(1)

Report DAAD07-89-C00212-FTR

REAL-TIME SIGNAL PROCESSING SYSTEM

Dr. Michael Andrews
Space Tech Corporation
125 Crestridge Drive
Fort Collins, CO 80525-3900

29 October 1992

Final Technical Report

Contract No. DAAD07-89-C-0212

5 October 1989 - 29 October 1992

The views, opinions, and/or findings contained in this report are those of the authors and should not be construed as official Department of the Army position, policy, or decision, unless so designated by other official documentation.

Prepared for
USA White Sands Missile Range
STEWS-ID
WSMR, NM 88002-5201

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF COLOR PAGES WHICH DO NOT REPRODUCE LEGIBLY ON BLACK AND WHITE MICROFICHE.

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for Public Release: Distribution is Unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) DAAD07-89-C-00212-FTR	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Space Tech Corporation 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION U.S. Army White Sands Missile Range
6c. ADDRESS (City, State, and ZIP Code) 125 Crestridge Drive Fort Collins, CO 80525-3900	7b. ADDRESS (City, State, and ZIP Code) Commander, U.S. Army White Sands Missile Range, ATTN: STEWS-ID-TA, WSMR, NM 88002-5144
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DAAD07-89-C-0212
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 665502 / PROJECT NO. 1P65502M40 / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Real-Time Signal Processing Systems

12. PERSONAL AUTHOR(S)
Michael Andrews

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Technical	FROM 891001 TO 92825	92 Oct 29	372

16. SUPPLEMENTARY NOTATION The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other official documentation.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Real-Time Processors, Microprogrammable Processors, Crossbar Switch Chips, Linear Systems, Given Rotations, Gram-Schmidt Decomposition, Matrix Inversion.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Develop Expandable Victor Accelerator (EVA) and its hardware and software capable of processing range and range rate data, digital focus, real-time Kalman filtering, real-time target motion resolution (TMR), and processing image/pattern information for real-time optical trackers of multi-munitions scenes.

EVA consists of two major but tightly coupled components, a Vector Processing Hardware (VPH) and a Cascadable Processing Hardware (CPH).

VPH is a speed optimized architecture capable of processing vectors of complex data. The architecture is based upon the utilization of multiple Zoran VSP-325 chips. CPH is also a speed optimized. However, the architecture is configured for those applications where the concern for high precision and wide dynamic range is at a premium. CPH hardware utilizes multiple Bipolar Integrated B2110/B2120 and multiple 12x14 CrossBar custom chips.

(continued on reverse)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL FOO W. LAM	22b. TELEPHONE (Include Area Code) (505) 678-3010 / 22c. OFFICE SYMBOL STEWS-ID-TA

DD Form 1473, JUN 86 Previous editions are obsolete. SECURITY CLASSIFICATION OF THIS PAGE

19. ABSTRACT (Continued)

VPH was fabricated, completed and demonstrated successfully as proposed. However, CPH is incomplete with only 70% of the design accomplished. The Cross Bar custom chips were fabricated and completed.

TABLE OF CONTENTS

LIST OF FIGURES

1.0 Introduction

A Phase II SBIR contract was awarded to Space Tech Corporation to develop a new computer architecture for WSMR STEWS-ID-TA. Foo Lam was technical monitor and was assisted by John Williams. Michael Andrews was the principal investigator at Space Tech. Several Space Tech employees were involved with this effort. Steve Hall was responsible for the early design concepts of the CPH. Larry Hall was responsible for the VPH design effort. Jeff Weideman worked on the cache, address generator, IOP, and VME buffer boards. James Ott worked on the cache board. Phil White tested the crossbars and finalized the backplane design. John Stevens generated the IO drivers and Steve Sharp contributed to the VPH coding.

Major DOD agencies found that to upgrade their hardware development systems to keep up with advancing technology remains a large effort. Yet, a major hidden cost is more than a simple acquisition of equipment. Engineer retraining and software redevelopment easily magnify the total system costs. In early 1980, Foo Lam at the Instrumentation Directorate at White Sands Missile Range discovered a uniquely innovative solution: build a hardware emulator that can be universally applied across several life times of architectural technologies and modify only the microcode. Hence, a fixed and constant cost will remain in contrast to an escalating level of effort each time the next hottest microprocessor comes out.

White Sands Missile Range like most other test ranges must constantly upgrade computing facilities to take advantage of cost effective solutions. A proliferation of different microprocessors and development systems spread among the several laboratories reduces the commonality of effort. Code written in one application is likely to be unsuitable to another. Testing such code is also challenging when dissimilar hardware is encountered. A type of universal or meta-machine would help minimize portability constraints.

In response to this need, Lam's meta-architecture was discovered that could emulate many diverse types of microprocessors from RISC to CISC. Aptly called the Cascadable Processor Hardware, the CPH machine can be easily microcoded. More importantly, the architecture can be made to emulate any wordlength from 8- to 128-bits. Fixed-point and floating-point arithmetic for IEEE and DEC formats are executed. Special fast DSP routines are microcoded so that mere calling routines need be executed. And because of the microcode capability, a user can program in the language of his desired microprocessor. Two significant cost savings accrue. First, the ARMY proponent need no longer purchase costly development systems each time another micro wants to be incorporated. Second, he need not have to sacrifice real-time emulation because the CPH is really a sixth generation architecture, mostly capable of emulating architectures int the early 2000s.

Initial architectural studies were completed by Dr. Javin Taylor at New Mexico State University. Later, Space Tech Corporation was awarded a Phase I and Phase II effort to respond to this requirement. As a result a novel architecture was designed that is fast, flexible, and cascadable. The long-term goals of Mr. Lam's visionary architecture achieves the following objectives. Cascadability is easily supported by merely plugging into the backplane another processor and no new microcode is necessary.

The heart of the architecture is a fully concurrent crossbar chip. The novel chip is a 12x14 port switcher which can be dynamically configured in only one clock cycle (currently 20 nsecs). The chip is also directly cascadable so that extensible wordlengths can be supported in hardware with no software cycle penalties. The crossbar chip is employed in the processor section and the address generator section thus attesting to its universality. No doubt, the crossbar will find equal applications in modem switchers, beam splitters, antenna beam formers, telemetry, telephony, and massively parallel processing architectures.

During this Phase II effort, a microprogramming development tool was designed called MICROASM. This tool development was jointly funded by support from a Phase II SBIR contract with WSMR and SDC-Huntsville. Mr. K. Pathak sponsored this work at SDC.

This report is organized as follows. The early sections describe the developmental history of the Phase I and II projects. A reading is helpful to understand the eventual device selections for the functional units. A very brief description of the units and the overall EVA architecture can be found in Section 2 as well. Section 3 begins the detailed explanation of the resources including the operation of those modules that have been fabricated such as the VPH. Section 4 introduces some of the concepts in programming the CPH. Section 5 describes the microprogramming tool, MicroAsm, which will be important when the CPH is to be coded. Sections 6 and 7 discuss the results and suggestions for future work.

1.1 Developmental History of EVA (Extendable Vector Architecture)

EVA is an extended vector architecture computer. It consists of two major functional subsystems, the CPH and the VPH. The CPH architecture evolved in the course of a ten year period with the current effort of a Phase I and Phase II SBIR. EVA is designed to support a cascadable system whereby users can insert multiple CPH boards into the system and extend the wordlength. The architecture has been in development over several device technology evolutions. It has seen change from the first 8-bit slice AMD 2900 chips through the current 64-bit slice BIT 2120 multipliers. That it has withheld change over these years attests to its conceptual strength. These developmental efforts are described next and will be important to the reader when the current architectural issues are discussed.

1.1.1 Phase I Research Effort

Details of the Phase I effort are found in the Phase I Final Technical Report. The technical objectives are cited next to outline the steps that were taken during Phase I.

1. Study and organize the EVA architecture into efficiently coupled modules for radar and signal processing. In this step, data transfer techniques were investigated to increase I/O transfers at the chip and board levels. Optimal trade-offs were determined among engineering parameters of power, board size, and speed of operation so as to render EVA machinery fast and efficient for laboratory and range instrumentation applications.

2. Determine the optimal trade-offs between fixed-point and floating-point number systems. Also, analyze the rounding and truncation issues and/or the overflow and underflow issues with respect to fixed-point and floating-point operations in the EVA. The objective was to identify efficient wordlengths for signal processors in EVA-like architectures.

3. Study optimal ALU configurations that speed up signal processing in EVA architectures. The objective here was to determine the ideal configuration (16x16, 32x32, or larger multipliers) which supports the processing bandwidths required.

4. Research the usage of fast controller circuits that may utilize centralized or distributed PLAs. The objective of this step was to improve arithmetic processing speeds while reducing or at least maintaining low control wire count from the control unit to the control points in the architecture.

5. Research microprograms for fixed-point and floating-point signal processing algorithms executable on EVA architectures. The objective was to developed sets of signal processing micro-routines that could be ported across architectural changes.

The following sections describe the efforts undertaken at Space Tech Corporation (STC) to satisfy the objectives set forth above. The basic architecture for the EVA organization as determined from Phase I is shown in Figure 1. The basic architecture derived for the VPH in Phase I follows in Figure 2. During Phase II the VPH architecture was modified to include a better VME interface controller chip, the MVME 6000, and PALs were used instead of the Motorola BAMs for speed reasons. The remaining VPH retained much of its Phase I characterization during Phase II. In fact, the VPH final design exceeded its Phase I speed estimates for the 1k FFT. The 730 usec benchmark was reduced to 604 usec in the final Phase II architecture.

The EVA is an architecture concept whereby high-speed yet versatile and efficient computations are a must. In order to reach an acceptable compromise between these conflicting needs, the process of selecting the building blocks for each component of the EVA architecture considered several issues. Minimum/maximum cascadable increments (8, 16, or 32 bits CPH only), execution speed, versatility, availability, amount of "glue logic" needed, overall chip count, and maximum utilization of available resources are just a representative sample of the issues considered.

Figure 1 depicts the block diagram of the 32-bit EVA architecture containing the Vector Processing Hardware (VPH) and the Cascadable Processing Hardware (CPH) modules. It has been determined that all of the modules will connect to the VMEbus. The VMEbus data transfers between modules can handle up to 32 bits in one transfer, however the CPH allows up to 64-bit on-board data manipulations when two CPH modules are incorporated. One CPH module will support up to 32-bit wordlengths. This cascadability allows users to maximize the use of available resources.

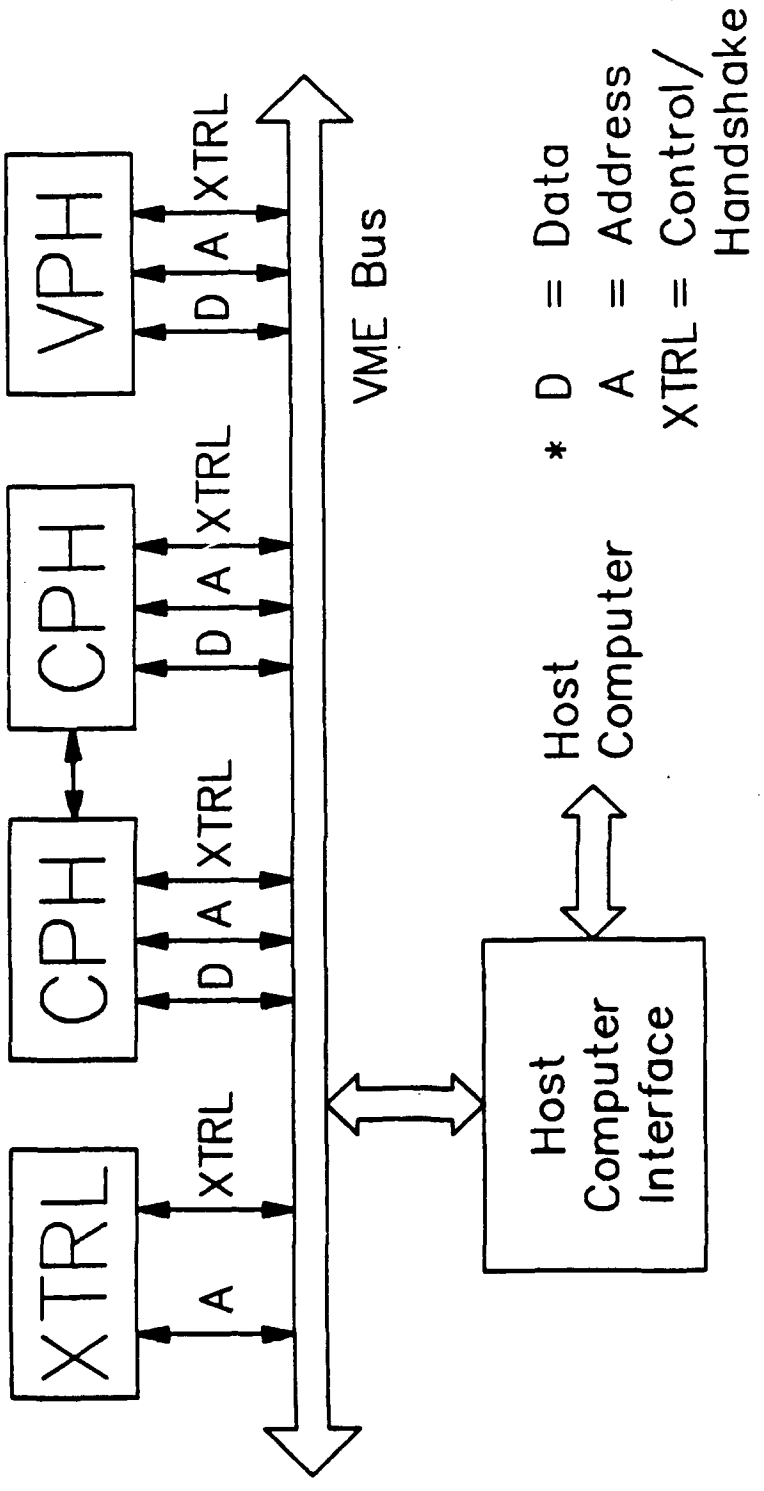


Figure 1. 32-Bit EVA Architecture

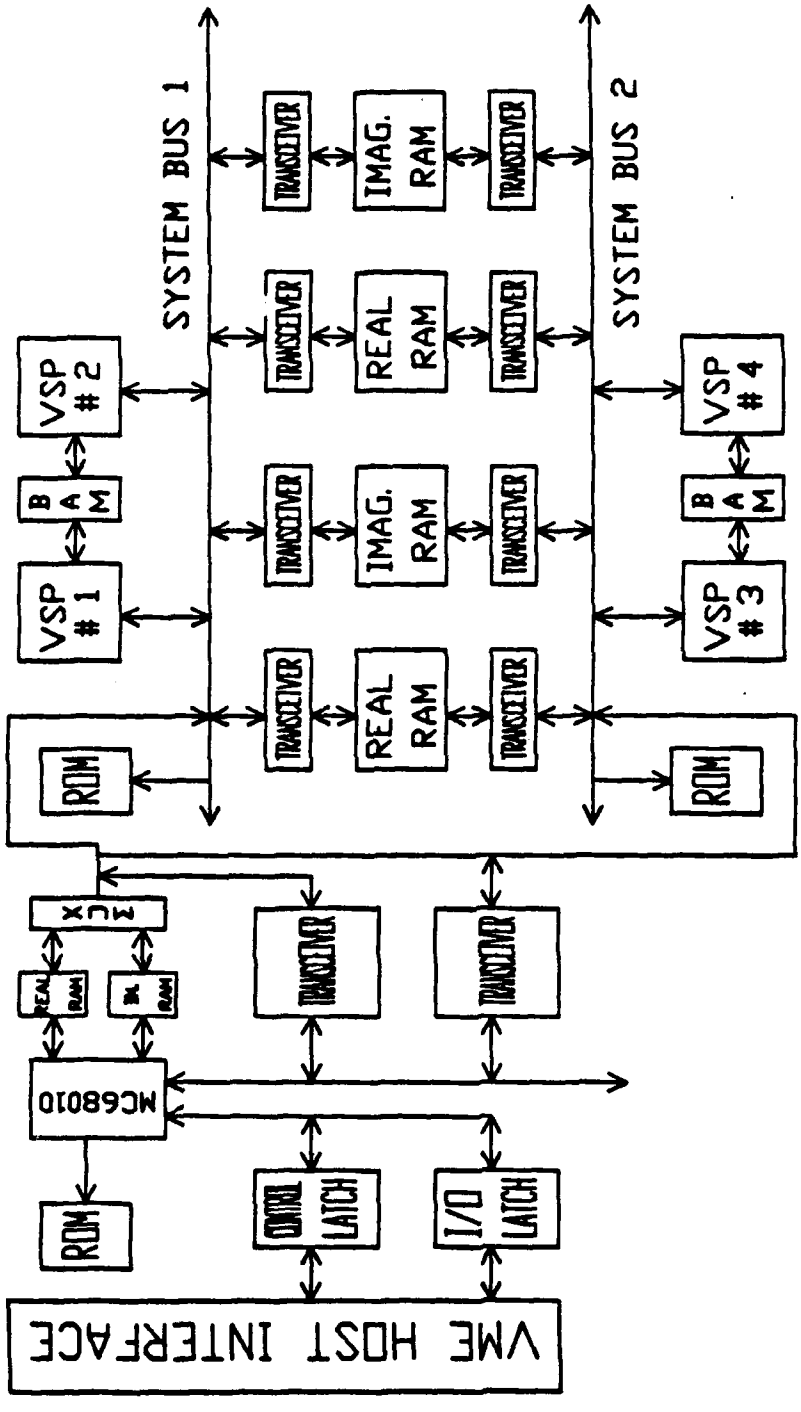


Figure 2. Vector Processing Hardware

The VPH is ideally suited for high-speed signal-processing applications where efficient, complex-data number-crunching is of the utmost importance. The heart of the VPH (the ZR34325 also referred to as the VSP-325 and shown in Figure 3) is capable of executing high-level, vector oriented instructions which embed the DSP algorithms directly into the device, allowing efficient algorithm execution. Moreover, a VSP-325 based architecture facilitates algorithm partitioning in the sense that multiple VSP-325s can be paralleled in order to share in the data processing requirements. Hence, while the VSP-325s perform parallel processing with interleaved I/O on the data from one RAM section, the host or the CPH can be up-loading or down-loading data into the other memory bank of the VPH. Once the current activities are completed, the roles of the VPH memory banks are reversed. This function-swapping is the primary reason for the efficiency and high throughputs attainable with the VPH.

In order to fully capitalize on the processing power of an EVA architecture, the system bus configuration must be equally capable of interfacing with the host, and within modules of the architecture. A study was made to identify the most optimal bus arrangement which allows maximum exploitation of the capabilities of the EVA architecture. The study did not consider 16-bit bus configurations such as the STD bus, MULTIBUS I, UNIBUS, and Qbus. The reason is that these systems do not satisfy current DSP and/or military real-time demands, nor are they capable of supporting the dynamic range required in such applications.

The Phase I effort concluded with an EVA architecture to support both DSP via the VPH and cascadability via the CPH. The Phase II effort began a year later. The gap in time offered STC and WSMR the opportunity to incorporate new technology advances. Phase II began with a review of those advances.

1.1.2 Phase II Developmental Effort

Through engineering analysis, STC proposed in Phase II to review, update, and modify the preliminary EVA designs developed during Phase I of this effort. The objective was to ensure integration of the latest technology and design techniques in order to guarantee longevity and usability of EVA over a wide range of applications. Of paramount importance was the determination of the optimal number of board and interboard cabling and control requirements for efficient operation of the cascadable architecture.

The EVA remains an architectural concept whereby high-speed, versatility, and efficient computation are balanced. The scope of this Phase II project was to develop a system that incorporates cascadability and high-speed data- and signal-processing. The building blocks, designed in Phase I, for each component of the EVA were expanded into efficient, working modules. A signal processing software library, containing algorithms that enhance the usability of the EVA architecture, was studied but not fully developed. Targeted applications for the EVA included range instrumentation, radar signal processing, digital focusing, spectral data processing, Kalman filtering, and real-time target motion resolution.

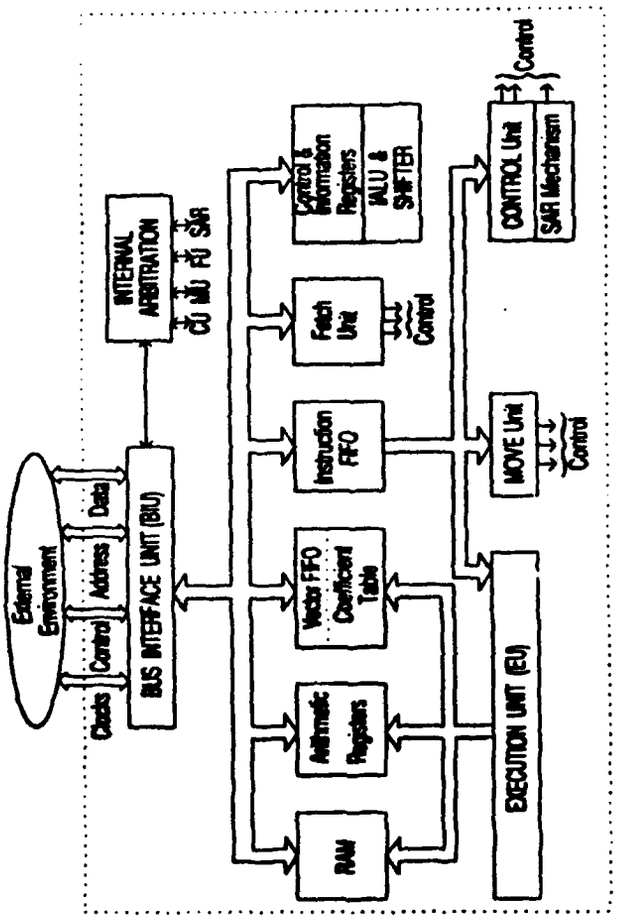


Figure VSP-325 Architecture.

Figure 3. ZR34325

In order to fully capitalize on the processing power of an EVA architecture, the system bus configuration must be equally capable of interfacing with the host, and within modules of the architecture. Phase I preliminary studies and Phase II review showed that the VME system provides the speed, versatility, and generality required in an EVA-like architecture. STC incorporated a bus configuration within the EVA to allow maximum exploitation of the architectural capabilities. Moreover, its asynchronous, non-multiplexed protocol insured longevity of the system. This is accomplished by providing the flexibility to incorporate faster devices into the system design, without having to redesign or upgrade the interface block. This allows the system performance to be upgraded as superior technology is developed. In addition, various processors and peripherals can operate at various speeds without having to wait for proper timing to get on/off the bus.

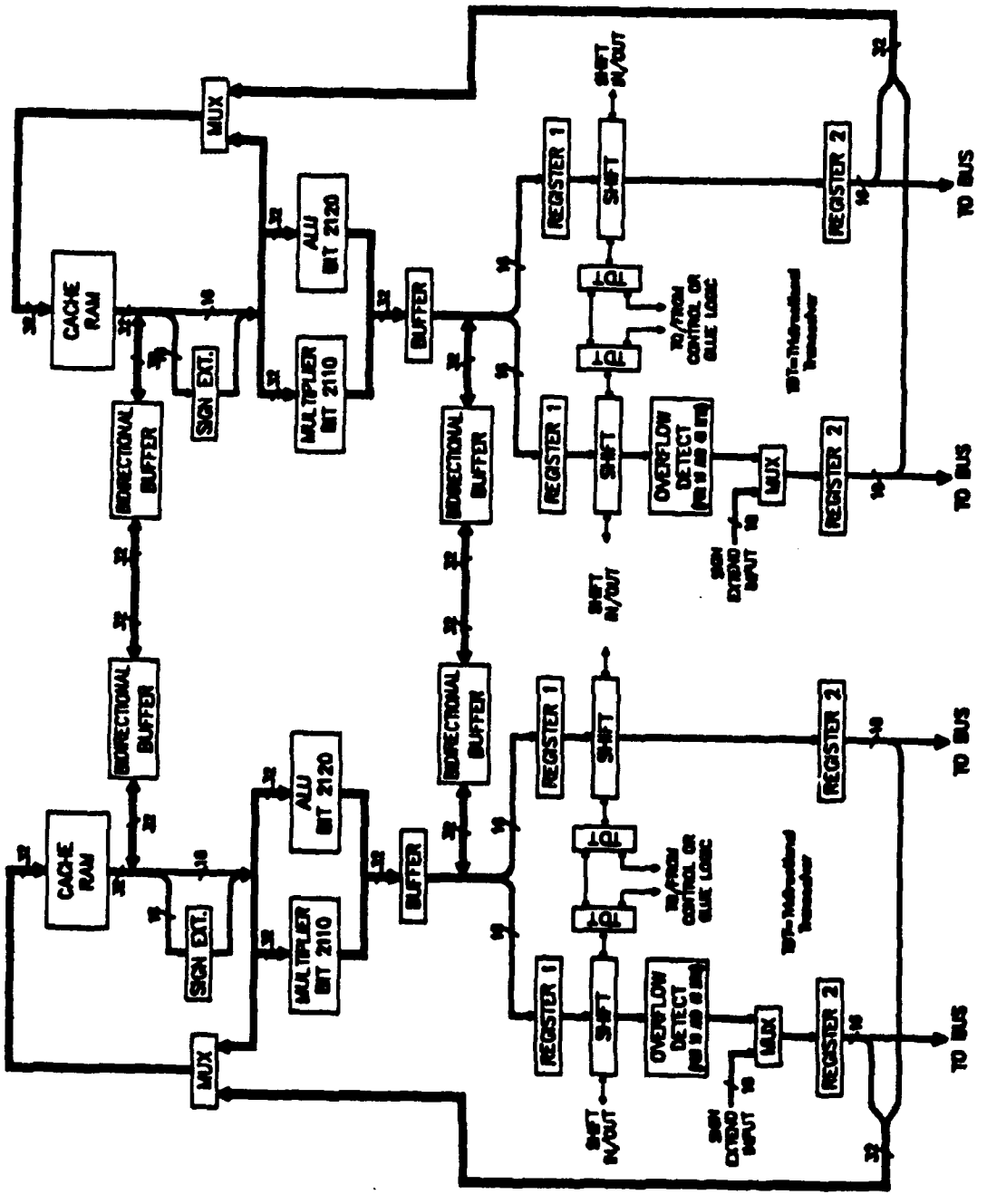
Initially, the Phase II proposal identified the following cascadable processing hardware as depicted in Figure 4. The VPH and EVA architectures were depicted in previous figures. During the course of Phase II, the cascadable processing hardware (CPH) underwent major changes described in Section 1.2. Those changes came as a result of significant component developments described next.

1.1.2.1 Significant EVA Component Considerations

From extensive discussions with the WSMR-ID-TA staff, it was determined that the BIT2110 and BIT2120 devices would serve as the main processing engines in the CPH. Each is ideally suited as a 32- and 64-bit device. Also, such devices provide pathways to future ALUs with minor changes to the microcode and boards. The VPH numerical engine selected was the Zoran 325 DSP device which became available during Phase II. The 325 chips performed as needed. In many cases they exceeded the speeds of other choices such as the Motorola 56000 and 96000. The AT&T DSP 32C and TI32020 devices were too slow for the WSMR applications and were discarded early in the design selection process of Phase II.

During Phase II GaAs technologies became mature such as the Gazelle serial transceivers. These GaAs chips provide data transfer rates in the gigaflop range and serve as the high speed link between the VPH and the CPH. This prompted further investigations into ultra high speed buses. The high speed IO or HSIO bus was designed on this basis. This bus, described in a later section under the CPH/VPH link section, was used to make 32- and 64-bit data transfers among the modules in the CPH. Those modules include the processor, cache memory, address generator, and IOP.

In 1991, the VPH design was impacted favorably by the introduction of economical 4-port memories. The 4-port memory circuit shown in Figure 5 made the VPH board requirements smaller. The device was incorporated into the design for the program space of the VPH so that the DSPs could share the data space with the 68020 and the ISA interface. This made a truly versatile architecture for multiple processing tasks.



Two-Board Architecture for the 32-Bit Module.

Figure 4. Phase II Proposal CPE

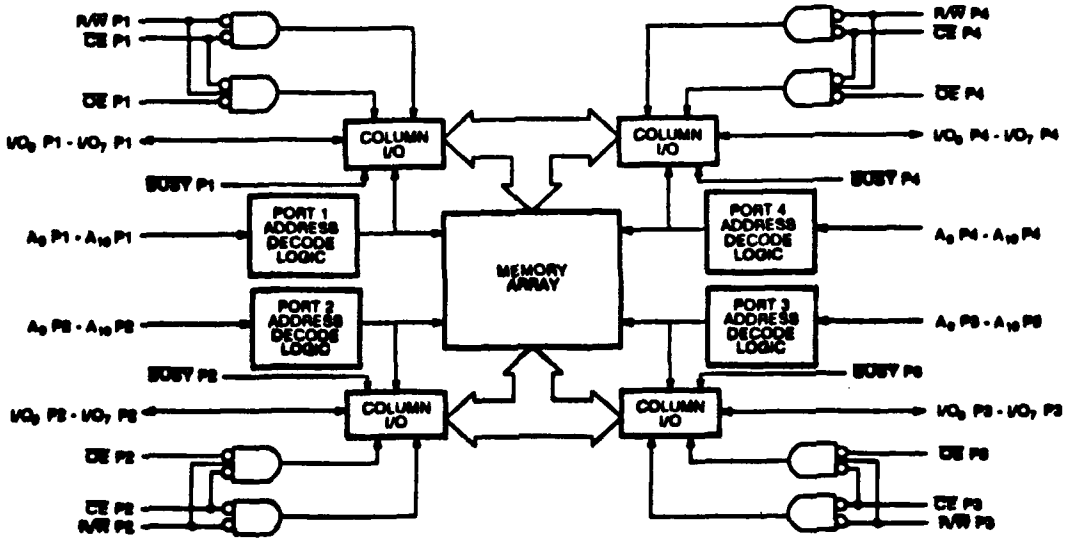


Figure 5. 4-Port Local Memory Architecture

Lastly, the EVA architecture became significantly fast when a custom crossbar was designed by Steve Hall. This crossbar depicted in Figure 6 was to make a significant impact on the large scale integration of the processor and address generator boards. The original organization was a 12x12 configuration as shown. Later modifications required an 12x14 organization. However, internally, the functional areas remain as in this figure.

1.1.2.2 Development of I/O Configuration

Before an indepth design of the CPH could have begun, the host interface design needed to be investigated. Hence, a major design issue was to determine how the CPH is to be viewed from the standpoint of the host or system controller. Three basic schemes described next were investigated early in Phase II. The CPH Bus-Based system was finally chosen.

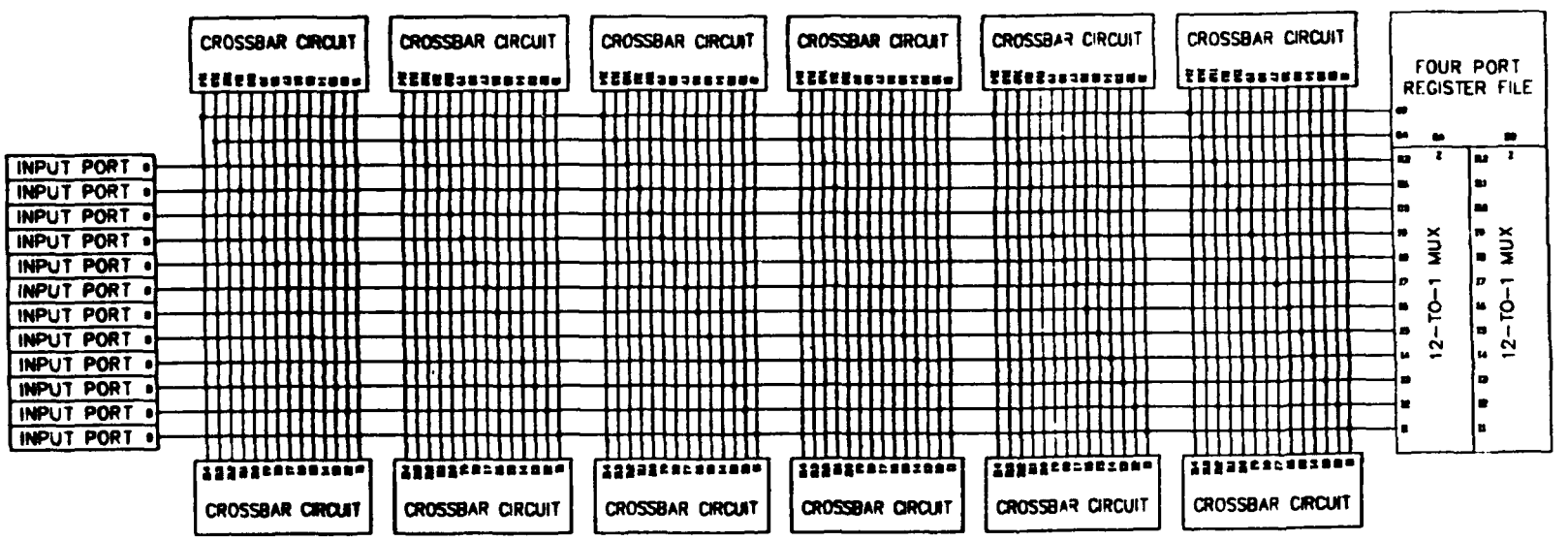


Figure 6. 12x12 Crossbar with Register File

Primitive Processing Unit

This is the simplest possible view of the CPH. In this scheme, the CPH functions as a processor with virtually no control intelligence. The host provides the data to be processed, the microprogram code to be executed, and explicit control instructions on where in CPH memory to place the data and microcode and where to begin execution. Output from the CPH to the host would be handled in a similar fashion. In this scheme, the host/CPH interface would involve some rudimentary handshaking logic to initiate transfers, and logic to allow the host to access CPH memory.

Intelligent IOP

This is the next more sophisticated view of the CPH. In this scheme, an I/O processor would be incorporated into the CPH which would have a fair level of control intelligence. The IOP would handle all transactions between the host and CPH. The IOP would have access to the CPH memory space, and would handle the task of informing the CPH where data is located, where to begin execution, and all handshaking between host and CPH. In this scheme, the host/CPH interface would require some processing ability of its own - probably a microprocessor such as a 68000. Some additional logic to support the microprocessor would be required.

CPH Bus-Based System

This is the most sophisticated view of the CPH. In this scheme, a high-speed bus would be developed for the CPH. A bus controller would link the CPH bus to the CPH backplane. An intelligent interface would link the CPH bus to the host. All transactions between CPH and host would be handled by both the host interface and the CPH bus controller. In this scheme, resource requirements would far exceed those of either method previously outlined.

Impacts, Comparisons, and Additional Considerations

If the primitive approach is taken, CPH throughput will be negatively affected, since a great deal of system overhead exists for the host to service the CPH. The tasks of processing and I/O cannot occur concurrently. If the IOP approach is taken, a marked increase in system throughput can be achieved. This is largely due to the fact that the IOP can handle I/O tasks while processing of other data is being done. The increase in throughput may indeed be significantly improved under this scheme, as it is likely that I/O time for a given task will be equivalent to the processing time required. Throughput may be increased by as much as a factor of two.

Implementation of a bus-based CPH could provide a similar increase in throughput, as well as increase overall system flexibility, since additional special-purpose modules could be designed to hang on the CPH system bus. In terms of impact on development costs, the IOP approach would add very little to development costs. A few more chips would be required than if the CPH is capable of only very rudimentary I/O, but the price of these additional chips is nothing when compared to the cost of system memory. Design time would be increased very little, as some type of I/O circuitry must be developed. While the implementation of an IOP is more sophisticated than the primitive approach, the task of design may actually be somewhat simplified because of

having a microprocessor to handle control and routing of data.

Development of a system bus for the CPH would be the most expensive in terms of both resources required and design time required. A number of additional considerations should be taken into account in determining which I/O approach to take. Among these is the idea of developing a macro or assembly language for the CPH. The CPH is a poor architecture for implementing looping or branching in programs. Also, processing of scalar operations is not one of the CPH's strong points. This means that under the primitive approach to I/O, separate and distinct microprograms must be written for every task it is to accomplish. Writing microprograms is a complicated, time-consuming task that requires an intimate knowledge of the architecture. In addition, implementing scalar operations in microcode results in inefficient use of processor time.

Designing an IOP for the CPH would allow development of a library of fundamental microcode routines which could be assembled into many useful, much larger routines. These assembled routines might not make the most efficient use of the processor, but in terms of time saved in not having to write long, complicated microprograms, this could be a very attractive feature to potential users. In addition, the microprocessor used in the IOP could be used to improve processing of scalar operations - something for which the microprocessor is more well-suited than the CPH. For the project at hand, development of the macro language does not have to be done, but if this capability is desired, it must be designed in now, or the system will have to be redesigned at a later time when the feature becomes desirable. This is a waste of both time and money.

Development of a system bus is important in a multi-CPH system, or in a turnkey or stand-alone CPH-based system. Currently, development of an IOP for the system seems a desirable and cost-effective approach to take. Microprogram storage RAM costs about $.40 per instruction, and data cache RAM costs about $.08 per word. External memory for storage of IOP data and programs would cost less than $.0025 per word. When viewed in this light, the IOP approach may be the least expensive approach to take, since RAM space for storing IOP programs is much less expensive than RAM space for microcode routines to handle I/O. The microprogram memory will not have to be as deep if an IOP is used, and the money saved on microprogram storage space will likely pay for the parts required to construct an IOP.

1.1.2.3 Development of EVA Control Store

In order to effectively use EVA with as many microprograms as possible, a writable control store organization was chosen. This organization allows the user to load in at runtime as many microprograms as is needed for a sequence of tasks. This type of control store then makes very efficient usage of the costly high speed RAM by loading and subsequently unloading precious space. Reusing the control store space requires different supporting hardware than an EPROM or fixed microcode memory.

A typical control store circuit is shown in Figure 7. With this design, one sees that interruption, micro-level subroutining, and context switching are supported as is necessary in writable control stores. An adder is included in order to compute address offsets so that relative addressing can

be supported at the microcode level. In writable control store architectures, relative addressing is necessary, otherwise users could not download microprograms without wasting writable control store space. To avoid the loss, every microprogram should fit in the next available location. However, that location would not be known a priori. So some hardware must be included in the controller to offset locations from the last microprogram loaded into the WCS.

Stack pointers can also be supported by the stack pointer registers in the upper left portion of Figure 7. This facility makes microprogram coding simpler and alleviates complicated address calculations by the user in advance. Stack pointers also facilitate subroutine calls and nesting. An address space exceeding 64k is desired because of the several simultaneously loaded microprograms which should be resident in the WCS. Thus, the counters and adder should handle 20-bits instead of 16-bits (16-bits spans only 64k).

Examining off-the-shelf components for a microsequencer 20-bit adder faster than 50 nsecs found no such devices. Even the counter must be built up from discrete devices in order to achieve 50 nsec speeds. An estimate of the chip count for discrete logic components for the complete sequencer indicates that at least 50 24-pin chips may be needed. The Phase II investigation proceeded to analyze faster and denser FPGA chips, among those included the chips from Plus Logic. It was found possible that one FPGA will replace 50 random logic devices. The board space savings became very attractive. But in addition, the ability to reprogram an FPGA without having to redesign the entire PCB became more attractive.

During 1990, software was received from Plus Logic to evaluate the FPGA devices STC anticipated for the microprogram sequencer and address generators. That code helped STC to lay out a chip from the standard cells available from Plus Logic. Using a FPGA is important because design changes can now be made to the device instead of the already manufactured PCB (which may be cost prohibitive). STC anticipated using the Plus Logic devices for a 20-bit adder and counter. The major issue in the speed was the need for carries and borrows across 20-bits.

Five 4-bit adders could have been used but carry lookahead circuits must be built. Xilinx, at first, appeared to be an adequate solution but later investigation showed that Xilinx cells were only suitable for random logic and not adders and counters. The basic Xilinx cell called a Configurable Logic Block (CLB) is depicted in Figure 8. Each cell is comprised of two FFs and a combinatorial logic section containing a program memory controlled multiplexer. Subsequently, the FPGA design for the two dimensional counters was completed with some custom library components provided by Plus Logic. Every I/O pin and functional block of the FPGA2020 was used.

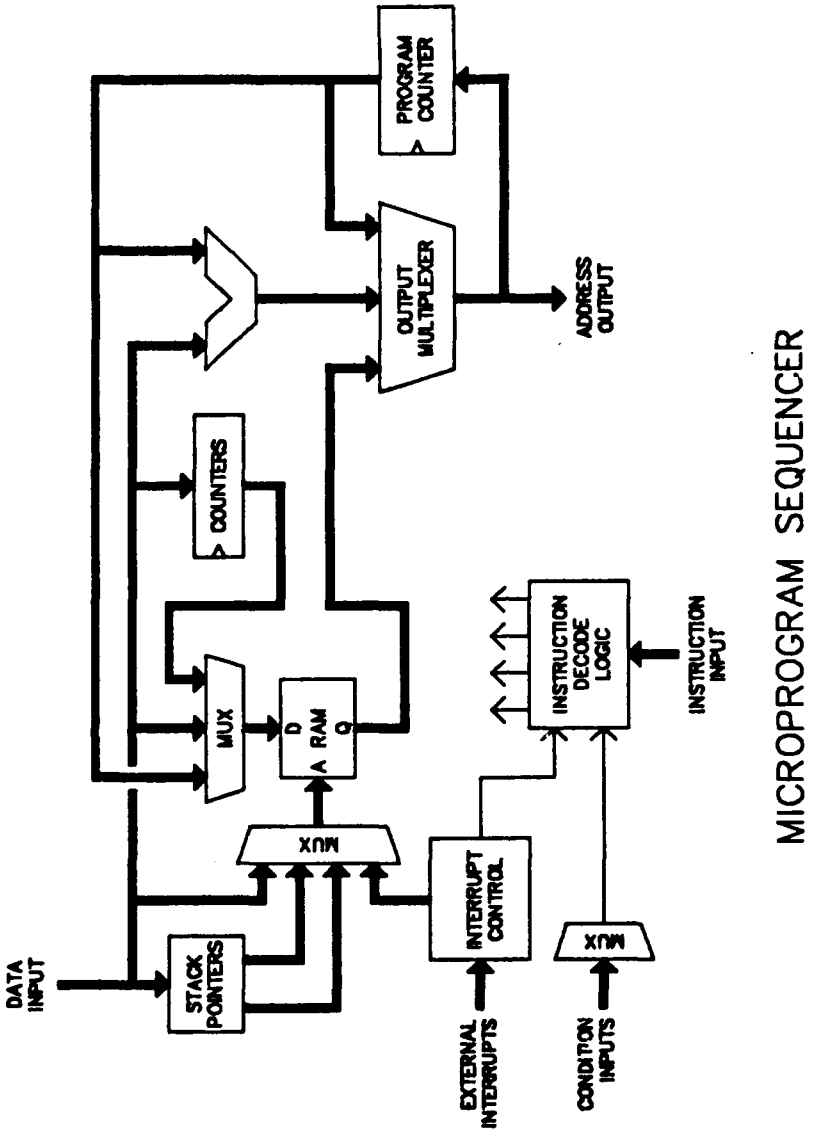


Figure 7. Typical Control Store Organization

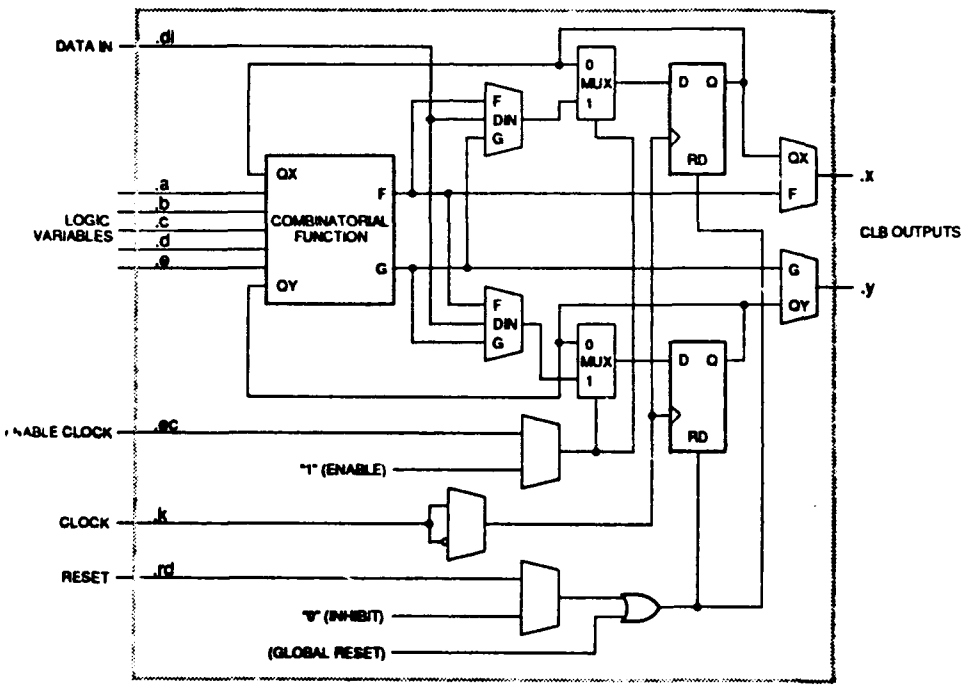


Figure 4. Each Configurable Logic Block includes a combinatorial logic section, two flip-flops and a program memory controlled multiplexer selection of function.

It has: five logic variable inputs .a, .b, .c, .d and .e.
a direct data in .di
an enable clock .ec
a clock (invertible) .k
an asynchronous reset .rd
two outputs .x and .y

Figure 8. XLINC CLB

It was desired that part of the EVA microprogram sequencer could be fit into an FPGA. Plus Logic began working on a custom component for another company which is an adder, mux, and incrementer all in one part. When this component was to be completed, Space Tech would evaluate it and determine if it could be used as part of the microprogram sequencer. It wasn't completed.

Several of the CPH's circuits required large numbers of small and medium scale integrated circuits. Some of these could be reduced down to a few chips with the use of Field Programmable Gate Arrays (FPGAs) from Plus Logic. FPGAs from other sources had been evaluated and found unsuitable for use in the CPH. High speed adders and counters are required. Plus Logic FPGAs can be used to implement counters of any number of bits which can be clocked at 40 MHz. Adders have a carry propagation time of 1 nsec per bit. This was significantly faster than any other FPGAs.

Plus Logic's FPGAs are constructed with an EPROM technology which allows them to be easily reprogrammed. This is another advantage of using FPGAs in the CPH. The ability to modify a section of circuitry on an FPGA as opposed to modifying a printed circuit board is an important feature. A mistake or modification to a printed circuit board could require a new board. This would mean an NRE charge of several thousand dollars. With extensive use of FPGAs and PALs it is possible to change a circuit without actually rewiring the circuit board. The larger the FPGAs, the better the chance of being able to make a change.

FPGAs also result in a significant parts reduction. For example, the section of the address generator board containing four two dimensional counters and an incrementer file would require 125 chips. With the use of Plus Logic FPGA2040 arrays the parts count could be reduced to 16. However, these chips are not yet available. The use of the proposed smaller (and available) FPGA2020 arrays would result in a part count of 36. The savings of board manufacturing costs and engineering costs alone offset the cost of the Plus Logic development system. The basic 2020 device is depicted in Figure 9.

1.1.2.4 Development of PC Interface Board

To coordinate design, development, and testing, a special PC interface board was designed first. An initial candidate for the PC interface board was designed based on the following assumptions. First, WSMR will use a Zenith 286 to interface to the CPH. Second, the same board will be used to test the CPH boards during code development at STC where a 286 PC will be used. Third, the interface control from the perspective of both machines (the PC as well as the CPH) is basically, "the PC (or CPH) sees a register from which to 'write to' or 'read from'". However, the PC is a 16-bit bus and the CPH is a 32-bit bus. Hence, the interface board must multiplex data accordingly depending on the direction of the data. Fourth, the board was designed to easily interface to typical bit-slice architectures such as the CPH. Fifth, the board shall be capable of driving high-speed data across long distances. Here, the IEEE RS-422 receivers are used. To invoke the simple handshake protocol earlier, FIFOs were used on the board. FIFO signals such as almost full and almost empty are to be monitored.

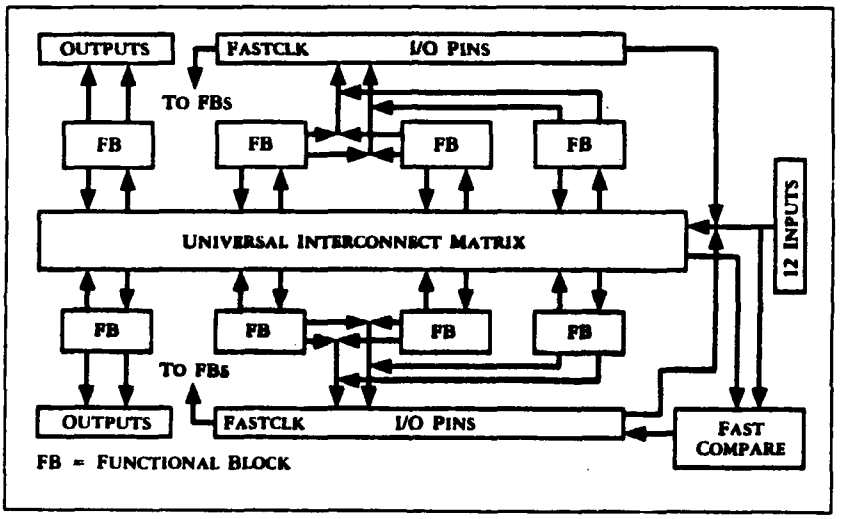


Figure 9. Basic 2020 FPGA Device

1.1.2.5 Study of PCB Manufacture Techniques

Central to the eventual Phase II objectives was a study of PCB techniques. A search and analysis of quality board manufacturers was done with the indepth feedback from Unicircuit in Englewood, Colorado. The factors with the greatest impact on cost and complexity of manufacture include the number of layers and the use of interstitial or blind vias. (A blind via is a hole which is buried inside the layers or only comes out one side of the board. Using such a via makes bed-of-nails testing almost impossible because the fixture cannot touch this via directly.) The physical dimensions of the eventual board have some effect when the board exceeds 8" x 10". Trace widths less than 8 mils and via sizes smaller than 15 mils would also significantly increase cost. When the boards are to be layed out, special vias will be reduced and replaced with another layer since this approach is less costly. Traces and spaces of 10 mils can be used effectively. Manufacturers suggested that this line width offers the best price per real estate.

1990 tooling charges were approximately $100 per layer. Fabrication costs for an 8-layer board with low complexity were approximately $200 for a board of approximately 8" x 15". Costs for creation of the bed-of-nails test fixture for checking board integrity are about $500 on the basis of a pin count of 3000.

Subsequently, PCB fabrication, assembly, and test were approximately $1700 per board, assuming 10-layer boards with pin counts up to 2000 per board. EVA architecture originally anticipated 4 boards, a CPH, an IOP, a cache memory, and the VPH. At a minimum, $1200 was to be expected for the PCB effort of a single board. It did not include parts or functional circuit testing at STC. Final costs rose to $2200 per board.

1.2 Results of the EVA Phase II Project

As mentioned earlier, the Phase II development effort underwent significant changes to the Cascadable Processor Hardware (CPH). Figure 10 depicts the current CPH. It differs from the previous architecture in that two ALUs and two multipliers are embedded on each board instead of one each per board. From design efforts early in Phase II, it was determined that doubling the processing power on a CPH board could reduce the data traffic bottlenecks for the HSIO and facilitate 64-bit processing on one board instead of two. In order to accomplish this integration, a new chip was designed called the Crossbar. This chip was fabricated by ILSI in Colorado Springs for the EVA architecture and is described in a later section. Such a chip was necessary to reduce the several multiplexers into one single device for the CPH. The datapath from ALUs to general purpose registers in Figure 11 was one example of significant crossbar usage. Later, it was discovered that the same chip could be used in the address generator board.

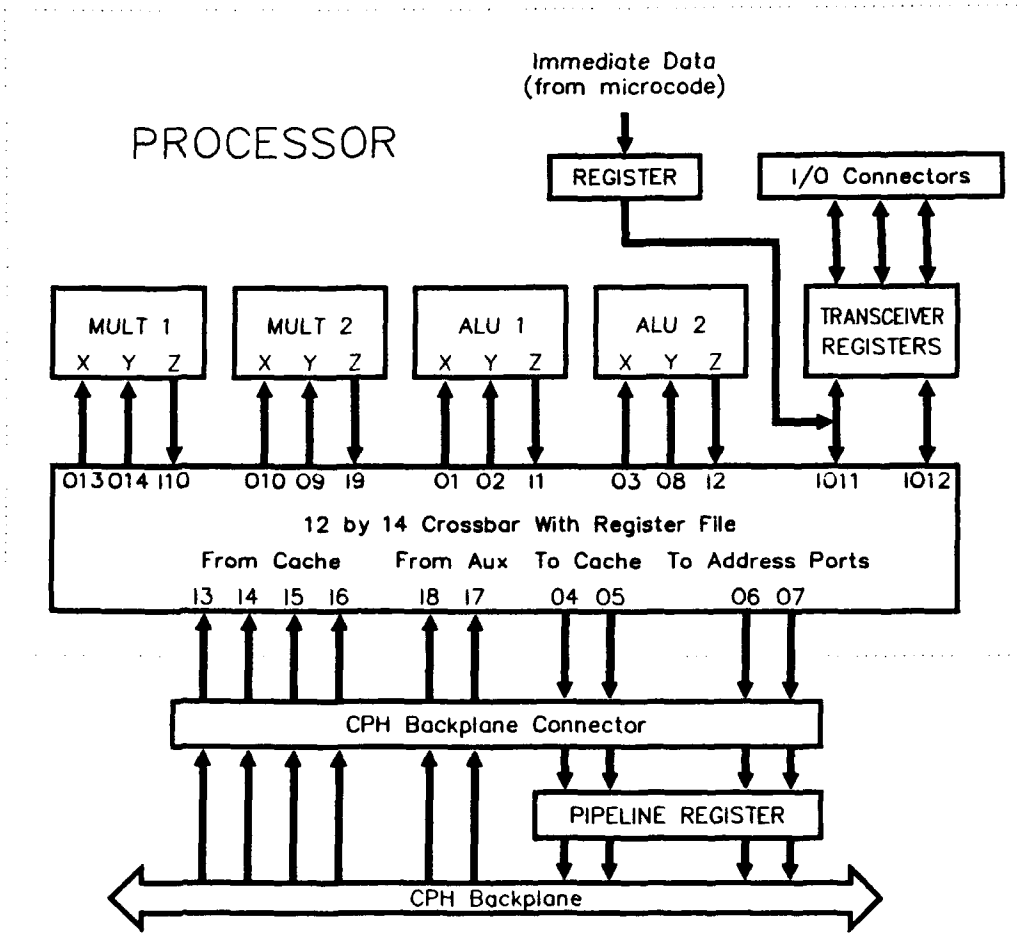


Figure 10. Phase II CPH Architecture

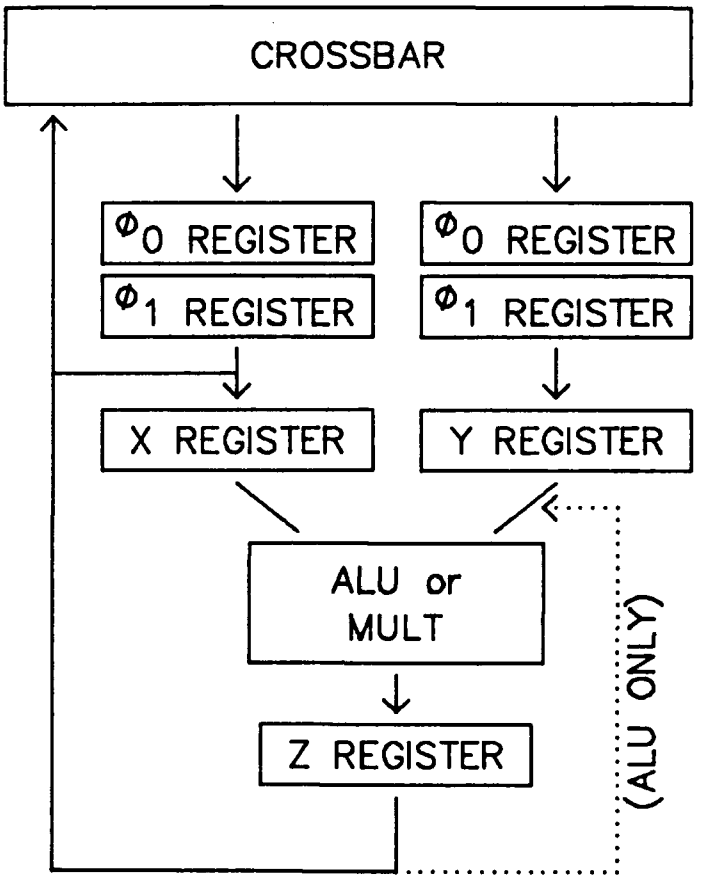


Figure 11. ALU to GPR Datapaths

The EVA organization began to solidify by the second year into several boards. A site visit by Mr. Lam and John Williams from WSMR-ID reviewed the new EVA architecture. Later, discussion with ID found a direct application with another WSMR SBIR contractor, Mentor. The Mentor application included radar tracker processing. The majority of that processing task centered around the Kalman filter. This directed the STC design team's attention to fast address generation for the complex matrix operations. An address generator was sought that would produce complex addresses in hardware at real-time speeds so that no computational overhead would result. And a study of matrix algorithms was initiated to ensure that EVA throughput was high. That algorithm study is discussed in Section 4.

Each board performed a separate and distinct function so that a cascadable design became feasible. As an introduction, those boards are briefly discussed in Section 2.0. The boards as organized developed into a very powerful computing engine and exceeded the performance specifications of the Phase II proposal by two orders of magnitude in some cases. A single EVA machine could perform over 30 operations per clock. Hence, if a 20 MHz clock were used, EVA would be a 600 mflop machine in a single desktop machine. The innovation became so attractive to Space Tech that the current EVA architecture was proposed.

Later results during the second year proved to be demanding to the design team at Space Tech. Advanced devices that were designed into the architecture had to be removed because the devices did not become available, were removed from production, or were functionally changed. The Plus Logic FPGA 2040 which was to be an integral part of the address generator never became available. The 2020 was substituted. The AMD 29540 FFT address generator chip was deleted from inventory. Finally, the BIT devices that were delivered lacked some of the vital control and status signals promised in the advanced specifications. As these were sole source suppliers, the EVA architecture design had to undo some of the effort and restart with less powerful chips like the FPGA 2020.

The VPH effort proceeded more smoothly since all parts remained available throughout the project. One major new chip discovery in December of 1989 which reduced board space needs was a four port RAM from IDT with a 7052S35G part number. This single device reduced space by 20% which allowed more functionality to be embedded on the VPH. Prior to that only the Micro Technology MT42C8128 was available and was seriously being considered. It was an expensive part.

During May of 1990, with considerable discussion with the technical monitor, the value of making the architecture more general purpose became more apparent. To that end, several changes were made to the schematic of the VPH.

The input bus to the board from the VME was originally designed to be only a 32-bit interface. Modifications have been made which allow the interface to be configured either as a 16- or 32-bit bus through the use of a simple jumper scheme. Due to the type of processing the VPH is designed to perform, namely DSP, and the computational speed it is capable of maintaining, I/O bandwidth becomes a serious concern. In fact, the VME bus would be sorely strained to keep the VPH busy. Because of this fact, it was originally proposed to make the 68020 processor bus available off the board. This was

proposed to allow for the development of external A/Ds and D/As which would interface to the 68020. From subsequent discussions with Mr. Lam, it became apparent that it would be beneficial if "off the shelf" D/As and A/Ds could be interfaced directly to the VPH. To that end and because the 68020 bus is so similar to the VME bus, it was decided in May of 1990 to provide a rudimentary VME bus, devoid of the layers of protocol, but able to support simple I/O boards. The final design of 1992 provides full VME bus, however, due to the desire to interface to single board computers (SBC) acting as masters.

The form factor of the VPH board was selected to be a VME 9U and has the capability of holding 4k words of data RAM. Because 4k words is not enough memory for some large data set problems, it was decided to allow for memory expansion. Expansion is accomplished by the addition of daughter cards which sandwich to the base board. Each daughter card contains an additional 4k words and all of the required bus buffering and decoding. Up to three additional boards may be added to the base board, bringing the data ram up to 16k words. Provisions have been made to support the new 4kx8 chips when they become available. This would double the data space.

The need for flexibility gave rise to a possible enhancement to the VPH. Because of the similarity of the VME and the IBM-AT and EISA bus architectures, investigations as to the possibility of mounting the VPH in an external box with power supply and minimal interfacing logic proceeded. This would allow the same board with no modifications, only additions, to be interfaced to a commonly available and inexpensive computational platform.

