicon"
IEE Proceedings, Vol. 135, Pt. E, No. 2, 1988, pp. 73 - 81

[41] R. Hobson, "An Outline of Objectives for the SAM-II Parallel Computer Prototype"
8th International Symposium on Parallel Processing:
Parallel Systems Fair, April 1994, pp. 37 - 42

[42] J. Tomlinson, "Avoid the Pitfalls of High-Speed Logic Design"
Electronic Design Magazine, November 1989

[43] B. Burgess, N. Ullah, P. Overen, D. Ogden, "The Power PC 603 Microprocessor"
ACM/Communications, June 1994 - Volume 37, Number 6, pp. 34 - 41

[44] A. Poursepanj, "The Power PC Performance Modelling Methodology"
ACM/Communications, June 1994 - Volume 37, Number 6, pp. 47 - 55

[45] Ray Weiss, Craig Haller, Nick Lethaby, "Special Report on Debugging Embedded Systems"
Computer Design Magazine, July 1995, pp. 69 - 86

[46] R. Hobson, J. Hoskin, J. Simmons, R. Spilsbury, "SAM-I: a prototype machine for dynamic, array-oriented programming languages"
IEE Proceedings-E, Vol. 139, No. 4, July 1992, pp. 335- 347