By June of 1990, a general VPH concurrent operating scheme for a status latch through which the five processors may share status information was agreed upon with WSMR. The need for such a status latch arose from the multi-processor nature of this system. Consider, as an example, the task of performing a two-dimensional FFT, with processing by all four Zorans. Roughly stated, the procedure is to first perform FFTs on the rows of the matrix, then perform FFTs on the resulting columns. The four Zorans share the work of performing these FFTs. Because of the way the problem will be partitioned, the Zorans will not complete the initial task of computing row FFTs at the same instant. Some delay must then exist for some of the processors before the column FFTs may be computed. The status latch concept will allow the Zorans to keep track of the status of their companion processors without the intervention of the 68020, keeping it free to perform other tasks. Later it was agreed that assigning each processor two status bits should allow for ample versatility.

Examination of a preliminary design for the status latch shared among the processors revealed that the design was deficient in several respects. The latch would allow any processor to write status bits to the latch, but in the case of the Zorans, whenever one Zoran wrote its status the status of its bus companion would be lost from the latch. To prevent loss of status bits from the latch, a duplicate image of the status bits for both Zorans on a bus would have to be maintained in the PRAM for that bus. A Zoran expecting to write its status would first read the status image in the PRAM, would write back to PRAM an updated status nibble reflecting the new status, and would finally write the updated nibble to the status latch. This sequence requires a read and a write to PRAM and a write to the status latch. The time involved is not a major concern, since writing out status info represents only a very

small fraction of the tasks performed. However, this sequence of operations contains a hazard which could result in problems. Between the time a Zoran reads and writes to the PRAM and then writes to the status latch, it is feasible that it might lose mastership of the bus. In the event that the new bus master is the companion Zoran updating its status, the original Zoran, upon regaining mastership of the bus, will write a status nibble to the latch which is erroneous. While the chances of this sequence of events occurring is rather slim, such an occurrence could prove fatal to a process, since an incorrect reflection of processor status could effectively "lock-up" a bus. It was determined that this design for the status latch would be scrapped in favor of a different design which will avoid the previously-discussed hazard, require only a single write to update status, and additionally, use less-expensive components in its implementation.

2.0 Brief Description of VPH and CPH Architectures

Much of the developmental history of EVA has been given in Section 1 so that one could have an appreciation for the design approach. In this section the reader will see the influence of the developmental history on the interfaces among the EVA functional units and the host. As stated earlier, EVA is composed of two main functional units, the VPH and the CPH architectures. EVA can be organized to expand in two dimensions, one through adding additional VPH boards and the other through adding additional CPH subsystems. Adding additional VPH boards is straightforward. All that is necessary is a simple insertion in the VME backplane. However, the CPH expansion uses different microprograms that share the common data buses. It is even possible for the CPH to share the same cache memory. In this manner a user saves two additional boards, a cache memory board and an address generator board. But, the additional cost savings should be compared with the larger and more complex microprograms needed for sharing a single cache memory space.

The design philosophy of EVA has been to provide a user friendly system that can be expanded easily. The advantage to this approach is obvious. The disadvantage is the increased system complexity of a very general organization. To understand the organization further, the following sections describe the interfaces to hosts and the internal control of the CPH. Both of these high level views will aid the reader in comprehending the EVA computer. The following paragraphs quickly outline the major functional capabilities on each of the boards. Section 2.1 concentrates on the multiple CPH interfaces. Section 2.2 focuses on the VPH interface and programming model. The VPH, as a separate unit, is intended for operation in any computing system with a VME backplane. Hence, it is important to grasp the VME interface capabilities of the VPH. More specific descriptions of the CPH and VPH follow in Section 3 and are useful for the microprogrammer.

PROCESSOR BOARD DESCRIPTION

The processor contains two multipliers, two ALUs, microprogram storage memory, a crossbar, a register file, and various I/O ports. Many configurations are possible by using different interconnections between processors and combinations of processors and memory banks. Descriptions of the processor's major components follow now.

ARITHMETIC COMPONENTS

The multipliers and ALUs support a wide range of number formats. These include 32 and 64 bit fixed-point, single and double precision IEEE floating-point, and DEC F and G formats. Each multiplier has a throughput of 20 megaflops for all number formats. The ALUs each have a throughput of 40 megaflops for all number formats, however, the bandwidth of the buses may limit double precision throughput to 20 megaflops. Total throughput of 120 megaflops could be possible with a single processor board.

MICROPROGRAM STORAGE RAM

The processor operates on a 50 nsec instruction cycle. Each microinstruction is 192 bits wide by two phases long. Each phase is like a separate instruction 25 nsec long, although they are always selected in pairs, giving a 50 nsec instruction cycle. The memory is 16,384 deep. That's 16,384 instructions by 2 phases by 192 bits. This memory can be written to through the I/O ports, 64 bits at a time.

RECONFIGURABLE REGISTER FILE

The register file has 64 double precision registers organized as an 8 by 8 array. Four independent ports allow high speed access to the registers. Two ports are write only and two are read only. Each port has its own address and a bandwidth of 40 MHz. Two reads and two writes can be done simultaneously. All accesses are synchronous, so a single location can be both read from and written to in the same instruction cycle.

The register file also has four different modes of operation. One is normal RAM access. The others link register locations into multiple pipelines. Configurations of 8 pipelines 8 deep, 4 pipelines 16 deep, and 2 pipelines 32 deep are possible. When configured as a pipeline, writing data to the first location of a pipe causes all data in that pipe to be shifted to the next register location. Data may be read out from any stage of the pipe.

CROSSBAR NETWORK

All arithmetic components, register file ports, and I/O ports are linked by an extensive crossbar network. Each arithmetic component has two input ports and one output port. These, along with external I/O ports and register file ports, have a dedicated port into the crossbar. This allows for all possible paths to occur simultaneously. All paths may be switched simultaneously at a rate of 40 MHz.

I/O PORTS

The processor board has 6 dedicated input ports, 4 dedicated output ports, and two bidirectional ports. Each port is 32 bits wide with a bandwidth of 40 MHz. These ports may be used to link the processor to memory banks or link multiple processors together or both.

ADDRESS GENERATOR BOARD DESCRIPTION

The address generator is a specialized processor with an architecture optimized to generate complex sequences of addresses for various vector and matrix operations. This will offload the arithmetic processor and allow higher throughputs. Microprograms for complex routines will be much shorter and easier to write. The address generator architecture has 4 two dimensional counters, 2 address look up table RAMs, microprogram storage memory, address output ports, a register file, and a crossbar. All data paths and components of the address generator are 16 bits wide.

TWO DIMENSIONAL COUNTER

Each two dimensional counter contains 2 preloadable up/down counters, two adders, two registers, and a multiplier. This hardware is designed to do array subscript expansion. After initializing, the counter can simultaneously index up or down the rows and columns of an array. This allows many complex routines to be programmed quickly and efficiently. Each of the four counters can be used to access a different array or vector in memory. Three of these counters contain an FFT address sequencer. This will allow various types of FFTs, including two dimensional FFTs, to be programmed efficiently.

ADDRESS LOOK UP TABLE RAMS

These RAMs can be used for indirect addressing or for storing sequences of addresses too complex to calculate in real-time. Each of these RAMs are 16 bits wide by either 32k or 64k deep. They can be accessed at a rate of 20 MHz.

MICROPROGRAM STORAGE MEMORY

The size of this memory is 16,384 instructions by 2 phases by 188 bits. It functions the same as the processor's memory.

MICROPROGRAM SEQUENCER

This sequencer generates addresses at a rate of 20 MHz to be used to access microprogram memory and provide program flow control. Both relative and direct addressing modes are possible. A stack of 4096 words is used for subroutine calls and a 16 bit counter is provided for loop counting.

ADDRESS OUTPUT PORTS

Three 18 bit ports are provided for outputting addresses. Each of these ports can run at a rate of 40 MHz. A 16 bit microprogram address output port is also provided. This feature allows the microword of the address generator to be combined with the processor and memory boards.

REGISTER FILE

The register file for the address generator is identical to the register file for the processor. Its primary use is for address pipelining and storing pointers.

CACHE MEMORY BOARDS

The cache memory is used to store reasonably large amounts of data for use by the processor. The memory is organized as two banks of triple ported static RAM, one bank for real data and the other for imaginary data. In each instruction cycle one complex word can be written and two complex words can be read from cache. All writes occur in the first clock phase and all reads in the second. This eliminates all possibility of conflict. A single location can be read from and written to in the same instruction cycle.

The cache memory hardware consists of memory blocks. Each block has two banks of triple ported RAM. Each bank is 32 bits wide and the depth is dependent upon which memory modules are used. Depths of 4k, 16k, and 64k are currently possible. Each cache memory board has space for two memory blocks.

Memory blocks, via software control, can be linked together into banks. Linking can be achieved both vertically, for greater depth, and horizontally, for wider word width. Two blocks can be linked horizontally for 64 bit word width. Any number of blocks can be linked vertically for a bank size up to 256k words. Up to 16 banks can be configured simultaneously, however, the processor can only access one bank at any instant in time. Banks can be toggled or paged through rapidly and any bank not being accessed by the processor can be accessed by I/O.

2.1 CPH Interface Architecture

The multiple interfaces among EVA are described in this section, beginning with the CPH. This is to allow the reader a view from the host computer's perspective and lay a foundation for the intimate hardware details of the CPH and VPH in Section 3. EVA is primarily interfaced to a host via the PC interface or ISA bus. Another interface was planned earlier for EVA with a DT Connect bus but this proved to be costly to the VPH board space and was subsequently not included in the design. However, the design effort is documented in the next section for completeness. In 1990, this bus appeared to become a defacto industry standard. By 1992, its popularity faded inhibiting further versatility to other CPH applications.

2.1.1 CPH/PC Interface

STC currently uses essentially the same ISA interface structure for both the CPH and VPH. Advantages of going this route, as opposed to using very different interface designs as was originally planned, include lower NRE for the ISA-end cards, since only a single board design needs to be manufactured. Also, the low-level ISA drivers are the same for the CPH and the VPH, so time in software development has been realized. Another advantage is the ability to interconnect the CPH and VPH through the common interface. This would allow for some development of a CPH/VPH coprocessing system. The limited bandwidth of this interface would obviously limit the usefulness of such an interconnection in any real application, but it would certainly be adequate for fundamental development.

The user view of the PC interface is depicted in Figure 12. In that figure, the reader can see that the interface is comprised of a set of FIFOs for READS and WRITES. Flags are available in a status register to monitor the FIFO contents. Those flags include "almost full" and "almost empty" so that very general device drivers can be used for the EVA computer. The interface can also be interrupt driven as well as program driven and interrupt flags can be found therein.

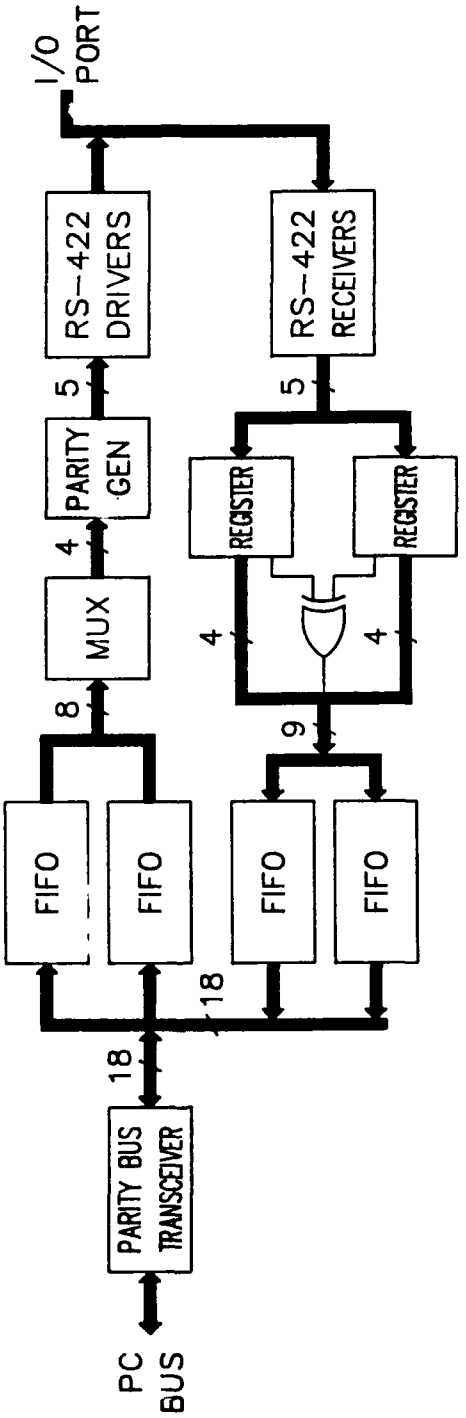


Figure 12. PC Interface Board Block Diagram

A parity bus transceiver connects to the PC bus so that even and odd parity can be checked. The selection is made via the status bits in the status register and the appropriate driver code. Both the PC and EVA ends must observe the chosen protocol. For physical distances greater than 3 feet, the RS-422 interface was chosen. Twisted pair shielded cable then insures noise free operation. Programming the interface board is described in Section 3.2.6. Also, to take advantage of the high nature of DSP applications, the VPH intended for DSP has a slightly different interface on its end. The differences are discussed in Section 3.2.6.1.

2.1.2 DT Connect Interface

One of the planned interfaces for the VPH was a DT Connect interface. Inclusion of this interface would allow systems to be implemented using Data Translation's data acquisition boards and possibly frame grabbers along with the VPH as the processing engine. Such a system might be desirable in light of the fact that Data Translations data acquisition products appear to be competitive in terms of bandwidth, etc., but their array processors aren't very fast. (The DT7020 array processor appears to be their quickest processor. This unit is rated at 8 Mflops peak, as compared to around 120 Mflops peak for the VPH.) The DT Connect interface is intended as a high-speed data path between acquisition devices and processors which are in close proximity to one another.

The DT Connect interface is very loosely defined. The definition consists of the pinouts on the connectors, the timing for the data and asynchronous handshake lines, and the electrical handshaking protocols implemented with the handshake lines. No limits on cable length are stated, but because the cabling is driven with conventional TTL drivers such as the 74ALS244 or 74AS244, and in light of the statement in the specification that the data can be clocked at something over 10 MHz, it is obvious that cable length will be limited to about 30 cm. This limitation could pose some serious restraints on putting together a system using Data Translation acquisition boards and a VPH.

The DT Connect interface is available only on Data Translation's products aimed at PC/AT-based systems. The need for a high-speed data path between acquisition devices and processors in a PC-based system is obvious due to the limited bandwidth of the ISA bus. Data Translation did the obvious thing to alleviate this problem in establishing the DT Connect pathway between their acquisition and processor boards. Because the VPH will not be on an AT form factor card, the usefulness of a DT Connect interface for the VPH is highly questionable.

As stated before, a practical limit on cable length is around 30 cm, and this is about the length of cable that would be needed just to get the cable out of the AT case. A cable long enough to exit the AT case and connect to a VPH placed close to the AT would be well in excess of the 30 cm limit.

A number of possible solutions or partial solutions to this obstacle present themselves. The simplest solution is possible due to the asynchronous nature of the DT Connect interface. The VPH end of the interface can easily govern the transfer rate. The transfer rate could therefore be limited to ensure reliable data transfer to occur across the cable. STC estimates that

the data clocking rate could be on the order of 2 MHz for an effective data rate of 4 MB/s. This data rate is lower than that of the ISA bus itself, which makes the solution seem very undesirable in light of the fact that our existing ISA interface can easily meet and probably beat the 4 MB/s rate of exchange. The only obvious advantage to using a rate-limited DT Connect interface is that the ISA bus would be free during transfers, which would in turn allow the AT to be performing some other processing task concurrent to the data transfer. If Data Translation software were being used on the AT, the transfers across the DT Connect interface could most likely be handled as standard DT Connect transfers by the software. On the other hand, it is questionable whether Data Translation software could handle control of the VPH. Depending on the ability of their software to link in user-generated routines for non-Data Translation system components, this solution might be totally unworkable if a user intends to use Data Translation software. If a user is willing to write all the software for driving the VPH and any Data Translation boards present in his system, this is a possible solution. As stated before, the penalty in speed reduction of the interface begs the question of the practicality of the solution, even in a situation where the user is willing to develop necessary software.

Another solution which seems somewhat more practical would be development of a combination ISA/DT Connect interface for the VPH. Such an interface would have a paddle card at the AT end which would plug into the ISA bus, would provide DT Connect ports into the interface, and provide a connector for cabling to the VPH. A number of advantages to such a scheme exist, including the likelihood that a design could be done which would impose much less significant limitations on maximum data transfer rates. It is also possible that such a scheme might allow the VPH to "look like" a Data Translation board so that no problems would occur when using software specific to Data Translation systems.

The disadvantages to this approach are primarily centered around the issue of development time. An ISA interface for the VPH is already in existence, and this design would need a good deal of modification in order to be made compatible with both ISA and DT Connect. An additional NRE and manufacturing charge would be incurred for production of the AT paddle card. In addition, if the approach of making the VPH look like a Data Translation board were taken, a great deal of research into protocols and architecture of Data Translation's processors would be necessary. It might prove very hard to get the necessary information. Also, a good deal of additional firmware development would be necessary if the VPH were to emulate a Data Translation processor. Considering these points, STC doesn't believe that this is a viable solution.

A partial solution would involve design of a fairly generic high-speed interface for the VPH. This interface could provide the ability to develop an AT paddle card to provide DT Connect translation at some future date. In terms of development costs, this seems like a much better approach. In addition, such a generic interface could provide the ability to develop translators for any number of other busses and/or interfaces to which we might want to connect at some future date. STC believes that the existing VPH-end ISA interface may be modified to provide such a generic interface. Modifications might include increasing the width of the I/O data paths and increasing the amount of control logic in order to make the adaptability of

the interface as robust as possible.

2.1.3 VPH/CPH Interface

A number of possible methods of implementing such an interface are possible, and the best solution is an "augmented" VME link between the VPH and CPH. This "augmented" VME link utilizes the standard 32-bit data path of the VME bus, and additionally uses 32 of the user-definable bits on the bus as additional data bits, making the effective width of the link 64 bits. This enables a maximum data transfer rate of 80 Mbytes/second between the VPH and CPH. This transfer rate stretches the limits of the VPH, and requires playing with how I/O occurs on the VPH board when VPH/CPH 64-bit transfers are occurring. This high data transfer rate makes the added circuitry worthwhile, since it greatly enhances the real-time capabilities of a CPH/VPH system, and effectively cuts the required number of required bus cycles for any given VPH/CPH transfer in half, thereby reducing loading of the host bus.

The CPH is also equipped with a VME interface through its VME buffer board, although its VME interface is somewhat more rudimentary than that of the VPH. This allows the VPH and CPH to be housed in a common enclosure. This common enclosure actually contains two separate backplanes - a VME backplane and a proprietary backplane for the CPH boards.

In previously proposed VPH architectures, the VPH/VME interface shared a port of the 4 port SRAM with the ISA interface. This arrangement allowed for VME communications to occur transparently as far as the 68020 was concerned, which would allow the 020 to do simple system traffic control concurrently with VME transfers. The likely kinds of traffic control that might be performed during VME communication would necessarily be limited to such things as status updating or polling of status of the Zoran processes. A limitation of this architecture is that the VME can only access the 4 port SRAM space. Data or program code that is being transferred into other memory areas would need to be transferred out of SRAM and into the actual destination by the 020. This puts additional demands on the 020 and also results in real transfer times being inflated due to the double transfers necessary.

In the current VPH architecture utilizing the MVME6000, the VME interface has access to the entire VPH address space. This will allow the VME interface to access data in any section of memory on the VPH board, including the memory on the Zorans, eliminating the need for 020 transfers from 4 port SRAM to actual destinations. The VME accesses the 4 port space through the 020's port via the 020 bus. This imposes the limitation that while VME transfers are occurring, the 020 is essentially locked out and can't perform any local processing tasks. This limitation is of only small consequence, especially when balanced against the elimination of double transfers that require 020 control.

Transfers between the VPH and CPH are performed by using the VME standard 32-bit data path and using 32 of the user-configurable bits to widen the effective data width to 64 bits. The additional 32 bits of data are written to/read from the ISA port of the 4 port SRAM. This allows for data transfer rates far in excess of the bandwidth of a single port into SRAM (about 50 MB/s) and effectively doubles the stated VME bus specification of 40 MB/s maximum.

Another advantage of the new VPH architecture using the MVME6000 is that the VPH, by virtue of the capabilities of the 6000 chip, may be used as a VME system controller. This is likely to have a large impact on marketability. The VPH is able to be a system controller, which will allow use of standard VME system components such as memory and data acquisition boards to function under VPH control, without the need for an expensive VME host system or VME controller. This could be very attractive to anyone who needs the capabilities of a VPH but doesn't have a VME host system. It could also be attractive to anyone who does have a VME host system, but would like their vector processor to be able to master the system.

In terms of the immediate goals of this project, the new architecture has a number of advantages. Primary among these advantages is the ability to configure a CPH/VPH system which does not require a VME host system. With the ISA interfaces resident on both the CPH and VPH, a very powerful processing station may be configured with a CPH, a VPH, a good ISA machine, and the previously described backplane and enclosure. A wide variety of off-the-shelf data acquisition and interfacing boards are available for VME, so interfacing such a CPH/VPH/ISA system to virtually any type of sensors or other data sources should be relatively straightforward. Unusual or highly specialized interfacing applications are handled by an appropriate VME-compatible interface board (the VME buffer board in Section 3.2.5).

2.2 VPH Architecture

The Vector Processor hardware or VPH consists of 4 Zoran 325 DSP devices and a 68020 floating-point processor configured to perform DSP operations in a wave fashion. The 68020 can operate independently of the DSPs. The VPH is a single board in a 9U VME quad high footprint. It can interface to a 9U or 6U VME platform. A MVE 6000 master slave controller device on the VPH assists data transfer across VME systems.

The VPH block diagram is shown in Figure 13. Here, one can see that the DSPs and the 68020 talk to a 4-port SRAM from data and program storage. A PC interface is also provided for code development and system monitor. The PC interface is a fast parallel port data transfer. For 6U VME transfer an additional VME buffer board is provided. The VPH is intended to be plug compatible with the SUN workstations to enhance intensive numerical computations via a set of provided math libraries.

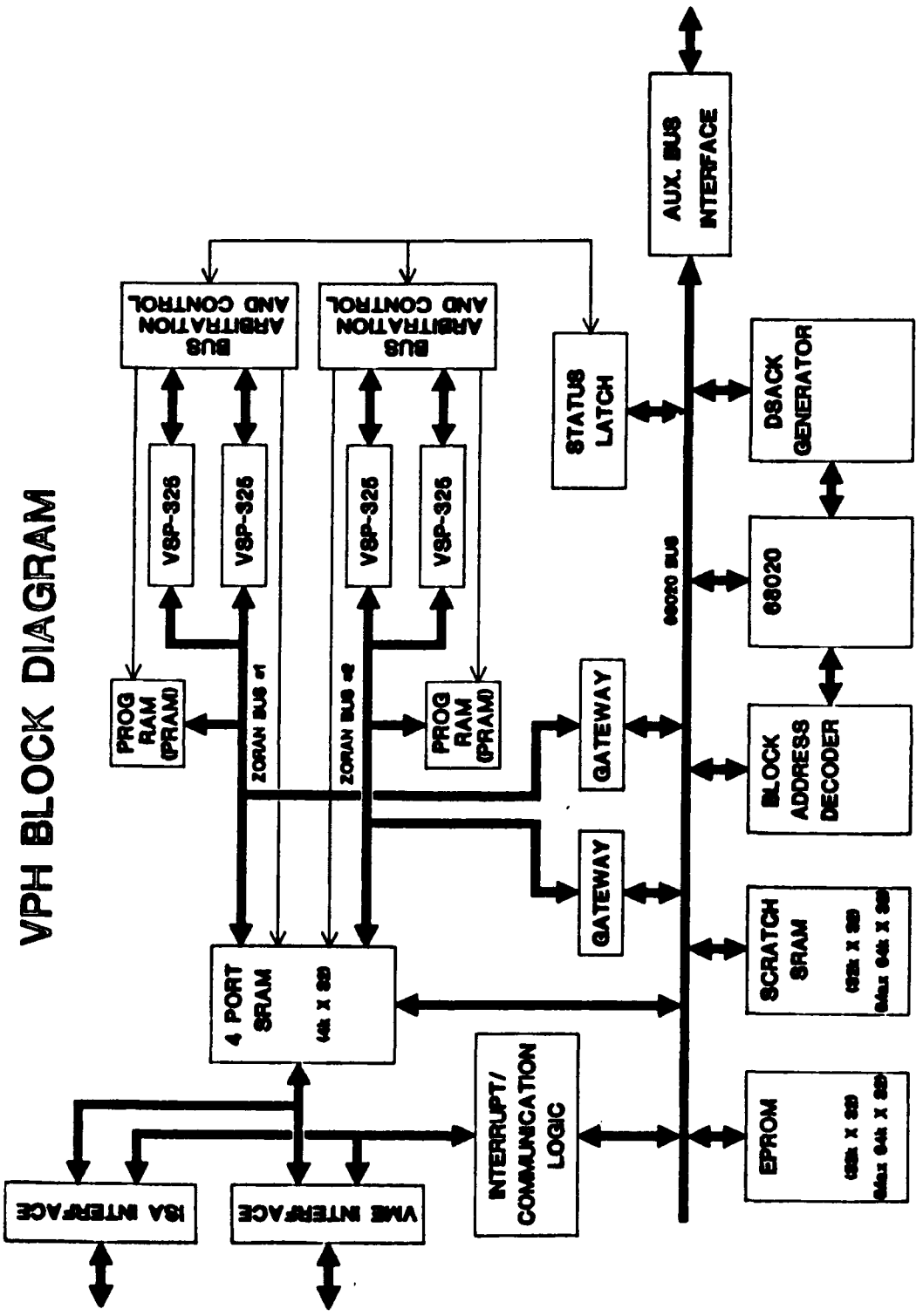


Figure 13. VPH Block Diagram

2.2.1 ISA Interface

An important task of the VPH project has been the study of the host-to-VPH interfaces. Two such interfaces are possible - the primary VME interface and a secondary ISA interface. The ISA interface will allow the VPH to be configured into a PC/AT system. This allows development of a variety of VPH software without the need for access to a VME machine. The ISA/VPH combination is not a very efficient way in which to utilize the VPH due to the limitations of the ISA bus, but should prove convenient for development purposes, and may even be useful for some applications. A number of extensions to the VME standard are also in existence. These extensions are designed to improve certain aspects of VME system performance, and to add flexibility to the VME bus. These extensions were examined to see if any of their features are suited to the VPH. The two extensions were examined: the VSB bus (VME Subsystem Bus) and the VXI bus (VMEbus eXtensions for Instrumentation).

The ISA interface is realized as a block of four 8-bit I/O ports on the ISA side of the interface. Two of these ports form a 16-bit data port into the VPH, while the other two ports form a 16-bit control/status register through which the VPH may relay status information to the ISA host. Also, the ISA host through this same port gives control/command information to the VPH. The basic command set includes Block Transfers to/from the VPH, Block Moves between memory domains within the VPH, RESET of the VPH subsystem, and commands to the 68020 to begin execution of internal code. The VPH will be capable of interrupting the ISA host to indicate task completion. The interrupt level used is user-selectable in order to configure the VPH into most ISA systems without creating conflicts with other boards. The VPH also posts task status in the status register area so that the host may poll this register to look for task completion, rather than being interrupted. This could be handy in some applications, but the main reason for this feature is to prevent interrupt conflicts in ISA systems that have other resources using all available user interrupts (This is a typical problem with ISA systems.).

The ISA host is capable of interrupting the 68020 to initiate transfers of data and/or commands, or it may poll the status registers to see if the VPH is in an "idle" state which will allow the host to effect various operations by setting specific bits in the command register.

Transfers of data to the VPH is accomplished with the help of a 16-bit presettable up/down counter in the interface which will allow transfer of data to contiguous locations in the 4-port SRAM with a single address being passed to define the starting point for the block transfer. This allows for the maximum possible data rates between the host and VPH.

The VPH interface is mapped into the ISA I/O space rather than PC memory space. The interface is essentially a contiguous block of four I/O locations. These locations will be user-selectable, since add-on cards use a wide variety of the available I/O addresses. Because of the fact that a block of only four locations will be required by the VPH, a user should have no problem successfully configuring the VPH into a system. This requirement of four contiguous locations is small when compared to most add-on cards - even something as simple as a serial port typically requires eight contiguous I/O locations.

The I/O mapped approach does not allow the ISA host to read or write a specific location in a single transaction cycle since the address bus won't be available to the interface. The address and data must be passed in two separate cycles. Rates of data transfer could be seriously impacted by this requirement. This problem is solved by giving the interface the ability to provide incremental addresses for accessing contiguous locations in the SRAM, eliminating the need for the host to provide an address for every word transferred. This has virtually eliminated the potential performance penalty of the I/O mapped approach, since the vast majority of transfers consist of blocks of data rather than individual words.

2.2.2 VME Interface to VPH

In VME interface design investigations, STC determined that one feature the VPH VME interface must have is the ability to perform VME block transfers. This will allow the highest data transfer rates possible. Designing this ability into the interface provided some challenges.

In order to perform block transfers, the interface must include an address counter for accessing the SRAM. This in itself is no real problem. A state machine must be designed which clocks (increments) the counter at the appropriate times within the block transfer. In addition, this state machine must generate the A01 address bit to the SRAM address decoder, since this bit is only valid on the first transfer cycle of a block transfer.

A block transfer begins with a normal byte, word, or longword transfer. The transfer becomes a block transfer if the DS0* and DS1* data strobes are released and then reasserted without a negation of the AS* address strobe in between. Once a block transfer has begun as described, the address strobe remains asserted until the block transfer is complete, with individual transfers being delineated by negation of both data strobes.

Once a block transfer has begun, the LWORD* and A01 and A02 - A31 bits from the VMEbus are invalid. They are valid only for the first cycle of a block transfer. The initial value of these bits sets up the block transfer, and on subsequent transfers the interface circuitry must supply a valid address and hold the LWORD* value which existed during the initial transfer cycle.

The state machine to perform these functions would seem at first glance to be relatively straightforward, but it was discovered that the machine is not easy to implement in any simple way and still be able to keep up with the timing requirements for maximum throughput. STC uses a design for implementing the state machine in a single 20RA10 PAL.

The MVME6000 is designed for interfacing 68020/30 processors to the VME bus. An analysis of this chip's specifications shows that the chip has a wide range of functionality. With only a small handful of additional logic, the MVME6000 may be used to create a VME/680x0 interface which conforms strictly to the VME bus specification, and which includes all VME functions except BLTs (block transfers), including all master/slave/system controller capabilities.

2.3 Summary of Interfaces

The EVA computer is comprised of several functional units each of which have multiple interfaces. Because of the versatile communication paths, the previous sections centered on those available to a user. Two boards serve as multiple interfaces. They are the IOP board which interfaces the CPH modules to the host, and the VME Buffer board which interfaces to the CPH, VPH, and a 6U VME backplane so that the CPH can communicate to a VME system. They are now listed for clarity.

Interface	Board	Description
PC to VPH	VPH daughterboard	VPH end of this Interface, see Sections 2.2.1, 3.2.6.1
PC to CPH	IOP	6U board plugs into CPH backplane, see Sections 2.1, 2.1.1, 3.2.4, 3.2.6
PC to ISA	PC-INT	ISA bus board plugs into 286 and 386, see Sections 1.1.2.3 and 2.1.1
VPH to CPH	VME Buffer	6U board plugs into CPH backplane, see Sections 2.1.3 and 3.2.5
VME to VPH	VPH	integral part of VPH board, see Sections 2.2.2
VME to CPH	VME Buffer	same board used to interface to CPH to VPH and also called SIO or Serial IO board, see Sections 3.2.5
Internal CPH	HSIO	high speed IO bus that communicates among the CPH modules (processor, AG, IOP, cache memory), see Section 3.2.7

3.0 Theory of Operation

With this introduction to interfaces, the theory of operation section describes the remaining architectural details of EVA. Section 3.1 starts with the VPH and its internal register resources. Operating the VPH will require a thorough understanding of the VPH to VME interface. Hence, the programmer's model and the VPH address map are presented so that a programmer may know which addresses on the VME bus correspond to internal VPH resources. The address map is presented early because addresses for the 68020 are different than those for the DSPs. (The DSPs are designed by the manufacturer to address words. The 68020 can address bytes.) They are numerous and include control, status registers, two program RAMs or PRAMs 1 and 2, a 4-port, and 68020 registers. Section 3.2 covers the CPH and its resources, again very numerous including the processor, cache, address generator, IOP, and VME Buffer. Because some boards (e.g. the VME Buffer board) serve multiple functions, it will be necessary to return to earlier sections at times. The versatility of EVA is evident in its many interfaces and operating modes. Those operating modes include VPH in VME systems (such as the TSI tracker), CPH/VPH as EVA, and CPH in VME systems. Note that the VME buffer board allows the CPH to be hosted by a system other than a PC.

The previous sections described the general architecture and interfaces of the CPH and VPH. With this introduction it is now possible to discuss the operation of both in more detail. The following sections begin with a description of the VPH resources and end with those of the CPH. In the process, additional architectural hardware details are presented as needed. These are accompanied by the microinstruction format and machine definition file for the CPH found in the appendices. To understand the theory of operation of each functional unit it will be necessary to know much about the individual address spaces, control signals, and assembly language, and microinstructions of the IOP, CPH, VME buffer board and PC interface board. Such information is also presented in this Section.

3.1 VPH

The VPH-20 is a multi-processor DSP board suited to FFTs, FIR and IIR filters, spectrum analysis, Kalman (and other) adaptive filters, and numerous other DSP tasks. The VPH-20's processing power comes from four Zoran ZR34325 Vector Processor chips (arranged two chips on each of two buses) and one Motorola 68020 microprocessor. The VPH-20's unique architecture allows concentration of all processors on a single task for the highest processing speed, or partitioning of the processing resources to handle multiple simultaneous tasks. The VPH-20 performs a 1024-point complex FFT in as little as 604 us at 20 MHz (483 us at 25 MHz).

The form factor of the VPH-20 is a standard 9U-4H (366.7 X 340.0 mm) board. This is the standard VXIbus "D"-size board. The VPH-20 requires a single slot in the VME/VXI backplane unless the optional PC interface daughterboard is attached, in which case two slots are required. The VPH-20 may be used in any environment where a standard VMEbus is in existence, including VXI systems. Since none of the user-definable pins are used by the VPH-20, it may be used in many systems which are based on a VMEbus with extensions, such as Sun Microsystems.

Integral to the VPH-20 is a standard VME bus interface which allows the VPH-20 to operate in either Master or Slave modes. The system may also be configured as a VME System Controller board. The system architecture allows for transactions to occur on the VME bus without interfering with signal processing operations.

An optional high-speed PC interface allows the VPH-20 to be tied to any standard PC/AT-compatible computer. This interface may be used in conjunction with the VME interface, allowing a PC to be used for any number of purposes such as process monitoring, data display, etc.

PROGRAMMER'S MODEL

A brief discussion of the system architecture including a system memory map and the programmer's model follows. Documents which may be of additional help include:

32-Bit Microprocessor User's Manual
Motorola #MC68020UM/AD

ZR34325 32-Bit Floating-Point Vector Signal Processor
Zoran Corporation #DS34325-0989-1.5K

MVME6000 VMEbus Interface User's Manual
Motorola #MVME6000UM/D1

The VPH-20's four vector processors are arranged with one pair of processors on each of two local buses. Each bus has 32k longwords of high-speed static RAM (SRAM) for the use of the two vector processors the bus serves. In addition, each VSP bus may access one port of the system's four-port SRAM. This four-port SRAM is a memory resource which is common to all system resources; the use of such a memory area allows multiple resources to access the same memory area simultaneously and without conflict - a single memory location may be read from each of the four ports at the same time. The size of the four-port SRAM is 4k longwords.

Another resource common to all five processors is a status latch which provides a simple means of providing for primitive semaphore communication between processors. Each processor may write two status bits to the status latch; a read of the latch yields the eight status bits from the other four processors.

The 68020 has access to all system resources, including the local memories on each of the VSP buses and the internal registers of the four VSP chips themselves. The VSPs have access only to their local memory, the global status latch, and the four-port memory. All off-board communication is handled by the 68020.

The VMEbus interface is based on the Motorola MVME6000 interface chip. This versatile arrangement allows the VPH-20 to function in the Master or Slave modes, and also allows the VPH-20 to be configured as the VME system controller. The VPH-20 may access the entire 32-bit VME address space. The VPH-20's location in the VME address space is user-configurable over a wide range.

The optional PC interface allows the VPH-20 to communicate with any PC/AT-compatible machine. The PC interface is designed to provide much faster communication between the PC and the VPH-20 than could be achieved with conventional serial or parallel communication techniques, thereby making the PC a handy and useful addition to a system utilizing the VPH-20.

An examination of the Programmer's Model diagram in Figure 14 shows that there remain two resources not yet discussed. The DSACK Generator handles the task of terminating 68020 bus cycles at the appropriate time. Its operation is normally transparent to the user, and need not be considered in most situations. The Expansion Bus allows for the addition of any of a number of 68020-compatible subsystems, such as A/D and data acquisition, etc. Any resource which is "tacked on" to the system expansion bus will have its bus cycles terminated by the DSACK generator according to values loaded into the DSACK RAM. These values define the cycle times (wait states) necessary for addresses within the region of the 68020 address space reserved for system expansion (the upper 2 Gbytes).

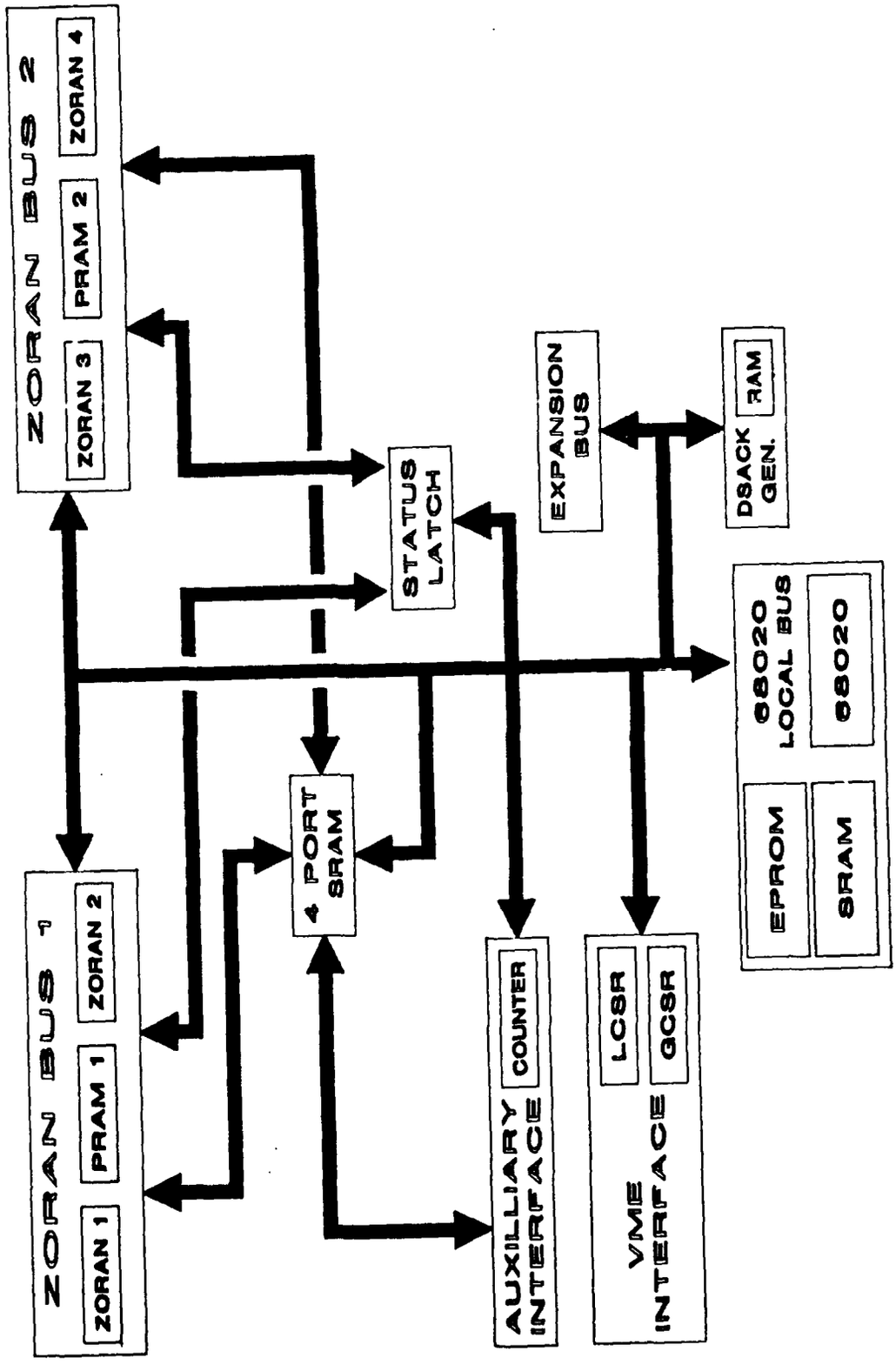


Figure 14. VPH Programmer's Model

The following 68020 address map shows where the various resources reside in the 68020 address space.

Hex Address	Resource
0 - FFFF	EPROM
4 0000 - 5 FFFF	68020 SRAM
8 0000 - 8 3FFF	Four-port SRAM
C 0000 - C 0FFF	Zoran 1 Internal Registers
C 1000 - C 1FFF	Zoran 2 Internal Registers
10 0000 - 11 FFFF	Zoran Bus 1 SRAM (PRAM)
14 0000 - 14 0FFF	Zoran 3 Internal Registers
14 1000 - 14 1FFF	Zoran 4 Internal Registers
18 0000 - 19 FFFF	Zoran Bus 2 SRAM (PRAM)
1C 0000	Global Status Latch
1C 0004	Zoran RESET Latch
20 0002	REQUEST (Write) or RELINQUISH (Read) *VMEbus* (byte or word access)
24 0000	PC Interface FIFO
24 0004	PC Interface Status/Control Register (longword access)
24 0008	PC Interface Interrupt Register (longword access)
28 0001 - 28 001B	MVME6000 LCSR (Odd Bytes) (byte access)
28 0021 - 28 002F	MVME6000 GCSR (Odd Bytes) (byte access)
2000 0000	DSACK SRAM Enable
6000 0000	DSACK SRAM Disable
8000 0000 & above	Expansion Space

A more detailed discussion of individual resources follows.

ZORAN BUS 1 & 2

Each Zoran bus (or VSP bus) serves two Zoran VSP chips and a 32K longword area of local SRAM. In addition, each VSP bus has a port into the four-port memory and can access the global status latch. The following VSP address map shows the location of resources as seen by any one of the VSP chips.

Hex Address	Resource
0000 - 7FFF	Local SRAM
2 0000 - 2 07FF	Four-port SRAM
4 0000	Global Status Latch

Note that VSP addresses 2 0000h - 2 0FFFh correspond *exactly* with 68020 addresses 8 0000h - 8 3FFFh for *both* VSP buses. In addition, VSP Bus 1 addresses 0000h - 7FFFh correspond to 68020 addresses 10 0000h - 11 FFFFh and VSP Bus 2 addresses 0000h - 7FFFh correspond to 68020 addresses 18 0000h - 19 FFFFh. The reason for the apparent difference in address ranges between the 68020 and the VSPs is due to their respective methods of addressing. The VSPs can only access longword memory locations, whereas the 68020 can access individual bytes. The 68020 then has, in effect, two more least significant address bits than the VSPs. The difference in address ranges and their locations is very important to the programmer. The following table should be of assistance in converting the addresses of common resources between the various buses; the programmer should thoroughly familiarize himself/herself with this table.

To Convert VSP:	To 68020:	Use Formula:
Bus 4-port Address	4-port Address	(VSP addr. - 2 0000h)*4 + 8 0000h
Bus 1 SRAM Address	Address	(VSP addr.)*4 + 10 0000h
Bus 2 SRAM Address	Address	(VSP addr.)*4 + 18 0000h

To Convert 68020:	To VSP:	Use Formula:
4-port Address	Bus 4-port Address	(020 addr. - 8 0000h)/4 + 2 0000h
Bus 1 Address	Address	(020 addr. - 10 0000h)/4
Bus 2 Address	Address	(020 addr. - 18 0000h)/4

3.1.1 VPH Internal Control

It is important to know how internal controls operate on the VPH since a user will be coding directly to Zoran status latches, Zoran program memory space (PRAMs 1 and 2), and the 4-PORT SRAM. The following information describes address and status latch maps.

To write to DSACK SRAM:

Write to any address such that A[31..29]=[001]. This disables address buffers and allows access to the DSACK SRAM, which is addressed with the vector A[31,24..18].

Write to any address such that A[31..29]=[011] to disable DSACK SRAM load mode and re-enable address buffers.

To gain/relinquish control of the VME bus:

Write to any address such that A[31..22,20..18]=0, A[21]=[1], and A[2,1]=[01] to request mastership (byte or word access.

Relinquish the VME bus by reading A[31..22,20..18]=0, A[21]=[1], and A[2,1]=[01] (byte or word access).

Status latch access:

The 68020 may access the status latch at address A[31..21]=0, A[20..18]=[111], A[2]=0. When reading the latch, the bit pattern is:

D[7] D[6]	D[5] D[4]	D[3] D[2]	D[1] D[0]
Bits from Zoran #4	Bits from Zoran #3	Bits from Zoran #2	Bits from Zoran #1

In addition, D[27..16] reflect PC Interface status bits STAT-[11..0] when the interface is on board.

When writing to the latch, the bit pattern is:

D[1] D[0] (All other bits are don't cares.)
Bits from 68020

The 68020 may send RESET commands to any of the Zorans by writing to A[31..21]=0, A[20..18]=[111], A[2]=1. The bit pattern is:

D[3]	D[2]	D[1]	D[0] (All other bits are don't cares.)
Zoran #4	Zoran #3	Zoran #2	Zoran #1

A '1' written to one of these bit positions causes the appropriate Zoran to be reset and put in the SLAVE mode.

PC Interface access:

The base address for access to the PC Interface is at A[31..22,20,19]=0, A[21,19]=[11]. In addition, A[3,2] are used to access specific resources within the interface. All accesses to PC Interface registers are longword accesses, but only D[15:0] are used.

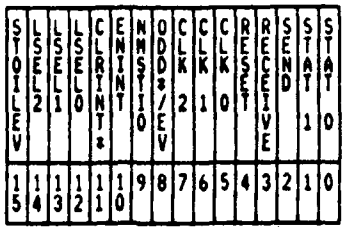
To read or write the FIFO, A[3,2]=[00].

To read the status register or write the control register, A[3,2]=[01]. (The status register may also be read by reading the status latch as described above.)

To read or write the interrupt register, A[3,2]=[1,0].

PC Interface registers:

Control Register



STAT 0 & 1 - These are general purpose interface bits. A bit written to STAT 0 or 1 in the Control Register appears as STAT 0 or 1 in the Status Register at the other end of the interface.

SEND - This bit is an enable for the sending of data across the interface. A 0 written to this bit does not disable the ability to write to the output FIFO, but does prevent data in the output FIFO from being sent until a 1 is written to this bit.

RECEIVE - This bit is an enable for the receiving of data across the interface. A 0 written to this bit does not disable the ability to read data in the FIFO, but does prevent the FIFO from receiving additional data until a 1 is written to this bit.

RESET - A 1 written to this bit resets the entire interface. The FIFOs are cleared, zeros are written to all bits of all three registers. (This effectively clears the RESET command once it has been effected.)

CLK 0,1,2 - These bits set the rate at which output data is clocked across the interface.

ODD*/EVEN - This bit selects odd or even parity across the interface.

NMSTIO - Setting this bit makes a high level on the incoming STAT 0 the highest priority interrupt, thus giving the PC priority over any VME interrupts. (The level of the request as passed to the 68020 is set by bit 15.)

ENINT - This is an enable for PC interrupts.

CLRINT* - A 1 written to this bit clears all PC interrupts. The bit does not self-clear, so a 0 must be written to this bit after interrupts have been cleared.

LSEL0,1,2 - These bits set the level of the interrupt passed to the 68020 in response to a PC interrupt request. (A request via the STAT 0 line has its interrupt level set by bit 15 rather than by these three bits.)

STOILEV - This bit determines the interrupt level passed to the 68020 (level 3 or 7) in response to a PC interrupt request on STAT 0.

Status Register

X	X	X	X	X	RFAF*	RFAE*	RFF*	RFE*	WFAF*	WFAE*	WFF*	WFE*	PARITY*	STAT 1	STAT 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Interrupt Mask Register

X	X	X	X	X	RFAF	RFAE	RFF	RFE	WFAF	WFAE	WFF	WFE	PARITY	STAT 1	STAT 0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

3.1.2 VPH Control Signals

When performing board level diagnostics or reprogramming PALs, the following signals may be needed. They are listed for completeness. Should future WSMR applications call for functional design changes, these sources of PAL signals will assist in the process. The device and signal names refer to VPH schematic labels. The schematic is an E-size drawing (3'x4') and is provided separately from the Final Technical Report.

4-PORT SRAM

68K PORT - /OE2 GROUNDED
/CE2 (FOR EACH BYTE) FROM 4PORTCS PAL
/WR2 FROM U139 (BUFFERED R/W)

ZORAN PORT 1 - /OE4 FROM ZDEC2 PAL (/CE1 OUTPUT)
/CE4 FROM ZDEC2 PAL (/CE2 OUTPUT)
/WR4 FROM ZDEC2 PAL (/WRA OUTPUT)

ZORAN PORT 2 - /OE1 FROM ZDEC2 PAL (/CE1 OUTPUT)
/CE1 FROM ZDEC2 PAL (/CE2 OUTPUT)
/WR1 FROM ZDEC2 PAL (/WRA OUTPUT)

BLT64 PORT - /OE1
/CE2
/WR2

ZORAN 1 & 2 PRAM

R/W - FROM ZDEC2 PAL (/WRA OUTPUT)
/OE - FROM ZDEC2 PAL (/OEA OUTPUT)
/CE - FROM ZDEC2 PAL (/CE3 OUTPUT)

68K EPROM

/OE & /CE (FOR EACH BYTE) FROM 68KMEMCS PAL

NOTE: PIN 1 ON EACH EPROM IS SELECTABLE VIA JMP1 JUMPER TO BE EITHER +5V OR AN UPPER ADDRESS BIT. THIS ALLOWS EITHER 128K OR 256K EPROMS TO BE USED.

68K SRAM

R/W - FROM U139 (BUFFERED R/W)
/CE & /OE - (FOR EACH BYTE) FROM 68KMEMCS PAL

ZORAN BUS ARBITRATION

Arbitration on each of the 2 Zoran buses is handled by a group of 4 PALs - ZARB, ZDEC1L, ZDEC1H, and ZDEC2. These PALs handle generation of all control signals related to operation of the bus, including processors, memory (both local and 4-port), and status latch. RESET is not handled by these PALs.

ZARB PAL - This PAL handles most of the bus arbitration functions. Inputs to the PAL include Block Select signals for the Zorans and PRAM on the bus, Bus Request signals from each Zoran, WRITE signals from each Zoran, and a R/W signal from the 020.

Outputs include Bus Grant signals to each Zoran, a GEN signal which enables the 020 to Zoran bus transceivers, ZDDIR and ZADIR signals which control direction of the Zoran address and data bus transceivers, and 2 qualified Block Select signals which are used by other control circuitry.

ZDEC1x PALs - These PALs provide decoding and generation of control signals to the Zorans. The ZDEC1L PAL handles the lower-numbered Zoran, the H PAL handles the higher-numbered one. The control signals these PALs handle are the Zoran Chip Selects, Data Strobes, Reads and Writes, and the Ready signals.

ZDEC2 PAL - This PAL handles generation of WRITE and Chip enables for local PRAM, Chip Enables for the 4-port SRAM and PRAM, and a Status Latch Enable.

DSACK GENERATOR

The DSACK generator handles generation of DSACK signals to the 020. These signals require different timing for the various different memory spaces in the system. The DSACK generator consists of 2 PALs, DSGEN and ROMPAL, a small SRAM, and a switch setup for setting default wait cycle lengths.

ROMPAL PAL - This PAL acts as a 9 X 4 ROM containing configuration data for 8 blocks of memory. A G output serves to disable the 020 address and data bus buffers when the DSACK generator SRAM is being loaded. The G signal also acts as an input to the DSGEN PAL for correct DSACK generation during SRAM loading. A Write Enable is output to the DSACK SRAM, as is an Output Enable. 4 configuration bits (CBIT0-3) are output to the DSGEN PAL.

DSGEN PAL - This PAL handles the generation of the actual DSACK0-1 signals to the 020.

3.1.3 VPH Configuration Procedures

There are a number of hardware and system level considerations to take into account when configuring the VPH. The following sections will address some possibly critical issues and outline the procedures for configuring the VPH hardware. Switch and jumper settings will be treated, as will "software" configuration of board and system functions.

3.1.3.1 System Controller Selection

In a VME system, slot 1 of the backplane (usually the leftmost slot as viewed from the front) is reserved as the system controller slot. The board performing the system controller function drives the VME 16 MHz system clock line, the IACK daisy chain, and the BG0-3 daisy chains. The system controller also provides bus arbitration for the system.

The VPH may be configured as either a standard VME board or as the VME system controller. This is accomplished with JMP2 on the VPH board. This jumper is located near the MVME6000 chip, which is the one with the cooling tower on it. With the jumper in position 1 (shorting pins 1 and 2) the board is NOT the system controller. With the jumper in position 2 (shorting pins 2 and 3) the VPH is configured as the system controller.

Configuration of the board's VME bus arbitration module is necessary when the VPH is configured as the system controller. A discussion of how to do this may be found in the section "LCSR DESCRIPTION".

Please note that a board configured as the system controller may be positioned ONLY in slot 1 of the VME backplane; a VME system may be comprised of many boards but only the board in slot 1 may be a system controller.

3.1.3.2 020 EPROM Size Selection

The VPH is designed so that a number of different sizes of EPROMS may be used. The EPROMS are socketed in ZIF sockets for ease of code development. 128, 256, or 512 kbit EPROMS may be used by proper setting of JMP1 and JMP5, which are located near the EPROMS. The table below indicates proper jumper

settings for each of the three EPROM sizes. Note that position 1 indicates that the jumper is shorting pins 1 and 2, position 2 indicates that pins 2 and 3 are shorted.

EPROM Size (kbits)	Jumper Position JMP1	JMP5
512	2	2
256	1	2
128	1	1

3.1.3.3 GCSR Base Address Selection

The GCSRs (Global Control and Status Registers) are a resource associated with the VME interface. This group of 8 registers is physically located on the MVME6000 chip. A detailed description of the GCSR may be found in the section "GCSR DESCRIPTION". This section is dedicated to setting the GCSR base address.

The VPH GCSRs, as viewed from the VME bus, are located in the VME's Short Supervisory Access space (AM code $2D), which utilizes 16-bit addresses. This address space is typically partitioned in the following manner.

The upper 8 VME address bits (A15-A8) are used to define a Group Address. The next four bits (A7-A4) are used to address a board within a group. The lower 3 bits (A3-A1) are used to address a specific resource of a board within a group. This partitioning concept isn't hard and fast, but many boards conform to this structure. The VPH's VME interface GCSRs are located in this address space, and configuration is necessary to position the GCSRs at a specific location in the short I/O space.

The GCSR base address, referred to above as the "group address", is determined by the setting of S1 on the VPH board. This switch is an 8-pole DIP switch located next to the top edge of the board. The lowest bit of this switch corresponds to VME A8; the highest bit of this switch corresponds to A15. A switch in the "on" position selects a zero for a given bit, the "off" position selects a one.

EXAMPLE: To set the GCSR group address to $8Dxx, the S1 switch settings, from highest (S1-8) to lowest (S1-1), would be:

off on on on off off on off

The GCSR board address is configured through software by writing the desired value for A7-A4 into the register at an offset of $1B from the base address of the LCSR. (This procedure is covered in the section "LCSR DESCRIPTION".) The lowest 3 bits (A3-A1) are decoded by the MVME6000 to access one of the 8 registers of the GCSR.

3.1.3.4 VME Slave Address Modifier Code Selection

The Address Modifier (AM) code that the VPH's VME slave will respond to is configured through a combination of hardware and software means. This section deals primarily with the hardware configuration; more information on the software configuration may be found in the section "LCSR DESCRIPTION".

Decoding of the VME AM bits is done by both the MVME6000 and U135 on the VPH board. This has been done in order to allow more versatility in mapping the VPH into the VME address space than is allowed by the MVME6000 alone. A discussion of the MVME6000's AM decoding may be found in the section "LCSR DESCRIPTION" or in the MVME6000 hardware manual. (Note that the MVME6000 always sees a zero on AM4 regardless of the level actually present on the bus.) The following section describes the decode functionality of the U135 PAL; two versions of this PAL have been supplied to provide two different mapping sets for the VPH VME slave. Information contained in this and other sections should allow creation of additional PALs to provide other slave mappings.

The function of U135 is to look at the AM code present on the VME bus and determine if the AM code present is correct for an access to the VPH's VME slave. When a valid AM code is detected, an enable signal (MATCH32) is passed on to the MVME6000 to enable the VME slave. The MVME6000 then re-qualifies the AM code, with AM4 presented as a zero regardless of the level on the bus. This allows the VPH slave to respond to the VME AM codes that the MVME6000 would normally reject.

The "MATCH" version of U135 maps the VPH slave to one of the normal VME AM code sets. In order to enable the slave, the AM code must have the upper two bits low. The lower four bits are compared to the setting of the switches on S2 to complete the decode. S2-1 through S2-4 correspond to AM0 through AM3, respectively. This allows the slave to respond to the AM codes in the range $00 through $0F. However, within this group of AM codes, $00 through $08 are reserved as is $0C. The MVME6000 can not be made to respond to these codes. In addition, the MVME6000 is not capable of block transfers, so codes $0B and $0F are also eliminated. The remaining four codes, their VME transfer types, and the value that must be loaded to the MVME6000's LCSR $0B slave address modifier register (020 address $28000B) are summarized below.

AM Code	VME Transfer Type	Register Value
$09	Extended Nonprivileged Data Access	0bX11XX0X1
$0A	Extended Nonprivileged Program Access	0bX11XX01X
$0D	Extended Supervisory Data Access	0b1X1XX0X1
$0E	Extended Supervisory Program Access	0b1X1XX01X

The "MATCHA" version of U135 allows mapping of the VPH slave into AM

codes $10 through $1F. These are "User Defined" address regions. Keep in mind that since the MVME6000 always sees a zero on AM4, the AM code seen by the MVME6000 will be $10 less than the value actually present on the bus. In order to ensure response from the MVME6000, it is recommended that only codes $19, $1A, $1D, and $1E be used. It is possible that other AM codes within this block would be acceptable to the MVME6000, but this would have to be established through experimentation; it is easier just to utilize one of the four prescribed patterns. These AM codes and the Address Modifier Register values are summarized below. Note that all VME transfer types are actually "User Defined" - the transfer type shown is the type assumed by the MVME6000.

AM Code	Transfer Type	Register Value
$19	Extended Nonprivileged Data Access	0bX11XX0X1
$1A	Extended Nonprivileged Program Access	0bX11XX01X
$1D	Extended Supervisory Data Access	0b1X1XX0X1
$1E	Extended Supervisory Program Access	0b1X1XX01X

Other mappings are certainly possible. DO NOT ATTEMPT TO MAP THE VPH SLAVE INTO ANY 16- OR 24-BIT ADDRESS SPACES! The VPH's address decoders require a full 32-bit address even though most of its resources are located within the lower 24-bit region. An attempt at mapping the slave into a 16- or 24-bit address space will likely result in system failure, since the upper address bits may not appear as expected. (One would expect the upper bits to be a sign extension of the 16- or 24-bit address, which for most 24-bit accesses would work. But if the upper bits float high, or if the sign bit is a "1", accesses would fail.)

New design files for U135 could be created easily to make the VPH slave respond to any of a group of AM codes. As an example, a possible alternate design file is shown below which would allow the slave to respond to any combination of AM codes $19, $1A, $1D, or $1E. (The appropriate value loaded to the slave address modifier register would depend upon the selected codes; 0b111XX011 would work for any selected combination for this example.) The function of S2 is shown below.

S2-1 Enable accesses on AM code $19 when "ON"
S2-2 Enable accesses on AM code $1A when "ON"
S2-3 Enable accesses on AM code $1D when "ON"
S2-4 Enable accesses on AM code $1E when "ON"

For instance, to allow slave access on codes $1D or $1E, turn switches 1 & 2 off, switches 3 & 4 on. The following PAL file for the MATCH PAL is vital to future changes to the VPH. It is included (verbatim) for complete understanding.

```
;PALASM DESIGN DESCRIPTION

;------------------- Declaration Segment------------
TITLE   MATCH32 AND MATCHGCSR DECODER PAL
PATTERN  MATCHB.PDS
REVISION 00                                          ;
AUTHOR   LARRY HALL
COMPANY  SPACE TECH CORP.                            ;
DATE     07/31/92                                  ;
                                                   ;
CHIP  MATCH  PAL22V10                                ;
                                                     ;
                                                     ;
; THIS PAL GENERATES TWO ENABLE SIGNALS WHICH ARE    ;
; USED BY THE MVME6000 TO DETERMINE IF AN ADDRESS    ;
; ON THE VME BUS BELONGS TO AN ON-BOARD RESOURCE.    ;
; IT ALSO PERFORMS 020 BUS ARBITRATION BETWEEN THE   ;
; 020 AND THE 6000, AND PROVIDES A 10 MHZ CLOCK FOR  ;
; THE 6000 BY DIVIDING THE 20 MHZ CLOCK BY TWO.      ;
; /MATGCSR INDICATES THAT THE 6000'S GCSR IS BEING   ;
; ACCESSED. /MATCH32 INDICATES THAT THE VME IS       ;
; ACCESSING THE VPH'S 32-BIT ADDRESS SPACE. THE      ;
; /MATCH INPUT IS THE OUTPUT FROM A 688 COMPARATOR   ;
; WHICH COMPARES THE A08-A15 BITS TO A VALUE SET     ;
; ON AN 8-BIT DIPSWITCH WHICH DEFINES THE "GROUP     ;
; ADDRESS" OF THE GCSR IN THE VME SHORT ADDRESS      ;
; SPACE. CLK IS THE 20MHZ CLOCK. THE B0-B3 INPUTS    ;
; ARE FROM A DIPSWITCH USED TO DEFINE THE AM CODE    ;
; USED TO ACCESS THE VPH FROM THE VME. THIS AM CODE  ;
; IS REQUIRED TO HAVE BIT 5 LOW AND BIT 4 HIGH. THE  ;
; ACCEPTABLE AM CODES ARE SUMMARIZED IN THE TABLE
; BELOW, ALONG WITH THE VME BUS SPEC'S DEFINITION OF
; THE AM CODE SEEN BY THE MVME6000 CHIP.
;
;   AM CODE  TRANSFER TYPE
;   ---------------------------------
;    $19     EXTENDED NONPRIVILEGED DATA ACCESS
;    $1A     EXTENDED NONPRIVILEGED PROGRAM ACCESS
       ;    $1D     EXTENDED SUPERVISORY DATA ACCESS
;    $1E     EXTENDED SUPERVISORY PROGRAM ACCESS
;
; /BGACK IS USED BOTH AS THE /BGACK INPUT TO THE 020
; AND AS THE /PBG INPUT TO THE 6000. /BR IS THE /BR
; INPUT TO THE 020. /DSACK0-1 ARE THE 020 /DSACK0-1
; LINES. /BG IS FROM THE 020. /PBR IS FROM THE 6000.

;------------------- PIN Declarations --------------
PIN  1         CLK                                ; INPUT
PIN  2          AM0                                ; INPUT
PIN  3          AM1                                ; INPUT
PIN  4          AM2                                ; INPUT
PIN  5          AM3                                ; INPUT
PIN  6          AM4                                ; INPUT
PIN  7          AM5                                ; INPUT
```

```
PIN  8          /AS                             ; INPUT
PIN  9          /B0                             ; INPUT
PIN  10         /B1                             ; INPUT
PIN  11         /B2                             ; INPUT
PIN  12         GND                             ;
PIN  13         /B3                             ; INPUT
PIN  14         /MATCH                          ; INPUT
PIN  15         /MATGCSR            COMBINATORIAL ; OUTPUT
PIN  16         /BGACK              REGISTERED ; OUTPUT
PIN  17         /BR                 REGISTERED ; OUTPUT
PIN  18         /MATCH32            COMBINATORIAL ; OUTPUT
       PIN  19         CLK10                REGISTERED ; OUTPUT
PIN  20         /PBR                            ; INPUT
PIN  21         /DSACK1                         ; INPUT
PIN  22         /DSACK0                         ; INPUT
PIN  23         /BG                             ; INPUT
PIN  24         VCC                             ;

;---------------------------------- Boolean Equation Segment ------
EQUATIONS

MATGCSR  =  AM5 * /AM4 * AM3 * AM2 * /AM1 * AM0 * MATCH

MATCH32  =  /AM5 * /AM4 * AM3 * /AM2 * /AM1 * AM0 * B0
          + /AM5 * /AM4 * AM3 * /AM2 * AM1 * /AM0 * B1
          + /AM5 * /AM4 * AM3 * AM2 * /AM1 * AM0 * B2
          + /AM5 * /AM4 * AM3 * AM2 * AM1 * /AM0 * B3

BR       =  PBR * /BGACK

BGACK    =  PBR * BG * /AS * /DSACK0 * /DSACK1
          + BGACK * AS
          + BGACK * DSACK0
          + BGACK * DSACK1
          + BGACK * PBR

CLK10    =  /CLK10

;---------------------------------- Simulation Segment ------------
SIMULATION

;-----------------------------------------------------------------
```

3.1.3.5 Initialization Considerations

It is expected that the need will exist to develop a wide range of application code for the VPH in the future. Since the board is not supplied with any type of an operating system, the system programmer developing code for the VPH needs to be aware of proper resource initialization procedures for various VPH resources. Such initializations are necessary at power-up, and possibly at any other time that the VPH is "reset" or reconfigured as required by some process. The following section discusses these considerations.

At power-up or other reset, the VPH's 68020 will begin execution at address 0 in EPROM. The initialization sequence is the standard sequence as described in the 68020 User's Manual; the first few locations in EPROM contain initial stack pointers, the execution start address, etc.

It is recommended that the boot sequence for the 020 load the SFC and DFC registers with $3, as this is the function code used for accesses to VME via the MOVES instruction.

If the PC interface is to be used, the control registers for the interface must be set up appropriately. See sections on the PC interface for more information.

When the VPH wakes up, the DWB bit at 020 address $200002 will be asserted. This causes the VPH to request the VME bus and, once granted, will not release until the DWB bit is negated. This should be done early in the boot sequence so as not to interfere with other boards' ability to complete their boot sequences. Negating the DWB bit may be accomplished by doing a byte read of location $200002 in VPH local memory space.

Proper initialization of Local and Global Status Registers will be required before the VPH's VME slave and/or master will function properly. Information on the MVME6000's LCSR and GCSR may be found elsewhere, either in this document or in the MVME6000 User's Manual. There is no hard and fast rule as to how to set up the MVME6000; the necessary initialization will depend upon the application and overall system configuration, and must be determined by the system programmer.

One thing that will need to be done in nearly any situation at boot is to clear the BRDFAIL bit in the System Controller Configuration Register in the LCSR. If this is not done, the SYSFAIL line on the VMEbus will be asserted, which will bring the system to its knees before it ever gets up and running. This negation may be accomplished by a byte write of $4 to 020 address $280001.

It is good practice to clear the Zoran interrupts, reset the Zorans, and clear the 020's status bits at boot. This may be accomplished by writing zero to 020 longword location $1C0000 and $F to $1C0004.

Also necessary at boot is loading configuration data to a couple of locations in the DSACK SRAM. These locations are for accesses to the MVME6000 and/or VME bus, and the PC interface (if used). The following code segment will accomplish the DSACK SRAM initialization.

```
MOVEA.L #$20000000,A0    ;DSACK SRAM ENABLE ADDRESS
MOVEA.L #$60000000,A1    ;DSACK SRAM DISABLE ADDRESS
MOVEA.L #$240000,A2      ;PC INTERFACE BASE ADDRESS
MOVEA.L #$280000,A3      ;MVME6000 REGISTER SET BASE ADDRESS
MOVE.L  #0,(A0)          ;ENABLE DSACK SRAM
MOVE.L  #$4,(A2)         ;WRITE CONFIGURATION NYBBLE TO SRAM
MOVE.L  #$1,(A3)         ;WRITE CONFIGURATION NYBBLE TO SRAM
MOVE.L  #0,(A1)          ;DISABLE DSACK SRAM
```

3.1.4 VPH Installation and Setup Procedures

The following procedure describes the installation and setup of the VPH and SBC. It shall be used for a cold start sequence (e.g. the unit directly out of the box). The instructions are also useful when the board settings of either the VPH or the SBC have been changed. Before any of the following steps are taken, you should read and study the VPH User Manual, the MVME6000 manual, and the SBC Manual for the 135 board. A thorough understanding of the address spaces of each board will be necessary if hardware or software modifications are to be made. This will help prevent inadvertent address space overlap.

MODE 1: SBC system controller/VPH non-system controller

1. Set the VPH switches as follows

s1	1-8	all off	(address map)
s2	1-4	off on on off	(AM code mods)
s3	1-4	on off of off	(default DSACK wait states, used in expansion bus)

2. Set VPH jumpers as follows

JMPR1	(set for EPROM size)
JMPR2 short 1 and 2	(VPH non-system mode)
JMPR3	(set for # of Zoran ext memory access wait states)
JMPR4	(set for # of Zoran ext memory access wait states)
JMPR5	(set for EPROM size)

3. Set the SBC switches as follows

s3	1-8	#4 on, all others off
s4	1-10	4, 8, 9 on, all others off

You are now configured for the SBC to operate as system controller. Plug it in slot #1 (left most slot of chassis). 135 Dbug will run at its base DRAM address. The SBC is configured to operate with 32-bit address and 32-bit data.

3.1.5 Typical VPH Operation

The following sections describe the typical execution sequence that is recommended for the VPH 325 chips. The current set of application code has adhered to these procedures. They serve to provide a uniform basis for future coding practices and will maintain better documentation if consistency is applied to the programming methodology.

The major programming convention is necessary to ensure that the four 325 chips initiate activity simultaneously. In this manner the code executed by each chip will start at the same point in the programs and end at the same point in the programs. Zorans describe execution across multiple chips as waves. Hence, synchronization of the wave processing is desired. We say that a chip or a set of chips completes a wave when each and every chip has executed its code segment relative to that wave.

Synchronization is depicted in Figure 15. Here, a starting routine is executed first. In the current suite of code, a routine called STARTUP.ASM is used for most of the applications. It is a generic routine for any of Zoran's application libraries as well. Startup initializes the status bits in the status latch so that the 68020 or 020 can synchronize Zorans. In startup the Zorans do not modify the status bits. In a polling loop, the 020 will modify these bits when it is ready to initiate Zoran starts simultaneously.

Once the 020 sets the status bits accordingly, the 325s begin wave 1 processing. Wave 1 processing consists of any routines a user wants the 325s to execute such as convolution or FFT. When every 325 that is processing has completed their tasks, they individually set their status latch bits. Now the 020 has been monitoring all bits in a poll status loop. Upon detecting that each and every 325 has completed wave 1 tasks, the 020 modifies the status bits to allow the 325s to begin wave 2 processing.

Figure 15 depicts only two waves, but the concept is not limited to only two waves. As many waves or routines as are desired may be used in this method. Further, the waves may be any routines desired by the user. They do not have to be the same code.

Another important programming convention is the consistent usage of the stack frames as depicted in Figure 16. The example discussed assumes that two 325s are sharing the same bus, probably 325s 1 and 2 using PRAM 1. The convention should be followed no matter how many 325s are used or how many 325 buses. The two key 325 registers are the stack pointer (SP) and the program counter (PC) of each 325. To synchronize execution across multiple 325s, it will be necessary to start them with correct program starting addresses. Those are popped off the stacks. A stack frame will then consist of addresses for important locations like the program starting addresses, locations of parameters to pass into and out of the routine or subroutine.

Those addresses are found in the MAP file of the code relevant to the current application. They are generated by the Zoran 325 assembler process. Each address must be linked into the program, so a specific procedure is followed. The Zoran Assembler Manual explains the method. The current application library has adhered to this procedure in every program.

The typical execution begins with each 325 with the correct PC and SP value in them. Note that the SP points to the first location below the starting location. Upon initiation of execution, the SP is incremented first and then the value is popped off the stack. The 4-port serves as the data space for each 325 which the stack pointers 1 and 2 (or as many as you need) point to. The PRAM contains the actual routine used in the current application. The code should always start at location 0000 as this makes assembly easier. Also, keep sufficient space between each stack pointer in the PRAM so that the 325s do not inadvertently write into your stack (as might occur with an interrupt).

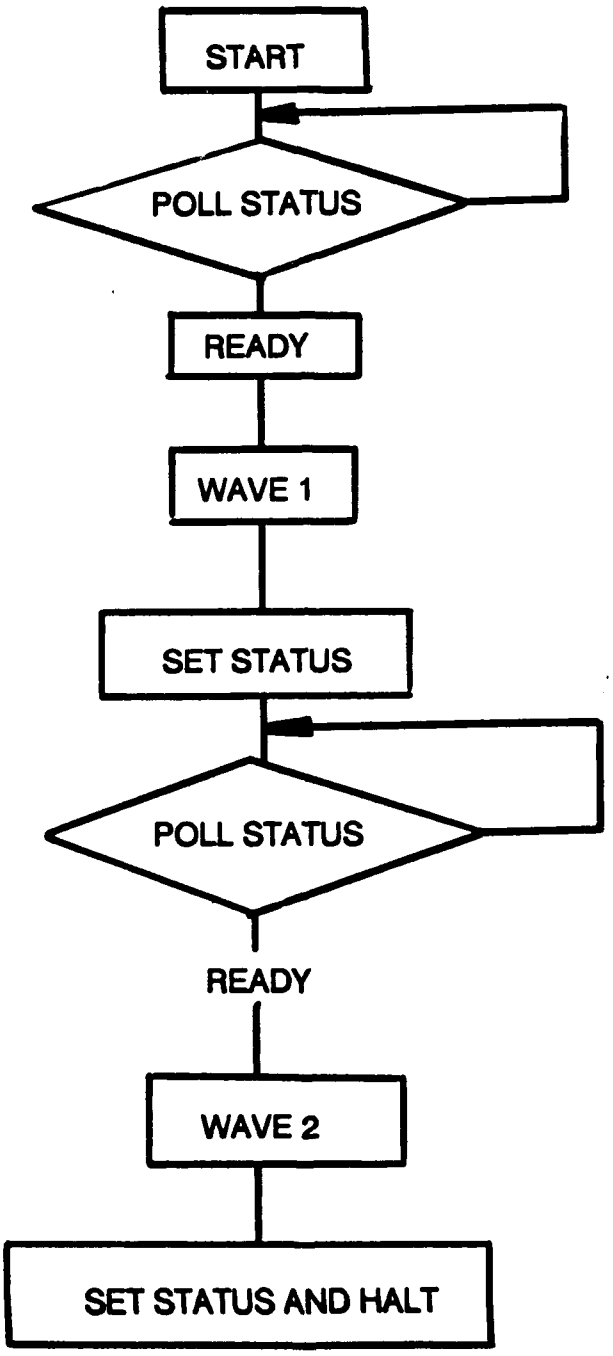


Figure 15. Synchronisation

PARAMETER PASSAGE CONVENTIONS

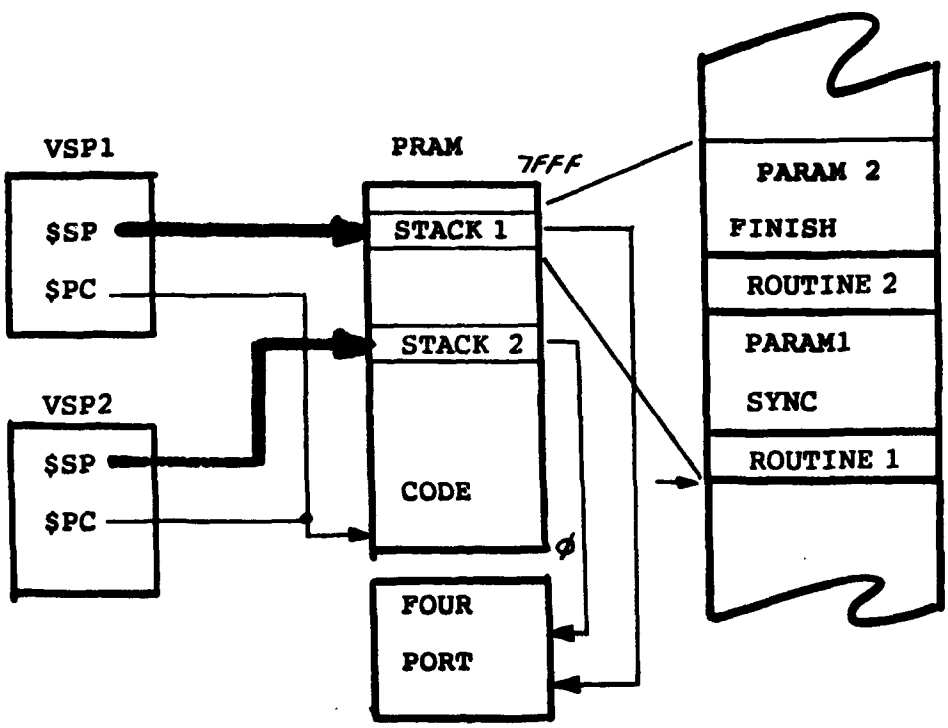


Figure 16. Parameter Passage to Routines VIA Stacks

A typical stack frame is shown on the left of Figure 16. Two routines are assumed, each with a synchronization call and list of parameters. The last routine will also execute a STC provided FINISH routine. FINISH cleans up the status latch bits to indicate to the 020 that the wave(s) by all 325s have been executed. The 020 then uploads the results into the correct 4-port space. This activity is shown in Figure 17. Again for consistency, all current programs follow this activity flow. Near the bottom of the chart is a decision box. If more routines are to be executed, the resultant path depends on the routines invoked. Typically, the path continues up to set 325 mode bits.

3.1.5.1 System Bootup

To bring up the VPH system with the 68020 monitor program, just turn on the power. If the VPH stops responding for some reason, it can be reset with the reset switch found on the board itself.

3.1.5.2 Initialization

If the io monitor program with a PC is being used, it is important to set up its status register manually. Then the Zoran interrupts must be cleared and the Zorans must be reset again. The steps for this are as follows. Keystrokes are shown in square braces.

1. set the port to the status register [P 362 <CR>]
2. clear the interface by writing ones [W FFFF <CR>]
3. set up the correct status values [W 186C <CR>]
4. set the port back to the FIFOs [P 360 <CR>]
5. clear the interrupts with a poke of 0 to address 1C000
   [W 12 <CR> W 0 <CR> W 1C <CR> W 0 <CR> W 0 <CR>]
6. reset the Zorans with a poke of F to address 1C0004
   [W 12 <CR> W 4 <CR> W 1C <CR> W F <CR> W 0 <CR>]

If a script is being used, all of these operations can be conveniently performed by a single call to the Init() function.

3.1.5.3 Transfer Programs to Zoran Program RAM (PRAM)

If the io monitor program is being used, programs can be downloaded with the Download command. As an example, assume that the file fft2d32.s is being downloaded to PRAM1 and PRAM2, which start at addresses 100000 and 180000. The command sequence would be

[D fft2d32.s <CR> 100000 <CR> D fft2d32.s <CR> 180000]

If a script is being used, programs can be downloaded with a call to the Download function. For the example, the call would be

```
Download("fft2d32.s", 0x100000);
Download("fft2d32.s", 0x180000);
```

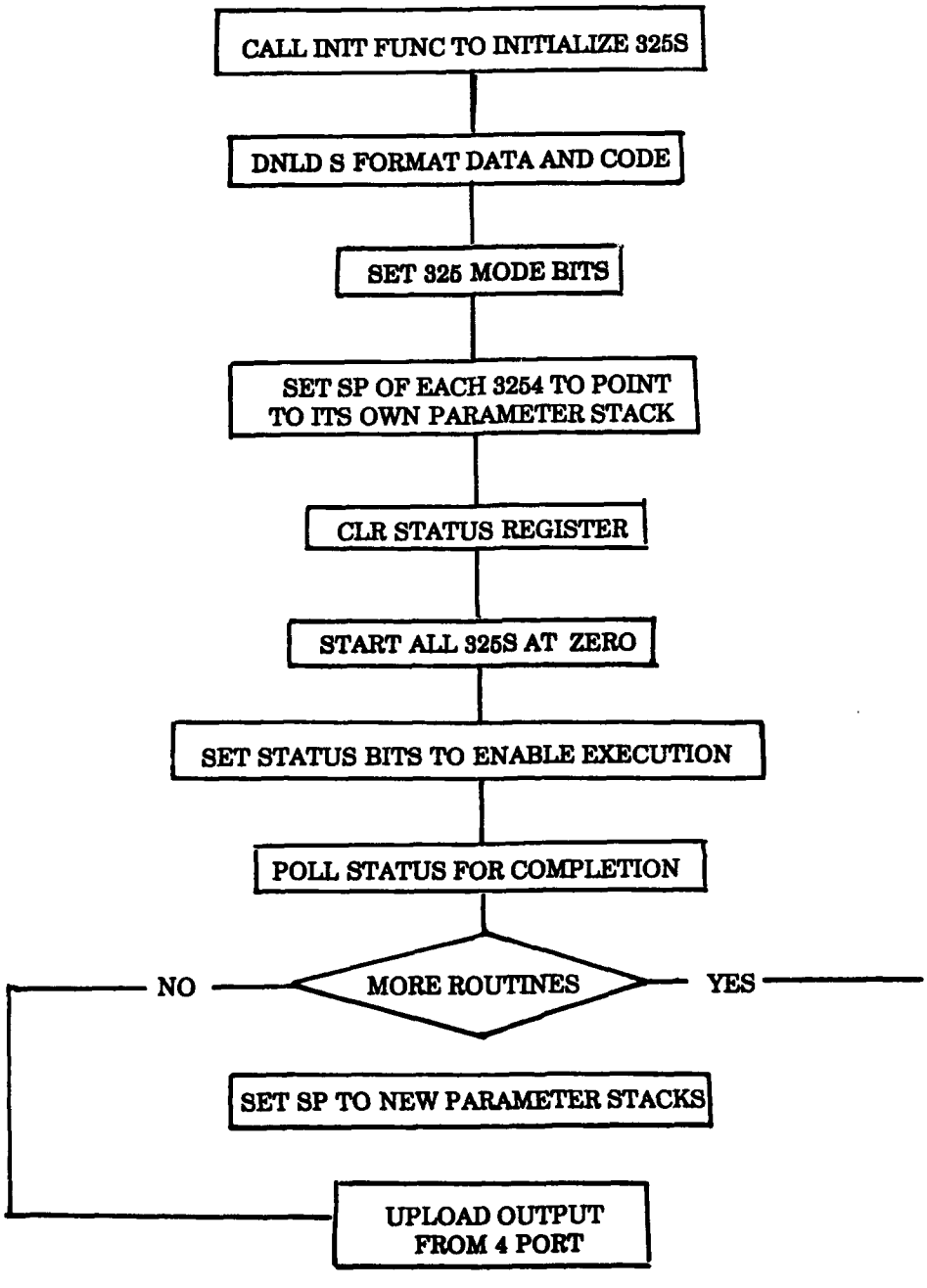


Figure 17. Typical VPH Activity Flow Chart

Macro definitions can be used to simplify this to

```
#define PRAM1 0x100000
#define PRAM2 0x180000
Download("fft2d32.s", PRAM1);
Download("fft2d32.s", PRAM2);
```

3.1.5.4 Data Transfer to/from Four Port Memory

If the io monitor program is being used, data files can be downloaded with the Download command as well. These files will generally be ASCII hexadecimal files. If the Zoran or Motorola assemblers are used to create data files to go into the four port memory, an address offset of zero is used instead of the values here. This is because the S format files already contain the correct addresses for each record. This was not the case for the program files being transferred to PRAM because address zero in the PRAM appears at 100000 or 180000 in the 68020 address space. Here is an example of downloading a data file to the four port, which starts at address 80000.

```
[D fft2d32.dat <CR> 80000 <CR>]
```

With a script, this would be performed by a call to the Download function as follows.

```
#define FOUR_PORT 0x80000
Download("fft2d32.dat", FOUR_PORT);
```

For uploading results, the Upload command is used. This command requires a size in longwords and produces an ASCII hexadecimal file as output. From the monitor, the command to upload the 2048 (800 hexadecimal) longwords of results of the fft2d32 program from four port would be as follows.

```
[U 80000 <CR> 800 <CR> fft2d32.out <CR>]
```

With a script, this would be performed by a call to the Upload function as follows.

```
Upload(FOUR_PORT, 2048, "fft2d32.out");
```

3.1.5.5 Setting the Zoran Registers

The Zoran internal registers can be accessed from the 68020. Each Zoran is mapped into a different set of memory locations. These are documented in the hardware memory map, but will be repeated here for convenience. Zoran 1 is at C0000, Zoran 2 is at C1000, Zoran 3 is at 140000, and Zoran 4 is at 141000. The register offsets from these starting addresses are listed in the Zoran Engineering Data Manual. These offsets must be shifted left two bits to convert them from addresses of longwords to addresses of bytes. Some of the more important resulting offsets are the stack pointer at 414, the program counter at 404, and the mode register at 408. A specific Zoran register can be accessed by adding the offset to the starting address. For example, the Zoran 2 stack pointer is at address C1414. To write the value 33 to that stack pointer from the monitor would require the following commands.

```
[W 12 <CR> W 1414 <CR> W C <CR> W 33 <CR> W 0 <CR>]
```

To perform the same operation from a script would require a call to the Poke function with appropriate parameters.

```
#define ZORAN2 0xc1000
#define SP_OFFSET 0x414
Poke(ZORAN2 + SP_OFFSET, 0x33);
```

Similar methods are used to write to the other registers. Writing to the PC causes the Zoran to begin executing at the address written. The mode register has many bits which should not be altered. The initial state is acceptable. If speed of execution is important, the number of wait states for memory access can be reduced from one to zero by writing the appropriate value. This is performed from a script as follows.

```
#define MODE_OFFSET 0x408
Poke(ZORAN2 + MODE_OFFSET, 0x70f251);
```

3.1.5.6 Accessing the Status Latch

The 68020 can modify its status latch values by writing to address 1C0000. The status latch bits are the bottom two. The 68020 can interrupt the Zorans by setting higher bits in the same location, so only the bottom two bits should be set when modifying the status latch. Commands from the monitor to set the upper status bit (status value 2) would be as follows.

```
[W 12 <CR> W 0 <CR> W 1C <CR> W 2 <CR> W 0]
```

From a script file, the same operation would be performed with a call to the Poke function.

```
#define STATUS_LATCH 0x1c0000
Poke(STATUS_LATCH, 0x2);
```

The 68020 can read back the status latch, but it will not contain the value that was written. Instead it will contain the values written by the Zorans in the bottom byte. To read it from the monitor would require the following commands.

```
[W 11 <CR> W 0 <CR> W 1C <CR> R R]
```

To read it from a script program and assign its value to a variable would require a call to the Peek function.

```
long value;
value = Peek(STATUS_LATCH);
```

All processors write to the bottom two bits of the status register. When they read from the status register, they see the values written by the other processors. **The 68020 sees the values in the order Zoran4 bits, Zoran3 bits, Zoran2 bits, Zoran1 bits, listed from most significant to least significant.** Each Zoran sees the values in an order that is symmetrical with respect to itself and the bus it is on. Most importantly, the 68020 bits are

seen at the same place by each Zoran. This allows more convenient coding for communication. The order is opposite bus high Zoran, opposite bus low Zoran, same bus other and Zoran, 68020 bits.

3.1.6 VPH Scripts

The VPH is delivered with a set of applications programs found in the appendices. Some of these programs have been collected into a type of "main" program called "scripts". A script is an organized collection of routines and subroutines that eliminate many of the keystrokes needed when a Command processor like the io monitor used by STC to demonstrate the VPH is invoked. A script assembles all of the necessary commands into a single command entry which is typically the filename of the application itself. For instance, if an FFT program were to be executed, several commands to the command processor are necessary. They are the data space setup commands, the status latch setup commands for the 68020 and the 325s, download commands and upload commands for the results. Six scripts have been provided with the VPH, including 2DFFTs for 8x8, 16x16, 32x32, a 1k FFT, real and complex convolution and correlation, and coordinate conversion routines.

3.2 CPH Functional Units

From a programmer's perspective (Figure 18), the CPH consists of two multipliers and two ALUs connected to cache and auxiliary memory via a crossbar switch. It is important to note that the crossbar switch is fully programmable in one clock cycle. Also, it is a fully parallel gateway. All selected paths are available in one clock cycle. Furthermore, the crossbar has an internal register file which is available to any other resource.

The address generation is performed by a separate board called the address generator board. Details of this board are described elsewhere. The address generator board contains a set of crossbars also. Microprogramming the CPH consists of using the 784-bit microword depicted in the appendix. All fields are simultaneously available. Hence, the CPH is a true Very Long Instruction Word machine (VLIW). Because the multipliers are faster than the memory chips, one stage of pipelining is added to all data paths and is shown in the figure. Microwords are emitted as two phases of 768/2 or 384-bits. The machine definition file in the appendix for the CPH shows which fields are active in each phase. When a field is active in both phases, the ASSIGN statement is repeated for those fields except that the physical bits differ per phase.

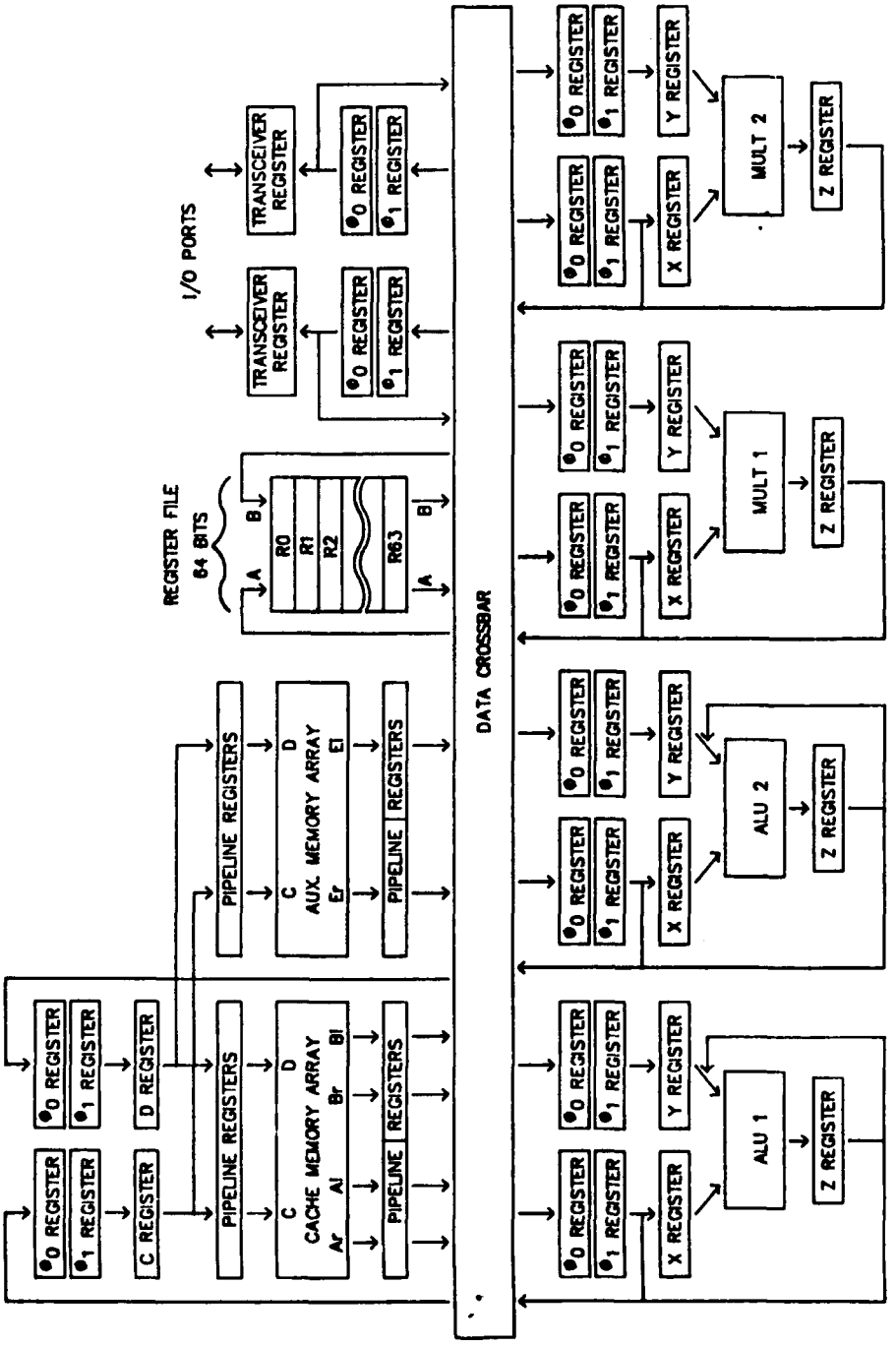


Figure 18. CPH Programmer's Model

3.2.1 Processor

The processor board is the numerical engine of the CPH architecture. Each board contains 2 BIT 2110 ALU devices and 2 BIT 2120 Multiplier devices. They are connected to other resources via nine xbar devices. Nine are used so that parity can be generated. Otherwise the 32-bit space would only require 8 xbars on the processor board. The organization is shown in Figure 18. It is useful as a programmer's model because it details the port assignments for each xbar and the microinstruction fields relevant to each port.

From this figure we see that the architecture is a two phase pipelined organization. All resources have the capability to pipe two levels of data. This was done so that the slower memory devices can conceptually keep up with the faster 2120s on the processor board. It is important to note that the ALUs do not have on-chip registers. So an external register file is provided which is embedded in the XBAR chips as a 64 word file arranged in an 8x8 array. The register file is general enough to allow FIFO, shift left and right operations to them. These are called register mode operations fully described in the xbar section of this report.

The processor board contains a writable control store for the control points on the board. Twelve microprogram memory modules are used. They are partitioned into real and imaginary fields and are signified by "MEM72" labels on the schematic. The WCS instructions are chosen so that complex arithmetic operations are facilitated by their respective real and imaginary parts. The WCS is downloaded from the IOP board. A WCS allows dynamic microprogramming so that multiple microroutines can be executed without excessive host interaction. The modules have been designed, fabricated and tested. A spare module also is being supplied. These modules are also identical to the WCS modules in the address generator board where the EVA master control store resides. The WCS essentially supports reconfigurability of the ALUS and multipliers by microprogram control. Some of the options are depicted in Figure 19. Those shown often are useful for inner and outer product operations on matrices.

The current status of the processor board design will require adding error FIFO flags (only if arithmetic status conditions are needed) and ECL clock distribution circuitry to the board. All other data and control paths have been assigned and entered into the schematic. Should a slower clock be used, ECL logic can then be replaced with CMOS clock distribution nets. The design will become much simpler in the process. Also, the high speed IO or HSIO control circuitry needs to be added to the schematic.

The original Phase I design for this board relied on the availability of end around carries being generated by the ALUs and multipliers. End around carries are necessary for two's complement arithmetic. However, when the final data specifications were completed by BIT, this signal was not provided. Hence, cascading these 32-bit chips via 32-bit boards became impossible. The current design then doubled the number of engines per board so that each board could behave as a 32-, 64- or 128-bit board under microprogram control. In this way, reasonable emulation speeds could be maintained and across-a-bus delays are eliminated.

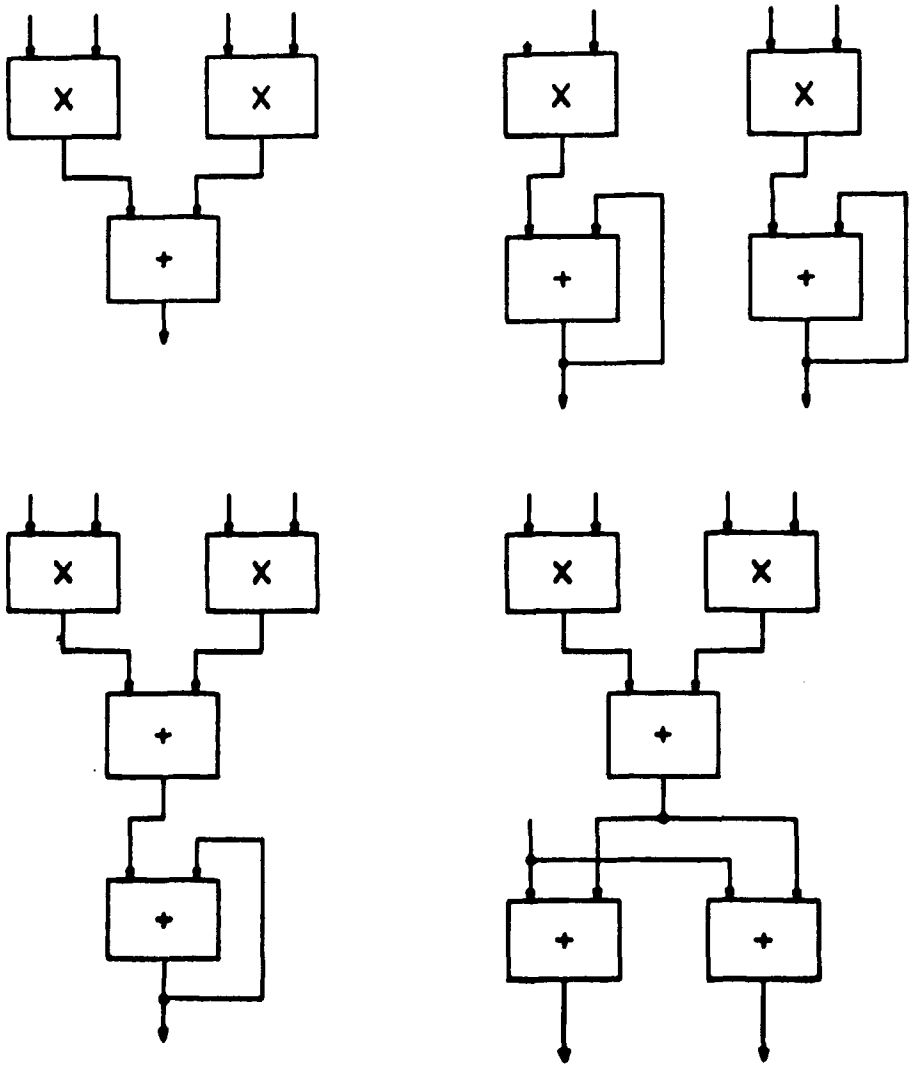


Figure 19. Dynamic ALU Configurability

The current processor board design connects the xbars to the cache memory via the CPH backplane as shown in the CPH Physical Layout in Figure 20. This figure is important when maximum execution speed is desirable in the microprograms. The slowest path will always be the one which takes the data off the board. Hence, when writing new microcode, the user should realize as shown in the figure that the cache accesses will take place across the CPH backplane. The same is true for the IO path obviously.

3.2.2 Cache Memory

The cache memory board is a versatile module for the CPH. It is designed to be cascaded so that memory space is limited only by the physical dimensions of the mainframe space. This cache can also be viewed as the main memory space of the EVA. It uses cache memory modules which have been designed, fabricated and fully tested. The board itself which houses the separate modules has not been fabricated. Each module is a SIMM or strip of discrete memory chips mounted on a small circuit board as shown in Figure 21. Fabricating the SIMMs this way allowed us to design very dense cache memory boards.

The individual memory cells of the modules uses a 3-port cell scheme as depicted in Figure 22. Here, we see that data ports A and B are output ports, while data port C is an input port. This is important to remember when microcoding the CPH because certain ports are only read and others are only write ports. The fields in the microinstruction reflect these conventions also. Note that the clock timing is a 4-phase clock with two phase 180 degrees out of phase and the other two clocks in quadrature with these two phases. A 4-phase clock scheme was chosen to maximize throughput of the modules. The cache memory bus timing also follows in Figure 23. Bus timing evaluation is necessary to complete the backplane clock distribution design.

The cache memory board is currently in design and its schematic is nearly 75% completed. Its RAM timing has been fully specified by Figure 24. Here, it is important to note that the 4-phase clock is still needed on the board itself. Also, when future microcoding starts, the code should observe the timing delays to be encountered by the clocks. For example, the last line shows that the "A DATA OUT" signal will generate the most significant data word first followed by the least significant data word. When microcoding the cache accesses, the coder should realize this multiplexing of the MS and LS words.

The cache memory board can be configured as follows:

Memory block - 16k X 36 (or 64j X 36) unit of memory. A jumper should reside on the board to set the size of each of the two blocks resident on the board. Pinouts of Cache Memory Modules are identical for both possible sizes - the only difference is that the two MSBs of the address are not used on the 16k modules.

Memory bank - a 256K deep region of memory. There may be a maximum of 16 banks each of Cache and Auxiliary memory.

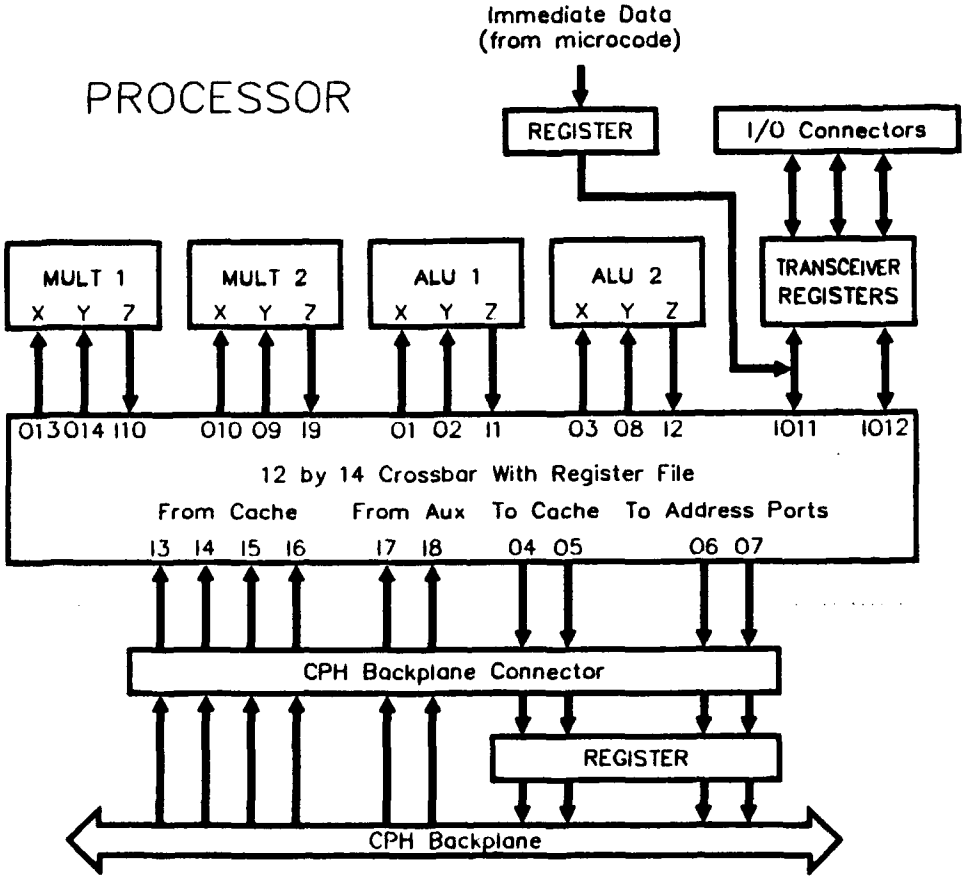


Figure 20. CPH Physical Layout

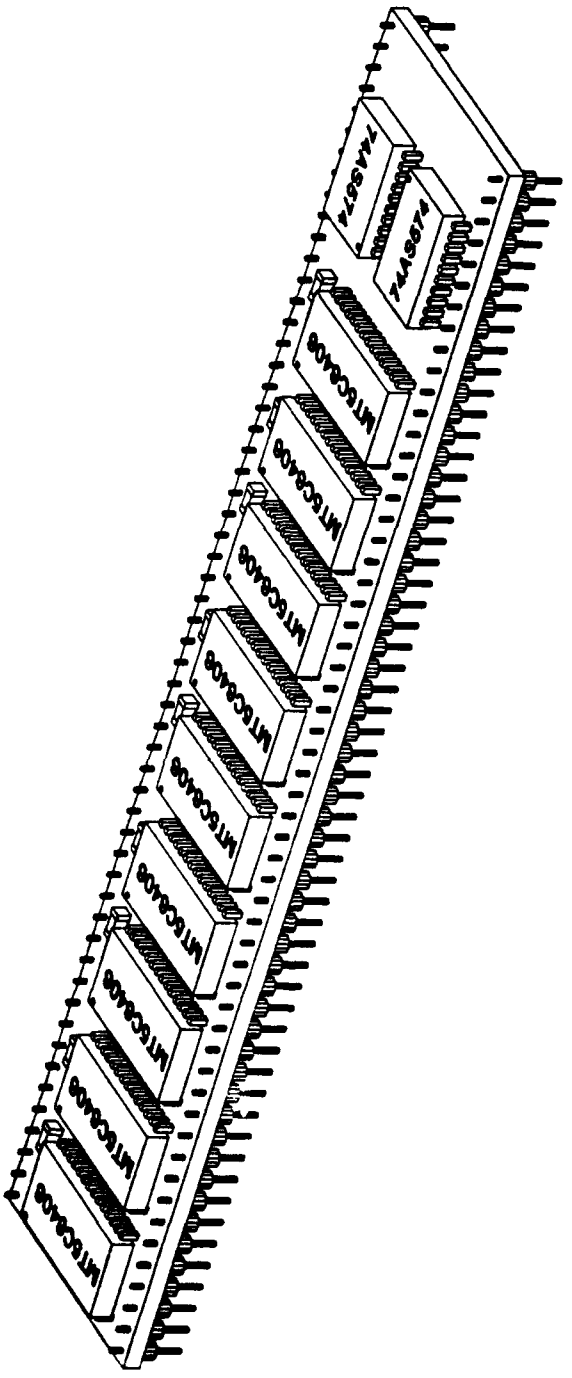
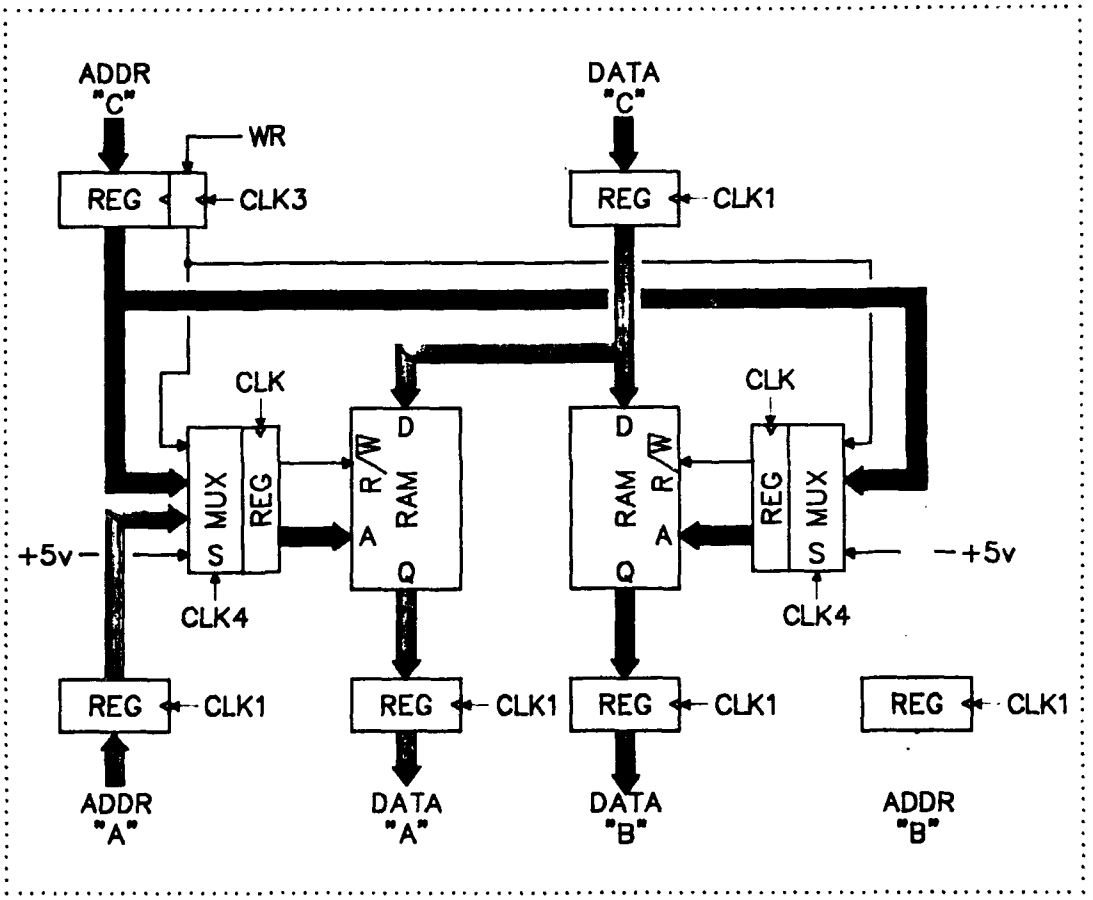


Figure 21. Cache Memory Module SIMMs



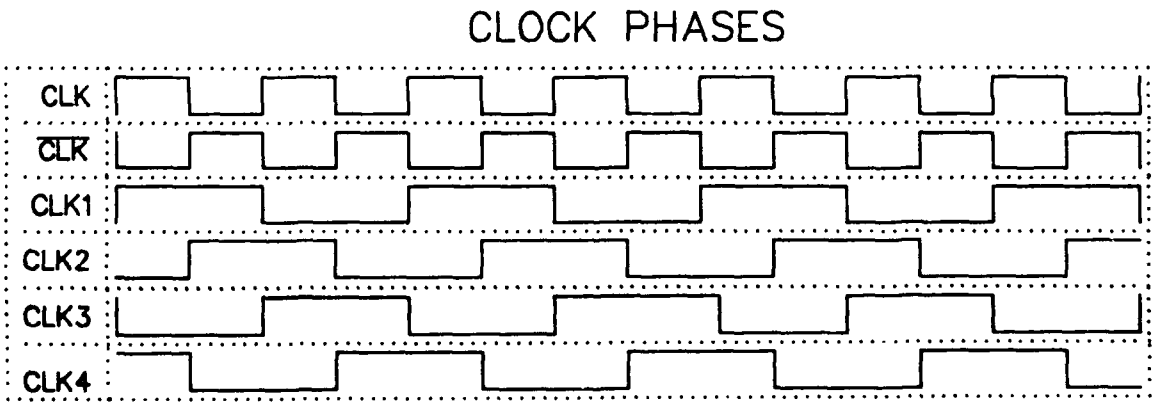


Figure 22. 3-Port Cells

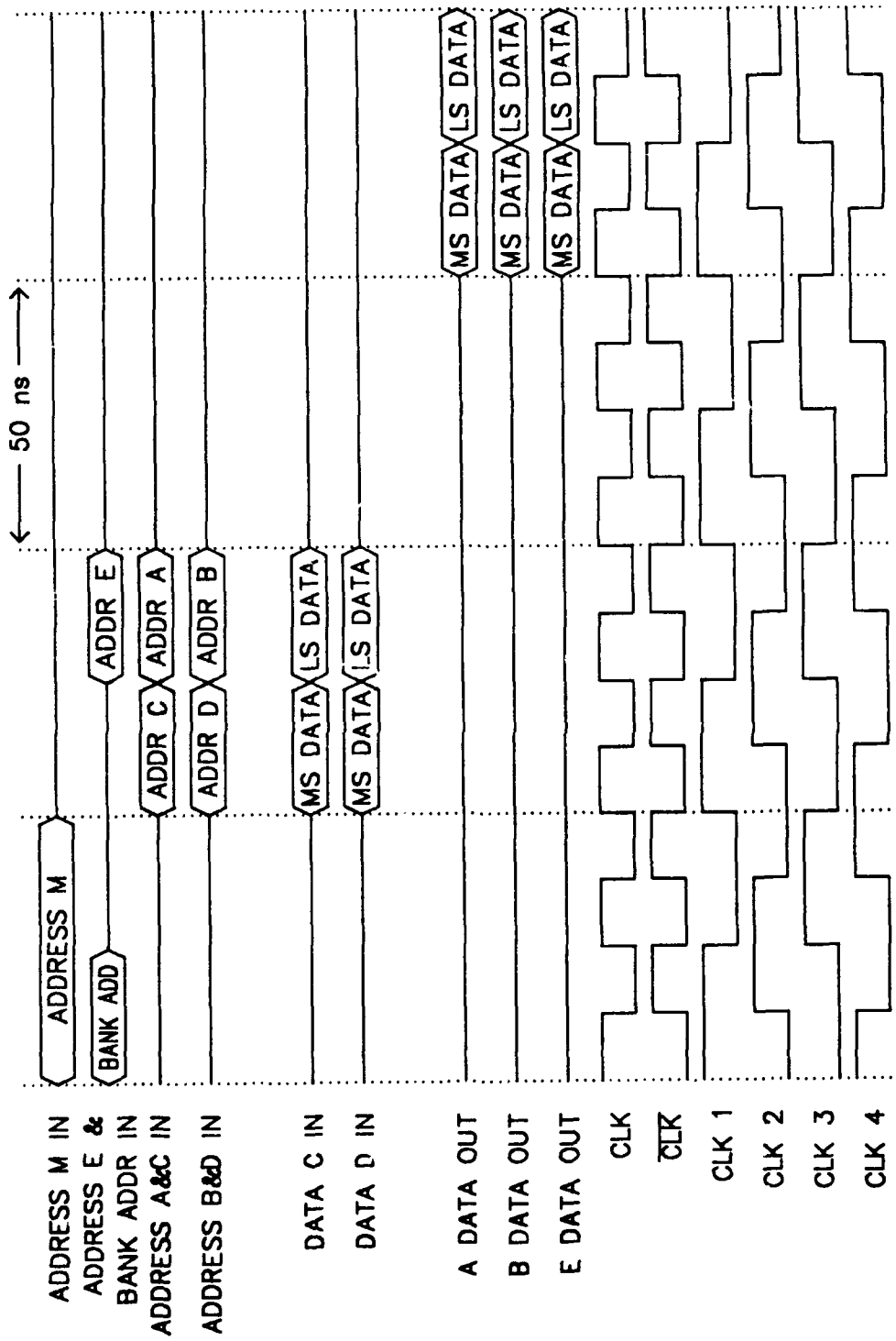


Figure 23. Cache Memory Bus Timing

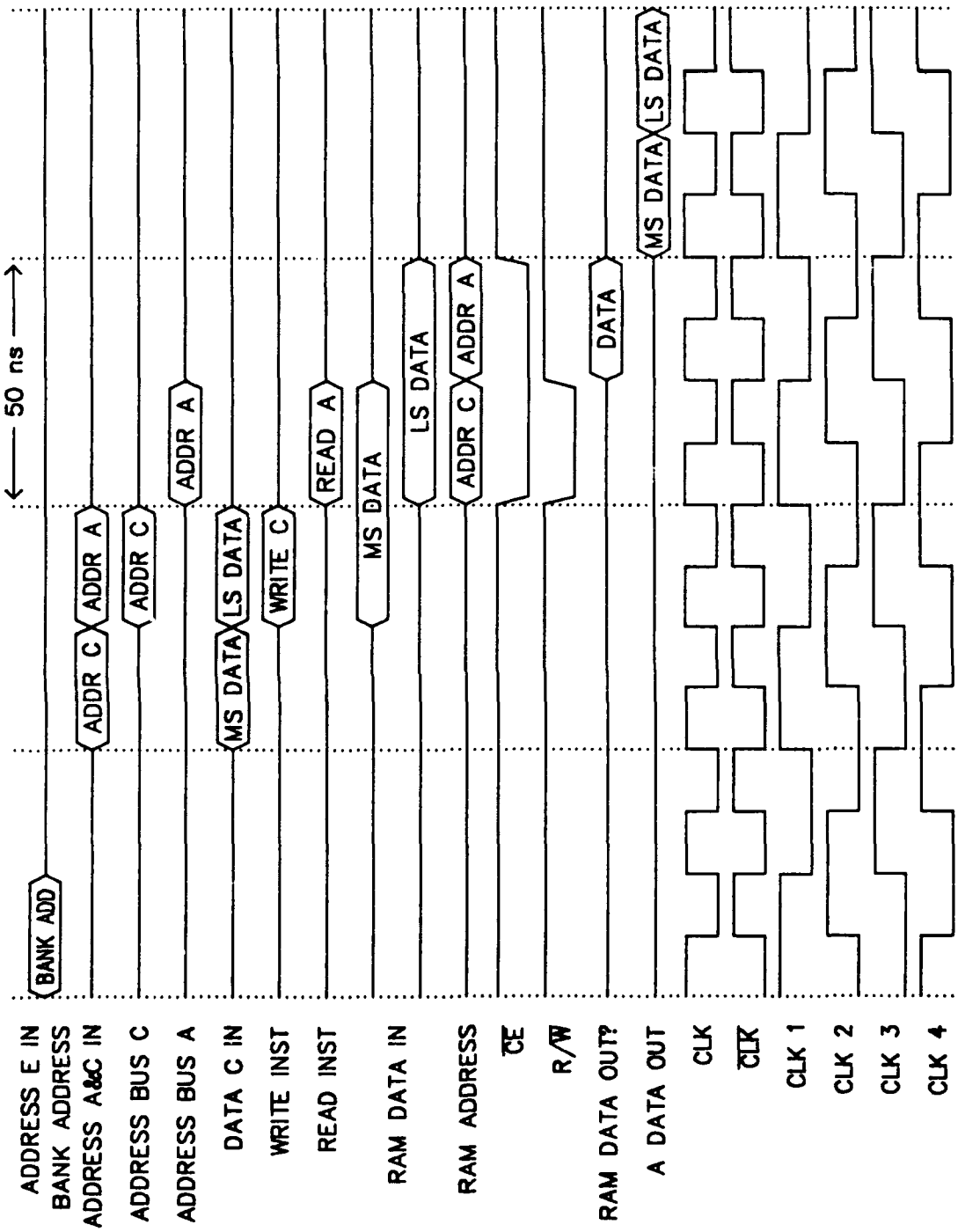


Figure 24. Cache Memory - RAM Timing

Both blocks of memory on each cache board must be configured as either cache or Auxiliary.

Address ports -

Port A - Cache Complex Read Port
Port B - Cache Complex Read Port
Port C - Cache Real Write Port
Port D - Cache Imaginary Write Port
Port E - Auxiliary Memory Port
Port F - H.S.I.O. Port

Data ports -

Port A - Cache Real Read Port A - Cache Imaginary Read
Port B - Cache Real Read Port C - Cache Imaginary Read

Port C - Cache Real Write
Port D - Cache Imaginary Write

Port E - Aux. Real Read Port E - Aux. Imaginary Read

Port F - HSIO Real (R/W) Port F - HSIO Imaginary (R/W)

Address port pairs A & C and B & D are time-multiplexed (they are physically the same backplane pins). During clock phase 0, ports C and D are active; during clock phase 1, ports A and B are active.

In addition, time multiplexing exists on the E address port. During phase 0, address port E carries bank addresses. Bits 0-3 are the cache bank address and bits 4-7 are the aux. bank address. During phase 1, address port E carries an aux. memory address.

The 8-bit configuration address, which is used to address each cache board uniquely during the system configuration process, may appear on either address port A, B, C, or D. This is your choice. The configuration address of each board is set for each board on a dipswitch.

In addition to bank address and selecting either cache or aux. memory, configuration data must include whether a block of memory is the most or least significant word. Also, the offset into the bank will be required for each block.

Separate decoding circuitry will be required for cache, Auxiliary, and HSIO addresses. Because the limitation exists that a given bank of memory may not be accessed by the processor and the IOP at the same time, if a valid cache or aux. bank address is presented to the board, the processor addresses are captured by the first level of decode circuitry, regardless of whether a valid HSIO address is present or not.

There are 4 bits of microcode resident on each board for each of the two clock phases. These bits are active /WRCAr, /WRCAi, /WRAUXr, and /WRAUXi during phase 0, and /RDA, /RDB, /RDEr, and /RDEi during phase 1.

3.2.3 Address Generator (AG)

A considerable effort was expended to enhance many of the address generator's circuits. High speed ALU and memory chips finally arrived by March 1990, but development of the required "glue logic" chips lagged behind. The address generator requires 16-bit wide counters and adders capable of a 40 MHz clock rate. These parts were unavailable in 1990. Many times, PALs could have been used to implement functions not available as standard devices. New larger and higher speed PAL type of devices have only recently been developed. Unfortunately, they are still too slow. The smaller PAL devices are capable of high speed, however, it is necessary to cascade multiple devices together. The combined delay was too great. The devices large enough to fit these functions on a single chip were too slow.

Several companies had large high speed PAL devices under development during 1990. Cypress, AMD, Plus Logic, and Altera released new devices that year. Some of these new parts are now fast enough to solve many of the speed problems. Also, Integrated Device Technology plans to make available many standard logic functions in a new high speed BiCMOS technology.

The address generator is designed to support multiple matrix addressing tasks directly in hardware. The purpose of the AG board is to reduce the overhead normally incurred by computing complex addresses in software. To keep the overhead down, 4 2-D counters are available on the board to assist memory access in a matrix. A dataword can be accessed randomly, in a row, down a column, down a diagonal, down a subdiagonal and all of the above in the opposite direction. The 2-D counter circuits are depicted in Figure 25.

The 2-D counters are designed with IDT7381L20 high speed adders. These adders were to be found in a Plus Logic 2040 FPGA but the 2040 did not become available during this Phase II effort. The IDT7217L25 multipliers are used for address offset computations executed directly in hardware. This hardware address generation method reduces the overhead of complex address generation to a minimum. Although the AMD 29540 is shown in the figure, the device has since been deleted from AMD inventory with no second sourcing. Should future availability occur, then these devices should be incorporated in the position shown in this figure. A discrete logic implementation of this device was executed. Over 40 16-pin devices are needed. Hence, the FFT hardware address generation feature of the CPH had to be deleted.

It is done by preloading the counters with the appropriate starting address and counting up or down as required. Control is accomplished with fields in the microinstruction such as 2-D counters #1, #2, #3, and #4. The microorders are fully parallel across the 4 counters. As a result, 4 concurrent addresses can be generated and sent anywhere in the CPH by virtue of the crossbar switch. The block diagram of the AG board follows in Figure 26. The AG board houses the microprogram control unit for the CPH. Here, one finds the microsequencer control for program control. Another microprogram memory resides on the processor board but this is simply writable control store. Once a program is downloaded to the processor board, execution of microinstructions on that board follows sequentially.

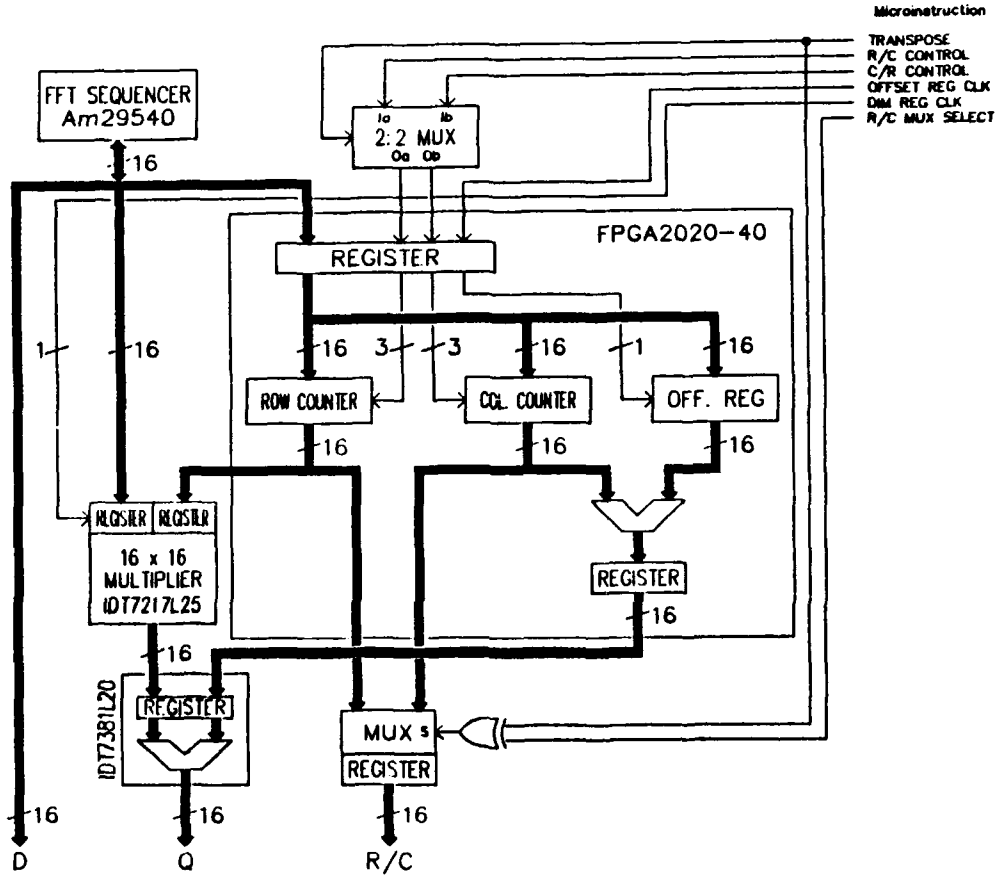
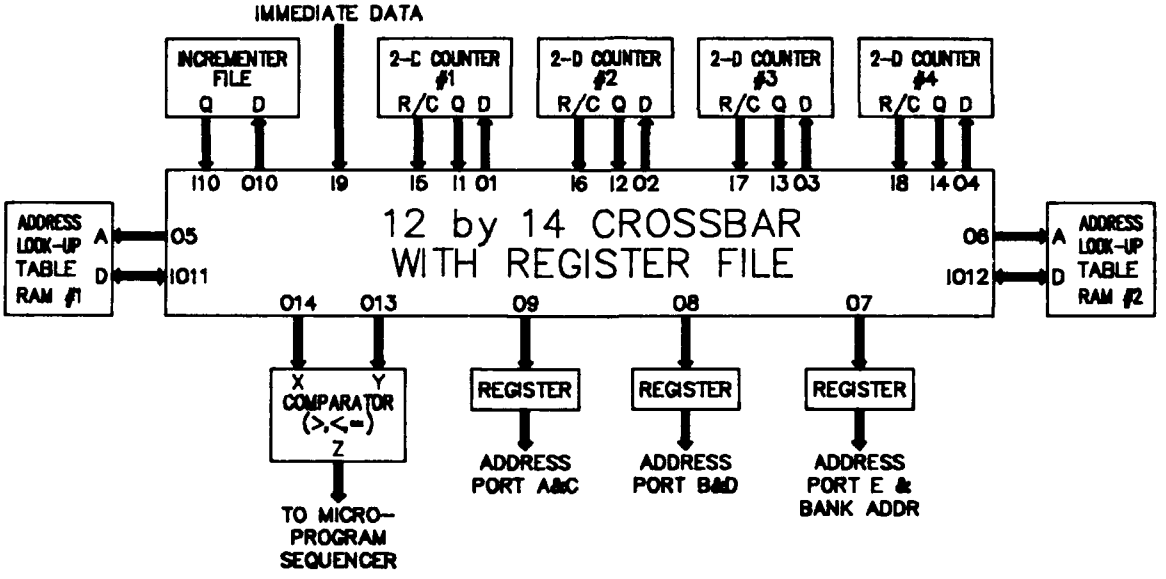


Figure 25. 2-D Counters



ADDRESS GENERATOR

Figure 26. AG Block Diagram

3.2.3.1 CPH Address Generator Board Download

Recall that the address generator board houses the central control store of the EVA. To download EVA microprograms from the I/O Processor (IOP) to the Address Generator Board (AG) the code running on the CPH system must request a program download by pulling the Download Request (DLRQST) line low on the high-speed I/O bus (HSIOB). This not only requests the IOP to download the program, it also causes the microsequencer to push the program counter onto the stack and to halt. The status of all counters, RAM (other than program RAM), and other circuits are preserved at that moment. The IOP then downloads the program to the program RAM as follows:

The IOP places the Program RAM address onto the HSIOB I/O address lines. Each board in the system decodes the address and the targeted board latches the address.

3.2.4 I/O Processor Purpose and Features

The IOP Processor (IOP) serves as the communication link between the CPH system via the High-Speed I/O (HSIO) bus, an IBM-PC via the I/O (PCIO) port, and the VME VPH processor via the Serial I/O (SIO) port. The SIO port communicates directly to a buffer/communications board residing in a VME chassis, so optionally this port can serve as the host rather than an IBM-PC if desired.

The microcontroller on-board the IOP is entirely interrupt driven. In response to an interrupt received from one of the I/O interfaces, it executes the interrupt service routine pointed to by its internal interrupt vector table. In the case of an interrupt from the host, this routine simply reads a command from the interface and executes it. This will generally be a command to transfer a block of data from/to the host. This is done by initializing one of two data transfer counters, initializing the appropriate interface control registers, and then setting the GO control bit on the "sending" interface's control register. The control logic for each interface handles the necessary handshaking to complete the data transfer, including monitoring flags and generating read and write signals, all independent of microcontroller intervention. Upon completion of the transfer, the "sending" interface generates an interrupt, and the microcontroller performs the necessary resource allocation cleanup.

3.2.4.1 IOP Control Signals

Addressing the control registers is accomplished by setting the microcode control address field to the address indicated below in each register description. Bits in the microcode data field may be either data write enable bits or data bits, as defined in each control register description. In order to modify a bit in a control register, the control bit associated with the data bit must be set LOW, the data bit(s) must be set to the desired value, and the correct address must be present. When all this occurs along with the Control Register Write (CRW) microcode bit set low, the change will occur.

RESOURCE ALLOCATION ADDRESS 0

PAL FILES: CTRL8.PDS
PAL DEVICE: PALCE26V12

This register indicates what resources are currently in use and which are available. These bits are undefined at power-up or after a reset and must therefore be initialized prior to operation. The resources are:

MICROCODE BITS control	data	
19	7	Counter A
18	6	Counter B
17	5	High-Speed I/O Interface Receive
16	4	High-Speed I/O Interface Send
15	3	Serial I/O Interface Receive
14	2	Serial I/O Interface Send
13	1	IBM-PC Interface Receive
12	0	IBM-PC Interface Send

Each software routine which uses a resource first checks its availability. Once the routine has determined that the resource is available by detecting a HIGH in the appropriate bit, it sets that bit LOW to indicate that it is in use. All interrupts must be disabled during this portion of the code. The bits are read using the microsequencer flag (condition) input.

IBM-PC INTERFACE CONTROL ADDRESS 1

PAL FILES:
PAL DEVICE:

This register contains all IBM-PC receiver interface controls, controls which are common to both the IBM-PC transmit and receiver interfaces, and controls that are initialized during reset and normally remain unchanged afterwards. Upon reset all outputs are set HIGH.

MICROCODE BITS control	data	
17	8	RECEIVE - Allows sending interface to send.
16	7	SOURCE - Selects the source interface when receiving data - LOW is SIO, HIGH is HSIO
15	6	CLRINT - Interrupts cleared when LOW
14	5	ENINT - Interrupts enabled when HIGH
13	4	ODD/EVEN - Parity ODD when LOW
13	3,2,1	CLK 2,CLK 1,CLK 0

CLK 2	CLK 1	CLK 0	
0	0	0	500 KHz
0	0	1	1 MHz
0	1	0	2 MHz
0	1	1	4 MHz
1	0	0	8 MHz
1	0	1	16 MHz
1	1	0	16 MHz
1	1	1	500 KHz

12	0	RESET - Reset the IBM-PC interface when LOW

IBM-PC INTERFACE TRANSMIT CONTROL ADDRESS 2

PAL FILES:

PAL DEVICE:

This register controls the operation of the IBM-PC transmit interface. Upon power-up reset or IBM-PC interface reset all bits are set HIGH.

MICROCODE BITS control	data	
18	7	PMRSTAT1 - STAT1 receive interrupt mask
17	6	PMRSTAT0 - STAT0 receive interrupt mask
16	5	GO - Enables sending data when LOW
15	4	PSELAB - Selects which counter is assigned to the IBM-PC interface for sending data - LOW is counter A, HIGH is counter B
14	3,2	REAL, IMAG

REAL	IMAG	
0	0	64-bit, low word first
0	1	32-bit, imaginary data
1	0	32-bit, real data
1	1	64-bit, high word first

13	1	XSTAT1 - Transmit status bit 1
12	0	XSTAT0 - Transmit status bit 0

IBM-PC INTERFACE INTERRUPT MASK ADDRESS 3

PAL FILES:

PAL DEVICE:

The interrupt is masked when the bit is set HIGH and enabled when set LOW. Upon power-up reset or IBM-PC interface reset all bits are set HIGH.

MICROCODE BITS control	data		
13	9	PREF	Receive Empty Flag
13	8	PRAEF	Receive Almost Empty Flag
13	7	PRHF	Receive Half Full Flag
13	6	PRAFF	Receive Almost Full Flag
13	5	PRFF	Receive Full Flag
12	4	PXEF	Transmit Empty Flag
12	3	PXAEF	Transmit Almost Empty Flag
12	2	PXHF	Transmit Half Full Flag
12	1	PXAFF	Transmit Almost Full Flag
12	0	PXFF	Transmit Full Flag

SERIAL I/O INTERFACE CONTROL ADDRESS 4

PAL FILES:

PAL DEVICE:

This register contains all Serial I/O receiver interface controls, controls which are common to both the Serial I/O transmit and receiver interfaces, and controls that are initialized during reset and normally remain unchanged afterwards. Upon power-up reset all outputs are set HIGH.

MICROCODE BITS control	data	
17	7	LOOPEN - Receive and transmit loopback outputs enabled when HIGH, Serial outputs when LOW
16	6	SOURCE - Selects the source interface when receiving data - LOW is IBM-PC, HIGH is HSIO
15	5	CLRINT - Interrupts cleared when LOW
14	4	ENINT - Interrupts enabled when LOW
13	3,2,1	XSEL2, XSEL1, XSEL0
12	0	RESET - Reset the interface when LOW

XSEL2	XSEL1	XSEL0	
0	0	0	HIGH
0	0	1	Receive FF
0	1	0	Receive AFF
0	1	1	Receive HFF
1	0	0	Receive AEF
1	0	1	Receive EF
1	1	0	XSTAT0
1	1	1	LOW

SERIAL I/O INTERFACE TRANSMIT CONTROL ADDRESS 5

PAL FILES:

PAL DEVICE:

This register controls the operation of the Serial I/O interface. Upon power-up reset or Serial I/O interface reset all bits are set to HIGH.

MICROCODE BITS control	data	
17	6	SXRESET - Reset the transmit interface
16	5	GO - Begins sending data when LOW
15	4	Selects which counter is assigned to the SIO interface for sending data - LOW is counter A, HIGH is counter B
14	3,2	REAL, IMAG
13	1	XSTAT1 - Transmit status bit 1
12	0	XSTAT0 - Transmit status bit 0

REAL	IMAG	
0	0	64-bit, low word first
0	1	32-bit, imaginary data
1	0	32-bit, real data
1	1	64-bit, high word first

SERIAL I/O INTERFACE TRANSMIT INTERRUPT MASK ADDRESS 6

PAL FILES:

PAL DEVICE:

The interrupt is masked when the bit is set HIGH and enabled when set LOW. Upon power-up reset or Serial I/O interface reset all bits are set HIGH.

MICROCODE BITS control	data		
13	9	PREF	Receive Empty Flag
13	8	PRAEF	Receive Almost Empty Flag
13	7	PRHF	Receive Half Full Flag
13	6	PRAFF	Receive Almost Full Flag
13	5	PRFF	Receive Full Flag
12	4	PXEF	Transmit Empty Flag
12	3	PXAEF	Transmit Almost Empty Flag
12	2	PXHF	Transmit Half Full Flag
12	1	PXAFF	Transmit Almost Full Flag
12	0	PXFF	Transmit Full Flag

HIGH-SPEED I/O INTERFACE CONTROL ADDRESS 7

PAL FILES:

PAL DEVICE:

This register controls the operation of the High-Speed I/O (HSIO) interface. After reset all bits are set to HIGH.

MICROCODE BITS control	data	
17	7	MEM - I/O HIGH, Memory LOW
16	6	WRITE - Read HIGH, Write LOW
15	4,5	SOURCE - Selects the source interface when receiving data

BIT5	BIT4	
0	0	Microprogram ROM
0	1	IBM-PC Interface
1	0	Serial I/O Interface
1	1	None

14	3	GO - Begins sending data when LOW
13	2	HSELAB - Selects which counter is assigned to the HSIO interface for sending data. LOW is counter A, HIGH is counter B
12	1	REAL
12	0	IMAG

REAL	IMAG	
0	0	64-bit, low word first
0	1	32-bit, imaginary data
1	0	32-bit, real data
1	1	64-bit, high word first

DATA TRANSFER COUNTER A ADDRESS 8

bits 19:0 Data transfer count to load

DATA TRANSFER COUNTER B ADDRESS 9

bits 19:0 Data transfer count to load

MACRO RAM ADDRESS REGISTER ADDRESS 10

bits 12:0 Directly addresses MACRO RAM

MACRO RAM ADDRESS COUNTER ADDRESS 11

bits 11:0 Parallel loads counter which directly addresses MACRO RAM

MACRO RAM COUNTER REGISTER ADDRESS 12

bits 11:0 May be used to load MACRO RAM ADDRESS COUNTER at a later time

CPH I/O ADDRESS COUNTER ADDRESS 13

bits 23:0 Addresses CPH I/O and memory space

CPH I/O SYSTEM ADDRESS REGISTER ADDRESS 14

bits 5:0 Used to generate system address when downloading microcode into CPH system(s)

IOP CONTROL REGISTER 0 ADDRESS 15

PAL FILES:
PAL DEVICE:

MICROCODE BITS					
control	data				
12	2,1,0	Interrupt Mapping Select			
		BIT2	BIT1	BIT0	
		0	0	0	Interrupt table 0
		0	0	1	Interrupt table 1
		0	1	0	Interrupt table 2
		0	1	1	Interrupt table 3
		1	0	0	Interrupt table 4
		1	0	1	Interrupt table 5
		1	1	0	Interrupt table 6
		1	1	1	Interrupt table 7

3.2.4.2 IOP Theory of Operation

SYSTEM INTERRUPT

A system interrupt indicates that one or more boards in a system requires servicing. The first step is to determine which system generated the interrupt.

The interrupt service routine must poll each board's configuration register bit 0 at the board's base I/O address + 1 to determine if that board caused the interrupt. If this bit reads 0 then that board is generating a system interrupt. At this point the action to take place is entirely under software control. The only requirement in hardware is that bit 0 of base I/O address + 1 on that board be written to with a 1 to clear the system interrupt.

IOP RESET

Upon IOP reset or power-up, if the BOOT RAM/ROM jumper is in the RAM position, a state machine presents a WCS 0000 instruction to the ADSP-1401 microsequencer. This places the microsequencer in the write control store mode and begins outputting addresses starting at 0000H counting upwards. Code is then loaded from the host (selected by a jumper) into the microsequencer microcode RAM. The entire RAM space of 0000 to 0FFF must be loaded with code or filled with IDLE instructions. Optionally ROM may be installed in place of RAM and the BOOT jumper set to ROM instead of RAM. In this case the above load is skipped.

For RAM BOOT jumper the address continues to increment now at 1000H. For the ROM BOOT jumper the microsequencer address is initialized using the WCS instruction to 1000H. At this point the microsequencer is no longer loading its own microprogram memory, but is loading the IOP macroinstruction memory. IOP macroinstruction memory must again be completely filled with code or filler. This continues until the microsequencer hits address 2000H where a microsequencer reset is generated by the hardware beginning execution of the code at microsequencer location 0000H. The code beginning at 0000H initializes the microsequencer and then jumps to the IOP macroinstruction at its program counter address 000H and continues from there.

The microsequencer reset is generated by the combination of the BOOT state machine in the BOOT state and the microsequencer address bit 13 high. When this occurs, both the microsequencer is reset and the BOOT state machine is placed in the RUN mode.

3.2.4.3 IOP Microsequencer

The IOP board has an extensive and independent microcontroller to manage the several datapaths among the various EVA functional units. The microsequencer is depicted in Figure 27 where it is shown that the PC (ISA), HSIO, and SIO (VME Buffer) are controlled by a 48-bit microinstruction as tabulated here.

Microinstruction Format

BITS	USAGE
7	microinstruction opcode
6	conditional select
11	literal data
16	data or relative jump address

A WCS is used for downloading IOP command sequences from the host computer. The A11 counter (CNTR) may be used for loops. A10 and A12 are additional address select registers for the sequencer where each may be assigned to the three external datapaths (PC,HSIO,SIO) for controlling the next sequence. The Analog Devices ADSP-1401 microsequencer chip has been selected because it supports interrupts, nested loops, and a stack. Booting up the 1401 requires us to put address 20H onto the sequencer program counter. This will always be the starting address for RESET as well.

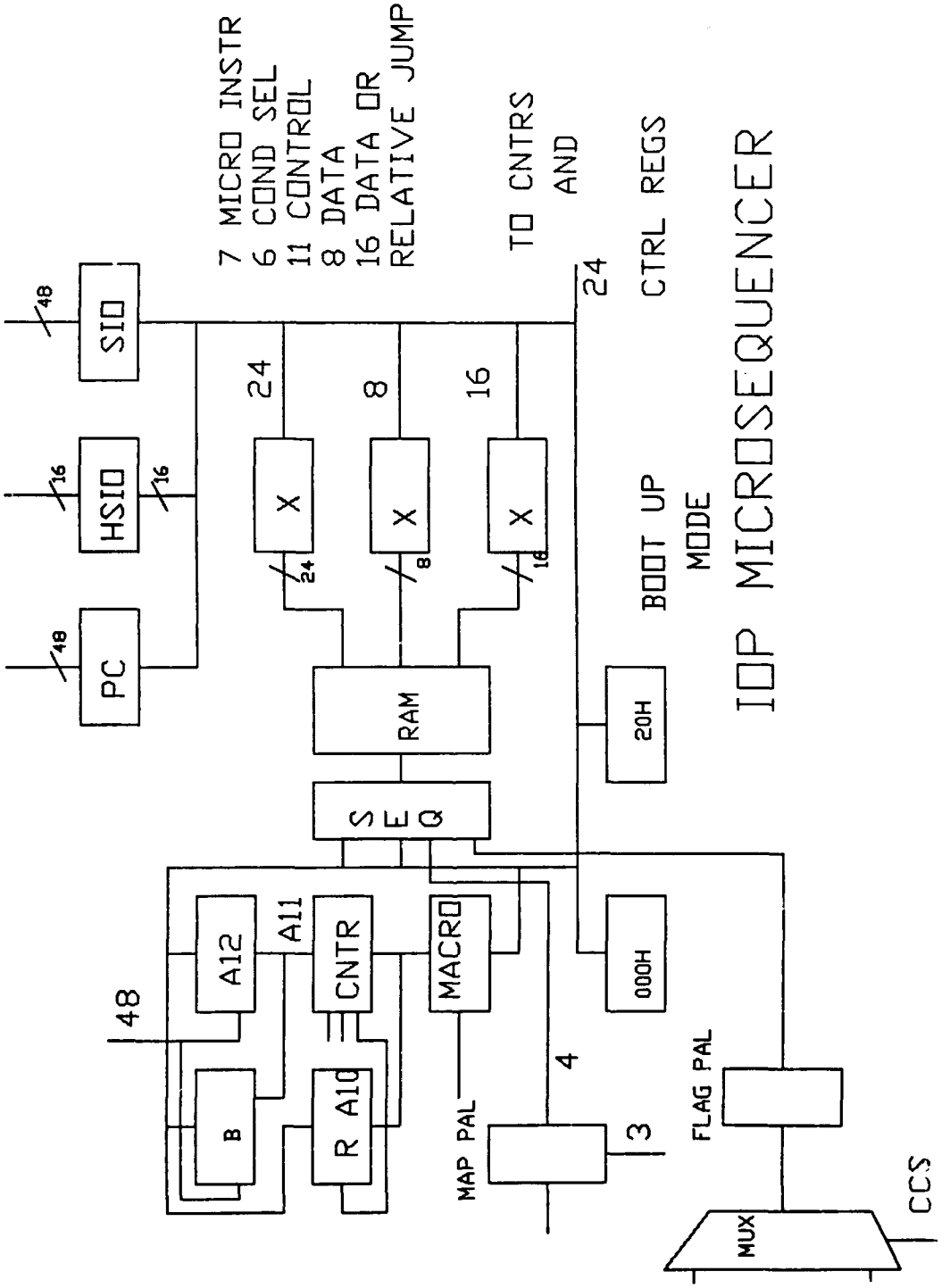


Figure 27. IOP Microsequencer

The IOP can detect the arithmetic status of the CPH ALUs. With this input via the condition code select MUX, the IOP can jump to error handling routines as needed. Both a flag PAL and a MAP PAL support future modifications to the IOP when device upgrades and subsequent address MAP changes are needed. The previous IOP sections have described the control register functions and the control signals which activate the datapaths through this IOP board. Once the IOP has served as the traffic director of the EVA, execution of code begins automatically and continues until the IOP detects a flag set on any of the EVA boards. A set flag denotes some action required of the IOP, such as "more data, computation done, or error condition".

OPERATION - BOOT

On power-up the microsequencer on the IOP board contains no instructions. The BOOT state machine controls the board at this point, enabling a path from the host interface (either the PC or SIO interface, whichever is programmed into the PAL) to the ADSP-1401 microsequencer's microprogram RAM. It also performs handshaking with the microsequencer's FLAG input and the host interface's FIFORD PAL to control the timing between the two, and loads the WCS 000H instruction into the microsequencer. The microprogram RAM is 8k 48-bit words long, and the BOOT state machine will load the first 8k 64-bit words of data appearing at the host interface into the RAM, discarding the upper 16-bits of each word. At this point, the BOOT state machine resets the microsequencer causing it to start executing code at address 000H. This boot code is required to start with a CONT instruction. The remaining boot code will load the MACRO RAM. The MACRO RAM performs the high-level instruction execution. It may be thought of as a sequence of subroutine calls to the microsequencer. The MACRO RAM is 8k 16-bits words long although only the bottom half will be used for MACRO instructions. The top 4k words will be used to store configuration data, etc. The boot code will expect the first instruction to appear at the host interface to be a LDMACRO which will contain a starting address, and the number of 16-bit data to be loaded. The upper 48-bits of each 64-bit data word from the interface will be discarded. To expedite initial CPH tests, since the configuration of the system will be known, the configuration data which would normally be read from each of the boards upon reset may be loaded from the host and programmed directly into the upper MACRO RAM. At this point all downloading has been completed, and normal operation is to begin. All interaction between the interfaces and the microsequencer are done under interrupt control. The microsequencer boot code initializes the interrupt table as follows:

IRQ8 IBM-PC Receive NEF	(HOST)
IRQ7 SYSTEM INT 0	(CPH)
IRQ6 SIO Receive NEF	(VPH)
IRQ5 IBM-PC STAT1	(HOST)
IRQ4 SIO STAT1	
IRQ3	
IRQ2 COUNTER A ZERO	
IRQ1 COUNTER B ZERO	

The boot code also reconfigures the interfaces if desired, such as increasing the clock rate from the initial low rate it defaults to on power-up.

3.2.4.4 Processor-to-I/O Processor Communication Protocol

The Processor-to-I/O Processor communication protocol is as follows. Three single-bit registers will exist for each bank of cache memory: BUSY, INT, and LOCK. The BUSY register is used by the Processor to indicate to the I/O Processor (IOP) that it is currently accessing that memory bank, the INT register will inform the IOP when the Processor is finished with that bank, and the LOCK register will prevent the Processor from accessing that bank until the IOP is finished. An example utilizing these registers is given from the viewpoint of first the Processor, and then the IOP.

PROCESSOR: The Processor examines the INT and LOCK bit and if both are inactive, sets the BUSY bit and begins processing that bank of memory. If the INT bit or the LOCK bit were active, it has to wait until both are inactive before setting the BUSY bit and processing the data. Once the Processor has completed its processing, it sets the INT bit.

IOP: The IOP examines the BUSY bit and if inactive, sets the LOCK bit active. It then reexamines the BUSY bit and if still inactive, it begins transferring the data. At completion of the data transfer, the INT bit is cleared. If when the IOP reexamines the BUSY bit, it is suddenly found to be active, the LOCK bit is immediately set to inactive assuming that the Processor has taken control of the memory bank during the time it took the IOP to set the LOCK bit. The Processor always has priority. If upon the initial examination the busy bit was active, the IOP must either use another memory bank or wait until the BUSY one generates an INT and the data is transferred out.

In addition, in order to prevent the IOP from having to read the LOCK register, OR or AND one bit, and write the LOCK register back, logic should be incorporated into the memory boards to accomplish these tasks. One method would be to have four register address bits to select which of sixteen bits will be changed, and one register control bit to indicate if the bit should be set or cleared.

The memory BUSY register and INT register must also be added to the High-Speed I/O (HSIO) bus memory address space, probably by utilizing the unused bank address 7.

3.2.5 VPH/CPH VME Buffer

The VME buffer board is the primary linkage between the CPH and the VPH. This, however, is not its only function. When operating apart from the VPH, the CPH can use the VME buffer board to connect to a 6U VME backplane. When used with the VPH, the VME buffer board plugs into the VPH backplane directly. This board also incorporates the augmented interface for the VPH so that parallel 64-bit data transfers between it and the CPH can take place. The board is completely fabricated but untested as yet. A schematic has been created for the board and is titled Serial IO board. As the board is basically a gateway for the VPH and CPH, the majority of the circuits are transceivers and PALs for controlling activity. The subsequent state machine design is basic. The major feature of this board is the Gazelle hot rod GaAs chips to maintain the 80 MHz throughput between the CPH and VPH.

3.2.5.1 Purpose

This VME buffer board floorplan shown in Figure 28 is designed to serve as a high-speed interface between the VPH Processor Board (designed for the VME bus) and the CPH's I/O Processor Board which connects to a proprietary backplane. The goal of this board is to link the two systems in an efficient manner to maximize data bandwidth and to minimize the amount of I/O necessary to control the data transfers. This board should accept data from both the VME bus (data width 4 bytes at 10 MHz) as well as the proprietary 32-bit data connector which connects directly to the VPH board. Since this extra 32-bit data connector is synchronized to the VME data transfer bus, it also transfers 4 bytes at 10 MHz for a total data transfer rate of 8 bytes at 10 MHz or 80 MBytes/sec between the VPH and Serial I/O Board. Actual performance is estimated to be approximately 67 MBytes/sec assuming an immediate response from the VPH to DTACK (Data Transfer Acknowledge). Faster rates may be obtainable by fine-tuning the Serial I/O Board's DTACK timing for both reads and writes once the boards are integrated into a system and actual timing measurements may be taken. Dipswitches have been designed in so that the DTACK timing may be adjusted individually for both reads and writes from/to the FIFOs in 10 nanosecond increments. Depending on the amount of the change, the FIFORD and/or FIFOWR PALs may also need to be reprogrammed.

At the serial interface, Gazelle HOT ROD ICs have been used which can transfer data serially at a rate of 500 Mbits/sec or 62.5 MBytes/sec. The actual serial baud rate is 625 MHz due to the 4-to-5 bit encoding scheme used. These bits are invisible due to their being inserted at the transmitter and stripped at the receiver. Data to the HOT ROD ICs is presented 40-bits at a time. 32 bits are data, 4 bits are parity, and 4 bits are control. These 40 bits are latched at a 12.5 MHz rate. Since only 32 of the bits are data, the actual data transfer rate calculates out to be 50 MBytes/sec. If this rate isn't fast enough, Gazelle also makes 800 Mbit/sec and will soon make 1000 Mbit/sec ICs which should be interchangeable with the ICs now in the design, as long as the PALs which control them are suitably fast. Faster Gazelle ICs would also mean faster FIFOs must be used. Only one speed upgrade is currently available from that which is already being used. 35 nsec FIFOs are now being used whereas 25 nsec are the fastest available at this time, and are significantly more expensive. Faster FIFOs may also bring the VME data transfer rate up to its maximum of 80 Mbytes/sec (including the proprietary 32-bit data connector). The Gazelle ICs directly drive 50-ohm coax cable for short distances. For longer distances, it is suggested that an amplifier be used for single-ended operation or that fiber-optic cable be used.

3.2.5.2 VME Buffer Board Bus Limitations

The VME buffer board uses a subset of the VME standard bus because the board functions only as a special augmented interface to the VPH. The board transfers the upper 32 data bits so that a 64-bit parallel bus couples the VPH and CPH. It has VME limitations now described.

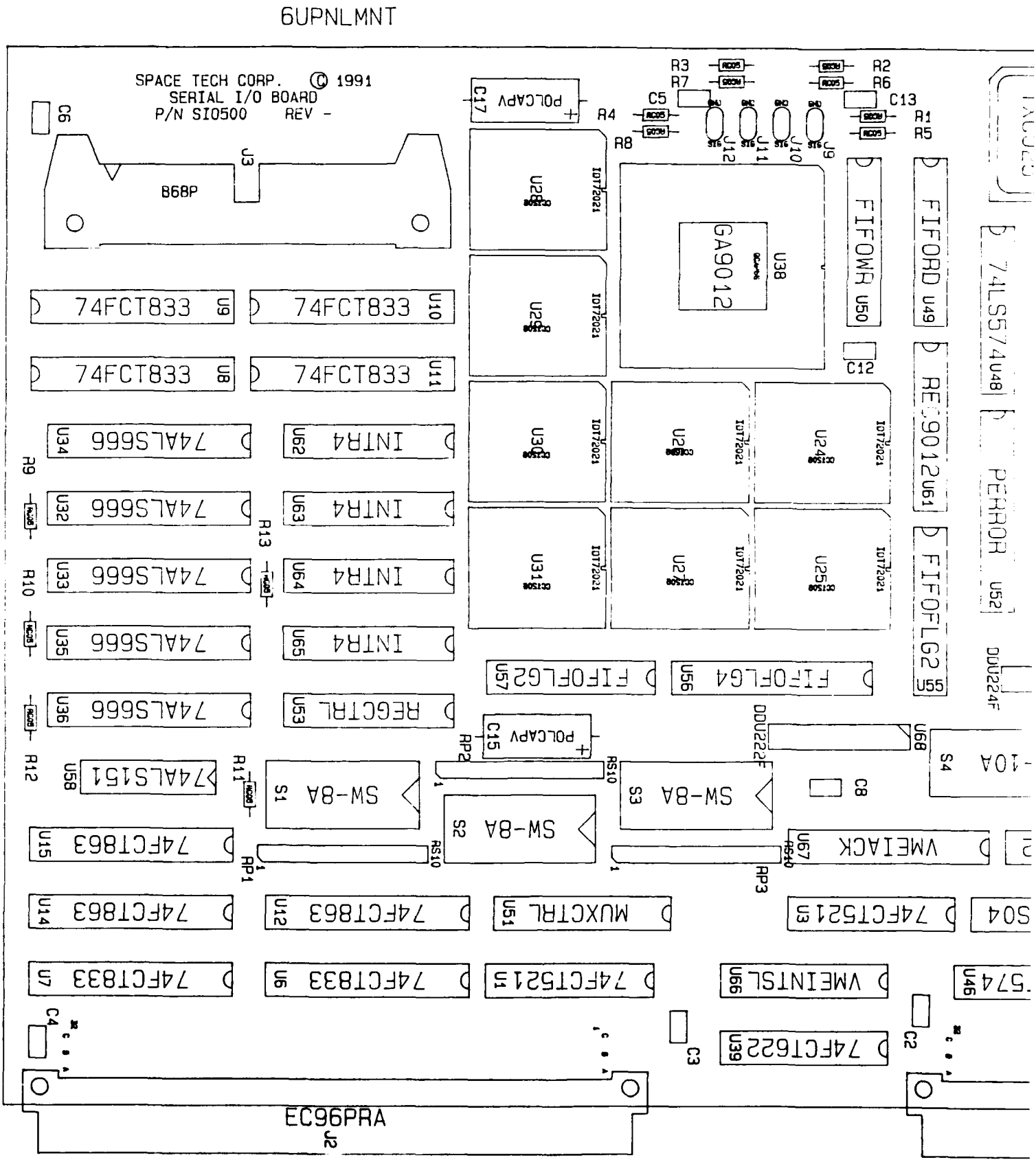


Figure 28. VME Buffer Board Floorplan

6' IPNLMNT

R2
R6
C13
R1
R5
X1
TXC025
C7
J5
J7
C10
J6
J8
POLCAPV
C14
C11
U37
GA9011
U38
GA9012
FIFOWR U50
FIFORD U49
74LS574 U48
COUNT9 U59
REC9012 U61
PERROR U52
XMIT9011 U60
FIFOFLG2 U55
U41
74LS08
U44
DDU224F
C9
POLCAPV
C16
U68
DDU222F
S4
SW-10A
C8
U47
VMEADDR
U54
FIFOFLG4
U67
VMEIACK
U42
74AS02
U43
BUFFER
U13
74FCT863
RP3
U3
74FCT521
U40
74AS04
U45
74FCT821
U5
74FCT833
U99
VMEINTSL
U46
74FCT574
U2
74FCT521
U4
74FCT833
C2
C1
U39
74FCT622
C3
EC96PRA
J1

Data Transfer Bus

- BERR* is not supported since all addresses are occupied. The only illegal board accesses are:

1. Attempt to write to the Interrupt Status Register
2. Attempt to read from the Interrupt Status ID Register

This board does not support D16:BLT nor D08(EO):BLT Double- nor Single-byte block transfers. When the board is configured without the extended 32-bits of data FIFO, it accepts all Quad-byte, Double-byte, and Single-byte data reads and writes. When configured with extended 32-bits of data FIFO, it supports proprietary Octal-byte reads and writes, although it appears to the VME bus as a Quad-byte transfer (D32:BLT). When the board is configured without the extended 32-bits of data FIFO, it accepts Quad-byte Block Transfers. When configured with extended 32-bits of data FIFO, it supports proprietary Octal-byte block transfers, although it appears to the VME bus as a Quad-byte block transfer.

This board does not support RMW (read-modify-write) simply because reading and writing is done from a separate FIFOs. When the board is configured without the extended 32-bits of data FIFO, it accepts Triple-byte reads and writes. Its Priority Interrupt Bus has the signals, I(1), I(2), I(3), I(4), I(5), I(6), I(7), and can generate an interrupt on any of the seven interrupt request lines IRQ1* through IRQ7*.

The VME signal, D08(O), drives D00-D07 in response to a valid 8-bit, 16-bit, or 32-bit interrupt Acknowledge cycle. Release On Acknowledge interrupter type (ROAK) is an interrupt request to be released upon a status ID register read.

3.2.5.3 Control Registers of the VME Buffer Board

The board contains control, interrupt status, interrupt mask, and interrupt status-ID registers. Their addresses and bit definitions are as follows:

CONTROL REGISTER

All bits are active high and are reset to zero on power-up or VME system reset.

Bit	Name	Description
0	IRESET	Reset Latched Interrupts
1	FRESET	Reset FIFOs
2	ENINT	Enable VME Interrupts (DEFAULT: Interrupts disabled).
3	INTSEL1 |	Selects which VME Interrupt Request Line is
4	INTSEL2 |	pulled low when an on-board interrupt is
5	INTSEL3 |	generated (DEFAULT: 000, no interrupt selected).
6	DT	Data Type 0 Standard 32-bit VME data (DEFAULT) 1 Extended to include 32-bit proprietary
7	SWINT	Software Interrupt
8	RLOOPEN	Enable Receiver L Input (DEFAULT: S Input)

9	XLOOPEN	Enable Transmitter L Output (DEFAULT: S Output)
10	CXSTAT0	Control Transmit Status Bit 0 (DEFAULT: LOW) NOTE: This signal is inverted prior to being transmitted.
11	XSTAT1	Transmit Status Bit 1
12	XSEL0	Transmitter Control Bit 0 Source Address
13	XSEL1	(DEFAULT: 000)
14	XSEL2	(see table below)
15		

TRANSMITTER CONTROL BIT 0 SOURCE ADDRESS

Address		Control Bit 0 Transmitted
0	LOW	Always LOW (DEFAULT)
1	RFF	Receiver Full Flag
2	RAFF	Receiver Almost-Full Flag
3	RHFF	Receiver Half-Full Flag
4	RAEF	Receiver Almost-Empty Flag
5	REF	Receiver Empty Flag
6	CXSTAT0	Control Transmit Bit 0
7	HIGH	Always HIGH

INTERRUPT STATUS REGISTER

Bit	Name	Description
0	XFF	Transmitter FIFO Full Flag
1	XAFF	Transmitter FIFO Almost-Full Flag
2	XHFF	Transmitter Half-Full Flag
3	XAEF	Transmitter Almost-Empty Flag
4	XEF	Transmitter Empty Flag
5	RFF	Receiver FIFO Full Flag
6	RAFF	Receiver FIFO Almost-Full Flag
7	RHFF	Receiver Half-Full Flag
8	RAEF	Receiver Almost-Empty Flag
9	REF	Receiver Empty Flag
10	PARITY	Parity Error
11	RSTAT0	Receiver Status Bit 0
12	RSTAT1	Receiver Status Bit 1
13	RECERR	Receiver Data Error
14	SWINT	Software Interrupt
15		

INTERRUPT MASK REGISTER

All bits are active high and are reset to one on power-up or VME system reset (all interrupts are initially masked).

Bit	Name	Description
0	XFF	Transmitter FIFO Full Flag Mask
1	XAFF	Transmitter FIFO Almost-Full Flag Mask
2	XHFF	Transmitter Half-Full Flag Mask
3	XAEF	Transmitter Almost-Empty Flag Mask
4	XEF	Transmitter Empty Flag Mask
5	RFF	Receiver FIFO Full Flag Mask
6	RAFF	Receiver FIFO Almost-Full Flag Mask
7	RHFF	Receiver Half-Full Flag Mask
8	RAEF	Receiver Almost-Empty Flag Mask
9	REF	Receiver Empty Flag Mask
10	PARITY	Parity Error Mask
11	RSTAT0	Receiver Status Bit 0 Mask
12	RSTAT1	Receiver Status Bit 1 Mask
13	RECERR	Receiver Data Error Mask
14	SWINT	Software Interrupt Mask
15		

INTERRUPT STATUS ID

This is simply an 8-bit register which is written to by a VME bus master. During an interrupt Acknowledge cycle, the contents of this register is placed onto the VME data transfer bus in response to a valid IACKIN address.

REGISTER ADDRESSES

Register	Read/Write	Address Offset
CONTROL REGISTER	R/W	100h
INTERRUPT STATUS	R	104h
INTERRUPT MASK	R/W	108h
INTERRUPT STATUS ID	W	10Ch

3.2.5.4 Address Select on the VME Buffer Board

Three 8-position dipswitches reside on the board for selecting both the FIFO address as well as the register address block. These two blocks must be contiguous with the FIFO block residing in the lowest 256-byte block and the registers in the upper. Neither the addressing for the FIFOs nor for the registers is fully decoded, leading to address foldover. The FIFO's respond to any address within their 256-byte block, and the registers each respond to sixteen different locations (they ignore the upper 4 address bits of the lowest byte).

The three dipswitches are:

S1 address bits A31 - A24

S2	address bits A23 - A16
S3	address bits A15 - A09

For each dipswitch OPEN represents a HIGH, CLOSED represents a LOW. Position 1 represents the most significant bit of that address byte, with position 8 representing the least.

3.2.5.5 VME Buffer Board Interrupts

The board may generate an interrupt to any of the following conditions:

0	XFF	Transmitter FIFO Full Flag
1	XAFF	Transmitter FIFO Almost-Full Flag
2	XHFF	Transmitter Half-Full Flag
3	XAEF	Transmitter Almost-Empty Flag
4	XEF	Transmitter Empty Flag
5	RFF	Receiver FIFO Full Flag
6	RAFF	Receiver FIFO Almost-Full Flag
7	RHFF	Receiver Half-Full Flag
8	RAEF	Receiver Almost-Empty Flag
9	REF	Receiver Empty Flag
10	PARITY	Parity Error
11	RSTAT0	Receiver Status Bit 0
12	RSTAT1	Receiver Status Bit 1
13	RECERR	Receiver Data Error
14	SWINT	Software Interrupt
15		

All of the above signals are active low. When active, a rising edge on the 25 MHz clock latches them into their respective INTR4 PALs (U62-U65), causing the INTR4 PAL to output a low on its INT output. The VMEINTSL PAL (U66), upon detecting one or more of its INTx inputs low, generates a high on the IRQy output that is addressed by the SELy inputs, and also a low on its INT output. The SELy inputs are programmable in the CONTROL REGISTER (U34) and select which VME interrupt request line is being used by the board. The CONTROL REGISTER ENINT (Enable Interrupt) bit must be set to one to enable the VME interrupt request open-collector drivers (U39).

RESPONDING TO INTERRUPT ACKNOWLEDGE DAISY-CHAIN INPUT

Upon detecting a low signal on its IACKIN input, the VMEIACK PAL (U67) sees if three conditions are met prior to responding. First, its INT input must be low indicating an on-board interrupt is pending. Secondly, the ENINT input must be high indicating that interrupts are enabled. And thirdly, the address received on the A01, A02, and A03 inputs must match those on the SEL0, SEL1, and SEL2 inputs (and must not be 0). If all of these conditions are met, then the IDEN output is set to active low, else the IACKOUT output is set to active low passing along the interrupt acknowledge to the next board in the system. If IDEN is set low, this signal is passed to the Status ID register (U36) OERB (output enable read-back) control input causing the register to output its contents onto the data bus. IDEN also connects to the MUXCTRL PAL (U51) enabling the VME bus transceivers (U4-U11).

3.2.6 PC Interface Board

The primary code development interface to EVA is via a PC interface board (PC-INT) shown in Figure 29. Space Tech's high-speed PC interface is designed for versatile interfacing to virtually any type of PC outboard hardware. The interface is symmetric; that is, the two "ends" of the interface circuitry are identical with the exception of glue logic tying the interface to the local environment. This interface is a bidirectional interface. Interconnect is done via twisted pair cable. RS-422 drivers/receivers are used to ensure noise immunity and allow high throughput; well-written drivers should allow this interface to handle data transfers at the full ISA bus data rate. An architectural/functional description of the interface as it appears to the PC/AT system follows.

The interface is accessed in the PC's I/O Address Space (as opposed to its Memory Address Space), and it occupies a 4-byte section of this space. The base address at which the interface resides is selectable via an 8-pole dipswitch on the interface board. It would be desirable that driver software can be configured to look for the interface at any address within the I/O Space dedicated to slave add-ons (the first 256 locations are dedicated to the platform itself, the next 768 locations are available for slave cards).

The interface is a 16-bit resource whose base address must be a multiple of 4. The least significant address bit will always be 0, since the board is a 16-bit device. Two addresses - the base address and the base address plus two - access different resources on the interface. These resources are:

Read FIFO
Write FIFO
Control Register
Status Register
Interrupt Mask Register
Interrupt Register

The Read and Write FIFOs are where input and output data, respectively, are queued up as they pass to and from the board. The FIFOs share an address; the cycle type (READ or WRITE) determines which FIFO is accessed. The Control register is a write-only location. Bits within this register determine the rate at which data is clocked across the interconnect, enable/disable of the FIFOs, enable/disable and set the sense of parity checking, enable/disable and clearing of interrupts, select whether an access to the FIFO/Interrupt Mask Register location is destined for the FIFOs or the Interrupt Registers, and setting the interrupt level passed on to the PC in response to a valid interrupt condition. Two additional bits are multipurpose, undedicated interface lines which travel directly across the interface without passing through the Write FIFO. (These two bits appear as two bits in the Status Register at the opposite end of the interface.)

The Status Register is a read-only location (address coincident with the Control Register) which provides access to status flags for the FIFOs. Both Read and Write FIFO flags may be observed via the Status Register. These flags are Full, Almost Full, Almost Empty, and Empty. Another bit indicates that a parity error has been detected. Two additional bits are a direct reflection of the two multipurpose bits from the Control Register at the opposite end.

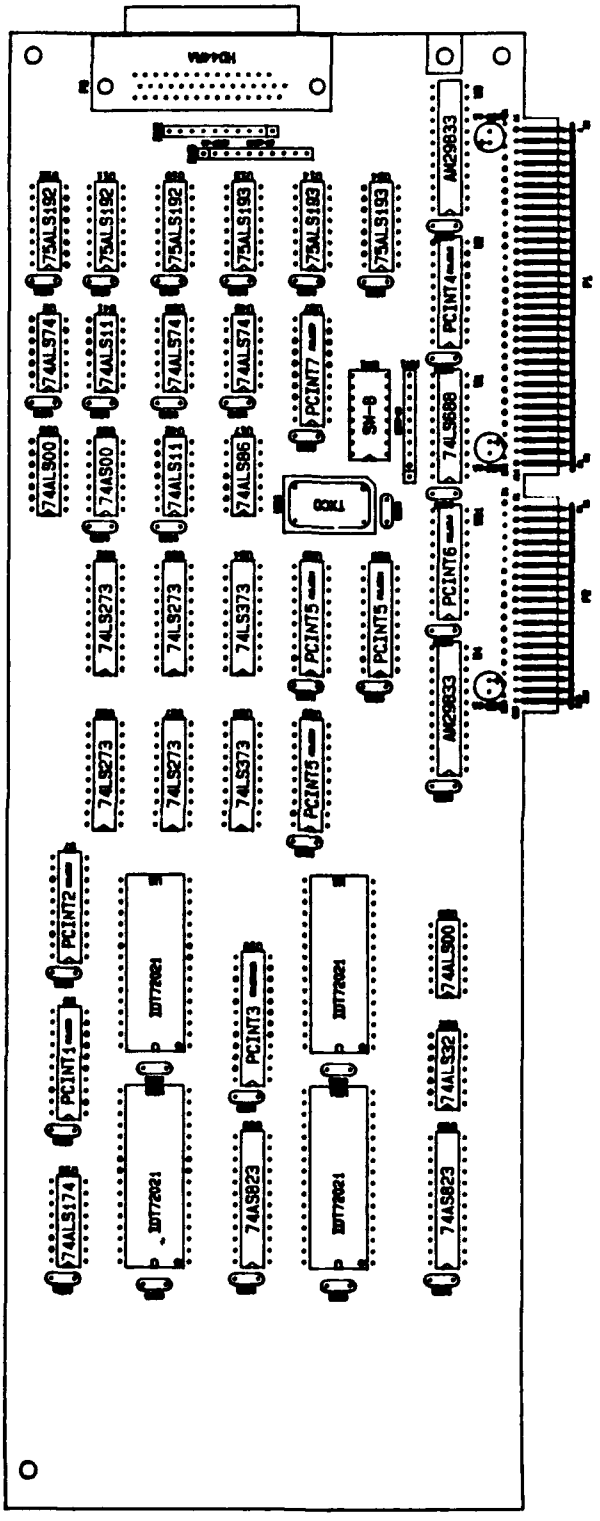


Figure 29. PC Interface Board Layout

The Interrupt Mask Register is a read/write location which provides a means of selectively generating a PC interrupt based on the conditions of the FIFO flags, the Parity bit in the Status Register, or the assertion of either of the multipurpose bits from the Status Register. A READ of the Interrupt Register provides a "snapshot" of the current interrupt conditions which have occurred since the last clearing of the Interrupt Register. (This provides a means of determining what type of service is required when more than a single condition may cause an interrupt.) The location of the Interrupt Mask Register and the Interrupt Register is coincident with the Read and Write FIFOs; a bit in the Control Register determines whether an access to this location is destined for the FIFOs or the Interrupt Registers.

A description of each of the registers and the bits they contain follows.

CONTROL REGISTER

X	X	IRQ1	IRQ2	CLRINT*	ENINT	SETMASK/SETEVEN	OD2	CLK1	CLK0	RESET	RESCOM	SETDONE	STAT1	STAT0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Bits 15 and 14 are not used, so are don't cares when writing the register.

Bits 13 and 12 determine which PC interrupt is asserted when a valid interrupt condition exists and interrupts are enabled. For:

Bit 13	Bit 12	Interrupt selected
0	0	IRQ10
0	1	IRQ11
1	0	IRQ12
1	1	IRQ15

Bit 11, when asserted, clears all interrupt flags. Also, while this bit is asserted all interrupts are disabled, so to clear interrupts but not disable them, this register must be written to twice - first with Bit 11 = 0 then with Bit 11 = 1.

Bit 10, when asserted, enables generation of interrupts. This is the intended method of enabling/disabling interrupts! If Bit 10 is negated, interrupts will not be generated, but the Interrupt Register will still be updated as valid interrupt conditions occur. If Bit 11 is asserted, interrupt

flags will NOT be updated and a valid interrupt condition will then be lost.

Bit 9 determines whether an access to the FIFO/Interrupt Mask Register address will be directed to the FIFOs or the Interrupt Registers. When the bit is asserted (=1), an access is directed to the Interrupt Registers.

Bit 8 determines the sense of parity sense. Bit 8 = 0 selects odd parity, and 1 selects even parity.

Bits 7, 6, and 5 select the clock rate used to clock data across the interface. The value of these bits determines the division applied to the local clock which runs at 16 MHz. The values and corresponding division factors are:

CLK2	CLK1	CLK0	Divisor
0	0	0	32
0	0	1	16
0	1	0	8
0	1	1	4
1	X	0	2
1	X	1	1

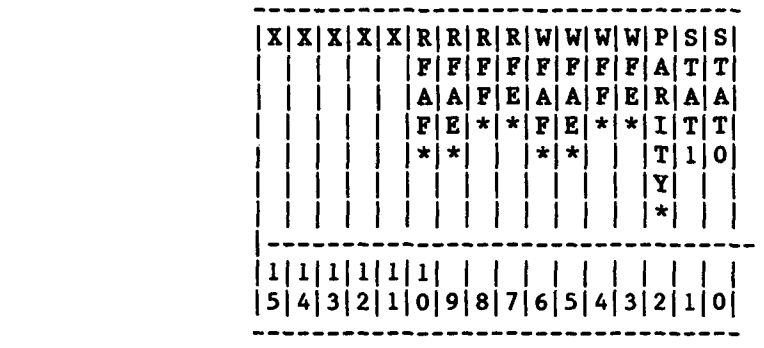
Bit 4 is the interface reset bit. A 1 written to this bit causes all FIFOs to be cleared and zeroes to be written to all bits of all registers. (This causes the bit to self clear.)

Bit 3 is the enable bit for the receive (READ) FIFO. A 0 written to this bit prevents the READ FIFO from receiving any new data across the interface, but does not prevent data already in the FIFO from being read by the PC.

Bit 2 is the enable bit for the send (WRITE) FIFO. A 0 written to this bit prevents the WRITE FIFO from sending data out across the interface, but does not prevent the PC from writing new data to the FIFO.

Bits 1 and 0 are the multipurpose interface bits. These bits propagate directly across the interface and appear as bits 1 and 0 in the Status Register at the other end of the interface. They may be used as interrupt lines, or for whatever kind of semaphores may be called for. These bits DO NOT pass through the FIFOs at either end.

STATUS REGISTER



Bits 11 - 15 are not used and should be disregarded when reading the Status Register.

Bit 10 is the Read FIFO Almost Full flag. A 0 in this bit indicates that the READ FIFO is almost full.

Bit 9 - Read FIFO Almost Empty flag.

Bit 8 - Read FIFO Full flag.

Bit 7 - Read FIFO Empty Flag.

Bit 6 - Write FIFO Almost Full flag.

Bit 5 - Write FIFO Almost Empty flag.

Bit 4 - Write FIFO Full flag.

Bit 3 - Write FIFO Empty flag.

Bit 2 is the parity error flag. A 0 in this bit indicates that a parity error has occurred.

Bits 1 and 0 are a direct reflection of the STAT1 and 0 bits from the Control Register at the opposite end.

INTERRUPT MASK REGISTER

The template for the Interrupt Mask Register is identical to the Status Register. A 1 in any bit position of the Interrupt Mask Register allows the corresponding bit in the Status Register to generate an interrupt; a 0 masks it out.

The addresses at which the various interface resources are located are shown below.

```
Base Address (SETMASK = 0) - READ or WRITE FIFO
Base Address (SETMASK = 1) - Interrupt Mask Register (Write)
                             or Interrupt Register (Read)
Base Address + 2 - Control Register (Write) or Status
                             Register (Read)
```

A better understanding of the register function can be obtained by reviewing the following pseudo-code for testing 2 PC interface boards. A simple program is suggested.

PROGRAM 1: Write 16-bit data out to one PC-INT board and receive it via another PC-INT board.

To test this program, install two PC-INT boards into the PC and connect the two board connectors together so that the output of one board is the input to the other. The procedure is to send the main memory data out one board and into the other. Set the sending board's base address to 340. Set the receiving board's base address to 360. Configure these addresses with the dipswitches on each board. Although the FIFOs are 2k words deep, only 256 words are being transferred. No check for parity errors are done. NOTE! Locations 342 and 362 are control registers when writing to them and the status register when reading from them.. Locations 340 and 360 are data registers when bit 9 in 340 and 360 are cleared. So data is then transferable via locations 340 and 360. However, when bit 9 is set to 1 in 342 and 362, then 340 and 360 are interrupt mask registers when writing to them and interrupt registers when reading from them.

The program is described in single step manner only to help you understand the procedures. An actual program would combine several of the steps into a single "load" assembly language instruction.

1. CLEAR and INIT ONBOARD REGISTERS (in 342 and 362)

```
set cr 4 to 1 in 342 and 362    /reset bit in the control
                                 registers, clears registers and
                                 FIFOs/

set cr 7,6,5 to 001 in 342 and 362 /500kps baud rate in both boards/
```

2. INIT CONTROL REGISTER base addresses 342 and 362 to talk next time to the interrupt mask register

```
set bit 9 to one in 342 and 362
                        /allows 340 and 360 to write to interrupt
                         mask reg instead of data registers/
```

3. INITIALIZE INTERRUPT MASK REGISTER

Load mask bits into 340 (note that 340 now writes to mask register instead of data register because bit 9 in the control register was just set to

one. (Later, we'll clear this bit in the control register in order to write to the data register.)

set bit 6 to one in 340
/the write FIFO will interrupt the PC when it is almost full/

4. ENABLE INTERRUPTS

set bits 13,12,11,10 in 342 and clear bit 9 in 342 so 340 is a "data" register now
/use IRQ 15 to interrupt PC when write FIFO is almost full in 342 (hence, stop transmitting)/

set bits 13,12,11,10 to 1011 in 362 and clear bit 9 in 362 so 360 is a data register
/use IRQ 12 to interrupt PC when read FIFO is almost full in 360/

5. WRITE DATA TO 340 (DATA PORT)(If FIFO is empty or almost empty, write a block <2kwords)

Move 16-bit words from main memory and write each word into address 340. Don't write more than 2k words, otherwise the FIFO will overflow in the board.

set bit 2 of 342 to 1 and bit 3 to 0
/location 340 becomes a transmitting board/

set bit 3 of 362 to 1 and bit 2 to 0
/location 360 becomes a receiving board/

set bit 9 of 362 to 1
/to be able to set interrupt mask into 360 instead of sending erroneous data out 360/

set bit 10 of 360 to 1
/enables the read FIFO almost full interrupt flag/

clear bit 9 of 342
/340 is now a data port again/

write 256 16-bit words to 340

read bit 4 in 342 and don't write til set (FIFO is not full if flag is set)

if set write next word and check bit 4 (ok to send a word)

6. READ DATA FROM 360

clear bit 9 of 362 /now 360 is a data port/
read 256 16-bit words from 360

read bit 7 of 342 before each read and if cleared then read the word

read bit 7 of 362 after each read. If set stop reading and wait til cleared.

7. IF INTERRUPTS OCCUR

If IRQ 15 occurs from the transmitting board (board sending data out of the PC), then pause writing to 340 to allow 340 to open space in its FIFO by dumping out to 360.

If IRQ 12 occurs from the receiving board (board sending data back into the PC), then stop writing to 340 because 360 is almost full and can't store any more data from the transmitting board.

3.2.6.1 VPH-End PC Interface

The PC interface at the VPH end differs slightly from the PC end interface. The architecture is essentially the same, but the interface resources are accessed a little differently than at the PC end. The resources at the VPH end are accessed at the following 68020 addresses:

Interface Base Address	-	$24 0000
Read/Write FIFOs	-	$24 0000
Status/Control Registers	-	$24 0004
Interrupt Registers	-	$24 0008

Accesses to all of these resources are longword (32-bit) accesses, although only the lowest 16 bits are utilized.

The Status, Interrupt, and Interrupt Mask Registers are identical to those at the PC end. The Control Register is slightly different due to the difference in local environments. The mapping of the Control Register is shown below.

Control Register - VPH end

STOILEV	LSEL2	LSEL1	LSEL0	CLRINT*	ENMINTI0	NMDSTOEV*	ODK2	CLK1	CLK0	CLKSET	RESRECEIVE	SENDD	STAT1	STAT0
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

STAT 0 & 1 - These are general purpose interface bits. A bit written to STAT 0 or 1 in the Control Register appears as STAT 0 or 1 in the Status Register at the other end of the interface.

SEND - This bit is an enable for the sending of data across the interface. A 0 written to this bit does not disable the ability to write to the output FIFO, but does prevent data in the output FIFO from being sent until a 1 is written to this bit.

RECEIVE - This bit is an enable for the receiving of data across the interface. A 0 written to this bit does not disable the ability to read data in the FIFO, but does prevent the FIFO from receiving additional data until a 1 is written to this bit.

RESET - A 1 written to this bit resets the entire interface. The FIFOs are cleared, zeros are written to all bits of all three registers. (This effectively clears the RESET command once it has been effected.)

CLK 0,1,2 - These bits set the rate at which output data is clocked across the interface.

ODD*/EVEN - This bit selects odd or even parity across the interface.

NMSTI0 - Setting this bit makes a high level on the incoming STAT 0 the highest priority interrupt, thus giving the PC priority over any VME interrupts. (The level of the request as passed to the 68020 is set by bit 15.)

ENINT - This is an enable for PC interrupts.

CLRINT* - A 1 written to this bit clears all PC interrupts. The bit does not self-clear, so a 0 must be written to this bit after interrupts have been cleared.

LSEL0,1,2 - These bits set the level of the interrupt passed to the 68020 in response to a PC interrupt request. (A request via the STAT 0 line has its interrupt level set by bit 15 rather than by these three bits.)

STOILEV - This bit determines the interrupt level passed to the 68020 (level 3 or 7) in response to a PC interrupt request on STAT 0.

Upon reset, the VPH PC interface wakes up with zeros in all control registers. This means that SEND and RECEIVE are disabled, the lowest data rate is selected, ODD parity is indicated, NMSTI0 on the incoming STAT0 is disabled, all interface-generated interrupts are disabled, all interrupts are cleared, the interface interrupt level is set to zero, and the STAT0 NMSTI0 interrupt level is set to 3. The Status and Interrupt Mask Registers are cleared, as are both FIFOs.

A RESET may be effected by writing a "1" to bit 4 of the Control Register.

To initialize the interface after a RESET, the required configuration must be written to the Control and Interrupt Mask Registers. The specifics of

how the interface is configured depends upon a previously agreed upon protocol or configuration. At the very least, the FIFOs must be enabled.

Following are a few guidelines for useful diagnostic code which have been written for testing the VPH end interface and can be found in the appendices.

Test Routines For PC Interface

1. Have VPH write a few words to the interface, verify that they are received by PC by reading PC end Status Register and then reading and verifying the received data.

2. Have VPH monitor the STAT0 and STAT1 lines in Status Register. The VPH should update the STAT0 and STAT1 bits in the Control Register to echo changes on incoming STAT lines. The echoed STAT values may be monitored at the PC end for verification.

3. Send several data values to the VPH. The VPH performs some simple manipulation on the data, and writes it back to the PC for verification.

Once these tests have been run, it can be assumed that basic PC interface operations are functional. More complex code may then be generated for testing the various interface generated interrupt capabilities. The PC layout of the VPH side of the PC interface is shown in Figure 30. It is a mezzanine board.

3.2.6.2 IO Command Processor

An IO command processor (also called IO Monitor) has been generated for the EVA system. The following list of "commands" should contain all necessary data. For each command, the 16-bit command word will be passed first, followed by any parameters required for that command. The order in which parameters are passed is the same as the order in which they appear in this list.

Some of the commands on this list may need to be duplicated in the user software in order to effect slightly different functionality. For instance, the "transfer to VPH memory" commands should be able to handle data which is resident in PC memory, or which is located in a disk file. The "transfer from VPH memory" would be similar.

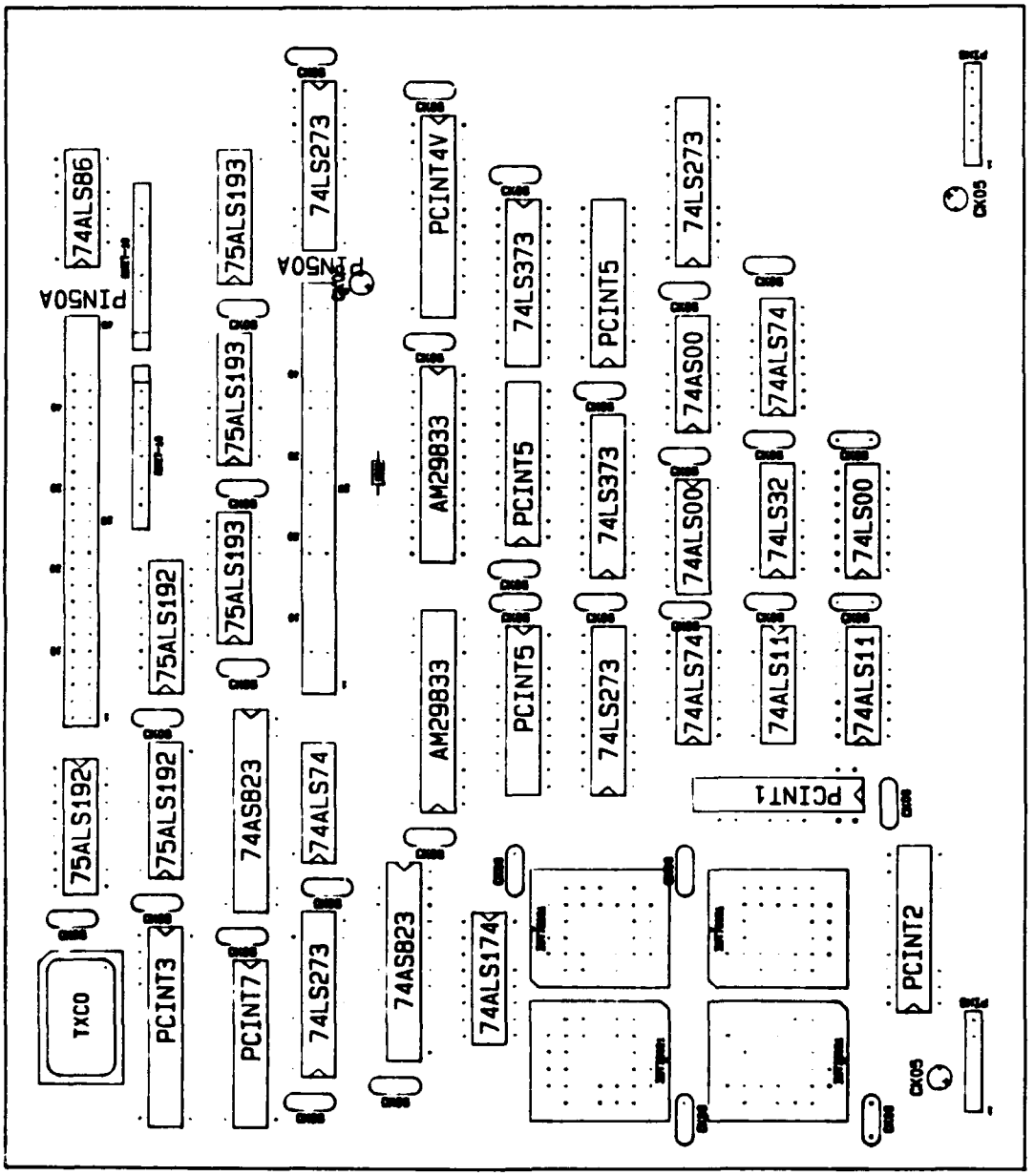


Figure 30. VPE-PC INT Layout

PC TO VPH COMMANDS

transfer to VPH memory (word writes)
command word = $0001
parameters: wordcount - 16 bit (this is the number of 16-bit words to be transferred)
VPH starting address - 32 bit
data type - lower bits of 16-bit word
$0 => 32-bit floating-point
$1 => 24-bit unsigned integer (sent as 32 bit with MSB padded with zeros)
$2 => 24-bit signed integer (sent as 32 bit with MSB padded with zeros)
$3 => 16-bit signed integer
$4 => program data (32-bit)
output: none
NOTE: data type is ignored

transfer from VPH memory (word reads)
command word = $0002
parameters: wordcount - 16 bit (this is the number of 16-bit words to be transferred)
VPH starting address - 32 bit
data type - lower bits of 16-bit word
$0 => 32-bit floating-point
$1 => 24-bit unsigned integer (sent as 32 bit with MSB padded with zeros)
$2 => 24-bit signed integer (sent as 32 bit with MSB padded with zeros)
$3 => 16-bit signed integer
$4 => program data (32-bit)
output: the number of 16-bit words requested in wordcount
NOTE: data type is ignored

request VME bus
command word = $0003
parameters: none
output: none

relinquish VME bus
command word = $0004
parameters: none
output: none

read DHB flag
command word = $0005
parameters: none
output: one 16-bit word (bit 6 is DHB bit)

read xCSR (byte read)
command word = $0006
parameters: address - 32-bit
output: one 16-bit word

```
write xCSR (byte write)
command word = $0007
      parameters: address - 32-bit
            value - 8-bit (sent as 16 bit with MSB padded with zeros)
      output: none

transfer from VPH to VME
command word = $0008
      parameters: number of words - 16-bit (this is the number of 32-bit
      words to transfer)
            VPH start address - 32-bit
            VME start address - 32-bit
      output: none

transfer from VME to VPH
command word = $0009
      parameters: number of words - 16-bit (this is the number of 32-bit
      words to transfer)
            VPH start address - 32-bit
            VME start address - 32-bit
      output: none

unused
command word = $000A

unused
command word = $000B

unused
command word = $000C

unused
command word = $000D

unused
command word = $000E

unused
command word = $000F

unused
command word = $0010

peek into VPH memory (longword read)
command word = $0011
      parameters: address to read - 32-bit
      output: one little endian 32-bit word

poke into VPH memory (longword write)
command word = $0012
      parameters: address to write - 32-bit
            value - 32-bit
      output: none
```

```
peek into 020 register
command word = $0013
      parameters: register to read - 16-bit
                                          $0 => D0
                                          $1 => D1
                                          $2 => D2
                                          $3 => D3
                                          $4 => D4
                                          $5 => D5
                                          $6 => D6
                                          $7 => D7
                                          $8 => A0
                                          $9 => A1
                                          $A => A2
                                          $B => A3
                                          $C => A4
                                          $D => A5
                                          $E => A6
                                          $F => A7
                                          $10 => PC
                                          $11 => CCR
                                          $12 => SR
                                          $13 => VBR
                                          $14 => SFC
                                          $15 => DFC
                                          $16 => CACR
                                          $17 => CAAR
                                          $18 => USP
                                          $19 => MSP
                                          $1A => ISP

      output: one little endian 32-bit word

poke into 020 register
command word = $0014
      parameters: register to write, size - 16-bit
```

	byte	word	longword
	$0000 => D0	$0100 => D0	$0200 => D0
	$0001 => D1	$0101 => D1	$0201 => D1
	$0002 => D2	$0102 => D2	$0202 => D2
NOTE: pokes	$0003 => D3	$0103 => D3	$0203 => D3
to CCR & SR	$0004 => D4	$0104 => D4	$0204 => D4
are always	$0005 => D5	$0105 => D5	$0205 => D5
word opera-	$0006 => D6	$0106 => D6	$0206 => D6
tions. Pokes	$0007 => D7	$0107 => D7	$0207 => D7
to VBR, SFC,	$0008 => A0	$0108 => A0	$0208 => A0
DFC, CACR,	$0009 => A1	$0109 => A1	$0209 => A1
CAAR, USP,	$000A => A2	$010A => A2	$020A => A2
MSP, and ISP	$000B => A3	$010B => A3	$020B => A3
are always	$000C => A4	$010C => A4	$020C => A4
longword op-	$000D => A5	$010D => A5	$020D => A5
erations. The	$000E => A6	$010E => A6	$020E => A6
VPH command	$000F => A7	$010F => A7	$020F => A7
processor will	$0010 => PC	$0110 => PC	$0210 => PC
accept any	$0011 => CCR	$0111 => CCR	$0211 => CCR
size for these	$0012 => SR	$0112 => SR	$0212 => SR
registers, but	$0013 => VBR	$0113 => VBR	$0213 => VBR
will always	$0014 => SFC	$0114 => SFC	$0214 => SFC
utilize the	$0015 => DFC	$0115 => DFC	$0215 => DFC
correct sizing	$0016 => CACR	$0116 => CACR	$0216 => CACR
when carrying	$0017 => CAAR	$0117 => CAAR	$0217 => CAAR
out the poke.	$0018 => USP	$0118 => USP	$0018 => USP
	$0118 => USP	$0218 => USP	$0218 => USP
	$0019 => MSP	$0119 => MSP	$0219 => MSP
	$001A => ISP	$011A => ISP	$021A => ISP

value - 32-bit (only the lower byte or word are used for byte or word writes)

output: none

reset VPH
command word = $0015
parameters: none
output: none

reset PC interface
command word = $0016
parameters: none
output: none

initialize PC interface
command word = $0017
parameters: control register value - 16-bit
output: none

set PC interface interrupt mask
command word = $0018
parameters: mask value - 16-bit
output: none

```
read PC interface status register
command word = $0019
      parameters: none
      output: one 16-bit word

read PC interface interrupt register
command word = $001A
      parameters: none
      output: one 16-bit word

read VPH status latch
command word = $001B
      parameters: none
      output: one 16-bit word

write VPH status latch
command word = $001C
      parameters: status latch value - 16-bit  bits 0,1 are status bits
      bits 4,5,6,7 are Zoran 1,2,3,4 interrupt flags
      all other bits are don't cares
      output: none

write Zoran reset latch
command word = $001D
      parameters: reset latch value - 16-bit  bits 0,1,2,3 are reset
      flags for Zoran 1,2,3,4
      output: none

load DSACK SRAM
command word = $001E
      parameters: address - 32-bit  the vector A[31,24..18] addresses
      the SRAM; all other bits are don't cares value - 16-bit the lowest
      nibble goes into SRAM; all other bits are don't cares
      output: none

execute starting at address
command word = $001F
      parameters: start address - 32-bit  (enter LSW first)
      output: none

transfer PC interface to VPH memory (longword writes)
command word = $0020
      parameters: longword count - 16-bit  the number of 32-bit words
      to transfer
                  start address - 32-bit  the starting
                  address in VPH (entered LSW first)
                  data type - 16-bit  (ignored)
      output: none

transfer VPH memory to PC interface (longword reads)
command word = $0021
      parameters: longword count - 16-bit  the number of 32-bit words
      to transfer
```

start address - 32-bit the starting
address in VPH (entered LSW first)
data type - 16-bit (ignored)
output: the number of longwords requested in longword count

3.2.7 HSIO Configuration

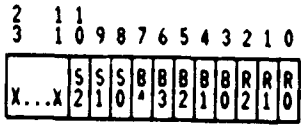
Each board within a CPH system has a small array of registers whose purpose is to allow downloading of configuration data and to provide a mechanism for the communication of control information. Some of these registers are not registers in the true sense of the word, but provide various functionality to provide the required range of special communication tasks required. A description of these registers as they must appear, for example, on the cache memory boards follows. The HSIO is the information highway for this communication.

Across the HSIO bus are also control and status information about the configuration of the current CPH system. This status information consists of the number of cache memory banks, number of CPH processor boards installed, and other such information. That status will be contained in the CPH processor status word which will operate as shown in Figure 31.

HSIO LINEAR ADDRESS SPACE/IO SPACE

The HSIO bus can access a 24-bit address space. This "linear address space" will be used to access resources in all of the CPH systems the IOP serves. In order to be able to access configuration information on any board in any system, an additional address space, referred to as the "IO Space," has been added. The IO Space will simplify system mapping and access to configuration/communication registers. A control bit on the HSIO bus will indicate when an IO Space access is to occur, as opposed to an access to the Linear Address Space. This line will be an active low line which when asserted dictates an access to the IO Space. This line is named the /HSIOMEM line.

When /HSIOMEM is asserted, the address put on the bus will have the following format:



Bits 11 through 23 are don't care.

Bits 8, 9, & 10 (S[2:0]) are the System Address bits. These bits select one of eight possible systems.

Bits 3 through 7 (B[4:0]) are the Board Address bits. These bits select one of thirty-two possible boards within a system.

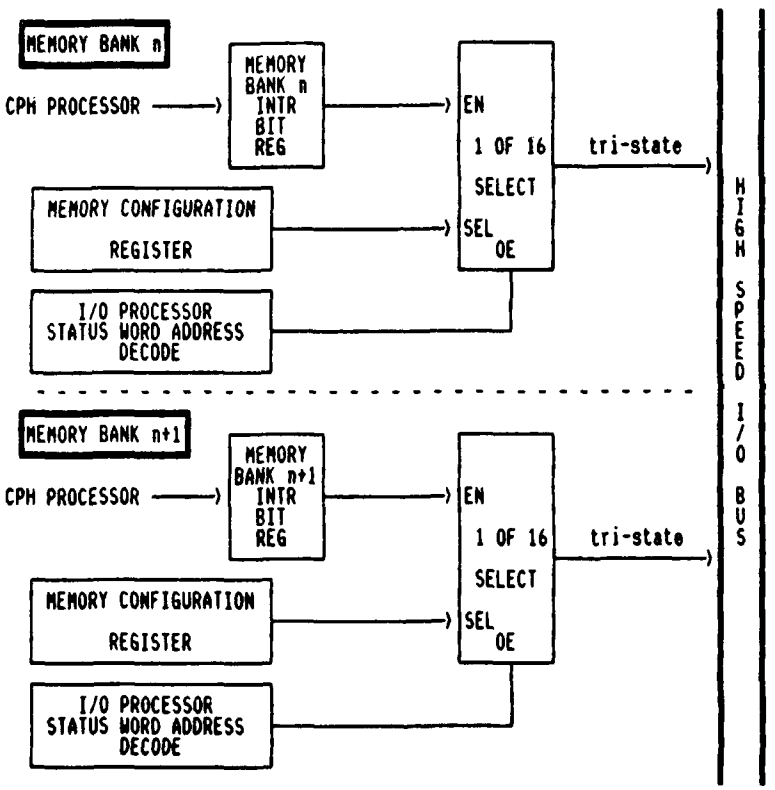


Figure 31. CPH Status Word

Bits 0, 1, & 2 (R[2:0]) are the Register Address bits. These bits select one of eight possible registers on a given board.

Each board will need a system of switches and/or jumpers to set the system and board addresses for that particular board.

On the backplane, a bit similar to /HSIOMEM exists. This is the /CONFIG microprogram bit which when asserted indicates that the address on Port A/C is destined for the configuration registers rather than the general address space of the CPH system. Data to be written to the configuration registers will be written in Port C and data read from the registers will appear on Port A. The /WRCAr and /RDA microprogram bits will be used to determine a processor configuration write and read, respectively.

REGISTER DESCRIPTION

Each of the registers within the IO Space on a particular board is a 16-bit register. Since all data paths are 32-bit paths, the convention will be adopted of using the least significant 16 bits of a given path when accessing an IO Space register. In addition, in the case of a complex (64-bit real/imaginary) path, the real portion of the path will be utilized.

The upper two registers are 16-bit mailbox registers which are accessible from the HSIO bus and the backplane. The register located at the board base address + 4 is accessible from the HSIO Bus only. Register base address + 5 is accessible from the backplane only. Each of these registers is read/write from its respective buses.

The register at the board base address is a read-only location which contains ID information for that board. This register is accessible from either the HSIO or the backplane. The format of the register is:

Bits 0:3 - a 4-bit board ID code.

Bits 4:7 - a 4-bit memory size code.

Bits 8:11 - a 4-bit block size code.

Bits 12:15 - a 4-bit read latency time code.

These bits may be hard-wired. However, in view of the fact that the codes have not yet been defined, and to allow for future re-definition, these 16 bits will be set with jumpers.

The register located at the base address + 1 is important. This register is a compound, special-purpose read/write register. Eight bits are semaphore bits, and eight bits are a "mailbox" register for passing control information between the HSIO and the backplane. A description of how the semaphores and mailbox must work follows.

Bit 0 is a system interrupt bit. This bit must therefore be passed through an inverting high-drive open-collector driver to the appropriate System Interrupt line on the HSIO. Again, jumpers will be used for routing this bit to the appropriate System Interrupt line.

Bit 1 is defined as "/VALID." This bit is active low to indicate if P/H (Bit 2) is valid. This bit is read/write from both the HSIO and backplane.

Bits 2:4 of this register are for semaphores which are set by the backplane and cleared by the HSIO. When a write to this register from the backplane occurs, a zero in any bit position causes the corresponding bit in the register to remain unchanged; a one in any bit position causes the corresponding bit in the register to be set (to one). When a write from the HSIO occurs, a zero in any bit position causes the corresponding bit in the register to remain unchanged; a one in any position causes the corresponding bit in the register to be cleared (set to zero). A read from either bus simply returns the state of the three bits. Bit 2 is defined as "P/H" and indicates control of the cache board. If Bit 2 is low, the HSIO has control of the board, but if the bit is high, the processor has control of the board. Bit 1 is used to determine if the state of this bit is valid. Bits 3:4 are undefined, general purpose semaphores.

Bits 5:7 of this register behave just as Bits 2:4, except that they set from the HSIO and clear from the backplane. All three of these bits are undefined, general purpose semaphores.

Bits 8:15 of this register are to form a mailbox between the HSIO and the backplane. That is, these eight bits are read/write from either bus. When a read occurs, the bits retrieved reflect the most recent write from the other bus. A write from one bus will not overwrite the most recent write from the other bus. This behavior is achieved with two 8-bit registers in parallel being oriented in opposite directions. An HSIO read or backplane write accesses one register, an HSIO write or backplane read accesses the other.

An interesting aspect of these registers' behavior is that access from the backplane to any of these registers is achieved by qualification of a bank address placed on the backplane with the /CONFIG bit asserted. When a valid bank address is presented during a READ cycle, only the least significant board at offset zero responds to the read request. During a WRITE, however, the data presented is written to ALL boards within that bank. The reason for this is that the processor views memory as banks with a maximum depth of 256k - it has no concern that there may be multiple boards within a bank. The IOP, on the other hand, has no conception of "banks" of memory - each board is a separate entity, regardless of what bank it belongs to, or whether it is configured as cache or Auxiliary. This means that any "message" to be passed from the IOP to the processor must be written to the correct board (least significant, offset zero). It will therefore be up to the programmer to keep track of such details.

The registers located at the base address + 2 and + 3 are configuration registers. These registers are loaded via the HSIO bus with information which assigns each of the blocks on the cache board a cache and/or Auxiliary memory bank address and offset into the block. Another bit per block assigns most or least significant status, and another bit selects the board as cache or Auxiliary memory. Bits are assigned as follows:

Bits 0:3 - Bank Address

Bits 4:7 - Offset Block 0

Bits 8:11 - Offset Block 1

Bit 12 - MSB/LSB Block 0

Bit 13 - MSB/LSB Block 1

Bit 14 - Aux/Cache

Bit 15 - Undefined

3.2.8 Crossbar

In order to minimize chip count and processor board space, a crossbar chip study was started. In December of 1989, AMCC formally quoted to STC their development costs for the ASIC crossbar design. A design quote by customer through netlist was $85,000 with 14 weeks schedule. A design quote by customer (STC) at AMCC was $95,000 with 14 weeks schedule. A custom 4:1 Mux with input enable was quoted at $10,000 with 4 weeks delivery. Production prices for up to 25 prototypes was $750 per piece and $504 in quantities of 100-499. They specified an 80 MHz clock in a 301 PGA configuration using BiCMOS. Space Tech then sought out ILSI more aggressively for their more economical ASIC design.

The new chip in cooperation with ILSI was developed as an innovative crossbar switch at an NRE cost of $35,000 that is particularly well-suited for high-speed, multiprocessor, microprogrammable, pipelined environments. It is now described.

This crossbar differs from others currently available in that it is both high speed (40 MHz) and has a large number of ports (12 by 14), all control lines are separately accessible, and it has an internal multiported, configurable register file.

The XB1210-40C crossbar switch is an ASIC fabricated with 1-micron CMOS technology. All pins use standard TTL levels. The device is packaged in a 256-pin PGA and supports Control Clock rates up to 40 MHz. It supports two-phase operation by means of two independent data clocks which are used to clock the output port pipeline registers.

This crossbar has 10 dedicated input ports, 12 dedicated output ports and 2 bidirectional ports. Each output port can access data from any input port. All ports are 4-bits wide externally and all internal data paths are 8-bits wide. Input ports have a 4-bit demultiplexing latch and output ports have a multiplexor to choose least significant or most significant bits from the pipeline. This device is particularly well suited to architectures employing the BIT Multiplier/ALU chipset, where 8 crossbar chips may be paralleled to achieve a crossbar system that is 32 bits wide externally and 64 bits wide internally.

All output ports are pipelined with a pair of parallel registers - one for the first phase and another for the second phase. A control line is provided for each output port to select data from either register. These

pipeline registers are clocked with two clocks - First Phase Clock and Second Phase Clock. The Second Phase clock may be tied low for single phase operation. All control lines are selectively pipelined and may be clocked using the Control Clock which is also used to clock the register file.

Since all control lines may be accessed simultaneously, the entire crossbar may be reconfigured every clock cycle as opposed to requiring many cycles to set up paths as in crossbars where the control signals are bused together.

The most unique feature of the XB1210-40C is an internal multiported, configurable register file. This register file is a four port synchronous static RAM organized as 64 words by 8 bits. It can also be used asynchronously by tying the Control Clock low. Each port has its own address and all ports may be used simultaneously. Each register file port may be accessed by any of the crossbar input ports. The register file may be configured in different ways - as normal static RAM, as 8 pipeline registers 8 deep, 4 pipeline registers 16 deep, 2 pipeline registers 32 deep or as a circular buffer. Figures 32 to 35 depict shift mode 1, 2, and 3, and XBAR to GPR data paths. These operating modes, non-pipelined synchronous and asynchronous, and pipelined synchronous are described later.

The crossbar consists of four major components - input ports, output ports, multiplexers, and a four port register file. All internal data paths are 8 bits wide while all I/O ports are 4 bits wide. Demultiplexing latches are provided on all input ports and multiplexers are used on all output ports. This architecture provides high speed and compatibility with various processors.

INPUT PORTS

The crossbar has ten dedicated input ports (I1_[0..3] to I10_[0..3]) and two bidirectional ports (IO11_[0..3] and IO12_[0..3]). Each input port has a 4-bit demultiplexing latch and an MSWEN control input associated with it. The most significant 4 bits of data are presented to the input port while MSWEN is brought high. MSWEN should then be brought low. Finally, the least significant four bits should be presented to the input port and held. This provides the 8-bit word presented to the internal bus.

MULTIPLEXERS

After passing through the input ports, data is passed onto an internal bus. This bus is 112 bits wide - 8 bits for each input port and 8 bits for each of two register file read ports. Any 8-bit path of this bus may be selected by the multiplexers as the data source for the fourteen output ports or two register file write ports.

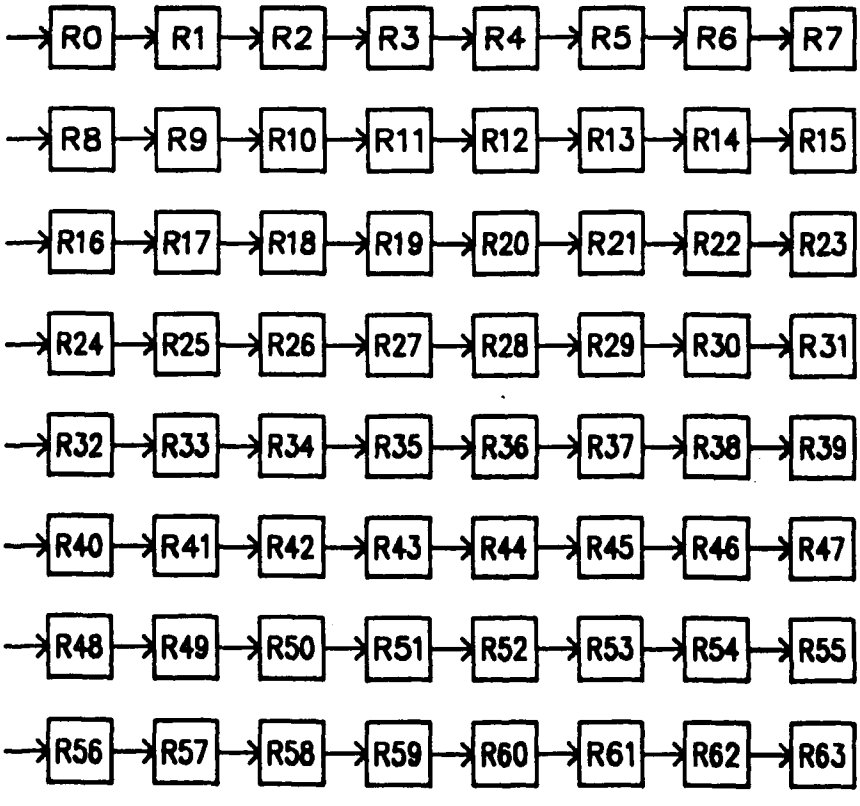


Figure 32. GPR Shift Sequence Mode 1

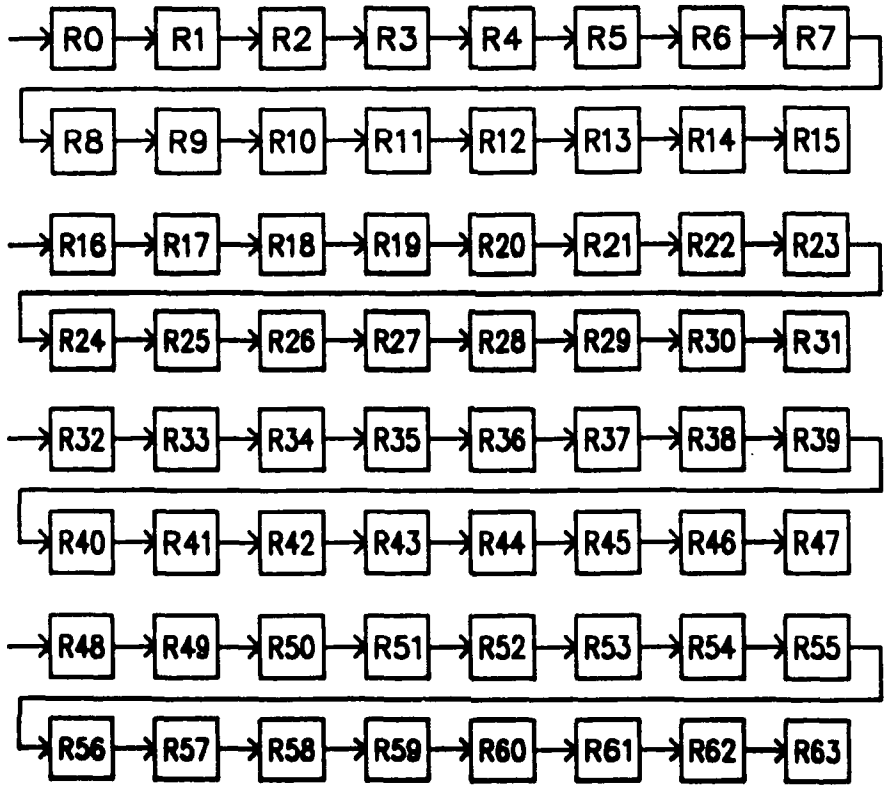


Figure 33. GPR Shift Sequence Mode 2

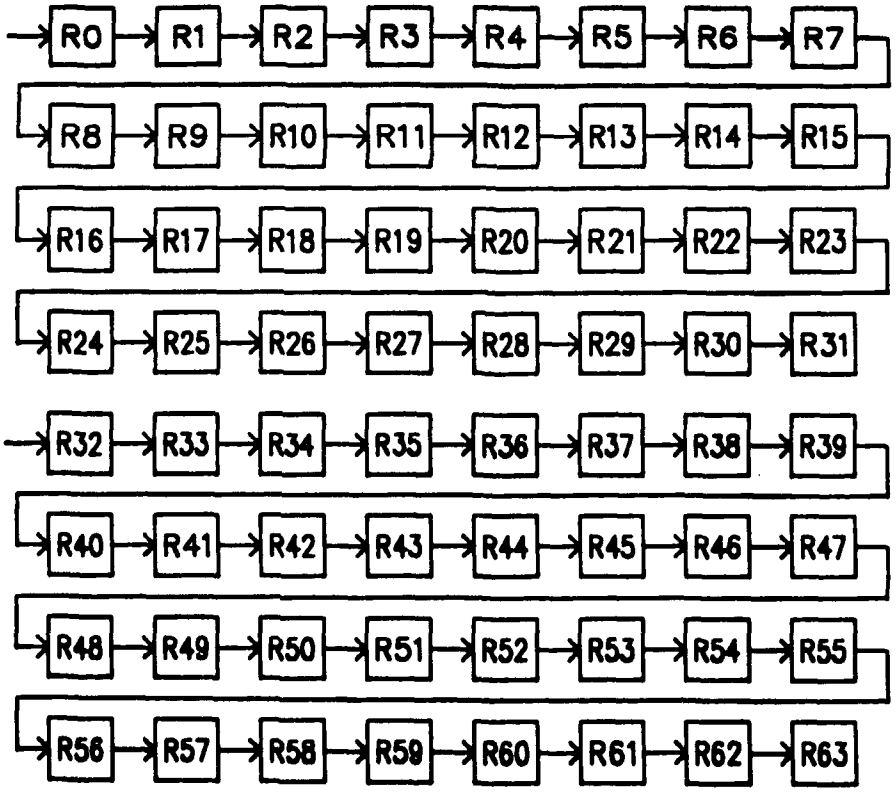


Figure 34. GPR Shift Sequence Mode 3

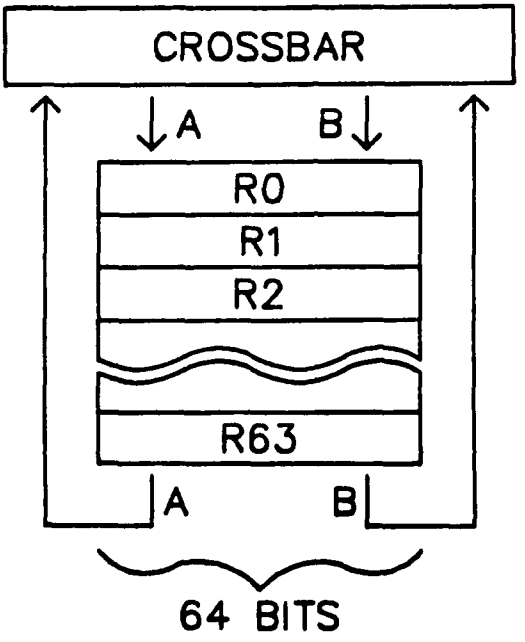


Figure 35. XBAR to GPR Path

Each output port has four select lines SELn[0..3] where n is the port number. The value placed on these inputs determines the source of the data to be sent to the output port registers. As an example, placing a hex value of "5" on any set of SEL inputs will select input port 5 as the data source. In addition, a hex value of "F" will disable the output port and a hex value of "0" will select the output port register as the data source. This will cause the ports' registers to hold their current state. The ports' registers will also hold their state when the output is disabled with an "F". The multiplexer select inputs for the register file write ports. (SELA[0..3] and SELB[0..3]) are similar to the ones for the output ports; however, a hex value of "0" will send all zeros to the register file and a hex value of "F" will send all ones.

OUTPUT PORTS

Each output port (O1_[0..3] to O14_[0..3]) and each I/O port (IO11_[0..3] and IO12_[0..3]) have two multiplexers and two 8-bit registers. The operation of the first multiplexer is described above and is used to select the source of data presented to the output registers. These registers are clocked by separate, anti-phase clocks. The phase 1 register is clocked by the low-to-high transition of CLK1, and the phase 2 register is similarly clocked by CLK2. The outputs from these registers are then input to the second multiplexer.

The second multiplexer has two control lines, PSEL and MSWSEL, which are used to select 4 bits for the output buffer. A low level on PSEL selects data from the phase 1 register while a high level selects data from the phase 2 register.

The MSWSEL input selects between the most and least significant 4-bit nibbles. A low level on MSWSEL selects the 4 least significant bits to be output.

REGISTER FILE

The register file is a four port synchronous static RAM memory organized as an 8 by 8 array of 8-bit registers. These registers are clocked by the rising edge of CLK3. The register file has two read ports (RPA AND RPB) and two write ports (WPA and WPB). Each port has its own address and all ports may be used simultaneously. Writing to the same location from both write ports simultaneously is allowed. Whenever this happens, the data from RPA is used.

The write address inputs are WRA_[0..5] and WRB_[0..5]. Each write port also has an active low enable, /WRENA or /WRENB. The read address inputs are RDA_[0..5] and RDB_[0..5]. The data read from the register file may be accessed by any output port or be written back into the register file. A hex value of "D" placed on any output port's SEL select lines will select RPA and a value of "E" will select RPB.

REGISTER FILE SHIFT MODES

Inputs SM1 and SM0 are used to configure the register file as a shift register. When both of these inputs are low, the register file functions like

a normal static RAM. When SM0 is brought high while SM1 remains low, each row of the register file becomes an eight deep shift register. Writing to the first register of each row causes the shift. The seven remaining registers of each row will be written to with the data from the preceding register. The old data in the last register is lost forever. Writing to a register other than the first register only updates that specific register. Reading never modifies any data.

Bringing SM1 high while leaving SM0 low links pairs of rows to give a configuration of four shift registers, each 16 registers deep. Bringing both SM1 and SM0 high links four rows together yielding two shift registers, each 32 registers deep.

OPERATING MODES

The crossbar has three possible modes of operation: non-pipelined synchronous, non-pipelined asynchronous, and pipelined synchronous. The MODE input selects whether certain other inputs pass through input pipeline registers, or if these registers are bypassed. The affected inputs are:

SELx[0..3], WRA_[0..5], WRB_[0..5], RDA_[0..5], RDB_[0..5], PSELx, SELA_[0..3], SELB_[0..3], /WRENA, /WRENB, SM1, AND SM0. Inputs which are not affected are: Ix[0..3], MSWENx, and MSWSELx.

A low level on MODE causes all inputs to bypass the input pipeline registers. With CLK3 left running, non-pipelined synchronous mode operation is achieved. This is the normal mode of operation and no special considerations are involved.

If CLK3 is tied low while MODE is held low, non-pipelined asynchronous operation is invoked. In this mode, the register file registers are clocked with the rising edge of /WRENA or /WRENB. Asynchronous register file writes can therefore be accomplished in this mode. Operation of the input ports, output ports, and multiplexers is unaffected by the absence of CLK3.

If MODE is brought high, pipelined synchronous mode operation is determined and CLK3 must be left running. This is because CLK3 is used to clock the input pipeline registers. The main consideration in this mode of operation is the affected inputs must be presented to the crossbar one CLK3 cycle sooner, and slightly different set-up and hold times may be involved.

A number of important control signals are listed next in Figure 36. Register file and port control follow in Figure 37. Then, timing charts for the mode 0 operations can be found in subsequent Figures 38 through 43. These data sheets formed the specifications for contracting the fabrication effort out to ILSI in Colorado Springs. Testing of the crossbars was accomplished at ILSI and later at Space Tech. The same test vectors by ILSI were on our emulyzer to verify ILSI tests. Those vectors can be found in the ILSI manual for the crossbars.

CLK1	Active high clock for phase one output port registers.
CLK2	Active high clock for phase two output port registers.
CLK3	Active high clock for register file and control input pipeline registers.
MODE	Bypasses control input pipeline registers when low.
I1_[0..3] to I10_[0..3]	Data input ports to the crossbar and register file.
MSWEN1 to MSWEN12	Controls input port demultiplexing latches. Latches are transparent when high.
SEL1_[0..3] to SEL14_[0..3]	Select inputs for output port registers.
PSEL1 to PSEL14	Selects phase one register for output when low and phase two when high.
MSWSEL1 to MSWSEL14	Multiplexer for output ports. Selects most significant four bits when high.
O1_[0..3] to O10_[0..3] O13_[0..3] O14_[0..3]	Data output ports from crossbar and register file.
IO11_[0..3] IO12_[0..3]	Bidirectional data ports.
SELA[0..3]	Select inputs for register file write port A.
SELB[0..3]	Select inputs for register file write port B.
$\overline{\text{WRENA}}$	Active low write enable for register file port A.
$\overline{\text{WRENB}}$	Active low write enable for register file port B.
WRA[0..5]	Address inputs for register file write port A.
WRB[0..5]	Address inputs for register file write port B.
RDA[0..5]	Address inputs for register file read port A.
RDB[0..5]	Address inputs for register file read port B.
SMODE0 SMODE1	Shift mode control inputs for register file.

Figure 36. Control Signals

Output Port Control

SELn_3	SELn_2	SELn_1	SELn_0	Output Port Register Source
0	0	0	0	Registers Hold Current Value
0	0	0	1	Input port #1
0	0	1	0	Input Port #2
0	0	1	1	Input Port #3
0	1	0	0	Input Port #4
0	1	0	1	Input Port #5
0	1	1	0	Input Port #6
0	1	1	1	Input Port #7
1	0	0	0	Input Port #8
1	0	0	1	Input Port #9
1	0	1	0	Input Port #10
1	0	1	1	Input/Output Port #11
1	1	0	0	Input/Output Port #12
1	1	0	1	Register File Read Port A
1	1	1	0	Register File Read Port B
1	1	1	1	Output High Impedance

Register File Control

SELn_3	SELn_2	SELn_1	SELn_0	Register File Write Source
0	0	0	0	All Zeros (Clear Register)
0	0	0	1	Input port #1
0	0	1	0	Input Port #2
0	0	1	1	Input Port #3
0	1	0	0	Input Port #4
0	1	0	1	Input Port #5
0	1	1	0	Input Port #6
0	1	1	1	Input Port #7
1	0	0	0	Input Port #8
1	0	0	1	Input Port #9
1	0	1	0	Input Port #10
1	0	1	1	Input/Output Port #11
1	1	0	0	Input/Output Port #12
1	1	0	1	Register File Read Port A
1	1	1	0	Register File Read Port B
1	1	1	1	All Ones (Set Register)

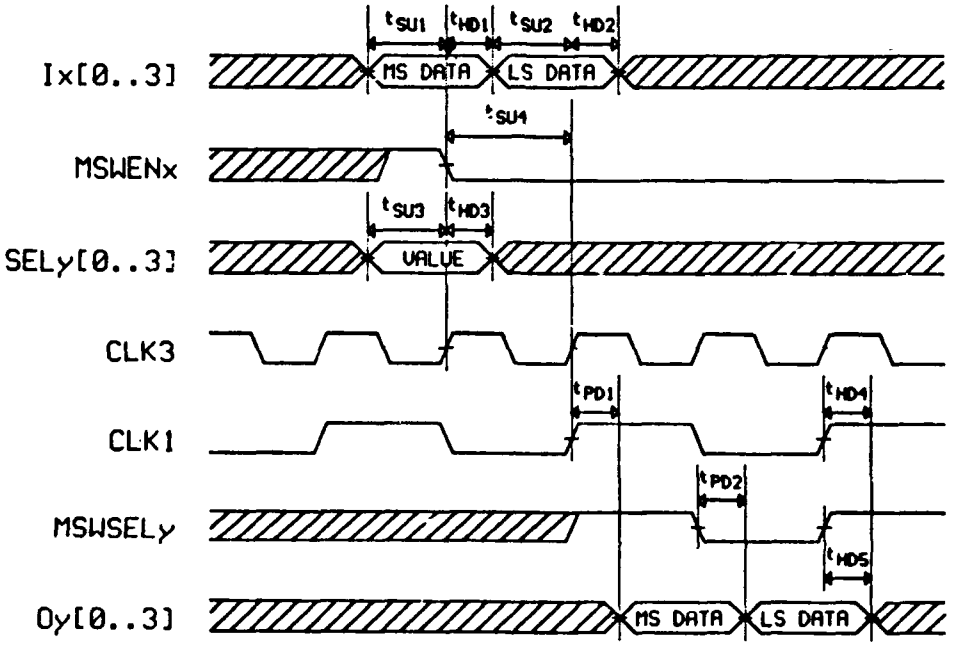
PSELn	MSWSELn	Source for Output Port On[0..3]
0	0	Least Significant Phase One Register
0	1	Most Significant Phase One Register
1	0	Least Significant Phase Two Register
0	1	Most Significant Phase Two Register

SMODE1	SMODE0	Register File Shift Mode Select
0	0	Normal "RAM" Mode
0	1	8 by 8 Shift Register Mode
1	0	4 by 16 Shift Register Mode
0	1	2 by 32 Shift Register Mode

Figure 37. Register File and Port Control

Input Port to Output Port Transaction for CLK1

MODE=1 PSEL=0

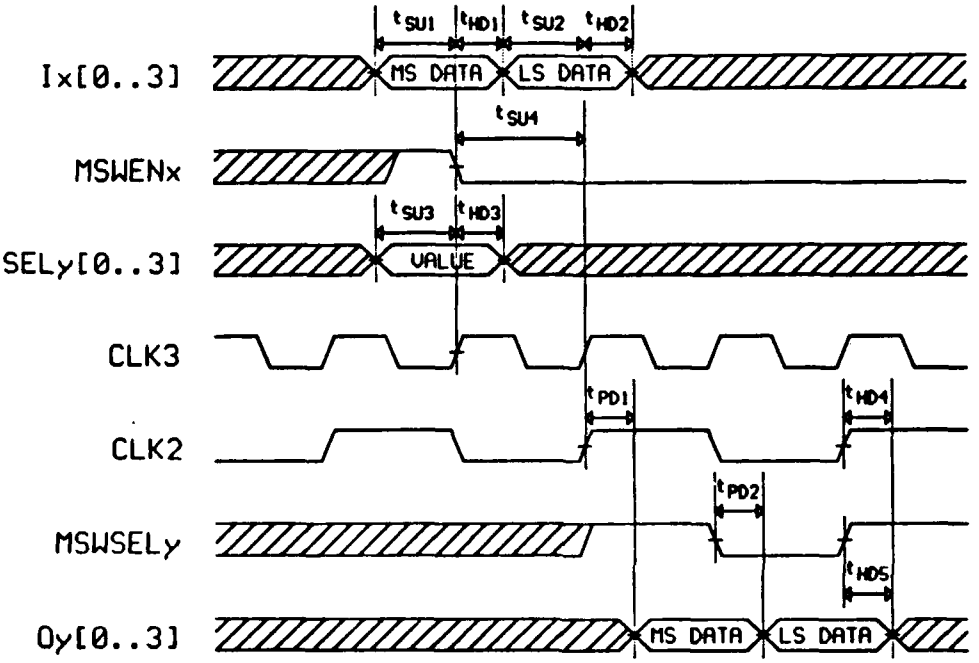


PARAMETER	DESCRIPTION	MIN	MAX	UNITS
t_{SU1}	Input Data to MSWEN LOW Set-up			ns
t_{HD1}	Input Hold from MSWEN LOW			ns
t_{SU2}	Input Data to CLK1 HIGH Set-up			ns
t_{HD2}	Input Hold From CLK1 HIGH			ns
t_{SU4}	Set-up From MSWEN LOW to CLK1 HIGH			ns
t_{SU3}	SEL Inputs to CLK3 HIGH Set-up			ns
t_{HD3}	SEL Inputs Hold From CLK3 HIGH			ns
t_{PD1}	CLK1 HIGH to Output Data Valid			ns
t_{HD4}	Output Data Hold From CLK1 HIGH			ns
t_{PD2}	MSWSEL to Output Data Valid			ns
t_{HD5}	Output Data Hold From MSWSEL Transition			ns

Figure 38. Timing Charts

Input Port to Output Port Transaction for CLK2

MODE=1 PSEL=1

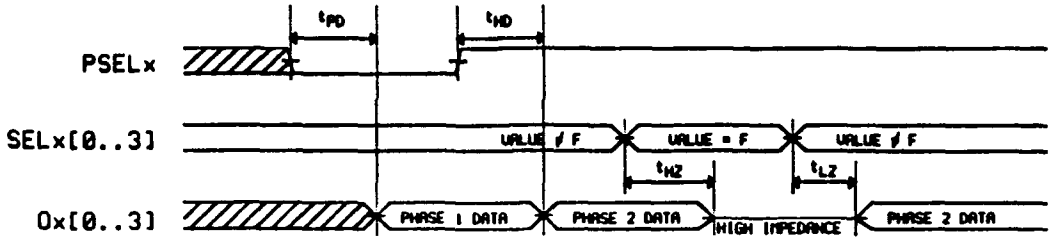


PARAMETER	DESCRIPTION	MIN	MAX	UNITS
t_{SU1}	Input Data to MSWEN LOW Set-up			ns
t_{HD1}	Input Hold from MSWEN LOW			ns
t_{SU2}	Input Data to CLK2 HIGH Set-up			ns
t_{HD2}	Input Hold From CLK2 HIGH			ns
t_{SU4}	Set-up From MSWEN LOW to CLK2 HIGH			ns
t_{SU3}	SEL Inputs to CLK3 HIGH Set-up			ns
t_{HD3}	SEL Inputs Hold From CLK3 HIGH			ns
t_{PD1}	CLK2 HIGH to Output Data Valid			ns
t_{HD4}	Output Data Hold From CLK2 HIGH			ns
t_{PD2}	MSWSEL to Output Data Valid			ns
t_{HD5}	Output Data Hold From MSWSEL Transition			ns

Figure 39. Timing Charts

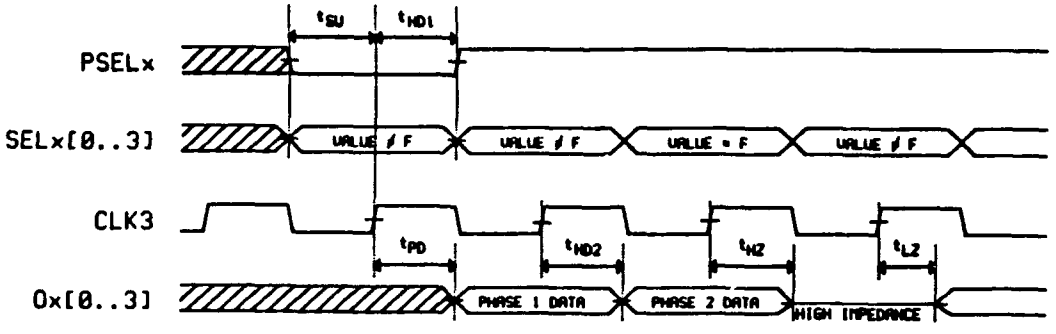
OUTPUT PORT CONTROL

MODE=0



PARAMETER	DESCRIPTION	MIN	MAX	UNITS
t_{PD}	PSEL Transition to Output Data Valid			ns
t_{HD}	Output Data Hold From PSEL Transition			ns
t_{HZ}	SEL = F to Output High Impedance			ns
t_{LZ}	SEL ≠F to Output Low Impedance			ns

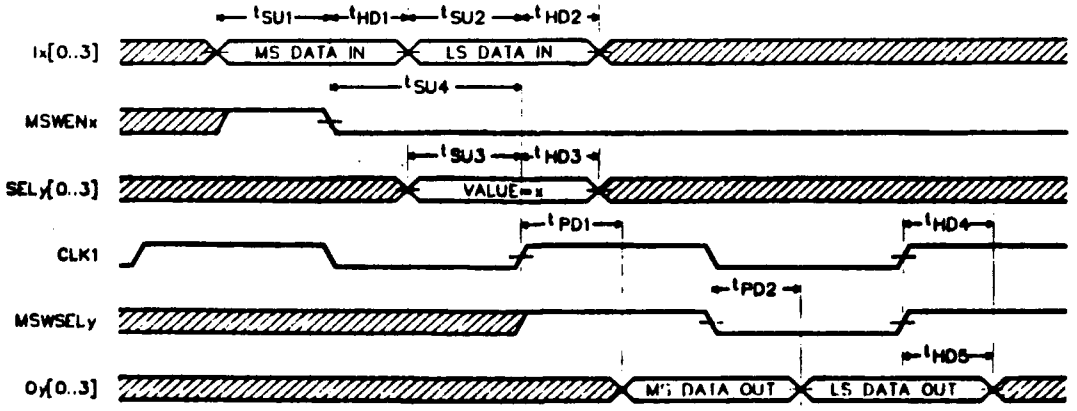
MODE=1



PARAMETER	DESCRIPTION	MIN	MAX	UNITS
t_{SU}	PSEL or SEL Inputs to CLK3 Set-up			ns
t_{HD1}	PSEL or SEL Inputs Hold From CLK3 HIGH			ns
t_{PD}	CLK3 HIGH to Output Data Valid			ns
t_{HD2}	Output Data Hold From CLK3 HIGH			ns
t_{HZ}	Output High Impedance From CLK3 HIGH			ns
t_{LZ}	Output Low Impedance From CLK3 HIGH			ns

Figure 40. Timing Charts

Input to Output Port Transaction for CLK1
MODE=0 PSELy=0

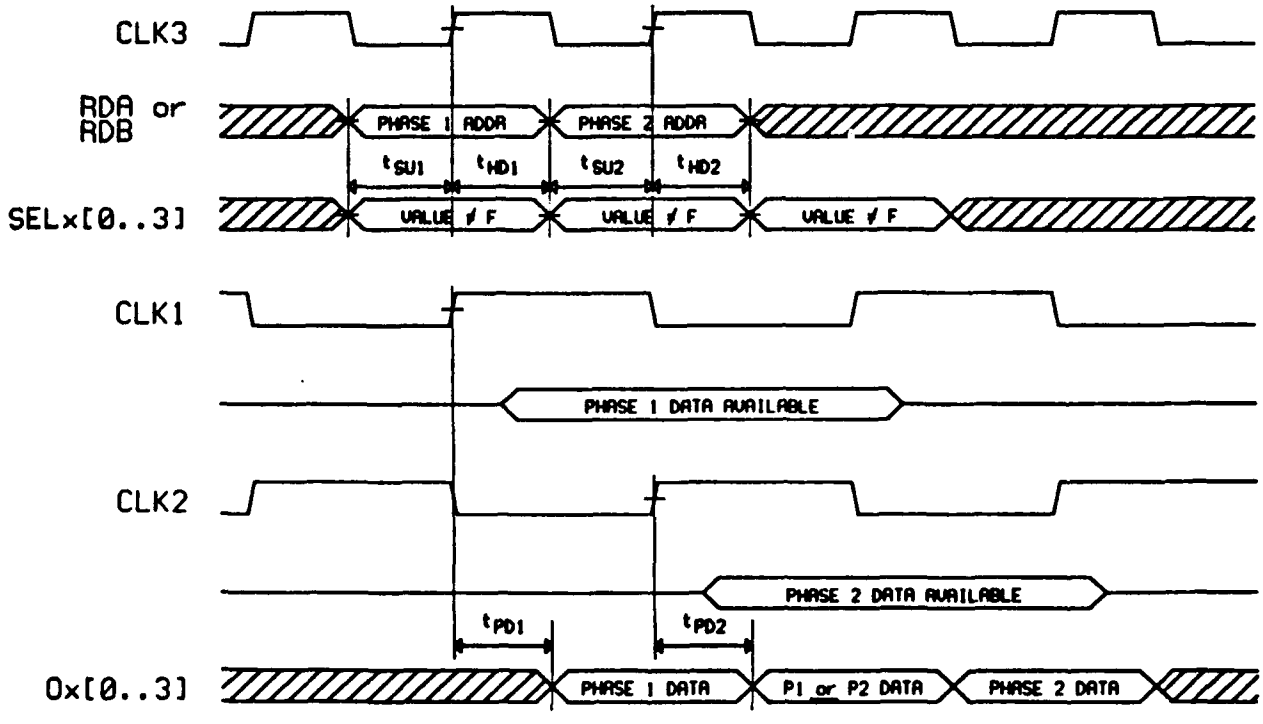


Parameter	Description	Min	Max	Units
t_{SU1}	Input Data to MSWEN LOW Set-up	4	NA	ns
t_{HD1}	Input Hold From MSWEN LOW	4	NA	ns
t_{SU2}	Input Data to CLK1 HIGH Set-up	12	NA	ns
t_{HD2}	Input Hold From CLK1 HIGH	0	NA	ns
t_{SU4}	Set-up From MSWEN LOW to CLK1 HIGH	16	NA	ns
t_{SU3}	SEL Inputs to CLK1 HIGH Set-up	17	NA	ns
t_{HD3}	SEL Inputs Hold From CLK1 HIGH	6	NA	ns
t_{PD1}	CLK1 HIGH to Output Data Valid	4	17	ns
t_{HD4}	Output Data Hold From CLK1 HIGH			ns
t_{PD2}	MSWSEL to Output Data Valid	3	9	ns
t_{HD5}	Output Data Hold From MSWSEL Transition			ns

Figure 41. Timing Charts

REGISTER FILE READ

MODE=0

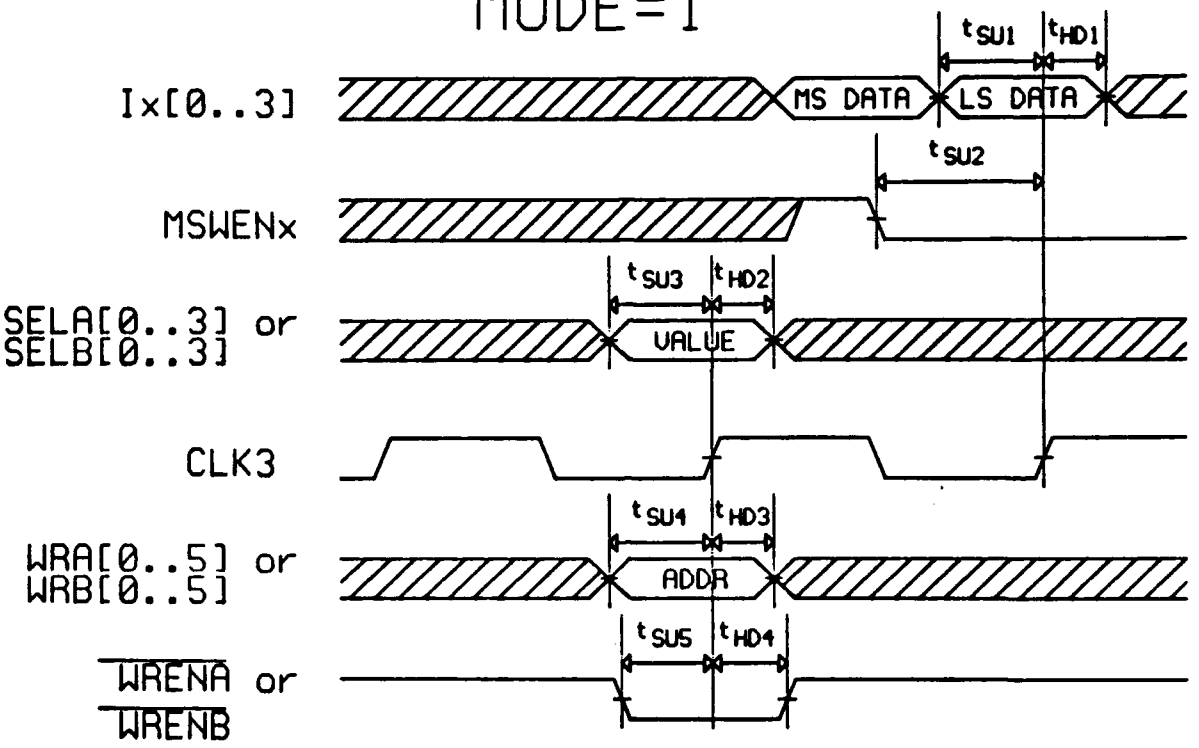


PARAMETER	DESCRIPTION	MIN	MAX	UNITS
t_{SU1}	RDA or SEL Inputs to CLK3 Set-up			ns
t_{HD1}	RDA or SEL Inputs Hold From CLK3 HIGH			ns
t_{SU2}	RDA or SEL Inputs to CLK3 Set-up			ns
t_{HD2}	RDA or SEL Inputs Hold From CLK3 HIGH			ns
t_{PD1}	CLK1 HIGH to Output Data Valid			ns
t_{PD2}	CLK2 HIGH to Output Data Valid			ns

Figure 42. Timing Charts

REGISTER FILE WRITE

MODE=1



PARAMETER	DESCRIPTION	MIN	MAX	UNITS
t_{SU1}	Input Data to CLK3 HIGH Set-up			ns
t_{HD1}	Input Data Hold From CLK3 HIGH			ns
t_{SU2}	MSWEN LOW to CLK3 HIGH Set-up			ns
t_{SU3}	SELA or SELB to CLK3 HIGH Set-up			ns
t_{HD2}	SELA or SELB Hold From CLK3 HIGH			ns
t_{SU4}	WRA or WRB to CLK3 HIGH Set-up			ns
t_{HD3}	WRA or WRB Hold From CLK3 HIGH			ns
t_{SU5}	$\overline{WRENA}$ or $\overline{WRENB}$ to CLK3 HIGH Set-up			ns
t_{HD4}	$\overline{WRENA}$ or $\overline{WRENB}$ Hold From CLK3 HIGH			ns

Figure 43. Timing Charts

3.2.8.1 Testing the Crossbars

Characterization tests were performed by ILSI at ILSI before shipment to Space Tech. Those test sequences and vectors are listed in the ILSI specifications manual under separate cover. Verification tests were performed at Space Tech with a Hi-Level Emulyzer connected to the input and output ports of each device. The same vectors were used at Space Tech as were used at ILSI to confirm the operation of each device. Of the ten shipped to us, only one failed and was dead on arrival. It was replaced by ILSI after they confirmed our results. The vectors used by Space Tech and ILSI set up 1s and 0s in adjacent bits alternating and repeating so that crosstalk could be discovered. Clocks were adjusted from 1 to 20 MHz and the chips passed at all clocks except 20 MHz in some modes. Those modes are not used in the CPH so they were important. The important modes were mode 0 modes and all passed these mode tests at all clock speeds.

The typical test setup of vectors used are shown in the following sheet from the engineer's notebook in Figure 44. Here, we can see that read and write ports A and B were activated with the several input data control lines and output data control lines. The testing took approximately 4 hours per device since 12x14 combinations of configurations were to be tested by numerous test vectors. The Space Tech test fixture is shown in the next drawing as Figure 45. The test fixture uses the pinout assignments for the crossbar chip as shown in Figure 46. A 6U Mupac VME board was used with PALs and registers to clock test signals and controls onto the crossbar under test.

A PAL function was created for the test jig, XBAR1M.POS, to input data into the I/O ports in a pipelined, synchronous manner. The test vectors of mode 0 could be used in testing mode 1 with the following modifications. The write pulse had to be shifted from the least significant vectors to the most significant positions. The write pulse had to be widened by several nanoseconds (accomplished by modifying XBAR2.PDS to include an additional input, namely async). The input data to the I/O ports had to be shifted one cycle sooner to offset the additional pipelining the PALs now present. And the SELx data of any F's (to high impedance output PORTx) had to be shifted one cycle sooner also (due to mode 1 internal pipelining of SELx data).

With the modifications described and one new set of vectors to test all of the internal pipelining, six sets of vectors were used to test mode 1 operation. After creating output reference files to compare XBAR outputs to, testing of the XBAR chips commenced in earnest.

While testing the XBAR, some sets of vectors ran better if a different amount of delay was used between SLK3 and PGCLK. Thus, a "gate delay line" was introduced to the jig to allow selective clock skewing. The delays needed for optimum testing are listed in the Engineer's Notebook which gives the complete testing procedure.

The result of testing was that 9 of 10 chips ran all 11 sets of test vectors with no erroneous output. The tenth chip, however, did not successfully run even one set of vectors. Several clock speeds and skews were tried and didn't get any improvement. The chip was then packaged up and sent back to ILSI for replacement.

Date: 1/23/92 File: F:\PHIL\TEMP\XBAR_IN.W

```
1 - ??????????????????????????????????????????????????????????????????????????????
2 -                    PATTERN GENERATOR OUTPUT WORDS
3 - ??????????????????????????????????????????????????????????????????????????????
4 -
5 -
6 -                MSN                                              LSN
7 - WORD FORMAT:   HEX1/HEX2/HEX3/ ... ... /HEX34/HEX35/HEX36
8 -
9 -
10 -   NIBBLE LEGEND:
11 -
12 -      HEX1  = /WCA, XXX, WRA-5, WRA-4                  ---+
13 -      HEX2  = WRA-3, WRA-2, WRA-1, WRA-0                  |-- WRITE PORT A CONTROL
14 -      HEX3  = SELA-3, SELA-2, SELA-1, SELA-0           ---+
15 -
16 -      HEX4  = /WCB, XXX, WRB-5, WRB-4                  ---+
17 -      HEX5  = WRB-3, WRB-2, WRB-1, WRB-0                  |-- WRITE PORT B CONTROL
18 -      HEX6  = SELB-3, SELB-2, SELB-1, SELB-0           ---+
19 -
20 -
21 -      HEX7  = XXX, XXX, RDA-5, RDA-4                   ---+-- READ PORT A CONTROL
22 -      HEX8  = RDA-3, RDA-2, RDA-1, RDA-0               ---+
23 -
24 -      HEX9  = XXX, XXX, RDB-5, RDB-4                   ---+-- READ PORT B CONTROL
25 -     HEX10  = RDB-3, RDB-2, RDB-1, RDB-0               ---+
26 -
27 -
28 -     HEX11  =  SEL1-3,  SEL1-2,  SEL1-1,  SEL1-0       ---+
29 -     HEX12  =  SEL2-3,  SEL2-2,  SEL2-1,  SEL2-0          |
30 -     HEX13  =  SEL3-3,  SEL3-2,  SEL3-1,  SEL3-0          |
31 -     HEX14  =  SEL4-3,  SEL4-2,  SEL4-1,  SEL4-0          |
32 -     HEX15  =  SEL5-3,  SEL5-2,  SEL5-1,  SEL5-0          |
33 -     HEX16  =  SEL6-3,  SEL6-2,  SEL6-1,  SEL6-0          |
34 -     HEX17  =  SEL7-3,  SEL7-2,  SEL7-1,  SEL7-0          |-- OUTPUT DATA SOURCE
35 -     HEX18  =  SEL8-3,  SEL8-2,  SEL8-1,  SEL8-0          |
36 -     HEX19  =  SEL9-3,  SEL9-2,  SEL9-1,  SEL9-0          |
37 -     HEX20  = SEL10-3, SEL10-2, SEL10-1, SEL10-0          |
38 -     HEX21  = SEL11-3, SEL11-2, SEL11-1, SEL11-0          |
39 -     HEX22  = SEL12-3, SEL12-2, SEL12-1, SEL12-0          |
40 -     HEX23  = SEL13-3, SEL13-2, SEL13-1, SEL13-0          |
41 -     HEX24  = SEL14-3, SEL14-2, SEL14-1, SEL14-0       ---+
42 -
43 -
44 -     HEX25  =   I1-3,   I1-2,   I1-1,   I1-0          ---+
45 -     HEX26  =   I2-3,   I2-2,   I2-1,   I2-0             |
46 -     HEX27  =   I3-3,   I3-2,   I3-1,   I3-0             |
47 -     HEX28  =   I4-3,   I4-2,   I4-1,   I4-0             |
48 -     HEX29  =   I5-3,   I5-2,   I5-1,   I5-0             |
49 -     HEX30  =   I6-3,   I6-2,   I6-1,   I6-0             |-- INPUT DATA
50 -     HEX31  =   I7-3,   I7-2,   I7-1,   I7-0             |
51 -     HEX32  =   I8-3,   I8-2,   I8-1,   I8-0             |
52 -     HEX33  =   I9-3,   I9-2,   I9-1,   I9-0             |
53 -     HEX34  =  I10-3,  I10-2,  I10-1,  I10-0             |
54 -     HEX35  = I011-3, I011-2, I011-1, I011-0  ss          |
55 -     HEX36  = I012-3, I012-2, I012-1, I012-0  ss       ---+
```

Figure 44. Engineer's Notebook Sheet

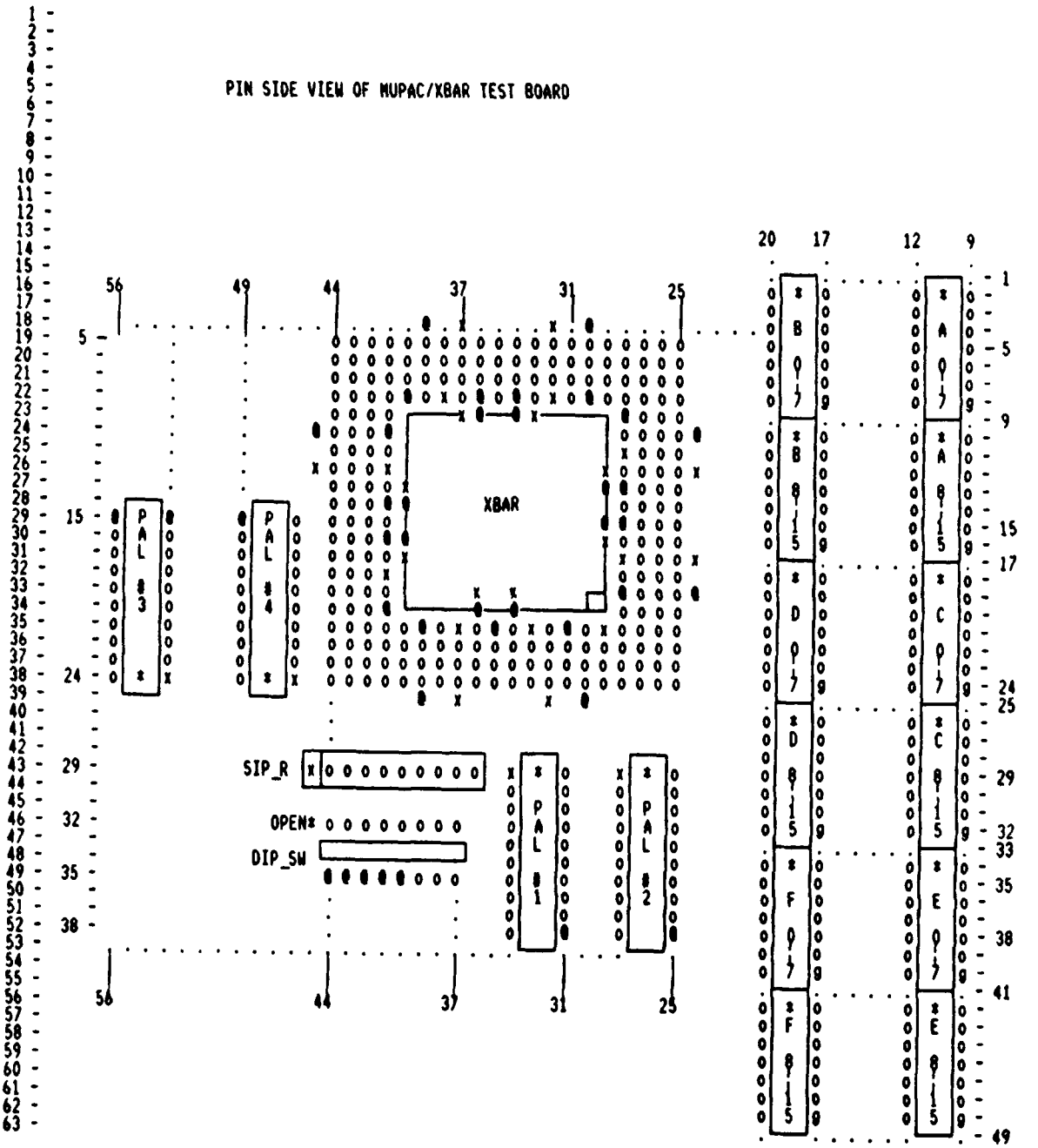


Figure 45. MUPAC Test Board

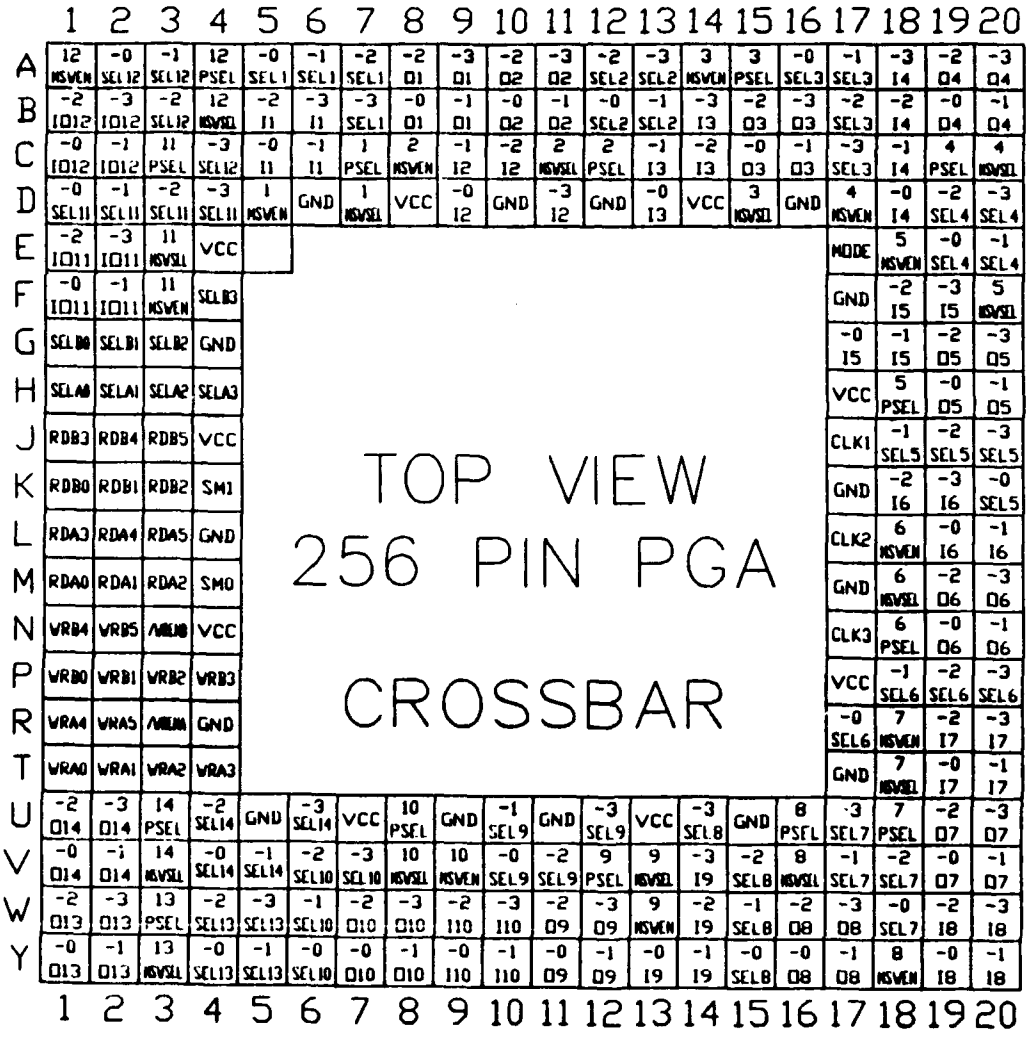


Figure 46. Crossbar Pinout

3.2.9 CPH Microsequencer

As with many of the other "glue logic" functions, a microprogram sequencer chip fast enough for the EVA architecture was not available in 1990. A sequencer that can also support relative addressing and interrupts was required. Several are available now but they remain too slow. Available sequencers that can handle the high speed don't support interrupts or the necessary addressing modes. One solution was to build the sequencer out of high speed PALS and logic chips. An architecture that could be built from available parts was designed. The problem with this approach is that over 50 chips are required. A few components could be added to one of the simple sequencer chips to support the required addressing modes. This would reduce the part count but the combined delay would be too great to meet the high speed requirement. Fortunately, IDT developed a suitable part by 1991.

The CPH Microprogram Sequencer (CPH-MS) is designed to perform its function in a 50 nsec maximum cycle time. Although the timing analysis is not complete, a preliminary analysis of the critical timing paths, those paths which pass through the slowest and/or greatest number of components seem to meet the timing criteria. A microinstruction set that has been selected is:

INITIALIZATION	
Load Loop Counter	16-bit count
Load Stack Pointer	10-bit address
Load Subroutine RAM Pointer	10-bit address
Load Subroutine RAM	16-bit data
IMMEDIATE	
Jump Immediate	16-bit address
Jump Immediate Conditional	16-bit address
Loop Immediate	16-bit address
RELATIVE	
Jump Relative	16-bit relative address
Jump Relative Conditional	16-bit relative address
Loop Relative	16-bit relative address
INDEXED	
Call	10-bit index
Call Conditional	10-bit index
INTERRUPTS	
Set Interrupt Mask	8-bit data
Reset Interrupt	8-bit data
OTHER	
No Operation	no data
Return	no data
Return Conditional	no data
Push	no data
Pop	no data

The method of using indexed subroutine calls allows each software module to be assembled, linked, and located at a base address of 0000h. The modules may then be loaded into program memory and called by their index number. Each call accesses the subroutine RAM by index number, and the subroutine RAM then loads the program counter with the address corresponding to the physical location of the module. Care must be taken when programming the modules not to use immediate instructions. Implementing the interrupt vector table into

the same RAM as the subroutine indices, and separate from the stack RAM, provides for the simultaneous access of both banks of RAM during a Call instruction. This allows the present address in the program counter to be pushed onto stack at the same time that the new 'call' address is presented to the program counter for a 50 nsec single cycle instruction. By placing the interrupt table in the subroutine RAM, the same single cycle instruction may push the program counter onto the stack upon detection of a hardware interrupt. This also simplifies hardware design, since the latches necessary to hold the RAM address while loading in data need not be present for the stack RAM.

The following features are supported:

A 2-to-1 MUX allows the immediate/relative address to come from a source external to the microsequencer. The stack and subroutine RAM is 4kx16 in size. An additional output MUX and a tri-state buffer were added to create two separate buses, one dedicated to the microsequencer and the second drives the external RAM. This helps guarantee that the tight timing requirements of the microsequencer won't be compromised.

Several restrictions on instruction sequences have been eliminated by designing the stack pointer out of PALs rather than discrete up/down counters. Prior to the change, CALL and PUSH type instructions which increment the stack after writing to it conflicted with RET instructions which increment the stack before reading from it. The solution required that a 40 MHz clock be brought in and logic added to compare the previous instruction to its successor and decide at each 20 MHz clock whether or not to increment or decrement for the CALL, PUSH, and RET type instructions. For POP, LS, TWBI, and TWBR instructions where the data is merely discarded from the stack, this is done using the 40 MHz clock at mid-instruction.

The full instruction set now follows. Since the instructions are 'microcoded' using PALs, and the PALs have many product terms remaining, additional instructions may have to be added as required without changing any hardware.

NOTE: In the following description /CNT0 refers to the loop counter's terminal count which goes low upon reaching zero, and /COND is a condition bit which indicates a true condition when low.

INSTRUCTION SET

NOP	No Operation
LDLC	Load Loop Counter
LDSP	Load Stack Pointer
LDSRP	Load Subroutine RAM Pointer
LDSUBR	Load Subroutine RAM
SIM	Set Interrupt Mask
RIM	Reset Interrupt Mask
RINT	Reset Interrupt
JI	Jump Immediate
JIC	Jump Immediate Conditional
JR	Jump Relative

JRC	Jump Relative Conditionally
LI	Loop Immediate
LR	Loop Relative
LS	Loop Stack
TWBI	Three-Way Branch Immediate
TWBR	Three-Way Branch Relative
CALL	Call
CALLC	Call Conditional
RET	Return
RETC	Return Conditional
PUSH	Push
PUSHC	Push Conditionally
PLDLC	Push and Load Loop Counter
PLDLCC	Push and Load Loop Counter Conditionally
POP	Pop (Discard Top of Stack)
POPC	Pop Conditionally (Discard Top of Stack)
EI	Enable Interrupts
DI	Disable Interrupts

When a data field of less than 16-bits is specified, the data is to be right justified into the lowest bits possible. For example, an 8-bit number A5h will become 00A5h in the 16-bit data field.

Mnemonic	OpCode	Data	Description
NOP	07Fh	--	Does nothing but consume time. The next address is the program counter + 1.
LDLC	07Eh	16-bits	Load loop counter with the data appearing in the data field. The next address is the program counter + 1.
LDSP	07Dh	12-bits	Load stack pointer with the data appearing in the data field. The next address is the program counter + 1.
LDSRP	07Ch	16-bits	Load subroutine RAM address pointer with the data appearing in the data field. The next address is the program counter + 1.
LDSUBR	07Bh	16-bits	Write the data to subroutine/ interrupt RAM location pointed to by the subroutine address pointer last loaded using the LDSRP instruction. The next address is the program counter + 1.
SIM	07Ah	8-bits	Set interrupt masks indicated in

			the data field. Each bit in the data field corresponds to one interrupt. The least significant bit corresponds to interrupt 0 (/INT0) which has the lowest priority, up through the most significant bit for interrupt 7 (/INT7) which has the highest priority. Wherever a bit is set to one in the data field the corresponding mask will be set. The next address is the program counter + 1.
RIM	079h	8-bits	Resets interrupt masks indicated in the data field. Each bit in the data field corresponds to one interrupt. The least significant bit corresponds to interrupt 0 (/INT0) which has the lowest priority, up through the most significant bit for interrupt 7 (/INT7) which has the highest priority. Wherever a bit is set to one in the data field the corresponding mask will be reset. The next address is the program counter + 1.
RINT	078h	8-bits	Resets the interrupts indicated in the data field. Each bit in the data field corresponds to one interrupt. The least significant bit corresponds to interrupt 0 (/INT0) which has the lowest priority, up through the most significant bit for interrupt 7 (/INT7) which has the highest priority. Wherever a bit is set to one in the data field the corresponding interrupt will be reset. The next address is the program counter + 1.
JI	077h	16-bits	Jump to the address specified in the data field.
JIC	076h	16-bits	Jump to the address specified in the data field only if the /COND signal is low, else the next address is the program counter + 1.
JR	075h	16-bits	Jump to the address created by

			adding the program counter to the data field.
JRC	074h	16-bits	Jump to the address created by adding the program counter to the data field only if the /COND signal is low, else the next address is the program counter + 1.
LI	073h	16-bits	If /CNT0 is high, indicating that the loop counter has not yet reached 0, then jump to the address specified in the data field. If /CNT0 is low the next address is the program counter + 1.
LR	072h	16-bits	If /CNT0 is high, indicating that the loop counter has not yet reached 0, then jump to the address created by adding the program counter to the data field. If /CNT0 is low the next address is the program counter + 1.
LS	071h	--	If /CNT0 is high, indicating that the loop counter has not yet reached 0, then jump to the address located on the top of the stack. This address is to remain on the top of the stack after the jump. If /CNT0 is low, then the jump address on the top of the stack is discarded and the next address is the program counter + 1.
TWBI	070h	16-bits	If /CNT0 is high, indicating that the loop counter has not yet reached 0, and /COND is high indicating a false condition, then jump to the address located on the top of the stack. If /CNT0 is low and /COND is high then jump to the address specified in the data field. The address on the top of the stack is discarded. If /COND is low then the next

			address is the program counter + 1 and the address appearing on top of the stack is discarded.
TWBR	06Fh	16-bits	If /CNT0 is high, indicating that the loop counter has not yet reached 0, and /COND is high indicating a false condition, then jump to the address located on the top of the stack. If /CNT0 is low and /COND is high then jump to the address created by adding the program counter to the data field. The address on the top of the stack is discarded. If /COND is low then the next address is the program counter + 1 and the address appearing on top of the stack is discarded.
CALL	06Eh	12-bits	The current program counter is incremented and stored onto the top of the stack. The program then jumps to the address appearing in the subroutine/ interrupt RAM at the SUBRAM address given in the data field.
CALLC	06Dh	16-bits	If /COND is low then the current program counter is incremented and stored onto the top of the stack. The program then jumps to the address appearing in the subroutine/interrupt RAM at the SUBRAM address given in the data field. If /COND is high then the next address is the program counter + 1.
RET	06Ch	--	Jump to the address appearing on the top of the stack.
RETC	06Bh	--	If /COND is low then jump to the address appearing on the top of the stack. If /COND is high then the next address is the program counter + 1.
PUSH	06Ah	--	Store the program counter + 1 on

			the top of the stack. The next address is the program counter + 1.
PUSHC	069h	--	If /COND is low then store the program counter + 1 on the top of the stack. The next address is the program counter + 1.
PLDLC	068h	16-bits	Store the program counter + 1 on the top of the stack. Load loop counter with the data appearing in the data field. The next address is the program counter + 1.
PLDLCC	067h	16-bits	Store the program counter + 1 on the top of the stack. NOTE: The preceding push was not conditional. If /COND is low, then load the loop counter with the data appearing in the data field. The next address is the program counter + 1.
POP	066h	--	Discard the data appearing on the top of the stack. The next instruction is the program counter + 1.
POPC	065h	--	If /COND is low then discard the data appearing on the top of the stack. The next instruction is the program counter + 1.
EI	064h	--	Enable future and pending unmasked interrupts to be serviced. The next instruction is the program counter + 1.
DI	063h	--	Disable all interrupts from being serviced. The next instruction is the program counter + 1.

Microinstruction productions for the CPH need to account for the timing delays in the crossbar, both in the processor and in the address generator. When selecting a pass through transfer or "in to out" in any direction, clock 1 selects the path (SEL). Clock 2 latches the input data. At Clock 4 the output data is available to the destination. To write data into the register file, Clock 1 selects the path (SEL), the register address, and the write enable signal (WRENA). At Clock 2 the data must be available to the crossbar for writing into the register. To read from a register, Clock 1 selects the port and the register address. At Clock 3, the data is available to the destination. (mode 1 operation only). The sample microprograms in the appendix take these delays into account. They should be examined carefully. Additional notes on microprogramming can be found in a later section.

For example, the IMMAD field or immediate address field is active in both phases. From the machine definition file in the appendix, one sees that the two ASSIGN statements are used. The first statement assigns physical bits 237 thru 339. The second statement assigns physical bits 621 thru 723. The higher order bits are reserved for the first phase and the lower order bits are reserved for the second phase. A particular phase at any clock cycle is selected transparent to the user. Clocking is done automatically.

3.2.10 Backplane

The CPH backplane depicted in Figure 47 entitled "Backplane" is a custom backplane with the footprint of a 9U VME board. However, all CPH boards require many more backplane pins then can be provided on the P1, P2, and P3 connectors of a standard VME bus. Special connectors from AMP were designed into the custom backplane. The plane must also have pinouts on the processor board which are different than those on the address generator and cache memory boards because the processor board can be cascaded with other processor boards. Each processor board must then generate different addresses to cache. The connector lists for the processor, cache, and address generator boards follow in Figures 48 and 49.

The physical configuration of the backplane consists of 9 slots and three left open for future expansion. Each connector will be placed on a 0.800 inch center to center spacing. The slot assignments are listed next.

Backplane Slot Assignment

Slot Number	System	Assignment
1	1	IOP
2	1	PROCESSOR
3	1	EMPTY
4	1	ADDR
5	1	EMPTY
6	1	CACHE MEMORY
7	2	PROCESSOR
8	2	EMPTY
9	2	CACHE MEMORY

Slots 3, 5, and 8 are empty to allow the tall boards to have clearance.

This backplane supports two CPH systems. The two systems share a common system clock, microsequencer address signals, and power, but all data and memory address buses are isolated between slots 6 and 7. This allows each system to access independent memory and data, and even to execute different microcode with the constraints that both systems have the same microsequencer generating a common program address.

The clock circuitry for the backplane remains to be designed. The initial design should support all phases of the CPH clock and should support single stepping. The single stepping feature can be installed on the frontplane with a debounce switch and as an alternating TTL signal from the IOP. The ECL-to-TTL conversion should be done on the backplane.

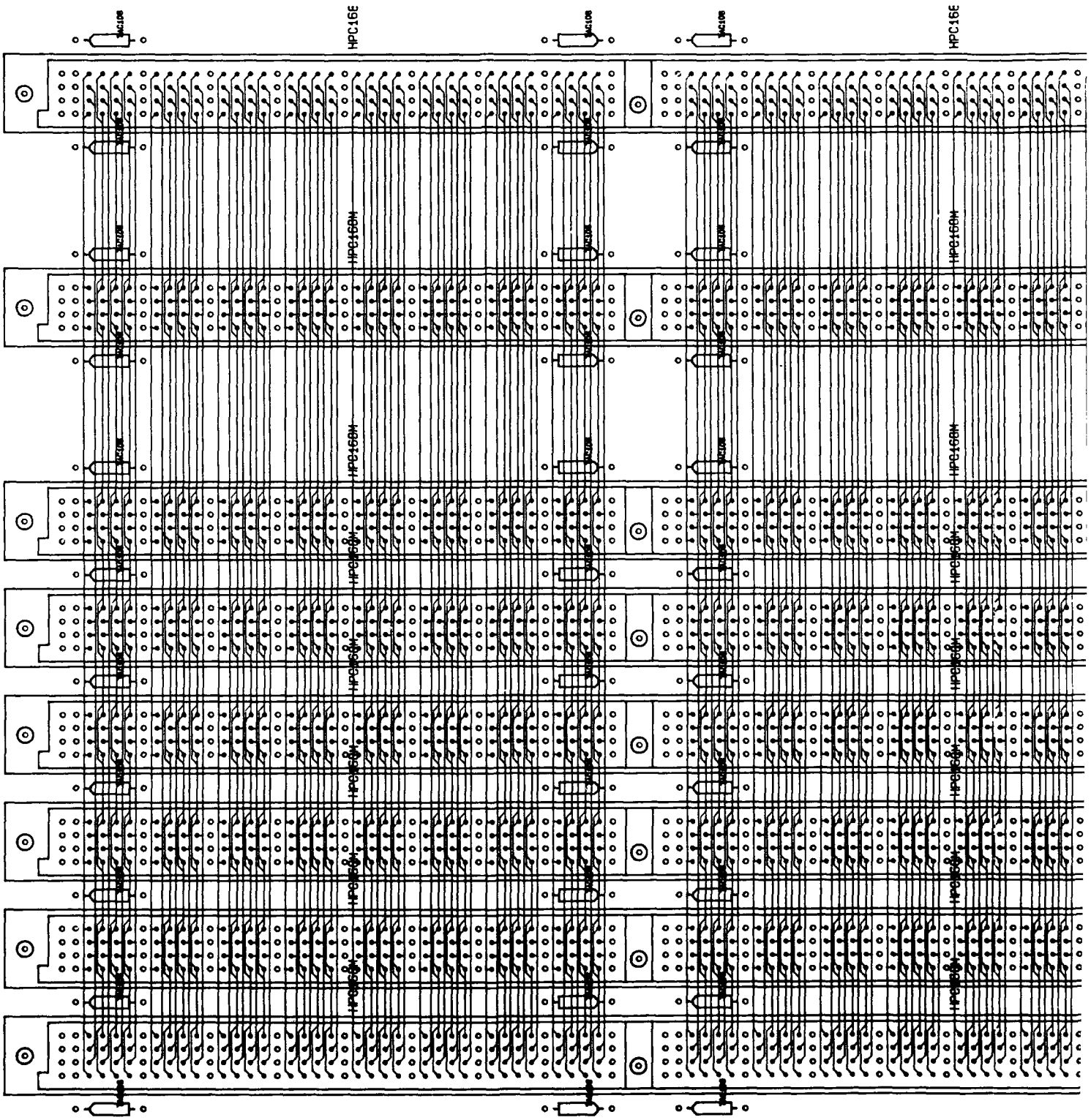
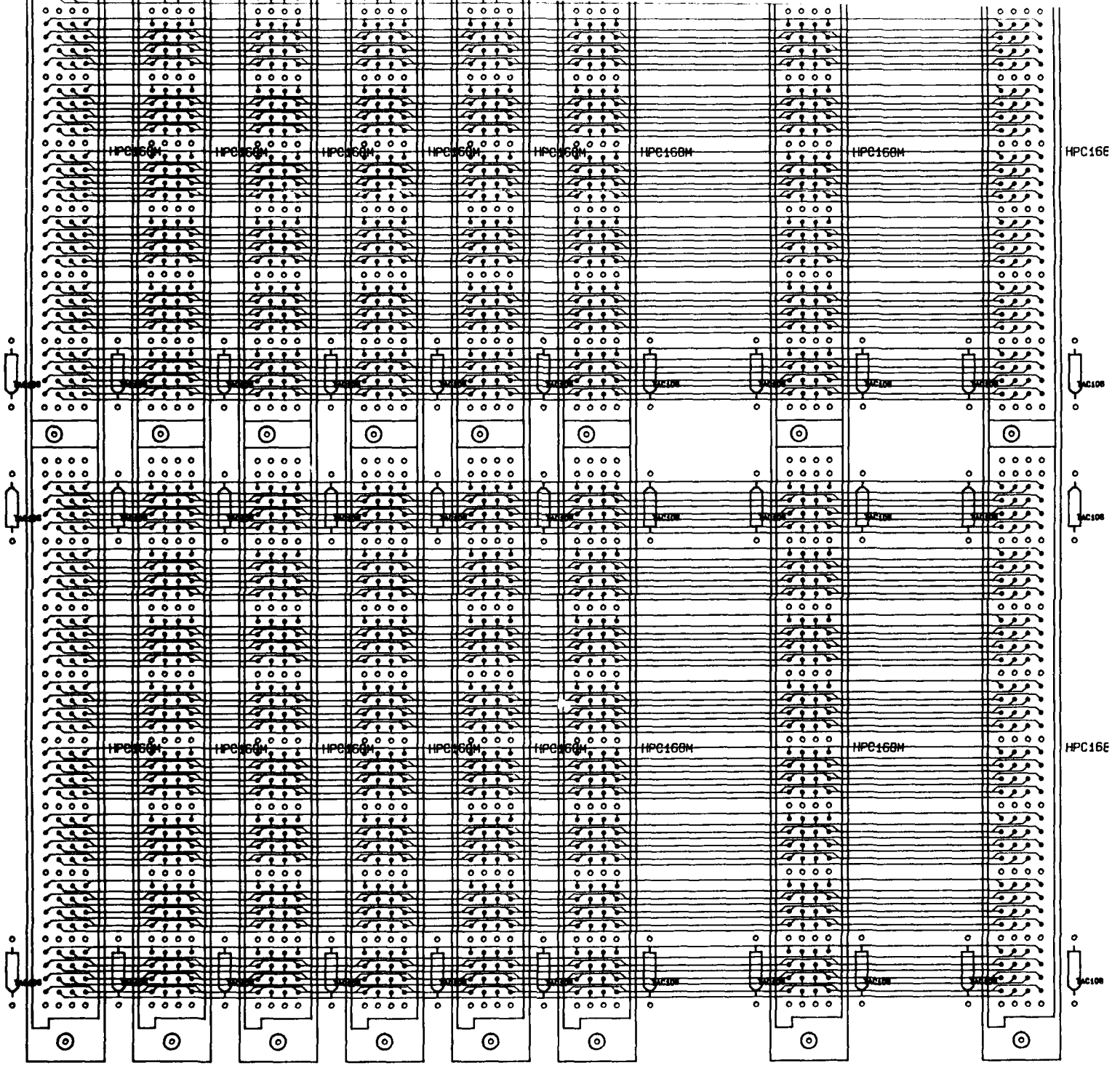


Figure 47. Backplane


HPC168M
HPC168M
HPC16E
HPC168M
HPC168M
HPC16E

THIS IS THE CONNECTOR LIST FOR THE PROCESSOR BOARD ONLY. IT DIFFERS FROM THE CONLST2.DOC LISTING. THAT LISTS THE ADDRRESS GENERATOR AND CACHE MEMORY CONNECTOR LIST. THIS CURRENT LIST FOR THE PROCESSOR DIFFERS BECAUSE EVA IS CAPABLE OF CASCADING MULTIPLE PROCESSOR BOARDS. HENCE, EACH BOARD MUST BE ISOLATED FROM THE OTHER PROCESSOR BOARDS.

CONNECTOR NET LIST FOR SECTION P1

PIN	NET	PIN	NET	PIN	NET	PIN	NET
A1	VCC	B1	VCC	C1	VCC	D1	VCC
A2	DATAAI15	B2	DATABI15	C2	DATAD15	D2	DATAEI15
A3	DATAAI14	B3	DATABI14	C3	DATAD14	D3	DATAEI14
A4	DATAAI13	B4	DATABI13	C4	DATAD13	D4	DATAEI13
A5	DATAAI12	B5	DATABI12	C5	DATAD12	D5	DATAEI12
A6	GND	B6	GND	C6	GND	D6	GND
A7	DATAAI11	B7	DATABI11	C7	DATAD11	D7	DATAEI11
A8	DATAAI10	B8	DATABI10	C8	DATAD10	D8	DATAEI10
A9	DATAAI9	B9	DATABI9	C9	DATAD9	D9	DATAEI9
A10	DATAAI8	B10	DATABI8	C10	DATAD8	D10	DATAEI8
A11	GND	B11	GND	C11	GND	D11	GND
A12	DATAAI7	B12	DATABI7	C12	DATAD7	D12	DATAEI7
A13	DATAAI6	B13	DATABI6	C13	DATAD6	D13	DATAEI6
A14	DATAAI5	B14	DATABI5	C14	DATAD5	D14	DATAEI5
A15	DATAAI4	B15	DATABI4	C15	DATAD4	D15	DATAEI4
A16	GND	B16	GND	C16	GND	D16	GND
A17	DATAAI3	B17	DATABI3	C17	DATAD3	D17	DATAEI3
A18	DATAAI2	B18	DATABI2	C18	DATAD2	D18	DATAEI2
A19	DATAAI1	B19	DATABI1	C19	DATAD1	D19	DATAEI1
A20	DATAAI0	B20	DATABI0	C20	DATAD0	D20	DATAEI0
A21	GND	B21	GND	C21	GND	D21	GND
A22	DATAAR15	B22	DATABR15	C22	DATAC15	D22	DATAER15
A23	DATAAR14	B23	DATABR14	C23	DATAC14	D23	DATAER14
A24	DATAAR13	B24	DATABR13	C24	DATAC13	D24	DATAER13
A25	DATAAR12	B25	DATABR12	C25	DATAC12	D25	DATAER12
A26	GND	B26	GND	C26	GND	D26	GND
A27	DATAAR11	B27	DATABR11	C27	DATAC11	D27	DATAER11
A28	DATAAR10	B28	DATABR10	C28	DATAC10	D28	DATAER10
A29	DATAAR9	B29	DATABR9	C29	DATAC9	D29	DATAER9
A30	DATAAR8	B30	DATABR8	C30	DATAC8	D30	DATAER8
A31	GND	B31	GND	C31	GND	D31	GND
A32	DATAAR7	B32	DATABR7	C32	DATAC7	D32	DATAER7
A33	DATAAR6	B33	DATABR6	C33	DATAC6	D33	DATAER6
A34	DATAAR5	B34	DATABR5	C34	DATAC5	D34	DATAER5
A35	DATAAR4	B35	DATABR4	C35	DATAC4	D35	DATAER4
A36	GND	B36	GND	C36	GND	D36	GND
A37	DATAAR3	B37	DATABR3	C37	DATAC3	D37	DATAER3
A38	DATAAR2	B38	DATABR2	C38	DATAC2	D38	DATAER2
A39	DATAAR1	B39	DATABR1	C39	DATAC1	D39	DATAER1
A40	DATAAR0	B40	DATABR0	C40	DATAC0	D40	DATAER0
A41	GND	B41	GND	C41	GND	D41	GND
A42	VCC	B42	VCC	C42	VCC	D42	VCC

CONNECTOR NET LIST FOR SECTION P2

PIN	NET	PIN	NET	PIN	NET	PIN	NET
A1	VCC	B1	VCC	C1	VCC	D1	VCC
A2	ADDAC35	B2	ADDAC23	C2	ADDAC11	D2	CLK1
A3	ADDAC34	B3	ADDAC22	C3	ADDAC10	D3	CLK3
A4	ADDAC33	B4	ADDAC21	C4	ADDAC9	D4	CLK2
A5	ADDAC32	B5	ADDAC20	C5	ADDAC8	D5	CLK4
A6	GND	B6	GND	C6	GND	D6	GND
A7	ADDAC31	B7	ADDAC19	C7	ADDAC7	D7	CLK
A8	ADDAC30	B8	ADDAC18	C8	ADDAC6	D8	/CLK
A9	ADDAC29	B9	ADDAC17	C9	ADDAC5	D9	/RESET
A10	ADDAC28	B10	ADDAC16	C10	ADDAC4	D10	/COMD
A11	GND	B11	GND	C11	GND	D11	GND
A12	ADDAC27	B12	ADDAC15	C12	ADDAC3	D12	ADDM0
A13	ADDAC26	B13	ADDAC14	C13	ADDAC2	D13	ADDM1
A14	ADDAC25	B14	ADDAC13	C14	ADDAC1	D14	ADDM2
A15	ADDAC24	B15	ADDAC12	C15	ADDAC0	D15	ADDM3
A16	GND	B16	GND	C16	GND	D16	GND
A17	ADDBD35	B17	ADDBD23	C17	ADDBD11	D17	ADDM4
A18	ADDBD34	B18	ADDBD22	C18	ADDBD10	D18	ADDM5
A19	ADDBD33	B19	ADDBD21	C19	ADDBD9	D19	ADDM6
A20	ADDBD32	B20	ADDBD20	C20	ADDBD8	D20	ADDM7
A21	GND	B21	GND	C21	GND	D21	GND
A22	ADDBD31	B22	ADDBD19	C22	ADDBD7	D22	ADDM8
A23	ADDBD30	B23	ADDBD18	C23	ADDBD6	D23	ADDM9
A24	ADDBD29	B24	ADDBD17	C24	ADDBD5	D24	ADDM10
A25	ADDBD28	B25	ADDBD16	C25	ADDBD4	D25	ADDM11
A26	GND	B26	GND	C26	GND	D26	GND
A27	ADDBD27	B27	ADDBD15	C27	ADDBD3	D27	ADDM12
A28	ADDBD26	B28	ADDBD14	C28	ADDBD2	D28	ADDM13
A29	ADDBD25	B29	ADDBD13	C29	ADDBD1	D29	ADDM14
A30	ADDBD24	B30	ADDBD12	C30	ADDBD0	D30	ADDM15
A31	GND	B31	GND	C31	GND	D31	GND
A32	DATAAR35	B32	DATABR35	C32	DATAC35	D32	DATAER35
A33	DATAAR34	B33	DATABR34	C33	DATAC34	D33	DATAER34
A34	DATAAR33	B34	DATABR33	C34	DATAC33	D34	DATAER33
A35	DATAAR32	B35	DATABR32	C35	DATAC32	D35	DATAER32
A36	GND	B36	GND	C36	GND	D36	GND
A37	DATAAI35	B37	DATABI35	C37	DATAD35	D37	DATAEI35
A38	DATAAI34	B38	DATABI34	C38	DATAD34	D38	DATAEI34
A39	DATAAI33	B39	DATABI33	C39	DATAD33	D39	DATAEI33
A40	DATAAI32	B40	DATABI32	C40	DATAD32	D40	DATAEI32
A41	GND	B41	GND	C41	GND	D41	GND
A42	VCC	B42	VCC	C42	VCC	D42	VCC

Figure 48. Processor Connector List

CONNECTOR NET LIST FOR SECTION P3

PIN	NET	PIN	NET	PIN	NET	PIN	NET
A1	VCC	B1	VCC	C1	VCC	D1	VCC
A2	DATAAI31	B2	DATABI31	C2	DATAD31	D2	DATAEI31
A3	DATAAI30	B3	DATABI30	C3	DATAD30	D3	DATAEI30
A4	DATAAI29	B4	DATABI29	C4	DATAD29	D4	DATAEI29
A5	DATAAI28	B5	DATABI28	C5	DATAD28	D5	DATAEI28
A6	GND	B6	GND	C6	GND	D6	GND
A7	DATAAI27	B7	DATABI27	C7	DATAD27	D7	DATAEI27
A8	DATAAI26	B8	DATABI26	C8	DATAD26	D8	DATAEI26
A9	DATAAI25	B9	DATABI25	C9	DATAD25	D9	DATAEI25
A10	DATAAI24	B10	DATABI24	C10	DATAD24	D10	DATAEI24
A11	GND	B11	GND	C11	GND	D11	GND
A12	DATAAI23	B12	DATABI23	C12	DATAD23	D12	DATAEI23
A13	DATAAI22	B13	DATABI22	C13	DATAD22	D13	DATAEI22
A14	DATAAI21	B14	DATABI21	C14	DATAD21	D14	DATAEI21
A15	DATAAI20	B15	DATABI20	C15	DATAD20	D15	DATAEI20
A16	GND	B16	GND	C16	GND	D16	GND
A17	DATAAI19	B17	DATABI19	C17	DATAD19	D17	DATAEI19
A18	DATAAI18	B18	DATABI18	C18	DATAD18	D18	DATAEI18
A19	DATAAI17	B19	DATABI17	C19	DATAD17	D19	DATAEI17
A20	DATAAI16	B20	DATABI16	C20	DATAD16	D20	DATAEI16
A21	GND	B21	GND	C21	GND	D21	GND
A22	DATAAR31	B22	DATABR31	C22	DATAC31	D22	DATAER31
A23	DATAAR30	B23	DATABR30	C23	DATAC30	D23	DATAER30
A24	DATAAR29	B24	DATABR29	C24	DATAC29	D24	DATAER29
A25	DATAAR28	B25	DATABR28	C25	DATAC28	D25	DATAER28
A26	GND	B26	GND	C26	GND	D26	GND
A27	DATAAR27	B27	DATABR27	C27	DATAC27	D27	DATAER27
A28	DATAAR26	B28	DATABR26	C28	DATAC26	D28	DATAER26
A29	DATAAR25	B29	DATABR25	C29	DATAC25	D29	DATAER25
A30	DATAAR24	B30	DATABR24	C30	DATAC24	D30	DATAER24
A31	GND	B31	GND	C31	GND	D31	GND
A32	DATAAR23	B32	DATABR23	C32	DATAC23	D32	DATAER23
A33	DATAAR22	B33	DATABR22	C33	DATAC22	D33	DATAER22
A34	DATAAR21	B34	DATABR21	C34	DATAC21	D34	DATAER21
A35	DATAAR20	B35	DATABR20	C35	DATAC20	D35	DATAER20
A36	GND	B36	GND	C36	GND	D36	GND
A37	DATAAR19	B37	DATABR19	C37	DATAC19	D37	DATAER19
A38	DATAAR18	B38	DATABR18	C38	DATAC18	D38	DATAER18
A39	DATAAR17	B39	DATABR17	C39	DATAC17	D39	DATAER17
A40	DATAAR16	B40	DATABR16	C40	DATAC16	D40	DATAER16
A41	GND	B41	GND	C41	GND	D41	GND
A42	VCC	B42	VCC	C42	VCC	D42	VCC

Figure 48. Processor Connector List Continued

PARTIAL LIST FOR BACKPLANE BITS
ADDRESS PORT E AND SOME CONTROL LINES MUST BE ADDED WHEN TIMING
DESIGN COMPLETED FOR THE BSIO BUS

CONNECTOR NET LIST FOR SECTION P1

PIN	NET	PIN	NET	PIN	NET	PIN	NET
A1	VCC	B1	VCC	C1	VCC	D1	VCC
A2	DATAAI15	B2	DATABI15	C2	DATAD15	D2	DATAEI15
A3	DATAAI14	B3	DATABI14	C3	DATAD14	D3	DATAEI14
A4	DATAAI13	B4	DATABI13	C4	DATAD13	D4	DATAEI13
A5	DATAAI12	B5	DATABI12	C5	DATAD12	D5	DATAEI12
A6	GND	B6	GND	C6	GND	D6	GND
A7	DATAAI11	B7	DATABI11	C7	DATAD11	D7	DATAEI11
A8	DATAAI10	B8	DATABI10	C8	DATAD10	D8	DATAEI10
A9	DATAAI9	B9	DATABI9	C9	DATAD9	D9	DATAEI9
A10	DATAAI8	B10	DATABI8	C10	DATAD8	D10	DATAEI8
A11	GND	B11	GND	C11	GND	D11	GND
A12	DATAAI7	B12	DATABI7	C12	DATAD7	D12	DATAEI7
A13	DATAAI6	B13	DATABI6	C13	DATAD6	D13	DATAEI6
A14	DATAAI5	B14	DATABI5	C14	DATAD5	D14	DATAEI5
A15	DATAAI4	B15	DATABI4	C15	DATAD4	D15	DATAEI4
A16	GND	B16	GND	C16	GND	D16	GND
A17	DATAAI3	B17	DATABI3	C17	DATAD3	D17	DATAEI3
A18	DATAAI2	B18	DATABI2	C18	DATAD2	D18	DATAEI2
A19	DATAAI1	B19	DATABI1	C19	DATAD1	D19	DATAEI1
A20	DATAAI0	B20	DATABI0	C20	DATAD0	D20	DATAEI0
A21	GND	B21	GND	C21	GND	D21	GND
A22	DATAAR15	B22	DATABR15	C22	DATAC15	D22	DATAER15
A23	DATAAR14	B23	DATABR14	C23	DATAC14	D23	DATAER14
A24	DATAAR13	B24	DATABR13	C24	DATAC13	D24	DATAER13
A25	DATAAR12	B25	DATABR12	C25	DATAC12	D25	DATAER12
A26	GND	B26	GND	C26	GND	D26	GND
A27	DATAAR11	B27	DATABR11	C27	DATAC11	D27	DATAER11
A28	DATAAR10	B28	DATABR10	C28	DATAC10	D28	DATAER10
A29	DATAAR9	B29	DATABR9	C29	DATAC9	D29	DATAER9
A30	DATAAR8	B30	DATABR8	C30	DATAC8	D30	DATAER8
A31	GND	B31	GND	C31	GND	D31	GND
A32	DATAAR7	B32	DATABR7	C32	DATAC7	D32	DATAER7
A33	DATAAR6	B33	DATABR6	C33	DATAC6	D33	DATAER6
A34	DATAAR5	B34	DATABR5	C34	DATAC5	D34	DATAER5
A35	DATAAR4	B35	DATABR4	C35	DATAC4	D35	DATAER4
A36	GND	B36	GND	C36	GND	D36	GND
A37	DATAAR3	B37	DATABR3	C37	DATAC3	D37	DATAER3
A38	DATAAR2	B38	DATABR2	C38	DATAC2	D38	DATAER2
A39	DATAAR1	B39	DATABR1	C39	DATAC1	D39	DATAER1
A40	DATAAR0	B40	DATABR0	C40	DATAC0	D40	DATAER0
A41	GND	B41	GND	C41	GND	D41	GND
A42	VCC	B42	VCC	C42	VCC	D42	VCC

CONNECTOR NET LIST FOR SECTION P2

PIN	NET	PIN	NET	PIN	NET	PIN	NET
A1	VCC	B1	VCC	C1	VCC	D1	VCC
A2		B2		C2	ADDAC11	D2	CLK1
A3		B3		C3	ADDAC10	D3	CLK3
A4		B4		C4	ADDAC9	D4	CLK2
A5		B5		C5	ADDAC8	D5	CLK4
A6	GND	B6	GND	C6	GND	D6	GND
A7		B7		C7	ADDAC7	D7	CLK
A8		B8		C8	ADDAC6	D8	/CLK
A9		B9		C9	ADDAC5	D9	/RESET
A10		B10		C10	ADDAC4	D10	/COMD
A11	GND	B11	GND	C11	GND	D11	GND
A12		B12	ADDAC15	C12	ADDAC3	D12	ADDM0
A13		B13	ADDAC14	C13	ADDAC2	D13	ADDM1
A14		B14	ADDAC13	C14	ADDAC1	D14	ADDM2
A15		B15	ADDAC12	C15	ADDAC0	D15	ADDM3
A16	GND	B16	GND	C16	GND	D16	GND
A17		B17		C17	ADDBD11	D17	ADDM4
A18		B18		C18	ADDBD10	D18	ADDM5
A19		B19		C19	ADDBD9	D19	ADDM6
A20		B20		C20	ADDBD8	D20	ADDM7
A21	GND	B21	GND	C21	GND	D21	GND
A22		B22		C22	ADDBD7	D22	ADDM8
A23		B23		C23	ADDBD6	D23	ADDM9
A24		B24		C24	ADDBD5	D24	ADDM10
A25		B25		C25	ADDBD4	D25	ADDM11
A26	GND	B26	GND	C26	GND	D26	GND
A27		B27	ADDBD15	C27	ADDBD3	D27	ADDM12
A28		B28	ADDBD14	C28	ADDBD2	D28	ADDM13
A29		B29	ADDBD13	C29	ADDBD1	D29	ADDM14
A30		B30	ADDBD12	C30	ADDBD0	D30	ADDM15
A31	GND	B31	GND	C31	GND	D31	GND
A32	DATAAR35	B32	DATABR35	C32	DATAC35	D32	DATAER35
A33	DATAAR34	B33	DATABR34	C33	DATAC34	D33	DATAER34
A34	DATAAR33	B34	DATABR33	C34	DATAC33	D34	DATAER33
A35	DATAAR32	B35	DATABR32	C35	DATAC32	D35	DATAER32
A36	GND	B36	GND	C36	GND	D36	GND
A37	DATAAI35	B37	DATABI35	C37	DATAD35	D37	DATAEI35
A38	DATAAI34	B38	DATABI34	C38	DATAD34	D38	DATAEI34
A39	DATAAI33	B39	DATABI33	C39	DATAD33	D39	DATAEI33
A40	DATAAI32	B40	DATABI32	C40	DATAD32	D40	DATAEI32
A41	GND	B41	GND	C41	GND	D41	GND
A42	VCC	B42	VCC	C42	VCC	D42	VCC

Figure 49. Cache/Address Generator Connector Lists

CONNECTOR NET LIST FOR SECTION P3

PIN	NET	PIN	NET	PIN	NET	PIN	NET
A1	VCC	B1	VCC	C1	VCC	D1	VCC
A2	DATAAI31	B2	DATABI31	C2	DATAD31	D2	DATAEI31
A3	DATAAI30	B3	DATABI30	C3	DATAD30	D3	DATAEI30
A4	DATAAI29	B4	DATABI29	C4	DATAD29	D4	DATAEI29
A5	DATAAI28	B5	DATABI28	C5	DATAD28	D5	DATAEI28
A6	GND	B6	GND	C6	GND	D6	GND
A7	DATAAI27	B7	DATABI27	C7	DATAD27	D7	DATAEI27
A8	DATAAI26	B8	DATABI26	C8	DATAD26	D8	DATAEI26
A9	DATAAI25	B9	DATABI25	C9	DATAD25	D9	DATAEI25
A10	DATAAI24	B10	DATABI24	C10	DATAD24	D10	DATAEI24
A11	GND	B11	GND	C11	GND	D11	GND
A12	DATAAI23	B12	DATABI23	C12	DATAD23	D12	DATAEI23
A13	DATAAI22	B13	DATABI22	C13	DATAD22	D13	DATAEI22
A14	DATAAI21	B14	DATABI21	C14	DATAD21	D14	DATAEI21
A15	DATAAI20	B15	DATABI20	C15	DATAD20	D15	DATAEI20
A16	GND	B16	GND	C16	GND	D16	GND
A17	DATAAI19	B17	DATABI19	C17	DATAD19	D17	DATAEI19
A18	DATAAI18	B18	DATABI18	C18	DATAD18	D18	DATAEI18
A19	DATAAI17	B19	DATABI17	C19	DATAD17	D19	DATAEI17
A20	DATAAI16	B20	DATABI16	C20	DATAD16	D20	DATAEI16
A21	GND	B21	GND	C21	GND	D21	GND
A22	DATAAR31	B22	DATABR31	C22	DATAC31	D22	DATAER31
A23	DATAAR30	B23	DATABR30	C23	DATAC30	D23	DATAER30
A24	DATAAR29	B24	DATABR29	C24	DATAC29	D24	DATAER29
A25	DATAAR28	B25	DATABR28	C25	DATAC28	D25	DATAER28
A26	GND	B26	GND	C26	GND	D26	GND
A27	DATAAR27	B27	DATABR27	C27	DATAC27	D27	DATAER27
A28	DATAAR26	B28	DATABR26	C28	DATAC26	D28	DATAER26
A29	DATAAR25	B29	DATABR25	C29	DATAC25	D29	DATAER25
A30	DATAAR24	B30	DATABR24	C30	DATAC24	D30	DATAER24
A31	GND	B31	GND	C31	GND	D31	GND
A32	DATAAR23	B32	DATABR23	C32	DATAC23	D32	DATAER23
A33	DATAAR22	B33	DATABR22	C33	DATAC22	D33	DATAER22
A34	DATAAR21	B34	DATABR21	C34	DATAC21	D34	DATAER21
A35	DATAAR20	B35	DATABR20	C35	DATAC20	D35	DATAER20
A36	GND	B36	GND	C36	GND	D36	GND
A37	DATAAR19	B37	DATABR19	C37	DATAC19	D37	DATAER19
A38	DATAAR18	B38	DATABR18	C38	DATAC18	D38	DATAER18
A39	DATAAR17	B39	DATABR17	C39	DATAC17	D39	DATAER17
A40	DATAAR16	B40	DATABR16	C40	DATAC16	D40	DATAER16
A41	GND	B41	GND	C41	GND	D41	GND
A42	VCC	B42	VCC	C42	VCC	D42	VCC

Figure 49. Cache/Address Generator Connector Lists Continued

4.0 Microprogramming the CPH

Microprogramming the CPH is done with the microassembler provided using MicroAsm. Here, a user would develop an assembly level program with the MicroAsm assembler syntax. A predefined description of the CPH has been entered into the Genasm files. A typical production of the microcode for the assembly level application program uses the following command line.

```
Microasm mulm.asm -cph -f
```

This command line uses the predefined machine definition tables of the cph file and generates the microcode for the mulm.asm assembly level code. Output will be in a file labeled as "mulm.ldf".

4.1 Theory of Operation

Generating microprograms for the CPH requires the MICROASM retargetable microassembler. There are three programs entitled, GENASM, MICROASM, and MPP. These three executable files should be in the current directory you are writing the assembly level programs. As an example, the following sequence of steps are necessary to produce a binary file for the machine. That output file will have the root name of your source and the extension, "LDF".

4.1.1 Sequence of Steps

To create and assemble a program, two steps are necessary as follows:

1. Create your assembly level program with any text editor. Save as an ASCII file only.

2. Keystroke the following command line

```
MICROASM <YOUR FILE NAME.ASM> -tCPH -f
```

This is the entire sequence. This example uses the already developed tables for the CPH which should be in your directory. The "-f" string tells MicroAsm to produce a binary output PROM file with the root name of your assembly program.

4.1.1.1 An Example

On the disk provided are 18 files, including Microasm.exe, Genasm.exe, MPP.exe, CPH.FIX, MULM.ASM, MULM.LDF, DAFY.FIX, and DAFY.LDF. To produce a PROM readable file in binary from the MULM.ASM assembly program, type the following:

```
MICROASM MULM.ASM -tCPH -f
```

This command line will assemble the program called MULM.ASM, using the machine description found in the CPH.FIX files and produce MULM.LDF. After completing the steps, examine the MULM.LDF file. It should have four microinstructions of 768 bits width. The source program, MULM.ASM, is found in the appendix along with the MULM.LDF and CPH.FIX machine description file. Verify that the micro orders in the LDF file agree with your syntax in MULM.ASM.

4.1.1.2 The LDF files

LDF files are produced by appending in the Microasm command line the symbols "-f". The output file will have the same root name as the ASM file but will have the LDF extension. This file is used to produce the PROM words. This LDF file can be viewed to verify the bits in each microorder selected by your assembly program. For example, an AAA.LDF. file was created from the AAA.ASM file in your example section. It is two microinstructions long. The very first bit in the upper left corner is physical bit 768. The lower rightmost bit is physical bit 1. The most significant 384 bits represent phase 1 microorders in each microinstruction while the least 384 bits represent the phase 0 microorders. To locate individual fields requires you to compare the MI format drawing with the LDF file. Be careful. Some of the fields are spread across isolated physical bits. The immediate address field is one. ADDRESS RAM1 is another. There is potential for confusion in several areas. These are clarified in the sections below.

4.1.2.1 Default Bits

In order to avoid having to specify all bits of a microinstruction in each assembly instruction, default values are specified in the CPH description. There is a default value for each of the fields as well as for each subfield of each field. There is also a global default bit value specified with the defbit directive that is used when the proper default is not available. Since all fields and subfields in the CPH description have defaults specified, this global default bit will never be used.

When a field is not specified at all in an instruction (no $<field_name>), then the default for the entire field is used. If there is no default for the entire field, the global default bit value is used instead. When the field is specified but a subfield is left out, either between commas or at the end, the default for the subfield is used. If there is no default for the subfield, the global default bit is used again rather than the field default. Any or all of the subfields can be left out and they will be replaced with the subfield defaults. For most of the fields, the default values are the same in the field as in the subfields. The exceptions are the $CCS, $IMM and $MWR fields. The $SEQ field is also unusual because assignments to the physical bits have been made from its subfields rather than the entire field. For that reason, the $SEQ field default has no effect and the $SEQ field must be specified in an instruction to keep it from getting a "don't care" value. It need not be given any subfield values, as they will default to a continue instruction, but a $SEQ must be present. Physical fields which are not assigned any bit values at all will get "don't care" values.

4.1.2.2 Immediate Data

To use the immediate field, it is necessary to specify $IMM or $IMM EN (DISable is the default value for the field, but ENable is the default value for the subfield). The data value is held in the $REG field and must be specified by filling in each of the subfields of the $REG field with the appropriate number of bits from its binary representation. For example, to specify the value

0B000011111000011110000111100001110000

would require

$REG 0X01,0X38,0X0F,0X03,0X3,0X03,0X30

Use only hex or octal format in Microasm. Do not use binary. This is inconvenient, but the immediate field should not be needed very often anyway.

The immediate address field can also be used to send literal addresses to the program counter. It is done similarly. For example, the microorder $IMMADD 0xFFFF will emit the bits, 0B111111111111 in the immediate address field.

4.1.2.3 CPH ROM Format

When assembling microcode for the CPH, the format shown in Figure 50 applies. An MI word is 384-bits long partitioned into 8 ROMs. A single MI is mapped as shown across several physical devices. Care must be exercised in downloading the code from the host so that the words map accordingly.

4.2 Algorithms

Severe computational requirements are placed upon WSMR radar and telemetry installations when multiple sensing and unreliable data acquisition occurs. Decentralized tracking via the new Square Root Information Filter (SRIF) offers exceptional promises. SRIFs easily handle sensor misalignment, adapting to unexpected randomness, and noisy telemetry. The optimal tracker, however, must be computationally efficient and fast. The tracker must also correlate multiple objects with measurements, requiring the tracking filter to be run on different sequences of measurements. To be reliable, the tracker must be numerically stable under extremely tight real time constraints. Figure 51 entitled, "Decentralized SRIF Architecture" depicts the typical processing chain and Figure 52 depicts the distributed/parallel architecture for combining local processors into the decentralized tracker scheme.

Both the CPH and the VPH boards can serve as the local processor for the SRIF. Where significant vector operations are required, the VPH excels in real-time performance. When significant matrix manipulations occur, the CPH is the better choice. It is anticipated that the major computational task is the matrix inversion which is highly sensitive to the ill-condition of the matrix. Matrix ill-conditioning can be quantified by the Mel-Penrose index. This index is the absolute value of the difference between the largest eigenvalue and the smallest eigenvalue. In practical terms, this index is a measure of the difference between the largest energy signal and the smallest energy signal.

Matrix inversion can be accomplished by LU factorization, Gaussian elimination, Gram-Schmidt Factorization, Hermitian matrix inversion, and scaled Givens rotations, general matrix inversion.

CPH ROM FORMAT
Each column represents a single x8 ROM
January 27, 1992

	ROM 7	ROM 6	ROM 5	ROM 4	ROM 3	ROM 2	ROM 1	ROM 0	ROM ADDR	SYSTEM	BANK ADDR	PHASE	RAM ADDR
INSTRUCTION 0	383..376	375..368	367..360	359..352	351..344	343..336	335..328	327..320	0	0	5	0	0
	383..376	375..368	367..360	359 352	351..344	343..336	335..328	327..320	1	0	5	1	0
	319..312	311..304	303..296	295..288	287..280	279..272	271..264	263..256	2	0	4	0	0
	319..312	311..304	303..296	295..288	287..280	279..272	271..264	263..256	3	0	4	1	0
	255..248	247..240	239..232	231..224	223..216	215..208	207..200	199..192	4	0	3	0	0
	255..248	247..240	239..232	231..224	223..216	215..208	207..200	199..192	5	0	3	1	0
	191..184	183..176	175..168	167..160	159..152	151..144	143..136	135..128	6	0	2	0	0
	191..184	183..176	175..168	167..160	159..152	151..144	143..136	135..128	7	0	2	1	0
	127..120	119..112	111..104	103...96	95....88	87....80	79....72	71....64	8	0	1	0	0
	127..120	119..112	111..104	103...96	95....88	87....80	79....72	71....64	9	0	1	1	0
	63....56	55....48	47....40	39....32	31....24	23....16	15.....8	7......0	10	0	0	0	0
	63....56	55....48	47....40	39....32	31....24	23....16	15.....8	7......0	11	0	0	1	0
INSTRUCTION 1	383..376	375..368	367..360	359..352	351..344	343..336	335..328	327..320	12	0	5	0	1
	383..376	375..368	367..360	359..352	351..344	343..336	335..328	327..320	13	0	5	1	1
	319..312	311..304	303..296	295..288	287..280	279..272	271..264	263..256	14	0	4	0	1
	319..312	311..304	303..296	295..288	287..280	279..272	271..264	263..256	15	0	4	1	1
	255..248	247..240	239..232	231..224	223..216	215..208	207..200	199..192	16	0	3	0	1
	255..248	247..240	239..232	231..224	223..216	215..208	207..200	199..192	17	0	3	1	1
	191..184	183..176	175..168	167..160	159..152	151..144	143..136	135..128	18	0	2	0	1
	191..184	183..176	175..168	167..160	159..152	151..144	143..136	135..128	19	0	2	1	1
	127..120	119..112	111..104	103...96	95....88	87....80	79....72	71....64	20	0	1	0	1
	127..120	119..112	111..104	103...96	95....88	87....80	79....72	71....64	21	0	1	1	1
	63....56	55....48	47....40	39....32	31....24	23....16	15.....8	7......0	22	0	0	0	1
	63....56	55....48	47....40	39....32	31....24	23....16	15.....8	7......0	23	0	0	1	1
	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .	. . .

Figure 50. CPH ROM Format

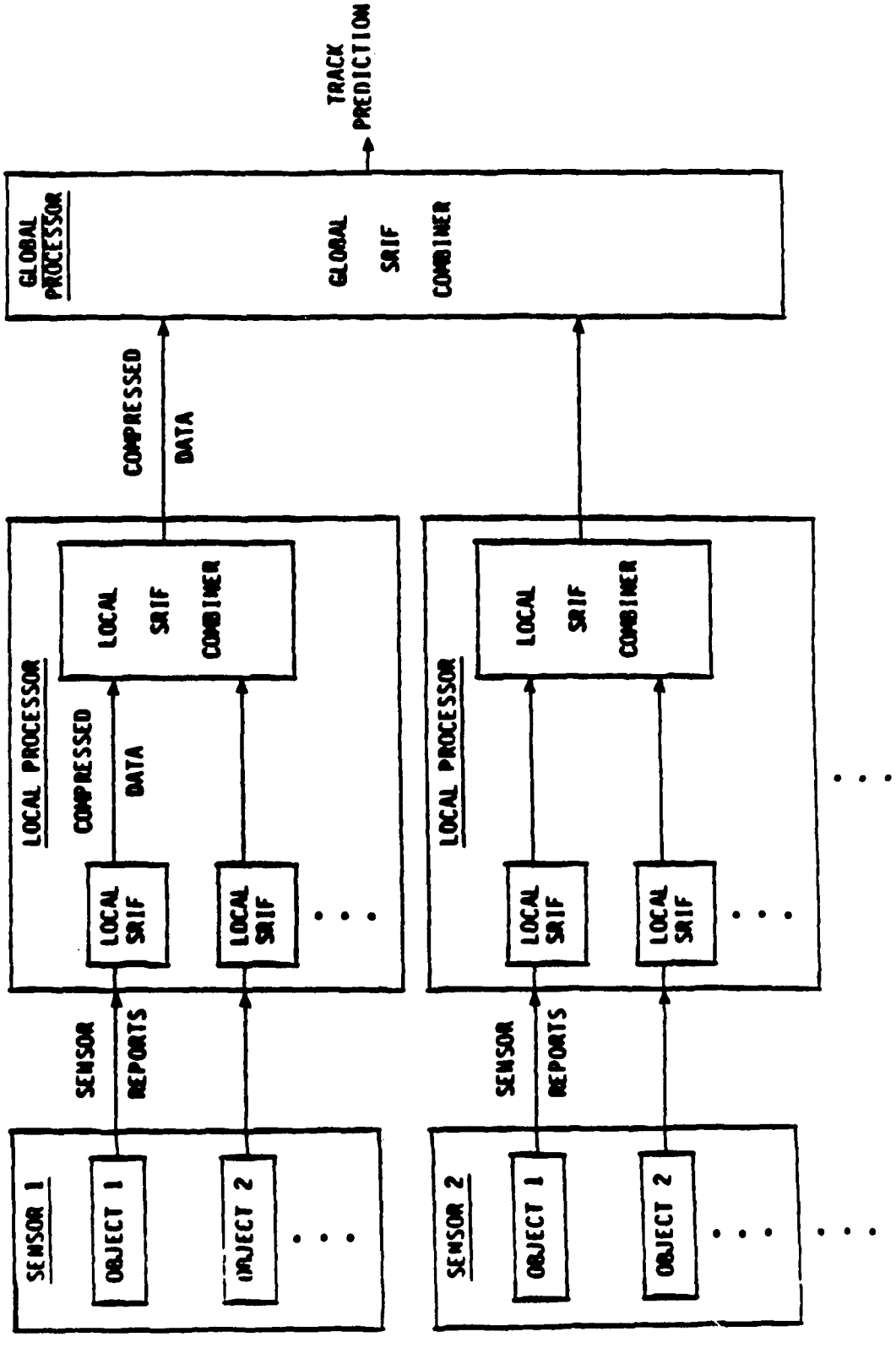
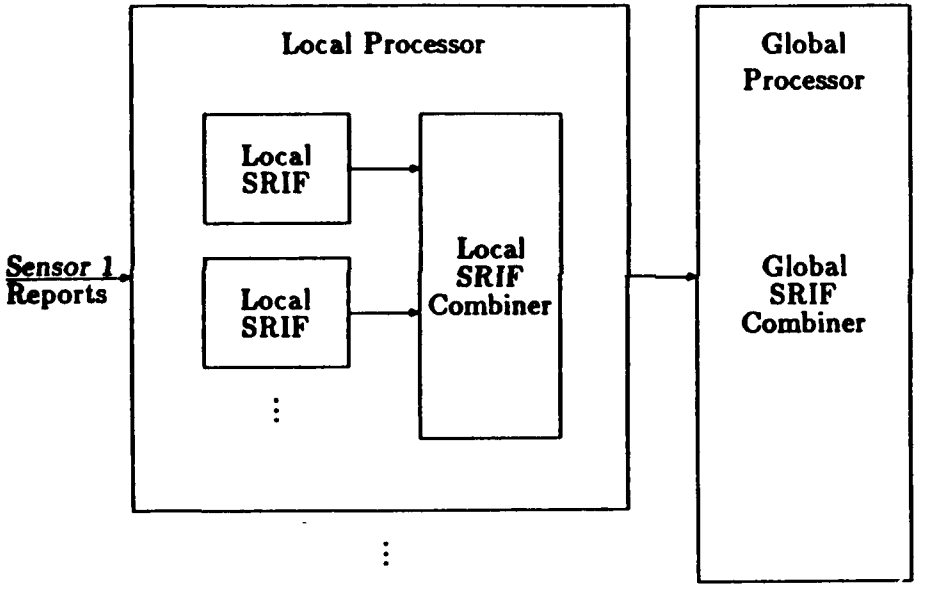


Figure 51. Decentralized SRIF Architecture

Decentralized Tracking via New Square Root Information Filter (SRIF) Concepts

Distributed/Parallel Architecture



Objectives

- Globally distributed with minimal communication.
- Locally parallel.
- Adapt to maneuvers without reprocessing data.
- Optimal structure.
- Reliable numerically stable form.

Accomplishments

- Algorithm development qualitatively meeting all objectives completed.
- High fidelity simulation demonstrating quantitative performance in progress.

Payoff

Technical breakthrough offering potential for dramatic improvement in meeting tracking needs with relaxed sensor and hardware requirements.

Figure 52. Distributed/Parallel Architecture

The computational budget for a complete SRIF is the following:

SRIF Computational Budget

Matrix Inversion	40%
Vector Multiplication	26%
Correlation	14%
Numerical Integration	6%
Scalar Manipulation	14%

The major tasks include adaptive tracking, nonlinear filtering, batch initialization, sensor control, and track correlation. Adaptive tracking can be accomplished via several methods some of which are listed in Figure 53 entitled "Adaptive Algorithms". They include the LMS, RLS, FLA, FTF, and SFTF. Note that the LMS is a slow tracker but its computational complexity (number of equivalent multiplication). The SFTF is fast but its computational complexity is 4.5 times worse than the LMS. The FTF is not stable. Therefore it is not suitable for the SRIF or the EVA architecture.

During April 1990, a new algorithm was investigated for the time motion resolution task at WSMR, because this is a very demanding application and time consuming to WSMR. It was found that the new algorithm could improve and enhance signal analysis of signals which are both time and frequency limited without the need for long windows as is required when using the Fourier transform. Because this new algorithm, called the Wigner-Ville transform, has significant improvements over the Fourier transform, an intensive analysis of its features was made and applied to the CPH. The CPH as currently configured appears to support this important new discovery.

The Mentor target tracker algorithms (a realization of DSRIF) were also examined carefully for implementation into either or both the VPH and CPH. The basic sequence of steps in the computations is as follows:

1. Take measurements (range, rate,...)

2. Execute local filters in parallel

3. Merge 1 at the global level

4. Local filter time update

5. Global merge

However, additional equations need to be computed in order to support steps 1 through 5. All matrices appear to be less than 25 x 25 elements in size. There are no real-time matrix inversion operations. One inversion is needed at the onset, however. Several orthogonal transformations are needed but appear to be straightforward. Givens rotations were suggested by Dr. Mitch Belza for some matrix manipulations.

- LMS : stochastic gradient algorithm (Least Mean Squares)
  RLS : ordinary Recursive Least-Squares algorithm
  FLA : fast RLS algorithm in lattice form
  FTF : fast RLS algorithm in transversal filter form
  SFTF: numericallv stable FTF

- LMS $\leftrightarrow$ RLS : RLS generally has faster *tracking* characteristics than LMS. The requirement of convergence imposes a bound on the gain of the LMS algorithm (bound depends on eigenvalue spread of autocorrelation matrix of $\boldsymbol{x}$).

- numerical stability : stability of the error propagation system

	problem solved	tracking speed	computational complexity (×)	numerical stability
LMS	LMS	slow	$2N$	exponential
RLS	RLS	fast	$2N^2 + 6N$	exponential
FLA	RLS	fast	$14N\,(\times) + 2N\,(\div) = 30N\,(\times)$	exponential
FTF	RLS	fast	$7N$	unstable
SFTF	RLS	fast	$9N$	exponential

4.2.1 Algorithms for Solving Linear Systems

STC's design review of the VPH, with respect to providing a full range of math functions, has yielded a healthy respect of its calculation capabilities. The VPH has 4 separate calculation units which can run in parallel, each of which can perform a square root in approximately 1.52 microseconds and a division in less than 1 microsecond. While these figures are not the fastest figures in the world, they are very respectable when viewed in the context of the architecture's main function, FFTs, which require complex multiply accumulates. This speed and flexibility allows the architecture to provide a wealth of processing speed which can be used for virtually any mathematical functions which might need to be performed. When the overall speed of the existing VPH architecture was compared with an architecture utilizing an additional processing unit such as the BIT chip, the cost to performance ratio of the speedup was very poor and the possible enhancement was discarded.

Many different algorithms solve matrix equations, and most of them rely on triangularizations of the input matrix. Triangularization is invariably followed by some sort of substitution to find the solution vector. Thus, the most efficient solutions are those which require the fewest calculations for their triangularization and subsequent backsubstitution. LU factorization and Gaussian elimination are now examined since they are important equation solvers.

4.2.2 LU Factorization

One effective method of solving a linear system Rw=s is to factor the coefficient matrix R into a product of two triangular matrices. The problem is then reduced to solving two triangular systems. The LU factorization produces a lower triangle matrix L, and an upper triangle matrix U, whose product is the original matrix: LU=R. This factorization is computationally simple because it consists primarily of inner product calculations. Once a factorization is found, the solution is simply a set of backsubstitutions.

In recent years, the LU decomposition has not received much attention, both because it is not very suitable for systolic array implementation, and because it is already so well known. However, because so much is known about it, and since the proposed implementation is a pipeline rather than an array, the LU algorithm appears to be the best solution.

4.2.3 Gaussian Elimination

Despite origins that date from at least 250 B.C., elimination methods are still viable as solution vehicles for linear equations. Gaussian elimination is widely know, being the primary method taught in introductory linear systems courses. The algorithm consists of a series of row interchanges (called pivots), combined with subtraction of matrix elements. It forms an upper triangle matrix by eliminating elements in the lower triangle of the coefficient matrix. The computational complexity of Gaussian elimination is identical to that of LU decomposition; in fact, if a specific pivoting strategy is followed, both methods will compute with the same accuracy.

4.2.4 Gram-Schmidt Decomposition

Another elimination method is the Gram-Schmidt algorithm which performs a Cholesky factorization on Hermitian positive-semidefinite matrices. Since a spatially distributed covariance matrix is Hermitian and positive-semidefinite, Gram-Schmidt is a valid algorithm for consideration. Since it is an elimination method, Gram-Schmidt operates like Gaussian elimination, first producing an upper triangle matrix, and then backsubstituting to find w. Unlike standard Cholesky factorization, the Gram-Schmidt method requires no square root calculations.

By 1990, researchers designed an array processor for adaptive beamforming based on the Gram-Schmidt algorithm. They replaced the reciprocal calculation with a shift, essentially the reciprocal of the nearest power of two. While this method avoids division, it solves a perturbed set of equations. Others were able to eliminate the divisions without disturbing the equations by generalizing the Gram-Schmidt method. Unfortunately, their method of eliminating the reciprocal tripled the number of multiplications.

4.2.5 Inversion of a Hermitian Matrix

Similar to the LU decomposition, inversion of a Hermitian matrix is much easier than inversion of an arbitrary matrix. First the matrix is triangularized, then the new matrix is formed by backsolving. The main difference between LU decomposition and Hermitian matrix inversion is the method of backsubstitution. Whereas LU decomposition reduces a triangular matrix down to a vector with O(N2) operations, the symmetric inversion expands a triangle matrix back to a full square matrix with O(N3) operations.

4.2.6 Scaled Givens Rotations

Despite a somewhat higher computational complexity, scaled Givens rotations have received much attention. The main advantages of this algorithm are:

1. easy implementation with a variety of parallel structures
2. flexibility to perform several matrix operations (e.g. singular value decomposition, diagonalization, and triangularization)
3. ability to compute plane rotations without square roots, and with half the multiplications of standard Givens rotations
4. high efficiency for sparse matrix operations
5. amenable to recursive least squares minimization techniques

Since these advantages have little effect on the solution of linear equations, we conclude that Givens rotations are more suitable for calculations other than a linear solution.

4.2.7 Comparison of Algorithms

Though all of the algorithms perform essentially the same operation, a determination of weight vector w, they are not equal in complexity. Table 1 gives a comparison of the number of operations (real multiplies, reciprocals, and additions) needed for each of the methods.

Table 1. Complexity of Solutions to Simultaneous Equations

Algorithm*	Number of Operations		Total for N=32
LU Factorisation[1]	Mult:	$2/3\ N^3 + 5N^2 - 7/3\ N$	27,040
	Recip:	N	32
	Add:	$2/3\ N^3 + 4N^2 - 2/3\ N$	25,920
Gaussian Elimination[1]	Mult:	$2/3\ N^3 + 5N^2 - 7/3\ N$	27,040
	Recip:	N	32
	Add:	$2/3\ N^3 + 4N^2 - 2/3\ N$	25,920
Gram-Schmidt Factorization[2] (and Division-Free Version[3])	Mult:	$2N^3 + 2N^2 - 4N$	67,456
		$(6N^3 - 2N^2 - 4N)$	(194,432)
	Recip:	N	32
		(0)	(0)
	Add:	$2N^3 + 2N^2 - 4N$	67,456
		$(4N^3 - 4N)$	(130,944)
Inversion of Hermitian matrix[4]	Mult:	$2N^3 + 11/2\ N^2 + 3/2\ N$	71,216
	Recip:	N	32
	Add:	$2N^3 + 4N^2 - 2N$	69,568
Scaled Givens Rotations[5]	Mult:	$8/3\ N^3 + 105/6\ N^2 + 89/6\ N$	105,776
	Recip:	$1/2\ N^2 - 1/2\ N$	496
	Add:	$8/3\ N^3 + 12N^2 + 28/3\ N$	99,968
General Matrix Inversion[1]	Mult:	$29/6\ N^3 + 3N^2 - 53/6\ N + 5$	161,173
	Recip:	N	32
	Add:	$29/6\ N^3 - 2N^2 - 11/6\ N + 5$	156,277

Table 1 shows the computational superiority of the LU factorization and Gaussian elimination methods, in terms of multiplications and additions. If reduction of divisions is the primary goal, then one of the Gram-Schmidt algorithms should be used. One can also see that matrix inversion is the most complex, and therefore the least desirable of the methods.

Because LU factorization is the fastest of the algorithms, and because Maron shows that it is easier to implement than Gaussian elimination, use of LU factorization is suggested. Our studies show that LU factorization is computationally simpler than other methods, and other publications recommend it as the optimum algorithm for solutions of simultaneous equations. For those reasons, implementations research currently focuses on efficient circuits for LU factorization.

Table 2 compares several least-squares computational techniques. The normal equations, Householder, Golub factorizations, standard Givens rotation, fast Givens rotation, scaled Givens rotation, and Gram-Schmidt methods are considered. Either the normal equations or the Householder Golub techniques require global communications. Additionally, these two techniques are sensitive to ill-conditioned matrices. Hence, the normal equations or the Householder Golub method are not amenable to systolic implementation. The Gram-Schmidt method, included for completeness, is not recursive and, therefore, is not considered for systolic implementation.

The remaining methods are based on the Givens rotation triangular decomposition. The standard Givens rotation requires pivoting as well as square-root computation. This slows the computation on systolic arrays. The square-root free Givens rotation eliminates the square-root computation but still requires pivoting. The scaled Givens rotation eliminates both the square-root computation and pivoting. Additionally, the scaled Givens rotation operates on matrix bands. It is not necessary to perform any computation on bands that contain only null elements. A computational savings is realized if the data matrix is in banded form. Note that the square-root free and scaled Givens rotations require half as many multiplies as the standard Givens rotation. The scaled Givens rotation only requires 1 division operation as opposed to 2 in the square-root free rotation. Apparently the scaled Givens rotation is superior to the other methods studied both in terms of computation speed and systolic implementation complexity.

Table 2. Weighted Least Squares Computational Methods

			Triangular Decompositions			
	Normal Equations	House-Holder Golub	Standard Givens Rotations	Fast Givens Rotations	Scaled Givens Rotations	Gram-Schmidt
Systolic Amenable	Non-nearest neighbor data paths	Requires global comm	Yes, but is slow and processor complex recursive separate back-substitution systolic array	If factored √ free operation, nearest neighbor pivoting increases data flow complexity	No pivots and √ free nearest neighbor comm	Not recursive
Additions/ Subtractions						
Mult./Stage			N	N/2	N/2	
Div./Stage			2		1	
Shifts/Im	Scal. Compl.	Scal. Compl.	Complex	Complex	2	
Latency				r+c+1		
Stable	Sensitive to Matrix Ill-Condition Number			Yes	Equiv. to Standard Givens	Well Cond.
Pivoting			2x1 Vector	2x1 Vector	None	
Fading Signal Capacity (Weighted)	Complex	Complex	Complex	Simple	Simple	
Row Removal	Complex	Complex	Complex	Complex	Simple	
Idle Processors			N/2	N/2	N/2	
Computation Time				2r+c+1	3m+ 3(q-1)+z+1	$^{1}0(m+z)$
Number of Processors				c(r+1)/2	q(w+z)	$0(w^2+zw)$

Table Notation: r -rows of rectilinear matrix, c - columns of rectilinear matrix, n - word length

4.2.8 VPH FFTs

A description of the FFT implemented on the VPH is now described. It serves as an introduction to the I/O compute overlap capabilities of the VPH and should be carefully studied. It will serve as the benchmark training program for the VPH. Hence, a full understanding of its operation is useful for future code development.

A 1024 point complex FFT is an ideal application for the VPH board. The Zoran DSP chips have the FFT coefficients in ROM for up to that size. In addition, a 1024 point FFT can be decomposed into two waves of thirty-two 32 point FFTs, each wave performing five of the ten passes required. Though the chips are capable of 64 point FFTs in a single instruction, processing 32 points at a time is more efficient when multiple FFTs are required. This is due to the ability of the chips to process data in half of the on-board RAM while transferring data between external memory and the other half of the on-board RAM. Since storing processed data and loading new data takes less than half the time that an FFT operation does, they can effectively be done for free even when sharing a bus between two chips working on the problem simultaneously.

The problem is very amenable to parallel use of all four DSP chips at once. Each chip can perform eight of the thirty-two FFT operations in each wave. The only time synchronization is needed between the processors is between waves. During each wave, each processor works with a distinct subset of the points. However, the points have to be redistributed among the processors between waves, so it is necessary to ensure that all of them finish the first wave before the second one starts. This inherent parallelism in the algorithm means that there is very little overhead required. The initial load and final store operations cannot be pipelined with the FFT operation and the parallel version has four times as many of these. They will also occur at almost the same time for the two processors sharing a bus, resulting in half the speed. These factors should have only about a 10% effect on the execution time. The VPH board should therefore be able to perform a 1024 point FFT almost four times as fast as a system with a single Zoran DSP chip.

The actual code works as follows. First the processors clear their semaphore flags to indicate that they are working. They then load their mode register with values that indicate that the internal RAM is to be divided into two banks and that bank references are to be inverted each time the loop counter is decremented. Then the two index register are set to point to the locations for incoming and outgoing data. In the current code, the first wave is done in place so they point to the same locations. A single index register could be used, but using both makes it easier to change to using a different location for the outgoing data if desired. The index registers on each processor are initialized to values offset by eight from the previous processor. This allows for each processor performing eight FFTs. The loop counter register is initialized to perform the seven fully pipelined iterations. The first set of data points are loaded from locations spaced 32 elements apart, as required by the FFT algorithm being used when the input data is in sequential order. Each subsequent set of data points will be loaded from a location one element after this one, so that after eight sets on each processor, all points will have been processed. Seven of the eight sets are handled in a loop that loads a set into the unused RAM bank, starts an

FFT, and then begins storing the results from the previous bank. After the loop, the final data set is stored. Outgoing data is stored in bit-reversed order to compensate for the reversal that occurs during the FFT calculation.

When each processor finishes the first wave, it uses one of its status bits to indicate that fact. The 68020 or one of the DSP chips designated to be master performs a full or partial handshaking operation using one of its own status bits to synchronize the end of the first wave and the start of the second. In the second wave, the FFTs are performed on sets of 32 adjacent elements. Each DSP chip again handles eight adjacent sets. The output results must be put in a separate output area this time because they are stored with a spacing of 32 again, instead of the spacing of one that the input is loaded from. This change in spacing performs a bit-reversal between the bits used to index the first and second waves, just as reversing each of the blocks during the store operations performs a bit-reversal of the index within a wave. This results in the output being in normal order instead of bit-reversed order. Each set is processed with an offset into the coefficient table to provide the correct value to account for it being part of a larger FFT. With these differences, the second wave is performed in the same manner as the first. When all processors indicate that they are finished with the second wave, the 1024 point FFT is complete.

The entire operation should take 133 clock cycles for the initial load and final store of each wave, doubled for the bus sharing, plus 334 cycles for each 32 point FFT. Allowing some extra time for synchronization, the entire FFT should take around 475 microseconds with a 25 MHz clock. This compares with a benchmark from Zoran of 1732 microseconds for a single chip.

4.2.9 VPH Software Conventions

In order to allow the software modules on the VPH board to work together properly, conventions must be established for their interaction. This is particularly important because the VPH has multiple processing elements that need to interact. The board provides a number of mechanisms for communication between these elements. Setting conventions for how they will be used is necessary for consistency.

VPH Resources

The processing elements on the VPH board are four Zoran VSP (Vector Signal Processor) chips and a Motorola 68020 microprocessor. A VME bus interface also allows an external processor to access the board.

There are two types of shared memory on the board. There are two local buses with two of the four VSP chips attached to each. Local memory on each bus is shared between the two VSPs that are attached to it. The VSP bus protocol allows bus locking to provide the mutual exclusion necessary to use the local memory for interprocessor communication. Each local VSP bus also has access to a four port memory shared by all the processors.

The 68020 has access to all system resources. This includes the local memories on the VSP buses and registers and control locations inside the VSP chips themselves. It cannot lock the local buses, but proper use of the VSP control locations should allow an equivalent ability. The 68020 can also

interrupt the VSP chips. With an appropriate interrupt routine, that allows the 68020 to preempt the buses as well.

There is also a status latch accessible by all processors. Each can write to two bits of the latch and read the other processors' bits from the latch. This does not provide any capabilities beyond those available through shared memory, though it is more convenient to use. In particular, it does not provide a mechanism for implementing true semaphores to control access to other resources.

Uses of Resources

The resources on the VPH are not sufficient to allow completely general synchronization of parallel tasks running on different processors without considerable overhead. However, they are adequate for the algorithms that are expected to execute on the VPH. Most of these algorithms will involve splitting up a task into almost identical subtasks, each of which will be executed on one of the VSP chips. All working VSPs will therefore need synchronization at the same points in their subtasks. This can be performed by using the status register and designating one of the processors as a synchronization arbitrator. In order to maintain the symmetry between the VSP chips, the 68020 will act in that capacity. This may not be the best choice for future use, since the 68020 may have other tasks to perform, but it is adequate for the present. One of the status bits for each VSP will be set to indicate that it is finished with its last assigned task. The other will be used to synchronize the VSPs by a full handshake with the 68020. This use of the second bit is not strictly necessary, since the same effect could be achieved by ending a task every time synchronization is needed. For the initial algorithms being written, this would probably be adequate. Only the FFTs need such synchronization and they only need it once. However, some future algorithms might need multiple synchronization points and the overhead of restarting the processors after each one might become excessive. Another possible method of synchronizing would be the use of the SYNC:[XE] instruction with a write to the $CAW location on each chip.

The bus lock on the shared local bus gives the shared local memories the most powerful communication mechanism. Their limitation is that they can only be used between the processors that share them. This is not useful for the global communication required by the algorithms being executed. Therefore this capability will not be used. The VSP chips will share code and static tables in these memories, but not data. Each one will maintain its own private data area. For simplicity, each will be preallocated a run-time stack area from which it can allocate storage.

The ability of the 68020 to access the VSP memories and registers can be used to communicate parameters such as the size and location of data to be processed. These parameters will allow for more functionality and for the slight differences in the tasks performed by each processor without any duplication of code. Placing such parameters directly into the VSP registers would give tiny performance improvements, but this is unlikely to justify the added complexity in the 68020 code. It does give the 68020 the ability to invoke subroutines that were written to expect parameters in registers without needing a separate version that performs the same task using parameters on the stack.

The parameters should be passed to each VSP by constructing a call frame on its run-time stack. The $SP register and $PC register must be set to the correct values so that it appears that a call has just been made. This will allow the same routine to be invoked from the 68020 or called by the VSP directly as part of another task. Making the call frame compatible with the Zoran library conventions will allow that code to be used when a single VSP chip is sufficient. In many cases, parallelism may be coarse enough that standard library functions can even be used as part of a subtask. For example, a dot product can be performed by four dot products on one fourth of the vector length, followed by summing the results.

In order to allow routines to act as both subroutines and main routines invoked by the 68020, the operations on the finished bits in the status latch must not be contained in the routines. The 68020 can reset the bits before starting execution by writing appropriate instructions into the VSP chips' instruction FIFOs while they are still in slave mode. The setting of the finished bits and halting of the VSPs can be performed by setting the return location in the constructed call frame to the beginning of a routine to perform those functions. The final return will cause the VSP to execute those instructions after completion of the main routine.

If a routine is going to be invoked repeatedly and it doesn't modify any of its parameters, the same stack frame can be used again. The parameters are still on the stack after the return. If interrupts are disabled, the return value is still on the stack as well. Otherwise it may have been written over by an interrupt after the return and before halting and will have to be "pushed" back on. If there are only a small number of sets of parameters needed and each routine needs minimal stack space, it would be possible to set up all necessary run-time stacks beforehand and select one simply by setting the $SP register to point to it. If the routines use too much stack space to allow dividing up local memory in this fashion, a data area pointer could be included in the stack frames to be used for allocation instead of the stack pointer.

The most useful shared memory is the 4 port SRAM, since it can be accessed by multiple processors simultaneously. For many algorithms it may be used for all signal data, with processed data being moved out from one buffer and replaced with new data while processing is performed on data in another buffer. The 4 port memory is relatively small, however. It only has room for two sets of 1k complex points. Two sets are adequate for buffering if the algorithm can be performed in place. The 32x32 2D FFT can be performed in place, but the 1k FFT cannot. With multiple data sets, a slower version of the 1k FFT that can be performed in place by using an extra reordering pass would probably allow greater overall throughput. Algorithms that use large data sets should be written to allow in-place operation when possible.

Invocation Conventions

The conventions for the use of the hardware determine the mechanisms available for communicating between software on different processors. The Zoran library calling conventions place further constraints on the format of VSP parameters being passed and the saving and restoring of VSP registers. In some cases where performance is particularly important, it may be useful to optimize the general calling sequence. Appendix C of the Zoran Software

Development Tools Manual details the calling sequence and possible optimizations. For the early demonstration code, a caller save convention is likely to be more efficient than the standard callee save. There may be no real subroutine calls at all and saving registers in code that is effectively the main routine when it is invoked by the 68020 is wasteful. Directives in the assembly code should make it easy to change to the standard convention if desired later. Using registers to pass parameters is another possible optimization.

Further conventions could guide the choice of what data to send in parameters. One of the biggest issues concerns the division of effort between the 68020 and the VSP chips. The VSP code may require values derived from the logical parameters. These could be supplied directly by the 68020 or determined by the VSP chips themselves. In particular, sharing a task between multiple VSP chips requires that each perform a different subtask. They could all be given identical task parameters along with a chip number and figure out for themselves what subtask they are to perform. Alternatively, each could be given different parameters determined by the 68020 to define its exact subtask. There are advantages to each approach that must be considered before making a choice.

Passing task parameters and chip numbers allows the 68020 to ignore the internal operation of the VSP algorithms. If the subtasks are changed, the 68020 code to invoke the task can still remain the same. If the VSP algorithms are invoked by a 68020 subroutine with the same parameters, it may be possible to copy the task parameters directly from the 68020 stack to the VSP stacks. Such a set of subroutines could be used to allow execution of VSP code to be transparent to a 68020 programmer, much like a remote procedure call. All of these subroutines could call a single subroutine to copy the stack frames instead of needing to perform task specific calculations. Calculation of subtask parameters would be performed simultaneously on each VSP chip, rather than serially on the 68020.

On the other hand, the 68020 instruction set is much more convenient for performing some of these calculations, and there is an assembler available to make it even more so. Being able to perform them in one place would make some of the calculations themselves simpler as well. The calculations could be done once and reused instead of redoing them every time the VSP code is invoked. The 68020 code could gain more functionality from the VSP code by combining VSP subtask primitives in more than one way. The standard Zoran library functions could be invoked directly to perform subtasks rather than having to be called indirectly from VSP routines that first determine the correct parameters. This saves calling overhead. By controlling the task division, the 68020 could assign differing numbers of VSP chips to a task to allow performance of multiple tasks at the same time. This would be constrained by the lack of general communication capabilities, but might be useful in some cases. It would also allow re-division of tasks to provide fault tolerance in the event of a VSP subsystem failure. These re-divisions would be less flexible and more awkward to implement with the other method.

Other Conventions

Some instruction parameters like ROM and first pass separation (FPS) are hardwired into instructions with no apparent way to set them from a parameter register. ROM needs to be set to different values for different subtasks in a large FFT. FPS, LPS (last pass separation) and ROM need to be set to different values for a single routine to be able to handle different sized FFTs. It may be possible to get the effect of one of FPS and LPS equal to 1 with the other less than 16 by using an appropriate $REPEAT and $NMPT combination. The problem of needing different values for different subtasks could be solved by having separate code for each subtask or by executing code conditionally based on an input parameter such as chip number. It is unclear whether the latter can be performed without multiple tests by using the ADDR instruction for a vectored jump. The problem of setting instruction parameters from an input parameter would be difficult to solve using self-modifying code because the only operations that can be performed on full word width data are floating-point. If a task only uses a routine with a single value for an instruction, the 68020 can modify the instruction appropriately before invoking the task. An initial ROM value can also be sent to a routine by executing an FFT instruction with that ROM value beforehand and using pre-addition or subtraction mode to "access" it. Some method needs to be decided upon if very general purpose FFT routines are to be used.

A smaller problem of the same type is that the $MBS_MSS register can't be used with partial bit reversal loads and stores the way the MBS and MSS parameters in instructions can. This can be solved by using the same methods used for the parameters that don't have registers, or by using extra instructions to get the desired reversals.

Conventions also need to be established for modifying special registers which affect the operation of the machine. The interrupt masks for arithmetic exceptions should not be modified by the VSP routines so that the 68020 can decide the level of error checking being performed. Some of the $MODE bits need to be modified by specific routines to get desired modes of operation. Some may need to have a particular value at all times. Others may need to remain at a value determined by the 68020 for reasons similar to the interrupt masks. If so, then all modifications to $MODE must be made by masking instead of loading. Some method of handling interrupts when they occur also needs to be determined. Many more such decisions will undoubtedly arise during system development.

Implementation Notes

Having the 68020 start the VSP chips one at a time executing application code presents many alternatives. Since most applications start vector loads early in the code, the 68020 may have difficulty getting the bus to start the second VSP chip on each bus. This will delay getting some of the chips started. With a start pattern that first starts one chip on each bus, this can be minimized but may still be significant. It would also be convenient for debugging under manual control if the application were started by a single event. For this reason, each VSP chip will be started in a polling loop and wait for a status bit from the 68020 to be set as the signal to proceed to the application code.

Just as with the setting of the finished bits at the end of the application, this synchronization should not be included in any of the application subroutines. The polling loop should be separate from them. There are two ways this can be accomplished. One is for the polling routine to end with a jump to the start of the application. The other is to add the starting address of the application to the bottom of the stack, start the stack pointer one lower, and perform a return instruction to get to the desired application. This is better because it simply requires adding to the artificial stack frame that must already be prepared rather than modifying a jump instruction in the polling routine. The same polling routine can be used for different applications. The same routine can also be used even if the two VSP chips sharing it must start at different addresses.

Here is the necessary starting procedure. Each VSP chip is assigned a stack area. This is initialized by "pushing" the start address of the FINISH routine, followed by any parameters being passed to the application, followed by the start address of the application. The $SP register of each VSP chip is set to point to the address below the end of the stack. The 68020 status bit used for starting is set false. Each VSP is started executing from the beginning of the START routine. When the start bit is set true, all of the VSP chips will exit the polling loop in START. They will "return" to the application code. When the application is done, they will "return" to the routine that sets the VSP status bits to indicate that they are finished and halt.

5.0 MicroAsm

A study was made of several commercially available micro assembler packages. Previous reports have referenced HALE (Hilevel Assembly Language Environment) and compared it to MicroASM. The following is a comparison of the MicroASM system and another popular microassembler - the Microtec Meta29M 2900 Macro Meta Assembler.

Meta29M was developed primarily for the AMD 2900 series microprogrammable microprocessors and thus is really aimed at different problems than MicroASM, however it is representative of most microassemblers available today. Like MicroASM, it utilizes a two-stage system consisting of a Definition phase and an Assembly phase.

The Definition phase allows instruction mnemonics and their associated formats to be defined along with constants and reserved symbolic names. The Definition program checks the definitions for validity and issues error messages when errors are found. The Definition program features conditional assembly directives, complex expression evaluation and a cross reference table listing.

The Assembly phase is a two-pass program that builds a symbol table, issues error messages, produces an easily read program listing and symbol table, and generates an object module. The Assembly program also features conditional assembly directives, complex expression evaluation, and a cross reference table listing.

Meta29M supports a macro facility. Through the use of macros, variable length microwords may be defined, fields may be broken up into non-contiguous bit patterns, and single mnemonics may be used to represent complex overlayed instruction formats. Conditional assembly statements may be used in conjunction with macros to implement multi-purpose macros. Macros may be recursive and may be redefined at any point in the program.

There are, however, some serious limitations to Meta29M that make it inappropriate for architectures such as the CPH and wide microword architectures in general. The Meta29M Definition language is really nothing more than a simple macro language consisting primarily of the "EQU" and "DEF" directives. These are used in the following manner:

```
ABAT: EQU   H#50                    ;Define a constant ABAT = 50 hex

ADD:  DEF   H#5,ABAT,4VH#           ;Define an instruction mnemonic ADD
```

Note that all mnemonics are globally defined - that is a mnemonic may be used in only one context. While this may be sufficient for microprocessors, it is a serious limitation in wide instruction word architectures where it is not uncommon to have in excess of 1024 instruction bits and multiple similar fields for similar resource control (say several identical multipliers). In these situations it is convenient to have identical mnemonics for each similar resource with no conflicts. In addition, wide-word architectures are typically "field-oriented" where the instructions are logically broken into fields for ease of programming. Thus an ADD may be accomplished in any number of ways using any number of resources (i.e., there may be multiple ADDs in

multiple fields). A simple macroing scheme is inadequate to this task.

MicroASM begins with the concept of logical fields. Any logical field may have any number or level of subfields. Mnemonics defined for any field (or subfield) are local to that field and may therefore be used in any number of different contexts without ambiguity. Logical fields are then mapped to the actual physical fields of the microword. This may be as simple as a direct one-to-one relationship or a complex relationship involving any number of logical fields. It should be noted that Meta29M also supports complex expressions for mnemonic definitions, however these expressions are limited in that they cannot use parentheses, cannot directly reference a "field" and support a very limited set operators. MicroASM supports the complete set of ANSI C language operators (arithmetic, logical and bitwise) with the addition of three MicroASM specific operators (EITHER, CAT and PARITY).

One of the more serious limitations of Meta29M is that it does not support polyphase system clocks, which are increasingly common in multiprocessor parallel architectures. Specifically the CPH uses a two-phase system clock, and thus cannot make use of an assembler like Meta29M.

MicroASM's definition stage actually defines the fields in the microword and constructs all of the necessary symbol tables for the Assembly phase. This allows the Assembly phase to execute far quicker than a system where the symbol tables must be constructed at run time. Also, MicroASM uses a macro preprocessor which allows conditional assembly as well as complete macro capabilities. Another capability provided by the MicroASM preprocessor and not supported by Meta29M is the ability to "include" other source files at assembly time. This allows the user much greater flexibility in source file control - i.e., all constants may be placed in a single "include" file and used with any number of other source files.

Another important feature not supported by Meta29M is the automatic support of different number formats. While both Meta29M and MicroASM allow the specification of numbers in Binary, Octal, Decimal and Hexadecimal, MicroASM also allows the specification of floating-point numbers in IEEE single and double precision as well as DEC F and DEC G formats. In addition, MicroASM supports a Pragma to specify whether numbers are big endian or little endian (see Section 4.6 of this report). Another important feature support by MicroASM alone is the "PARITY" field operator whereby any physical field may be mapped as the parity of any combination of logical fields. This is increasingly important for the efficient programming of fault tolerant architectures. Specifically, the CPH uses parity for memory checking, thus this feature is important.

Finally the level of error checking that is possible with MicroASM is a significant improvement over Meta29M which can only check to see that the final value of the microword is the proper length and that the internal Meta29M syntax has not been violated. MicroASM can detect fields that are referenced in the wrong phase, or for the wrong number of phases. It can enforce specific latency times for different fields or mnemonics. It allows the definition of default values at any level and even warns the user of suspicious activity (i.e., using a decimal number to define a mnemonic - not illegal but certainly uncommon).

5.1 Overview

The MicroASM system consists of two programs. *GENASM* generates the symbol tables specific to each micro-architecture that is defined using the MicroASM definition language. *GENASM* compiles this language and populates the symbol tables. *MICROASM* uses the symbol tables to assemble code that uses the mnemonics and logical fields defined and compiled by *GENASM*.

5.2 GENASM Program - Definition of Microword Fields and Mnemonics

The central concept of MicroASM is the idea of Logical Fields and Physical Fields. Logical fields are fields defined by the microprogrammer and are actually referenced in the micro-assembly code itself. Physical fields represent the actual physical segments of the microword. The definition phase of MicroASM involves defining the Logical fields, subfields and mnemonics that conceptually describe the underlying hardware and then mapping these Logical fields to the Physical fields. This is done by using the MicroASM definition language which is compiled by the GENASM program to produce the tables required by the MICROASM micro-assembler program.

5.3 MicroASM Definition Language

The GENASM definition language is designed as a structured, block oriented language in the spirit of C. In fact actual C syntax is used for some definition syntax. This **language is completely position independent and all white space is ignored by the compiler** thus easily readable programming "styles" are encouraged but not enforced. This language essentially does two things: it allows the definition of Logical fields, along with their associated subfields and mnemonics with no concern as to the "physical position" of the fields, and them allows the mapping of these Logical fields onto the actual physical microword.

5.3.1 GENASM Case Sensitivity

GENASM can compile the definition language either case sensitive using a command line switch (-c) or case insensitive (default). **When in case sensitive mode, nothing is translated and all keywords are defined in lower case.** When in case insensitive mode all characters are converted to lower case.

5.3.2 Comments

Comments in GENASM (and MICROASM) are delimited exactly the same as they are in the C language: Comments begin with /* and end with */. Any other character sequences including new lines or carriage returns are acceptable as comments within the delimiters and are simply ignored at compile or assembly time. Nesting of comments is not allowed.

Example:
```
/* This is a comment */

/* This is also a comment that
   ends down here.          */
```

5.3.3 Numerical Values

Any numerical value associated with the MicroASM definition phase will always be an unsigned integer. The definition language supports four common number bases, binary, octal, decimal and hexadecimal. For octal, decimal and hexadecimal numbers the specification is identical to that used by the C language, binary numbers are specified in a similar, consistent manner. The syntax for specification of each is as follows:

Binary: *0bbin_num* where *bin_num* is any valid binary number (i.e., each digit must be either a 0 or a 1) prefixed by *0b*. Example: 0b11011

Octal: *0oct_num* where *oct_num* is any valid octal number (i.e., each digit must be between 0 and 7) prefixed by *0*. Example: 0642
Decimal: *dec_num* where *dec_num* is any valid decimal number (i.e., each digit must be between 0 and 9) NOT prefixed by *0*. Example: 642

Hexadecimal: *0xhex_num* where *hex_num* is any valid hexadecimal number (i.e., each digit must be between 0 and 9 or between A and F) prefixed by *0x*. Example: 0x642A

5.3.4 Definition of Global Parameters

In any MicroASM definition there are three global parameters: **width**, **phases**, and **defbit**.

width specifies the actual width in bits of the physical microword using the following syntax:

width = *num*

where *num* is an integer (between 1 and 2^{32}) in any of the acceptable number bases. Failure to specify microword **width** results in an error.

defbit specifies the default bit value to be used whenever a value is not explicitly specified for any field. The specification syntax is as follows:

defbit = *num*

where *num* is either 0 or 1 in any of the acceptable number bases. **defbit** is optional, but there is NO DEFAULT VALUE. Thus if **defbit** is omitted, any unspecified bits in the assembly phase will generate an error. To aid in program debugging, a warning is generated each time the global **defbit** value is used automatically.

5.3.5 Logical Field Definition

A logical field is a segment of a microword that may be named to reflect its nature - i.e., "ALU_1" or "SEQUENCER". A logical field may have associated with it mnemonics, and a default value that is implied whenever the field is active but no value is explicitly assigned to it. A logical field may also have any number of nested subfields - each with their own mnemonics and defaults. In addition a logical field (or subfield) may be defined to be "active" for a specified number of clock phases. The syntax for logical field

definitions is as follows:

```
fieldname_1 [fld_width] #act_phases1
fieldname_2 [fld_width] #act_phases2

.
.
.
fieldname_n [fld_width] #act_phasesn
{
   field definition - subfields and mnemonics
}
```

fieldnames are valid unique identifiers (relative to their parent block). The syntax of multiple *fieldnames* is used to specify fields, probably mapped to different parts of the physical microword, that have the same subfields and mnemonics without having to duplicate the entire field definition. *fld_width* is an integer (between 1 and 2^{32}) in any of the acceptable number bases delimited by square braces "[" and "]". Note that the sum of the widths of all children fields must be less than or equal to the width of the parent field.

The syntax for logical subfield definitions is identical to parent field definitions - i.e., all field definitions are identical. The only difference is that subfields are defined within the parent field's definition block.

Note that the logical "position" or "offset" within the parent field is determined by the order in which the subfield is defined. This is important in that when *mnemonics* are specified in MICROASM (the assembly phase) the order of fields referenced are determined by this definition order.

5.3.6 Direct Field Definition

In the case where it is desired to define a block (of subfields and mnemonics) for a set of differing subfields of different fields, the MicroASM indirection syntax may be used. This syntax is similar to the C "struct" reference syntax.

```
parent1.child1.childn [fld_width] #act_phases1
parent2.child2.childm [fld_width] #act_phases2
.
.
.
parentn.childy [fld_width] #act_phasesn
{
   field definition - subfields and mnemonics
}
```

where *childn* is referenced as a child subfield of *child1* which is, in turn, a child subfield of *parent1*. Parental precedence descends from right to left with the leftmost field specified is the global field level parent and the rightmost field being the new subfield to be defined, with each field name separated by a period ".". *fld_width* is an integer (between 1 and 2^{32}) in any of the acceptable number bases delimited by square braces "[" and "]".

Note that the sum of the widths of all children fields must be less than or equal to the width of the parent field. The *act_phases* specifiers are optional and specify the number of phases during which the associated subfield must be "active" or hold a value. If *act_phases* is specified then all of the subfield's children (subfields and mnemonics) will be assumed to be active for *act_phases* as well. If *act_phases* is not specified then each child (or block of children) may be specified with differing active phase specifiers. *act_phases* is preceded by "#". The *field definition* can include subfield definitions, mnemonic definitions and default values with the entire definition block delimited with braces "{" and "}".

5.3.7 Mnemonic Definitions

A mnemonic is similar to a macro in that it serves to substitute a numeric value for an identifier name. In MICROASM it differs from a macro in that mnemonics are always local to their block (parent field), and serve to define a FINITE SET of identifier-referenced values for the parent field. In other words, if a set of mnemonics is defined for a field (this includes global mnemonics or parent block mnemonics), then no other mnemonics will be allowed to be used in reference to that field.

5.3.8 Defining Fields to Accept Address Labels

Some fields may need to accept address labels as well as mnemonics. These labels are defined during the assembly phase in the micro assembly code itself. These types of logical fields usually refer to address sequencers or program counters. The syntax for defining a field that accepts labels is:

```
field_def
{
  labels
  .
  .
  .
}
```

The *labels* keyword may be included with mnemonic definitions in a field definition. The *labels* keyword may appear anywhere that a mnemonic definition can with the exception of global mnemonics. In other words, the global microword may NOT accept labels. In addition, the field for which labels have been specified must be of the proper size (as with any mnemonic definition).

5.3.9 Complete Field and Mnemonic Definition Example

The following is an example to illustrate the use of the GENASM definition language.

```
/*GENASM Definition example */

width = 64                  /* 64-bit wide microword */
phases = 4                  /* 4-phase system clock */
defbit = 0                  /* When in doubt assign a 0 */

/* Logical Field Definitions */
```

```
mult1 [22]      /* Global level field 22 bits wide called mult1 */
{
  default = 0x0000          /* Default value for  mult1 */
  xsel[3] #1                /* Subfields of mult1 */
  ysel[3] #1
  {
    cache_a = 0b000         /* Mnemonics for xsel & ysel */
    cache_b = 0b010
    alu_1 = 0b110
  }
  insta[8]                  /* Subfield starting at bit 6 */
  {
    mult = 0b1111000 #2     /* Active for 2 phases */
    div = 0b0011000 #4      /* Active for 4 phases */
    ins_flag[1] #1          /* Single bit subfield of insta */
    {
      real = 0b0
      imag = 0b1
    }
    rest[7]
    {
      tia = 0b0001110
      tib = 0b1110010
    }
  }
  instb[8] #2
  {
    clear = 0b0000000
    load = 0b1111111
  }
  check [2]                /* Subfield using 2 bits */
  {
    ready = 0b01
    set = 0b10
    go = 0b11
  }
}

alu_1 [5]                   /* Field 5 bits wide called alu_1 */
{
  default = 0b11111
  cont [3] #2               /* Three bit subfield */
  {
    on = 0b111              /* Local mnemonics for subfield */
    off = 0b000
  }
  check [2]
  {
    go = 0b11
    clear = 0b00
  }
}
```

5.3.10 Specification of Logical Field to Physical Field Mapping

Once the logical fields are defined they must then be mapped onto the actual physical microword. Unlike logical fields which may have as many bits as is conceptually expedient, physical fields are constrained by the actual hardware for which the MicroASM tables are being defined.

5.3.11 Assigning Logical Fields to Physical Fields

The assignment of logical fields to physical fields is done using the *assign* statement. These statements have the following syntax:

assign (*offset_spec*) @(*phase_spec*) = *field_spec*;

The *assign* keyword is followed by the *offset_spec* which defines the absolute position within the physical microword that the physical field occupies. *offset_spec* can take any combination of the two distinct offset forms - contiguous form and individual form - delimited by parenthesis "(" and ")". *phase_spec* uses a syntax identical to the *offset_spec* to specify absolutely which phases the physical field may **become active** in. The *phase_spec* is always preceded by an "@". The *field_spec* is a logical field, list of logical subfields, or bitwise logical/arithmetic expression with logical fields as operands. The entire expression is always followed by a semicolon ";". The semicolon syntax for "end of statement" is included since in many cases these *assign* statements will occupy multiple lines and the "end of statement" is easier and more compact than "line continuation" schemes.

5.3.12 Absolute Phase Specifiers (not implemented yet)

Absolute phase specifiers determine the phases during which a physical field may become active. This allows the definition of physical fields that control completely different hardware functions in different clock phases, or the definition of fields that can alternately carry instructions and immediate data in different phases. Absolute phase specifiers for physical fields can take two forms. The syntax for both forms is as follows:

Contiguous form: (*first*:*last*)

where *first* is the first phase during which the physical field may become active and *last* is the last phase during which the physical field may become active.

Individual form: (*phase1*,*phase2*,*phasen*)

where *phase1* through *phasen* are individual absolute phases during which the physical field may become active.

A valid absolute phase specifier may include combinations of both forms as in the following: (*phase1*,*first*:*last*,*phase2*)

Note that the combination of phase length specifiers from the logical field definitions and these absolute phase specifiers can easily cause timing clashes which cannot be effectively prevented or detected by the compiler. Many polyphase machines have such complex timing schemes that there is no way

to automatically distinguish a timing mistake from a complicated system - other than one works and one doesn't.

Note also that GENASM always SORTS and COMPRESSES any phase or offset specification. Thus (0,5,4:1) is converted to (0:5). While, in general, this simply promotes rational definition it can lead to unexpected results.

5.3.13 Field Specifications

The *field_spec* section of the assign syntax may be as simple as a single logical or as complex as a complete logical expression with any number of logical fields as operands. These expressions are important for horizontal compaction of microwords where single physical fields *must* be used in multiple contexts to conserve microword width. The operators allowed in field expressions are identical in syntax to the bitwise operators in C, with three additional operators. These are:

& - bitwise AND operator
| - bitwise OR operator
^ - bitwise XOR operator
! - bitwise negation (NOT)

The additional operators are:

cat - Concatenation operator
either - Allows physical field to be referenced by one of two logical fields but not both simultaneously.
parity() - parity of some field spec.

When the GENASM compiler encounters a field expression it stores the expression in a table. The expression is evaluated at runtime by MICROASM whenever the pertinent fields are referenced. Any logical field may be involved in any number of expressions as long as there are no obvious conflicts, however care should be taken when using logical fields in multiple expressions as undetectable clashes are possible.

5.3.14 Assigning Logical Fields to Physical Fields Example

The following uses the fields defined as an example of how the **assign** syntax is used.

```
/* Physical field assignments */

                /* Assign multl fields except instb to absolute */
                /* bits 0-15 becoming active in phase 0 or 1.   */
assign (0:15) @(0:1) = multl.xsel,ysel,insta;

                /* Assign multl fields except insta to absolute */
                /* bits 0-15 becoming active in phase 2 or 3.  */
assign (0:15) @(2,3) = multl.xsel,ysel,instb;

                /* Abs bits 16 - 18  = bit subfield xsel of */
                /* multl ANDed with subfield cont of alu_1. */
assign (16:18) @(0) = multl.xsel & alu_1.cont;
```

```
                /* Abs bits 20 and 22 are either multl.check */
                /* or alu_1.check but not both.              */
assign (20,22) @(0) = multl.check EITHER alu_1.check;
```

5.4 MICROASM Program

The MICROASM program allows the user to write programs referencing the logical fields and mnemonics as defined in the GENASM program. The basic format for MICROASM statements is as follows:

```
@act_phas1 fld_spec1 m1,(m11,m12),m3,mn
.
.
.
@act_phasn fld_specn mn1,mn2,(mn11,mn12),mn3;
```

Where *act_phas* is the phase for which the following mnemonics are applied. It is preceded by @. *fld_spec* is a parent field specifier and may be a simple as global field name ("multl") or it may be a direct subfield reference (multl.xsel). The following mnemonics (*m1*, .. *mn*) are arranged in the order that their parent fields were defined. When a parenthesis is added this indicates that the mnemonics contained within the parenthesis belong to a child field of the current level. The following illustrates these concepts:

```
@0 multl    cache_a,cache_b,(real,tia),set
```

Note that *cache_a* is a mnemonic defined for multl.xsel, *cache_b* is a mnemonic defined for multl.ysel, *real* is a mnemonic defined for multl.insta.ins_flag, *tia* is a mnemonic defined for multl.insta.rest, and *set* is defined for multl.check.

An alternative structure is:

```
@act_phasn fld_specn = mn;
```

where the "=" implies that mnemonic *mn* belongs directly to *fld_specn*.

5.4.1 References to Immediate Data Values

Since a mnemonic is actually an identifier associated with an actual numeric value, any mnemonic can be replaced by an actual numeric value (assuming the field referenced is large enough). In addition to the integer number base specifications, MICROASM accepts floating-point data that is automatically converted to the floating-point format specified. The following format syntax is supported:

0s*fp_num* - Single precision (32-bit) IEEE floating-point
0d*fp_num* - Double precision (64-bit) IEEE floating-point
0f*fp_num* - DEC F Single precision (32-bit) floating-point
0g*fp_num* - DEC G Double precision (64-bit) floating-point

Example: 0d156.4632e4 would be represented in the microcode as a double precision IEEE format number.

This syntax allows the use of various floating-point formats unambiguously in the same microprogram.

5.4.2 Labels

Labels are used to mark positions within the microprogram for sequencer jumps and program branches. The syntax is:

label:

Where *label* is an unambiguous identifier to be associated with the address of its occurrence in the microprogram, followed by a colon ":". Labels may be referenced in the same way as mnemonics in fields which have been defined to accept labels. Use of a label in reference to a field which has not been defined as accepting labels will generate an error.

5.4.3 Absolute and Relative Addressing

There are several methods of programming program jumps and branches - absolute addressing and relative addressing. Absolute addressing simply jumps to the address (i.e., label reference) specified. Relative addressing, however, calculates the offset from the current position to the address specified and this offset is the value stored in the microcode. Note that offsets can be negative for backward jumps. The syntax used for absolute addressing is:

addr_spec

where *addr_spec* is a label reference or an immediate value.

The syntax for relative addressing is:

[addr_spec]

where *addr_spec* is a label reference or an immediate value.

Following example illustrates:

```
start:
@0    mult1       cache_a,cache_b,mult,go
.
.
.
/* Loop to start by jumping to start's address*/
      seq              long_jmp,start;

@0    mult1       cache_a,cache_b,mult,go
.
.
.
/* Loop to start by adding the offset of difference between the */
/* current location and start's address to the sequencer.      */
      seq              short_jmp,[start];
```

5.4.4 Expressions

Any MICROASM statement may contain arithmetic or Boolean expressions that follow the same operator precedence and construction rules as C. There is no limit on the complexity or nesting of the operations. Obviously there is a limit on the size of the result. Any result that overflows the size defined for it will generate an error. The following operators, listed in descending order of precedence are supported:

!	Boolean bitwise negation (NOT)
*	Arithmetic multiplication
/	Arithmetic division
%	Arithmetic remainder (modulus)
+	Arithmetic addition
-	Arithmetic subtraction
&	Boolean bitwise AND
^	Boolean bitwise Exclusive OR (XOR)
|	Boolean bitwise OR

5.5 MicroASM

MicroASM uses a C type preprocessor to implement macros and conditional assembly. This is a text processor that manipulates the text of a source file as the first stage of assembly. Although MICROASM ordinarily invokes the preprocessor in its first pass, the preprocessor can also be invoked as a stand-alone program.

5.5.1 Preprocessor Directives

The MicroASM preprocessor recognizes the following directives:

#define
#undef
#if
#ifdef
#ifndef
#elif
#else
#endif
#include
#pragma

The pound sign "#" must be the first non-white-space character on the line containing the directive. Several of these directives require an argument or value. Any text that follows a directive that is not part of its argument or value must be enclosed in comment delimiters "/*" and "*/".

5.5.2 Constants and Macros

The **#define** directive is used to create constants and macros. Its syntax is:

#define *mac_name subst_text*

#define substitutes *subst_text* for all subsequent occurrences of *mac_name* that can be interpreted as tokens that are encountered in the source text. In other words *mac_name* is replaced by *subst_text* wherever it is encountered in the text following the **#define** directive unless it is enclosed in parenthesis or is part of a longer identifier. The following example illustrates:

```
/* Original Source Code */

#define PI 0s3.14159
.
.
.
@0 alu      add,cachea,cacheb,PI

/* Source Code after Preprocessing */
.
.
.
@0 alu      add,cachea,cacheb,0s3.14159
```

5.5.3 Undefining Macros or Constants

The **#undef** directive removes the definition of an identifier. Once the definition is removed it can be redefined to a different value. This allow the use of the same macro or constant name to be used with different values in different contexts in the same source code. The syntax is:

#undef *mac_name*

This syntax will remove the previous definition of *mac_name* which was defined using a **#define** statement. The #undef directive is usually paired with a **#define** directive to implement conditional or special case assembly.

5.5.4 Include Files

The **#include** directive inserts the contents of the specified file into the source file at the point where the **#include** reference occurs. This allows the organization of common constants and macros into "include files" which may be **#included** into any number of MicroASM source files. There is a "standard" include file called "std.inc" that comes predefined with MicroASM. This file contains commonly used constants **#defined** in all of the different number formats.

Another important use of include files involves including source modules into a main driver module. This allows the use of smaller easily manageable source files which can all be included into a larger program.

The syntax is:

#include "*file_spec*"

or

#include <*file_spec*>

These two forms differ in the path search initiated by the preprocessor for the file specified by *file_spec* if *file_spec* does not include a complete path. The first form which uses double quote delimiters searches the parent source file's directory first and then searches the "standard directories" as defined via command line or system setup. The second form which uses the bracket delimiters "<" and ">" begins it's search with the standard directories.

Include files can be nested, i.e., and include file may itself contain **#include** directives. When include files are nested directory searching begins with the directories of the parent and then proceeds through the directories of any grandparents and finally it searches the standard directories.

5.5.5 Conditional Assembly

One of the most powerful features of the MicroASM preprocessor is conditional assembly. This allows the use of a single source file for several different applications (i.e., a single routine source may be assembled into two versions, one using IEEE floating-point and the other using DEC floating-point by simply changing a single statement). The basic directives that implement this feature are:

#if
#elif
#else
#endif

In addition the **defined()** operator is used along with the shortened concatenated forms

#ifdef
#ifndef

The syntax is:

#if *const_expr*
prog_text
#elif *const_expr*
prog_text
.
.
.
#elif *const_expr*
prog_text
#else
prog_text
#endif

Each **#if** directive must be matched by a closing #endif directive. Any number of **#elif** directives can appear between the **#if** and **#endif**, but at most one **#else** directive is allowed. The **#else** directive must be the last directive prior to **#endif**. The preprocessor selects only one of the blocks of *prog_text* which can be any sequence of text occupying any number of lines.

Typically *prog_text* is MICROASM source code or preprocessor directives. If the selected *prog_text* is contains preprocessor directives, the preprocessor carries them out, otherwise *prog_text* is passed to the assembler. Any *prog_text* not selected by the preprocessor is ignored and thus is not assembled or processed.

const_expr is a restricted constant expression that must involve strictly constants (which may be **#defined**) and **defined()** values that resolve to an integer value. The preprocessor selects a single *prog_text* block by evaluating the *const_expr* restricted constant expression following each **#if** or **#elif** directive until it finds a non-zero value. It the selects all text from the **#if**, **#elif** or **#else** directive up to the next **#elif**, #else or #endif directive.

The defined() operator and it's shortened forms #ifdef and #ifndef use the following syntax:

#if defined(*mac_name*)
prog_text
#elif defined(*mac_name*)
prog_text
.
.
.
#elif defined(*mac_name*)
prog_text
#else
prog_text
#endif

or alternatively

#ifdef *mac_name*
prog_text
#elif defined(*mac_name*)
prog_text
.
.
.
#elif defined(*mac_name*)
prog_text
#else
prog_text
#endif

These conditional blocks operate in exactly the same fashion as other #if statements. The difference is that the condition is simply whether *mac_name* has been previously **#defined**. The other forms, **!defined()** and **#ifndef** are satisfied if *mac_name* has NOT been **#defined** and are used in identical fashion.

5.5.6 Local Assembler Directives

The preprocessor supports a method of embedding assembler directives into the source assembler code. This is done using the **#pragma** directive. The syntax is

#pragma *direct_name*

Following the **#pragma** directive, *direct_name* is a single identifier identifying the assembler directive to be active beyond that point in the code. At this time the only *direct_names* supported by MICROASM are the floating-point byte/word order specifiers:

LITTLEENDIAN	Swaps low byte/high byte
BIGENDIAN	No byte or word swapping is done

6.0 Conclusions

The EVA architecture composed of the VPH and the CPH subsystems is capable of gigaflop throughput for several reasons. Careful attention was paid to the internal buses so that maximum data transfer can occur among the boards. The typical board level IO bottleneck was reduced significantly. Use of Gazelle hot rod chips with gigaflop clock rates and the fully parallel crossbar chip made all the difference.

Several innovations were achieved in this SBIR Phase II. Among those include the crossbar device with unparalleled speeds. The CPH architecture is ultrafast due to the massively parallel internal datapath options (using the crossbar). The VPH is a multi wave processing architecture. Fully concurrent DSP processing is made possible. Use of novel packaging helped to reduce data transfer bottlenecks. A photograph of the micromemory modules shown next in Figure 54 made it possible to integrate more memory on the cache boards. Many interfaces were necessary to interconnect the CPH to a PC, VME, and VPH. A VME buffer board was designed and built to let the CPH converse with the VPH and a VME bus. It is shown in the next photograph (Figure 55). Most important of all was the crossbar chip also shown in an accompanying photograph (Figure 56). The crossbar chip, a 256 pin PGA ASIC reduced board space by eliminating numerous multiplexer devices.

6.1 VPH Performance and Demonstration

It was predicted at the end of the Phase I project that the VPH would perform a 1k complex FFT in 800 usec. The board actually executes this FFT in 600 usec. This is largely due to careful hardware design and adroit programming of the VPH by Larry Hall and Steve Sharp. Programming the 325s proved to be a challenge because the available application library fit only one device and not multiple devices. Nevertheless, once the wave concept was mastered and used consistently, programming to optimize performance became routine.

Code for convolutions, correlations, and coordinate transformations was completed quickly. Using conventions for startup and terminating DSPs helped reduce the effort. The STARTUP and FINISH routines were created for generic code segments so that they could be used over and over. The 68020 also proved to be advantageous in controlling the synchronization. As a result, all of the Phase I performance predictions were exceeded by at least 25%. Some of the code performance is tabulated below.

Algorithm (4 DSPs)	Execution Time (usec)
1k Complex FFT	604
64 Point Correlation	40
64 Point Convolution	42
8x8 2D FFT	65
16x16 2D FFT	270
32x32 2D FFT	724
Polar to Rectangular	25
Rectangular to Polar	48

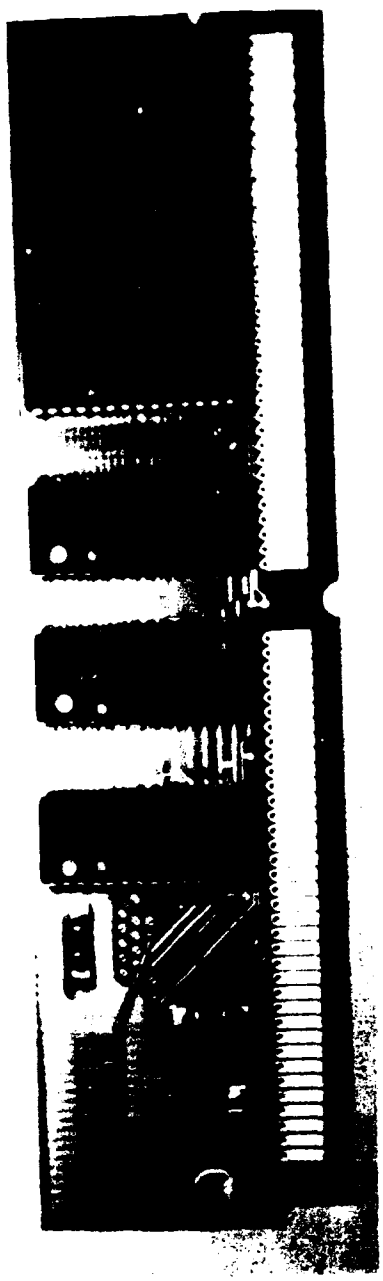


Figure 54. MicroMemory Module

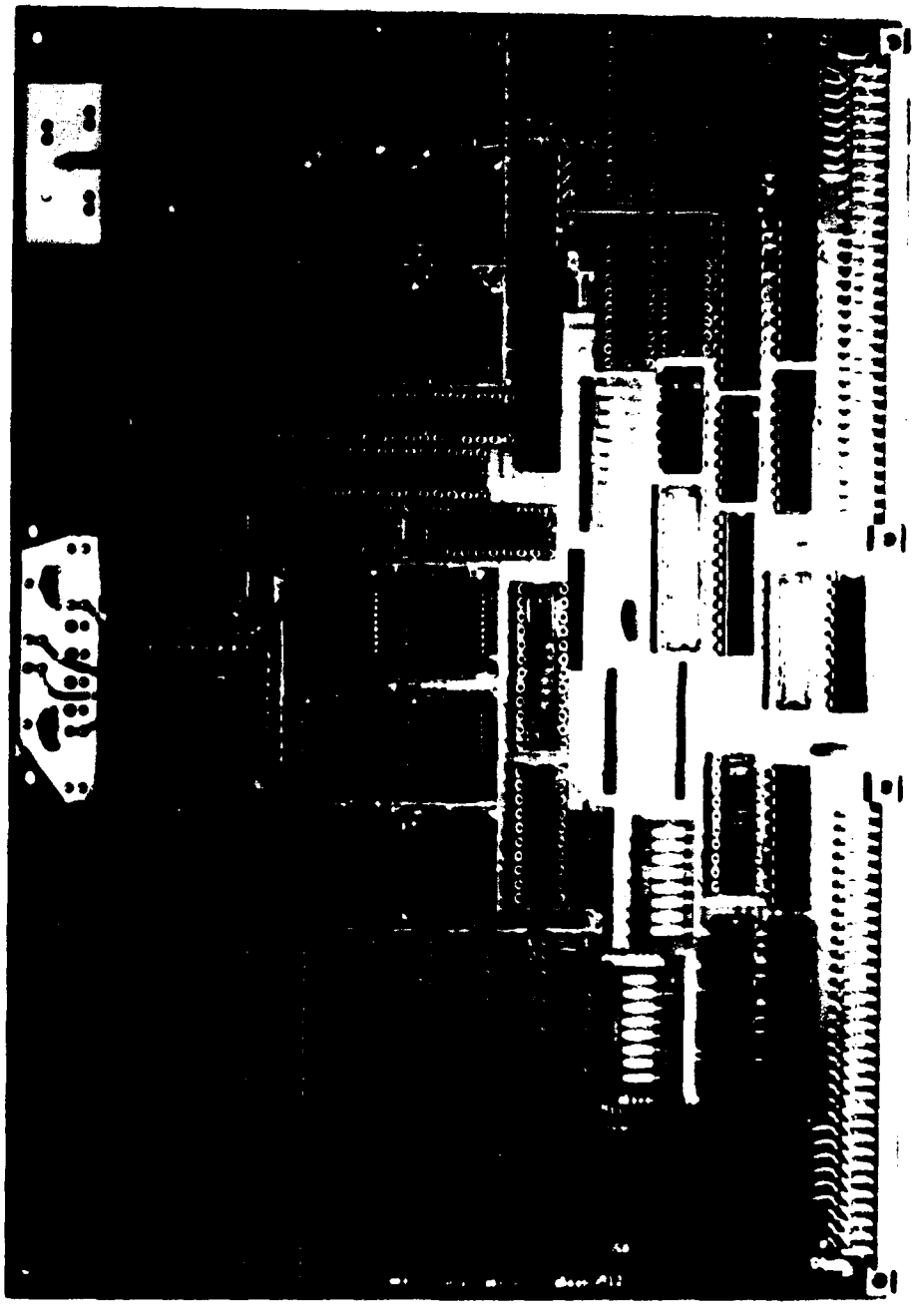


Figure 55. Serial I/O Board

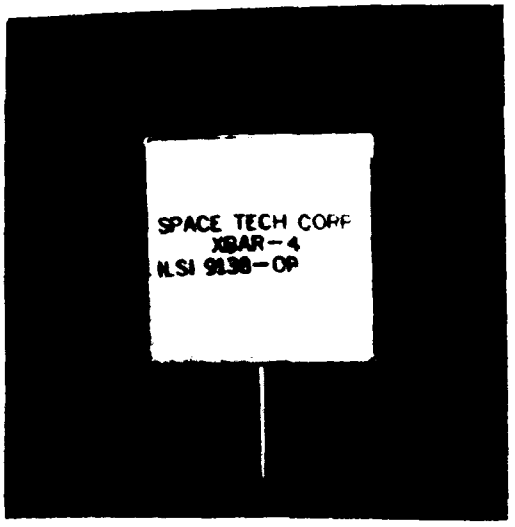


Figure 56. Crossbar Device

Note that the 2D FFTs are fast enough for real-time frame grabbers and CRT displays where 30 frames a second are often viewed without flicker. Hence, there is a real possibility that the TSI tracker and the Space Tech VPH board can track, focus, and translate in real-time instead of off-line.

The VPH, depicted in the following photograph (Figure 57), was demonstrated at WSMR in the Instrumentation Development Directorate on 25 August 1992. The VPH was interfaced to a TSI single board computer inserted into the VPH mainframe. A PC was used as a terminal for the VPH and the TSI SBC had its own terminal. The demo consisted of transmission of data between the VPH board shown next in Figure 58 and the SBC in either direction, executing digital signal processing programs, and sending results to the PC terminal and the SBC terminal.

Special drivers and utilities were generated. These drivers manipulated data and programs from the PC so that the debugger in the SBC could access them and display results on the SBC monitor. A section of the SBC memory space was allocated for the VPH results and processed data was sent there. Likewise, programs were downloaded from the SBC to the VPH to be executed by the VPH. This demonstrated that the SBC could serve as system VME master or controller. This also demonstrated that the VPH could be a VME slave in a generic VME system. This is important for the VPH as it is also intended to be interfaced to SUN workstations. An important device in the VPH greatly facilitated the SBC/VPH interface, namely, the MVME 6000 VME Interface chip from Motorola.

Figure 57. EVA Chassis

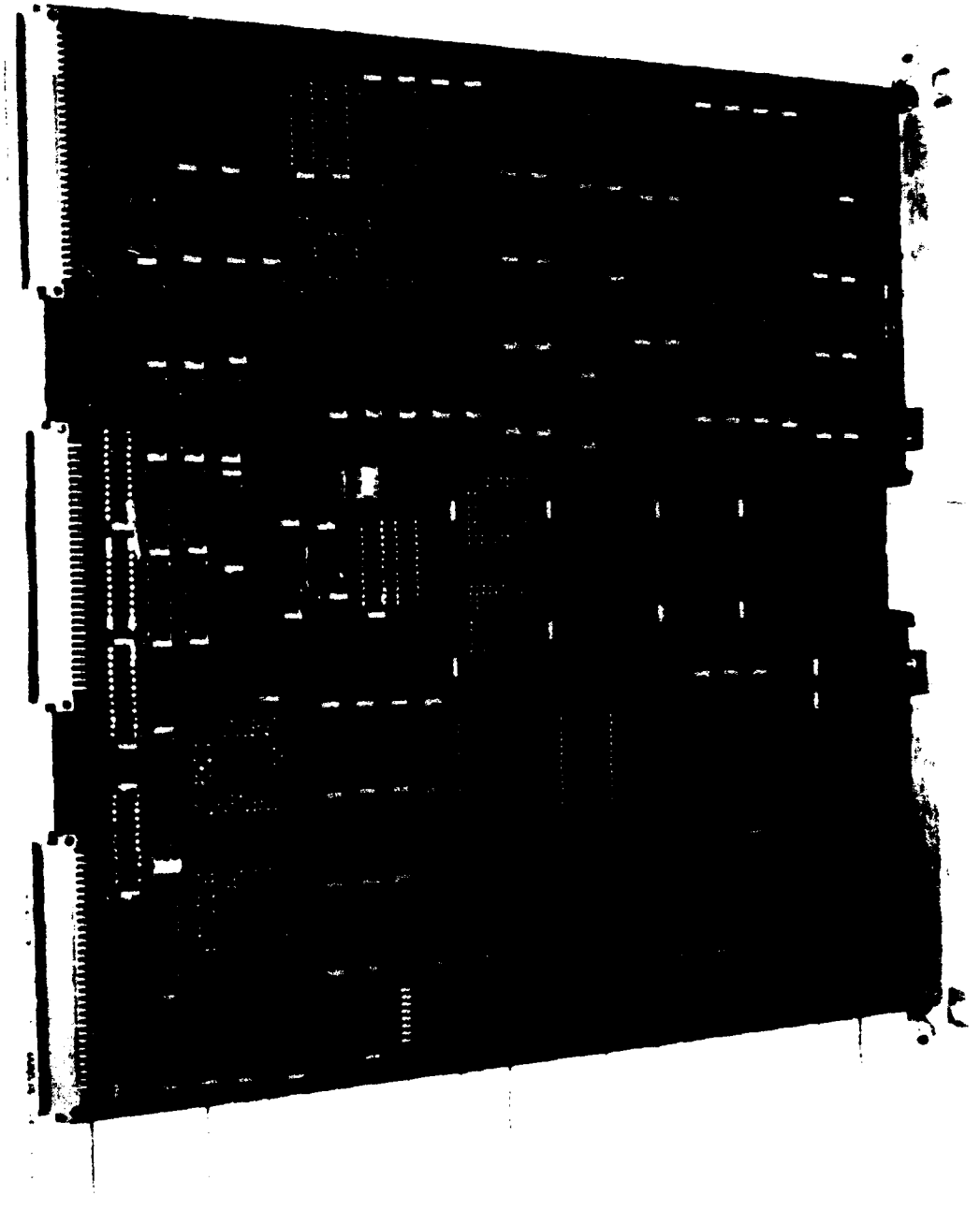


Figure 58. VPH Board

The DSP programs that Space Tech created for the demo were a 1KFFT and a correlation. Both programs were verified as functionally correct by comparing results from independently generated outputs from C routines found in the text, Numerical Recipes for C. A roundoff utility, a compare word by word, and an internal 16-bit fixed to IEEE floating-point data type conversion were used to test the results. The 1KFFT output was within 5 decimal digits of accuracy in all but 5 data points. The other 5 were within 4 fractional decimal digits. The correlation results were within the same range of precision. This is to be expected in both cases since the PC has a 16-bit internal processor and the VPH has a 32-bit internal processor. The execution times for these and other DSP routines are shown in listing above.

An important discovery of this demo is the need to map and translate memory maps across the several domains. Those physical domains include the EPROM space, register space, and data space of the ZORANs, the data and program space of the 68020, and the data and program space of the SBC. Care must be exercised when translating the correct hexadecimal literal values. Tables are included in earlier sections to make the translations for the VPH. It took Space Tech some time to determine that space for the SBC and the MVME on it because little documentation existed.

The demonstration was executed by inserting the TSI single board computer board into the VPH chassis as depicted in Figure 59. A Packard Bell PC was used for the VPH CRT and keyboard while the SBC had its own terminal and keyboard. The memory mapping described in the previous paragraph is illuminated in this figure when we observe that the address space of the SBC is 16-bits while that of the VPH is 32-bits. Hence, address modifier bits in the MVME 6000 were used to perform much of the translation between the VPH memory and the SBC memory. All of the standard VME bus control signals are available on the VPH backplane and all were used in the demonstration. However, only the frequently used control signals are shown in the figure.

The demonstration also consisted of exercising one, two, and four ZORAN DSP chips separately and together. Because a transparent bus arbitration PAL and scheme was designed into the VPH, it was relatively simple to turn single or multiple DSPs on and off. The procedure is to set up the status register in the VPH by the 68020 and let each ZORAN monitor their own "start" bit. If the bit is set, the respective ZORAN chip would initiate execution. Otherwise, it is suspended. Likewise, when a DSP chip has completed its current wave, it sets its "done" bit and stops. The 68020 monitors these bits as the board master. It is also possible for any resource on or off the VPH board to monitor these bits. Hence, the SBC can scan these bits as they are found in the public domain of the VME backplane. This feature is very useful when more than one master is exercising VME resources. This capability will support the SBC tracking and the VPH processing data in real-time. The intent of this design is to enable the VPH to process in the background while a front end, like the SBC, is acquiring the data. The "Status.ASM" code in Appendix B was used to demonstrate this capability.

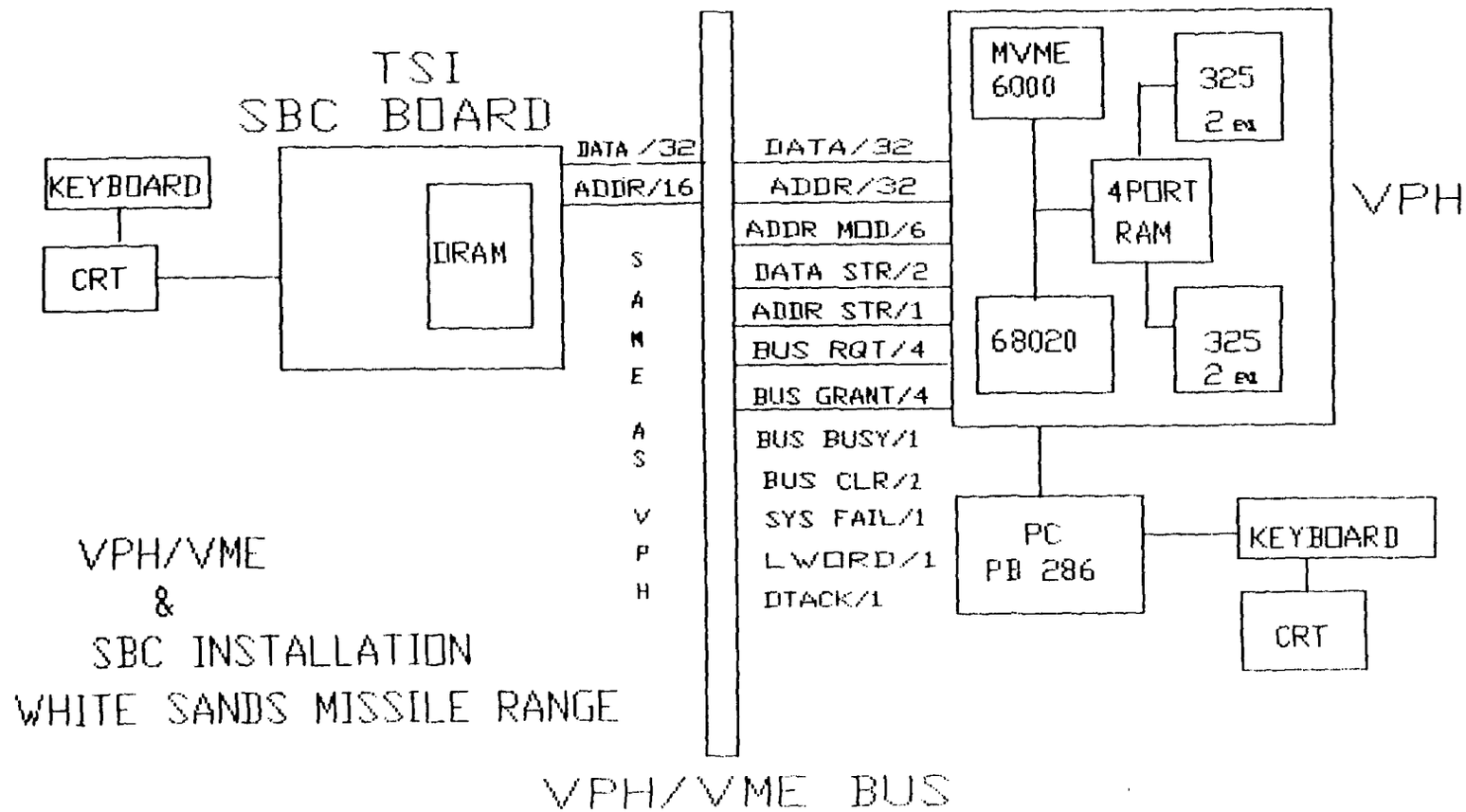


Figure 59. VPH Demo Setup

A number of other code segments were demonstrated. They include CConv.asm, Rect2pol.asm, FFT2d8.asm, Pol2rect.asm, FFT2d16.asm, FFT2d32.asm, Ccorr.asm, FFT1k.asm, Recip.asm, and Rconv.asm. All of these routines are found in Appendix B. Others are included there and are useful for performing diagnostics on the VPH. Furthermore, they can be used to help understand coding the DSPs. Specifically, "Test1.asm" and "Test2.asm" are useful for diagnosing the ZORAN chips and their interrupts, respectively.

For the 1KFFT, a random set of input points were chosen rather than a known set of points. In this way the DSPs were demonstrated as to accuracy and precision without any bias towards a known solution or output. The results of the FFT were then compared with those using the same input values to a standard C routine. The correlation program input used two signals. One was a square wave followed by a triangular wave. The other was an impulse function. The output of the correlator worked as expected. To verify our intuitive conclusions, it was necessary to zero pad the front end of the input data stream so that aliasing would not corrupt the interpretation.

6.2 CPH Conclusions

The CPH design effort was constantly buffeted by the technology envelope. An aggressiveness design stance was chosen at first to capture any and all new devices or promised devices. Among those included FPGAs and ASICs with performance specifications untried by designers. When the point of no return for fixing the design of the CPH came, some of the critical devices did not live up to advanced performance specifications. As a result the CPH design underwent more iterations than anticipated. The only conclusion to be drawn is that designers should not push the technology envelope.

Unexpectedly, available devices became unavailable as manufacturers became cost sensitized. Reduction of inventory became commonplace. The AMD 29540 FFT address sequencer, a staple for any FFT designer, was removed from AMD's catalogs. A work around required over 40 16-pin chips. The large number and size could not be supported by the available board space. Subsequently, the address generator board would have to produce FFT addresses in microcode. The VPH then became an important board to the EVA machine because it was capable of very fast FFTs.

7.0 Suggestions for Phase III

The CPH should be completed and fabricated. In doing so, the following is recommended. Clock distribution on the backplane could be done with a single TTL clock which will have low skew from board to board. Then, each board could have a digital delay line to adjust for the skew. Use a double rate clock so each board will develop 180 degree clocks for the quad. A single stepper on the backplane would be useful. Install a switch or jumper to make the selection between RUN and Single Step modes. Then debounce the switch manually or use a trigger so single stepping can be done from some external interface.

7.1 Backplane Design

The current EVA chassis has a 9U VME backplane and a custom backplane for the CPH side. The custom backplane is not complete. In the next development phase, this effort will require the selection of system clock circuitry. That circuitry may be distributed physically across this backplane. It will have to be, especially if ECL clocks are used. Six differential ECL clocks should be used with the same clock timings as shown for the cache memory board section. A provision for single stepping the clocks should also be provided for diagnosing system faults. A status bit on the PC-INT board may also be used here to control single stepping. Another desirable option would be to freeze the clock at 40 MHz and return it to the state just before the freeze. Obviously this will only be useful if entirely glitch free operation can occur. Finally, terminations need to be designed into the clock circuitry at the end of the lines so that overshoot is suppressed.

7.2 Integration of the EVA Computer

The VPH can be a standalone board or hosted via its VME bus directly to a SUN workstation or indirectly to a 6U VME system with a single board computer. Integration involves more than hardware, however. The system software of the host must be modified to make calls to the VPH, upload and download code and data, and manage the throughput of the VPH. Because VPH is so fast, the VME bus is not the best choice. Another high speed bus can be used if the host has the port. The VPH to CPH bus via the SIO channel uses the Gazelle hot rod chip set with gigabyte transfer rates. A future effort could examine the implementation of this path to a host.

If EVA is to be integrated fully to the TSI tracker, a good approach would be to put the TSI 6U VME backplane into the EVA chassis. This will allow the VPH to plug directly into the TSI backplane in one chassis and speed up operations further. It is a simple matter to mechanically modify the EVA chassis. Two new rails are necessary.

7.3 Crossbar Applications

The crossbar is an ultra fast switcher. It is general purpose so that any digital data gateway can benefit from its dynamically reconfigurable switching. The 12x14 In OUT paths are changeable in one single clock cycle. No other crossbar can do this. Also, the crossbar is cascadable so that each 4-bit slice can be expanded into any wordlength desired. Telemetry gateways

may benefit from this remarkably fast switcher. Wide area telephone net works could benefit from this powerful device. WSMR should consider a Phase III technology transfer with this chip to applications Army-wide.

7.4 Cascadability

Cascadability can be supported for fixed-point arithmetic. Use one block of memory and a processor board for the lower 32 bits of the 64-bit number. Use another configuration for the upper 32 bits. Microcode will have to be very sophisticated because the BIT chips do not provide all the necessary signals and flags. Also, the CPH throughput will fall off drastically. The better approach would be to use the 64-bit capability of the BIT chip directly.

Cascadability will require a local address bus so that the local CPH can use the HSIO bus without conflicting with the other CPH HSIO bus. Currently, the design supports an HSIO address that is broadcast everywhere. Additional hardware will be needed.

The system initialization bit is on the cache memory board in IO space. This bit will need to be set on the backplane and cleared by the HSIO. Hence, the AG has to be the principal owner of this bit. Each cache bank will have minor ownership. The IOP will need to monitor this bit so as to determine the system configuration (where multiple CPHs are installed).

7.5 EVA Extensions

The EVA architecture will prove to be a durable concept for many years. It should be completed to the extent possible by the new technological advances. Newer FPGAs and ASICs will greatly reduce the board space. Better transceivers will be available in late 1992. They should be considered for the HSIO bus. Also, since the BIT 3130 and 3120 ECL ALUs are available, a redesign of the CPH to include these 80 MHz devices may be advisable now that the system issues of EVA are formulated. However, selecting ECL ALUs may eliminate the need for the crossbars or modify them for nonpipelined application. Caution is advised in choosing 3130s etc., because these chips may also become unavailable in the future possibly being overcome in superior performance by the GaAs devices.

To fully support EVA, the MicroAsm microprogramming tool should include a linker and PROM formatter for the new PROMs. If the multiphase clocks for the WCS are to be kept, then MicroAsm should be updated to support multiphase microinstructions.

7.6 IOP Completion

The IOP is a general purpose IO traffic controller. The design can be completed by adding the boot state machine and some MUX data clocks (PCMUX,SIOMUX,HSIOMUX). Counters A and B enables should be added to the schematic. Also, the microsequencer design control signals need to fully time analyzed and certified for race and hazard free operation. This is on sheet 10 of the IOP schematic set.

To download microcode from the IOP to the processor, use the HSIO signal

lines labeled "DNLD ENABLE". At this point all boards should be in the "available" mode. Two new signal lines should be designated on the HSIO bus. They are DNLD REQT (from CPH to IOP) and DNLD FINISH (from IOP to CPH).

To upload condition codes of the processor board, use microcode bits to enable same. The AG will read the error FIFOs on the processor. Since there are many flags on the BIT chips (12), a 48 to 1 mux could be used to pass 1 flag only. Another flag could be the interrupt flag.

7.7 Wave Processing

The VPH application programs have been heavily optimized. However, there is always room for improvement especially when multiprocessing occurs. Some of those improvements were noted in the VPH User's Manual. If additional VPH boards are inserted into EVA, then wave processing can occur over 8 or 16 DSP chips with an attendant increase in performance. New code can then take advantage of this hardware extension.

7.8 VPH Augmented Bus

The VPH communicates with the CPH through the extra 32 bits in the VME space via an augmented bus. In this manner true parallel 64-bit transfers take place. For the SBC interaction across the 32-bit VME bus, this augmented bus is not needed. Hence, firmware in the VPH PALs would have to be regenerated if this augmented bus were to be activated. The PALs must allow for redirecting the upper half of normally unused memory space (for the SBC) back to the CPH address space. This is straightforward and a simple PAL reprogramming is necessary.

7.9 Phase III Opportunities

The VPH stands an excellent chance of technology transfer into many digital signal processing applications. Chief among those are those found in biomedical imaging applications and seismic signal processing. Both commercial applications need ultrafast FFTs. Both need over 1k length FFTs. Seismic data processing requires 1kx1k 2D FFTs. The VPH can handle very large FFTs but it might be better to add additional memory to the board first. This will reduce the off board data traffic. New and denser memory chips are now available and can be used in a mezzanine board for this purpose.

The crossbar Phase III opportunities have been presented already. The device itself should find many practical applications outside of computing.

APPENDIX A

CPH PROGRAMS

```
 1 - /*MULM.ASM*/
 2 - PROGRAM CODESEG MSRAM
 3 -
 4 - ORG 0
 5 -
 6 - start:  $SEQ  ,CONT
 7 - $IMMADD 0x0001
 8 - $ADDRA IMM,P0
 9 - $ADDRB IMM,P0
10 - $CACHE ,RDAB
11 - $REGAR,,BR;
12 -
13 - $SEQ   ,CONT
14 -     $M1 AR,P0MS,BR,P0MS,IMULT,EN.MS;
15 -
16 -       $SEQ ,CONT
17 -
18 -      $MWR AR,P0MS;
19 -
20 - PROGRAM ENDS
```

```
    1 - /*IMT.ASM, TESTS INTEGER MULTIPLICATION, R0XR1 = R2, INTEGER RESULT IS OUTPUT VIA IOR, LEAST SIGNIFICANT BITS ONLY
          (32-BIT MULT)*/
    2 -
    3 - PROGRAM CODESEG MSRAM
    4 - ORG 0
    5 - START: $SEQ ,CONT$REG CLEAR,0X00,CLEAR,0X01;
    6 - $SEQ ,CONT$REG ,,,,,0X00,0X01;
    7 - $SEQ ,CONT;
    8 - $SEQ ,CONT;    /*WAIT TWO CLOCKS FOR XBAR DATA AVA OUT*/
    9 - $SEQ ,CONT       $M1 REGA,P0LS,REGB,P0LS,IMULT,EN,LS;
   10 - $SEQ ,CONT       $REG M1,0X02;
   11 - $SEQ ,CONT;   /*HOLD M1 OUTPUT RESULT TO HERE FOR XBAR*/
   12 - $SEQ ,CONT       $REG ,,,,,0X02;
   13 - $SEQ ,CONT;
   14 - $SEQ ,CONT;
   15 - $SEQ ,CONT      $ IOR REGA,P0LS;
   16 - PROGRAM ENDS
```

```
 1 - /* IORZADD.ASM
 2 - R0=0
 3 - R1=0
 4 - R2=R1+R0
 5 - IOR=R2
 6 - Tests the ALU1 with the data register file. Adds two numbers from reg 0 and reg 1.Clears reg 0 and 1 with data from
      immediate field.  then adds reg0 to reg1. puts sum in reg2. operands are zero. result should be zero. result appea
      IO port real side (least significant only).*/
 7 -
 8 - PROGRAM CODESEG MSRAM
 9 - ORG 0
10 - START: $SEQ ,CONT
11 - $REG CLEAR,0X00,CLEAR 0X01;
12 - $SEQ ,CONT$REG ,,,,,0X00,0X01;
13 - $SEQ ,CONT;
14 - $SEQ ,CONT $A1 REGA,POLS,REGB,POLS,,IADD,EN,LS;
15 - $SEQ ,CONT $A1 ,,,,,,HOLD,LS$REG A1,0X02;
16 - $SEQ ,CONT$REG ,,,,,0X02;
17 - $SEQ ,CONT $IOR REGA,POLS;
18 - PROGRAM ENDS
```

```
    1 - /*IORADD.ASMIOR=IOR+IOIIN A CONTINUOUS LOOP, WARNING!NEED TO CHECK HOW IOR CONNECTS TO ALU1 PORT X AND IOI CONNECTS
          TO ALU1 PORT Y SIMULTANEOUSLY,ALSO, CHECK THE LOOP IMMEDIATE COMMAND FOR THE MICROSEQUENCER*/
    2 -
    3 - PROGRAM CODESEG MSRAM
    4 - ORG 0
    5 - LOOP: $SEQ ,CONT$IOR A1  $IOI A1;
    6 - $SEQ ,CONT;
    7 - $SEQ ,CONT;
    8 - $SEQ ,CONT $A1 IOR,POLS,IOI,POLS,,IADD,EN,LS;
    9 - $SEQ ,CONT;
   10 - $SEQ ,CONT;
   11 - $SEQ ,CONT $IOR A1,INLS;
   12 - $SEQ LOOP,LI;
   13 - PROGRAM ENDS
```

```
 1 - /*File:subfield.asm
 2 - Program showing examples of double use of SEQ.DATA field and the results
 3 - of the varying default definitions when different fields are omitted.  In
 4 - general, a default value will be used when its parent is specified but
 5 - the field itself is omitted.  Later specifications overwrite earlier ones.
 6 - */
 7 -
 8 - PROGRAMCODESEGTEST
 9 -
10 - ORG0
11 -
12 - TOP:
13 - /*This works properly, filling in label value. */
14 - $SEQ JMP,BOTTOM;
15 -
16 - /*This will use SEQ.INSTR & SEQ.DATA defaults since omitted. */
17 - $SEQ ;
18 -
19 - /*This will use SEQ.DATA default, since omitted. */
20 - $SEQ NOP
21 - /*Use dot to access subfield, overwrites SEQ.DATA default value. */
22 - $SEQ.DATA FIRST,0x3;
23 -
24 - /*With $SEQ not specified, SEQ default will be used and */
25 - /*the SEQ.INSTR and SEQ.DATA defaults will not be used. */
26 - /*Subfield definition overwrites SEQ.DATA part of SEQ default. */
27 - /*VALUE omitted so SEQ.DATA.VALUE default will be used. */
28 - $SEQ.DATA SECOND;
29 -
30 - /*This defines the SEQ.DATA subfield. */
31 - $SEQ.DATA THIRD,0x5
32 - /*Wrong order: SEQ.DATA overwritten with its default value. */
33 - $SEQ NOP;
34 -
35 - /*This defines the entire SEQ field. */
36 - $SEQ JMP,TOP
37 - /*Wrong: this overwrites the jump address without warning. */
38 - $SEQ.DATA THIRD,0x5;
39 -
40 - BOTTOM:
41 -
42 - PROGRAMENDS
```

APPENDIX B

VPH PROGRAMS

```
  1 - /*VPH code for convolution of a real sequence of up to 64 points with
  2 - another longer real sequence, producing up to 1024 outputs.  This
  3 - size can be done with a single FIR instruction.  This code can be
  4 - called repeatedly on a single processor to handle convolutions where
  5 - more than 1024 output points are required as long as the shorter
  6 - sequence is still less than 64 points.  However, a different routine
  7 - designed for a longer convolution would be more efficient.  This
  8 - same code can be used on multiple VSP chips simultaneously to give
  9 - a considerable speed increase.  There may be no benefit to executing
 10 - on more than one VSP chip per bus because the FIR instruction may not
 11 - give up the bus between output points.
 12 -
 13 - To get a full convolution of the input requires padding both ends of
 14 - the longer input sequence with a number of zeroes equal to the length
 15 - of the shorter sequence minus one.  This is required in order to
 16 - explicitly provide the zeroes that are assumed to be multiplied by
 17 - elements of the shorter sequence that extend beyond the ends of the
 18 - longer one during the convolution process.  The length of the output
 19 - sequence should be equal to the sum of the lengths of the (unpadded)
 20 - input sequences minus one.  If a circular convolution is desired
 21 - instead of a linear one, the zero padding should be replaced with
 22 - points from the other end of the input sequence.
 23 -
 24 - The shorter input length is passed in Coef_Length.  The output length
 25 - (equal to input length before padding plus coefficient length minus one)
 26 - is passed as Out Length.  Coefficients points to the shorter sequence
 27 - (typically FIR filter coefficients).  In_Data points to the start
 28 - of the longer sequence (possibly a zero pad).  The output is placed
 29 - at Out_Data.  Typical call for a four tap filter:
 30 - CALL RCONV(4, 1024, &Coef, &In, &Out)
 31 -
 32 - The convolution can be performed in place with careful choices of
 33 - parameter values.  If the convolution requires multiple calls on a
 34 - single VSP chip, the output must begin at the first location of the
 35 - long input.  This avoids overwriting inputs that will be needed for
 36 - the next call.  However, if multiple chips are being used, the output
 37 - must overwrite the last input used in its computation.  This works
 38 - because the VSP chip has already read the input into internal RAM
 39 - for further use.  It is necessary because that input is the first
 40 - one which will not be needed by the chip working on the previous
 41 - portion of the convolution.  Some further care is needed in the
 42 - initial startup of in-place multiple chip convolution to ensure that
 43 - a chip does not write over any input values before the subsequent
 44 - chip reads them in.  A multiple call, multiple chip convolution
 45 - cannot be done in place because the constraints are contradictory.
 46 - However, such a large data set would not fit into shared memory.
 47 -
 48 - Splitting up a convolution between NUM_CHIPS chips would require
 49 - something like the following invocation for chip ranging from zero
 50 - to (NUM_CHIPS - 1):
 51 -
 52 - CALL RCONV(COEF_LEN, OUT_SIZE(chip), &Coef,
 53 - &(In + DATA_OFFSET(chip)), &(Out + DATA_OFFSET(chip)));
 54 -
 55 - with the definitions
 56 -
 57 - #define OUT_LEN (IN_LEN + COEF_LEN - 1)
 58 - #define DATA_OFFSET(CHIP) (((CHIP) * OUT_LEN) / NUM_CHIPS)
 59 - #define OUT_SIZE(CHIP) (DATA_OFFSET(CHIP+1) - DATA_OFFSET(CHIP))
 60 -
 61 - Note that since all this routine does is to load various values into
 62 - internal registers and RAM and then execute a single instruction, it
 63 - might be faster for the 68020 to load the values directly and execute
 64 - the FIR instruction in slave mode.  The same applies to the complex
 65 - convolution and the correlations.
 66 - */
 67 -
 68 - zsp325()
 69 - {
 70 - /#
 71 - SUBROUTINE RCONV(zr325int Coef_Length,
 72 - zr325int Out_Length,
 73 - zr325ref Coefficients,
 74 - zr325ref In_Data,
 75 - zr325ref Out_Data)
 76 - {
 77 - /*set up mode properly, one RAM bank, 24 bit integers */
 78 - SET [ =RMS, =!XOR , =!FMT ];
 79 -
 80 - /*set $SAR to put output in correct place */
 81 - LDR Out_Data => $SAR;
 82 -
 83 - /*to get real coefficients in zig-zag order, need to load half
 84 - as many (rounded up) "complex" coefficients
 85 - */
 86 - SHLSETR:[SHIFT=17] Coef_Length => $PR;
 87 - ADDR $PR, #0x020000;
 88 -
 89 - /*load coefficients in reverse zig-zag real order */
 90 - LDR Coefficients => $A;
 91 - ADDR $A, Coef_Length;
 92 - SUBR $A, #2;
 93 - LD_(I,R):($MMPT) $A:(-1,1) => $C0;
 94 -
 95 - /*now set up actual lengths for FIR instruction */
 96 - SHLSETR:[SHIFT=16] Coef_Length => $PR;
 97 - ADDR $PR, Out_Length;
 98 -
 99 - /*convolve with input sequence  */
100 - FIR_R:($MMPT, $REPEAT) $Z0, *In_Data;
101 -
102 - }
103 - #/
104 - }
```

```
 1 - /*VPH code for convolution of a complex sequence of up to 32 points with
 2 - another longer complex sequence, producing up to 1024 outputs.  This
 3 - size can be done with a single FIR instruction.  This code can be
 4 - called repeatedly on a single processor to handle convolutions where
 5 - more than 1024 output points are required as long as the shorter
 6 - sequence is still no more than 32 points.  However, a different routine
 7 - designed for a longer convolution would be more efficient.  This
 8 - same code can be used on multiple VSP chips simultaneously to give
 9 - a considerable speed increase.  There may be no benefit to executing
10 - on more than one VSP chip per bus since the FIR instruction may not
11 - give up the bus between output points.
12 -
13 - To get a full convolution of the input requires padding both ends of
14 - the longer input sequence with a number of complex zeroes equal to the
15 - length of the shorter sequence minus one.  This is required in order
16 - to explicitly provide the zeroes that are assumed to be multiplied by
17 - elements of the shorter sequence that extend beyond the ends of the
18 - longer one during the convolution process.  The length of the output
19 - sequence should be equal to the sum of the lengths of the (unpadded)
20 - input sequences minus one.  If a circular convolution is desired
21 - instead of a linear one, the zero padding should be replaced with
22 - points from the other end of the input sequence.
23 -
24 - The shorter input length is passed in Coef_Length.  The output length
25 - (equal to input length before padding plus coefficient length minus one)
26 - is passed as Out_Length.  Coefficients points to the shorter sequence.
27 - In_Data points to the start of the longer sequence (possibly a zero
28 - pad).  The output is placed at Out_Data.  Typical call:
29 - CALL CCONV(4, 1024, &Coef, &In, &Out)
30 -
31 - The convolution can be performed in place with careful choices of
32 - parameter values.  If the convolution requires multiple calls on a
33 - single VSP chip, the output must begin at the first location of the
34 - long input.  This avoids overwriting inputs that will be needed for
35 - the next call.  However, if multiple chips are being used, the output
36 - must overwrite the last input used in its computation.  This works
37 - because the VSP chip has already read the input into internal RAM
38 - for further use.  It is necessary because that input is the first
39 - one which will not be needed by the chip working on the previous
40 - portion of the convolution.  Some further care is needed in the
41 - initial startup of in-place multiple chip convolution to ensure that
42 - a chip does not write over any input values before the subsequent
43 - chip reads them in.  A multiple call, multiple chip convolution
44 - cannot be done in place because the constraints are contradictory.
45 - However, such a large data set would not fit into shared memory.
46 -
47 - Splitting up a convolution between NUM_CHIPS chips would require
48 - something like the following invocation for chip ranging from zero
49 - to (NUM_CHIPS - 1):
50 -
51 - CALL CCONV(COEF_LEN, OUT_SIZE(chip), &Coef,
52 - &(In + 2*DATA_OFFSET(chip)), &(Out + 2*DATA_OFFSET(chip)));
53 -
54 - with the definitions
55 -
56 - #define OUT_LEN (IN_LEN + COEF_LEN - 1)
57 - #define DATA_OFFSET(CHIP) (((CHIP) * OUT_LEN) / NUM_CHIPS)
58 - #define OUT_SIZE(CHIP) (DATA_OFFSET(CHIP+1) - DATA_OFFSET(CHIP))
59 -
60 - DATA_OFFSET is doubled when used with pointer parameters because
61 - each complex element requires two machine words.
62 - */
63 -
64 - zsp325()
65 - {
66 - /#
67 - SUBROUTINE CCONV(zr325int Coef_Length,
68 - zr325int Out_Length,
69 - zr325ref Coefficients,
70 - zr325ref In_Data,
71 - zr325ref Out_Data)
72 - {
73 - /*set up mode properly, one RAM bank, 24 bit integers */
74 - SET [ =NMS, =!XOR , =!FMT ];
75 -
76 - /*set $SAR to put output in correct place */
77 - LDR Out_Data => $SAR;
78 -
79 - /*now set up lengths for LD and FIR instructions */
80 - SHLSETR:[SHIFT=18] Coef_Length => $PR;
81 - ADDR $PR, Out_Length;
82 -
83 - /*load coefficients in reverse order */
84 - LDR Coefficients => $A;
85 - ADDR $A, Coef_Length;
86 - SUBR $A, #2;
87 - LD_C:($NMPT) $A:(-1,1) => $C0;
88 -
89 - /*convolve with input sequence  */
90 - FIR_C:($NMPT, $REPEAT) $C0, *In_Data;
91 -
92 - }
93 - #/
94 - }
```

```
  1 - /*Routine to perform rectangular to polar conversion on a complex vector.
  2 - Uses a Cordic-like algorithm for magnitude and an arctangent lookup
  3 - table for angle in radians.  Maximum error in magnitude is 2% for
  4 - three iterations, which can easily be reduced to a value as low as
  5 - 0.0002% by increasing the number of iterations to eight.  Maximum error
  6 - in angle is 2.33% for 5 bits from each mantissa, which requires a table
  7 - of 1K entries for first quadrant angles only.  The table size must be
  8 - quadrupled for each doubling in precision, so this approach is not
  9 - practical for high precision.
 10 -
 11 - This program computes only first quadrant angles.  Other angles are
 12 - moved into the first quadrant by taking the absolute value of both
 13 - components.  This means that the angle will be correct for the first
 14 - quadrant, equal to pi minus the true angle in the second quadrant,
 15 - equal to the true angle minus pi in the third quadrant and equal to
 16 - minus the true angle in the fourth quadrant.  These angles are the
 17 - absolute values of the angles between the complex numbers and the
 18 - nearest real axis.  If full angles are needed, the table can just be
 19 - quadrupled to handle sign bits in the index.
 20 -
 21 - The vector length is passed in the parameter Length.  The parameter
 22 - In_Data points to the vector to be converted.  The output is placed
 23 - at Out_Data.  The conversion can be performed in place if desired.
 24 -
 25 - */
 26 -
 27 - /*need arctangent function for table */
 28 - #include <math.h>
 29 -
 30 - /*number of bits from each mantissa to be used in arctangent table lookup */
 31 - #define TAB_BITS 5
 32 -
 33 - /*number of Cordic iterations for magnitude calculations */
 34 - #define MAG_ITER 3
 35 -
 36 - /*function to return arctangent table value for index number */
 37 - /*only handles first quadrant angles, but could be modified for all four */
 38 - float tabentry(int i)
 39 - {
 40 - int fbits[2];
 41 - int part;
 42 - int index;
 43 -
 44 - /*determine numbers that would have produced the given index */
 45 - for (part = 0; part <= 1; part++)
 46 - {
 47 - /*extract interleaved mantissa bits from index */
 48 - fbits[part] = 0;
 49 - for (index = 0; index < TAB_BITS; index++)
 50 - {
 51 - fbits[part] |= (1 << index) & (i >> index + part);
 52 - }
 53 - }
 54 -
 55 - /*return middle angle of the possible range */
 56 - return (atan2((double) fbits[0] + 1, (double) fbits[1]) +
 57 - atan2((double) fbits[0], (double) fbits[1] + 1)) / 2.0;
 58 - }
 59 -
 60 -
 61 - /*actual assembly generation function */
 62 - zsp325()
 63 - {
 64 - int index;
 65 -
 66 - /*Generate arctangent table.  Because of normalization, only first
 67 - entry and last three quarters of table are actually used.
 68 - */
 69 - /#
 70 - AtanTab::
 71 - #/
 72 - for (index = 0; index < (1 << TAB_BITS*2); index++)
 73 - {
 74 - /#
 75 - .DATA { (IEEE_Float(tabentry(index))) };
 76 - #/
 77 - }
 78 -
 79 - /#
 80 - SUBROUTINE RECT2POL(zr325int Length, zr325ref In_Data, zr325ref Out_Data)
 81 - {
 82 -
 83 - /*set up two RAM sections, swapping on each loop iteration */
 84 - SET [ =INMS, =XOR ];
 85 -
 86 - /*load data pointers, parameter order gets In_Data into $A */
 87 - LDR Out_Data => [$B, $A];
 88 -
 89 - /*initialize loop count to number of 32s, skip loop if none */
 90 - SHRSETR:[SHIFT=5] Length => $LC;
 91 - JMPC [ZR], Do_Rest;
 92 -
 93 - /*first part of loop to fill software pipeline */
 94 -
 95 - /*load to bank 1, take absolute value to put in first quadrant */
 96 - LD_||:(32) $A => $C1;
 97 - /*align mantissas and interleave to create atan index in $I0 */
 98 - ALIGN:(32) $R1, $I1 => $I0;
 99 - /*do cordic iterations to get magnitude in $R1, takes a while */
100 - MAG:(32,MAG_ITER) $C1;
101 - /*look up arctangent in table, overlaps with MAG */
102 - LUT:(32):[SHIFT=(23 - 2*TAB_BITS)] AtanTab, $I0 => $I0;
103 - /*store angle, overlaps with MAG */
104 - ST_I:(32) $I0 => $B+=1:(2,1);
105 -
106 - /*decrement $LC, end loop if done */
107 - JMPC:[IE,DL] [LZ], Do_Store;
108 -
109 - /*software pipelined loop, allows next load to overlap MAG */
```

```
110 - Loop::
111 - LD ||:(32) $A+=64 => $C1;
112 - /*store magnitude from previous vector */
113 - ST_R:(32) $R0 => $B-1:(2,1);
114 - ALIGN:(32) $R1, $I1 => $I0;
115 - MAG:(32,MAG_ITER) $C1;
116 - LUT:(32):[SHIFT=(23 - 2*TAB_BITS)] AtanTab, $I0 => $I0;
117 - ST_I:(32) $I0 => $B+=64:(2,I);
118 - /*decrement counter and branch to top if not done */
119 - JMPC:[IE:1,DL:1] [ILZ], Loop;
120 -
121 - Do_Store::
122 - /*rest of loop to empty software pipeline */
123 - /*store magnitude from last vector */
124 - ST_R:(32) $R0 => $B-1:(2,1);
125 -
126 - Do_Rest::
127 - /*handle remainder left after blocks of 32 */
128 -
129 - /*shift remainder into $NMPT, use [TC] to zero high bit */
130 - SHLSETR:[SHIFT=18,TC] Length => $PR;
131 - JMPC [ZR], End;
132 -
133 - /*need MAG_ITER in $REPEAT to use $PR with MAG */
134 - ADDR $PR, #MAG_ITER;
135 -
136 - /*finish up remainder */
137 - LD ||:($NMPT) $A+=64 => $C1;
138 - ALIGN:($NMPT) $R1, $I1 => $I0;
139 - MAG:($NMPT,$REPEAT) $C1;
140 - LUT:($NMPT):[SHIFT=(23 - 2*TAB_BITS)] AtanTab, $I0 => $I0;
141 - ST_I:($NMPT) $I0 => $B+=64:(2,I);
142 - ST_R:($NMPT) $R1 => $B-1:(2,1);
143 -
144 - End::
145 -
146 - }
147 - #/
148 - }
```

```
  1 - /*Program to compute 8x8 2D complex FFT using one VSP chip.
  2 -
  3 - The parameter In_Data points to the input vector.  The output vector
  4 - is placed at Out_Data.  The operation can be performed in place if
  5 - desired.  Both input and output vectors are in normal order.
  6 -
  7 - To get an inverse FFT, just change the subroutine name and change the
  8 - FFT instructions to IFFT instructions.
  9 -
 10 - To use real data, change LD_C to LD_(R,0).
 11 -
 12 - Might be able to squeeze a little more speed out by starting with
 13 - two RAM sections, load first, FFT first rows, load second, FFT second
 14 - rows, switch to one RAM section, FFT columns, store.
 15 -
 16 - */
 17 -
 18 - zsp325()
 19 - {
 20 - /#
 21 - SUBROUTINE FFT2D8(zr325ref In_Data, zr325ref Out_Data)
 22 - {
 23 - /*set up one RAM section */
 24 - SET [ =NMS, =!XOR ];
 25 -
 26 - /*load all 64 entries, with rows bit reversed */
 27 - LD_C:(64) *In_Data:(8,8~) => $0;
 28 -
 29 - /*FFT the rows, result in normal order */
 30 - FFT_C:(8,8):[FPS:1,LPS:4] $0~, $ROM=0:512;
 31 -
 32 - /*FFT the columns, result in bit reversed order */
 33 - FFT_C:(64):[FPS:32,LPS:8] $0;
 34 -
 35 - /*store result, bit reversing columns into normal order */
 36 - ST_C:(64) $0 => *Out_Data:(8~,8);
 37 - }
 38 - #/
 39 - }
```

```
 1 - /*Routine to find peak values in a real matrix.  By varying parameters, it
 2 - can produce a vector of the max value in each row or column or the max
 3 - value in the entire matrix.  The calculation can be divided between
 4 - multiple VSP chips by giving each one a contiguous subset of the problem.
 5 - The maximum amplitude of a complex matrix can be found by first computing
 6 - the power (magnitude squared) and finding the maximum of that.  If the
 7 - magnitude itself is required, it is probably still faster to find the
 8 - peaks first and then compute the magnitude for only those points rather
 9 - than computing all the magnitudes and finding the peaks.
10 -
11 - The routine has a large number of parameters to allow it to be used in a
12 - flexible manner.  The parameter Number gives the number of separate
13 - vectors (rows or columns) to find the maximum for.  The parameter Length
14 - gives the length of each vector (row or column).  Length must be no more
15 - than 1024 for this routine, though a slight modification would allow up
16 - to 64K.  The input parameter Spacing gives the distance between starting
17 - elements of consecutive vectors.  The input parameter Interleave gives
18 - the distance between consecutive elements within a vector.  Due to some
19 - constraints on the $MBS_MSS register, bit 24 must also be set in the
20 - parameter.  Such a machine word can only be created at assembly time.
21 - It can be created directly by using a parameter ARG(value) with the
22 - macro definition
23 -
24 - #define ARG(X) (0x1000000 | (X))
25 -
26 - or by using a parameter that points to such a value created at assembly
27 - time.  As a slight compensation, a value other than 1 can be placed in
28 - the field from bit 24 to 30.  This value will be used as the $MBS value
29 - while the rest of the Interleave value is used as $MSS.  This allows
30 - for each vector to be addressed more generally.  If the $MBS_MSS register
31 - already contains an appropriate value, it can be passed.  The parameter
32 - In_Data points to the start of the first input vector.  The output will
33 - be placed at Out_Data.  The output will consist of a vector of length
34 - Number of pairs of maximum values and the index between 0 and Length of
35 - where that value appeared.
36 -
37 -
38 - To find the maximum row values for a ROWxCOL matrix using N VSP chips
39 - PEAK2D(COL/N, ROW, ROW, ARG(1), &(In+CHIP*ROW*COL/N), &(Out+CHIP*2*COL/N))
40 -
41 - To find the maximum column values for a ROWxCOL matrix using N VSP chips
42 - PEAK2D(ROW/N, COL, 1, ARG(ROW), &(In+CHIP*ROW/N), &(Out+CHIP*2*COL/N))
43 -
44 - assuming that ROW and COL are evenly divisible by N.  Using more than
45 - one chip on each local bus will probably not improve performance because
46 - the operation is bus-bandwidth bound.  When using two chips, CHIP should
47 - be set to 0 or 1 in the above formulas.
48 -
49 - To find the overall maximum, treat as one long row using 1 VSP chip
50 - PEAK2D(1, ROW*COL, any_value, ARG(1), &In, &Out)
51 -
52 - To find minimum values, just change the MAX instruction to MIN.
53 -
54 -
55 - Note: It is technically possible to accomplish the setting of the upper
56 - bits of $MBS_MSS at execution time with sufficient ingenuity.  It requires
57 - using (slow) floating point operations to manipulate the higher bits.  A
58 - lookup table is another possibility.
59 -
60 - */
61 -
62 - zsp325()
63 - {
64 - }#
65 - SUBROUTINE PEAK2D(zr325int Number,
66 - zr325int Length,
67 - zr325int Spacing,
68 - zr325val Interleave,
69 - zr325ref In_Data,
70 - zr325ref Out_Data)
71 - {
72 -
73 - /*set up automatic save to $SAR */
74 - SET [ =SAR ];
75 -
76 - /*set up parameters in correct registers
77 - note: LDRs depend on parameter order to put In_Data into
78 -       $A, Interleave into $MBS_MSS and Number into $LC.
79 - */
80 - LDR Out_Data => [$SAR, $A, $MBS_MSS];
81 - LDR Length => [$PR, $LC];
82 -
83 -
84 - /*loop Number times, handling Length each time, addressing properly */
85 - MAX_R:($NMPT,$REPEAT) $A:($MSS,$MBS) => $MNMX;
86 - ADDR $A, Spacing;
87 - LOOP:[IE,DL] [!LZ], #2;
88 -
89 - }
90 - #/
91 -
92 - }
```

```
 1 - /*Routine to compute magnitude squared for a complex vector.  If the vector
 2 - is the FFT of a signal, this is the power spectrum of the signal.  This
 3 - routine is faster than the rectangular to polar conversion and should be
 4 - used if the magnitude squared is as useful as the magnitude.  For example,
 5 - the point of maximum magnitude is also the point of maximum power.
 6 -
 7 - This routine can be performed in place, producing an output vector half
 8 - the length of the input.  This would leave gaps if multiple VSP chips
 9 - were being used.  If the calculation is not performed in place, or gaps
10 - are acceptable, there is no problem using multiple chips to calculate
11 - parts of the output vectors.
12 -
13 - Note: this routine is I/O bound even on a single VSP.  With two sharing
14 - a bus, it will be even worse.  If it is being used immediately after an
15 - FFT operation, it would be more efficient to perform the magnitude
16 - squared operation as the last step of an FFT routine before storing the
17 - result.  This would save a store and reload.
18 -
19 - The input parameter Length contains the number of elements in the
20 - input vector.  The parameter In_Data points to the start of the
21 - input vector.  The output will be placed at Out_Data.
22 -
23 - */
24 -
25 - zsp325()
26 - {
27 - /#
28 - SUBROUTINE POWER(zr325int Length, zr325ref In_Data, zr325ref Out_Data)
29 - {
30 -
31 - /*use both RAM banks to improve throughput */
32 -
33 - /*set up two RAM sections, swapped by $LC */
34 - SET [ =!NMS, =XOR ];
35 -
36 - /*set up pointers to data areas, compensate $B for pre-increment */
37 - /*Note: Load depends on parameter order to get In_Data into $A */
38 - LDR Out_Data => [$B, $A] ;
39 - SUBR $B, #32;
40 -
41 - /*initialize loop count to number of 32s, skip loop if none */
42 - SHRSETR:[SHIFT=5] Length => $LC;
43 - JMPC [ZR], Do_Rest;
44 -
45 - /*start up with first RAM bank */
46 - LD_C:(32) $A => $C0;
47 - MGSQ_R:(32) $C0 => $R0;
48 -
49 - /*if no more to do, skip rest of loop */
50 - JMPC:[IE,DL] [LZ], Do_Store;
51 -
52 - /*loop with software pipeline, XOR with $LC alternates RAM */
53 - LD_C:(32) $A+=64 => $C0;
54 - MGSQ_R:(32) $C0 => $R0;
55 - ST_R:(32) $R1 => $B+=32;
56 - LOOP:[IE,DL] [!LZ], #3;
57 -
58 - Do_Store::
59 - /* save last RAM bank */
60 - ST_R:(32) $R1 => $B+=32;
61 -
62 - Do_Rest::
63 - /*handle remainder left after blocks of 32 */
64 -
65 - /*shift remainder into $NMPT, use [TC] to zero high bit (32s) */
66 - SHLSETR:[SHIFT=18,TC] Length => $PR;
67 - JMPC [ZR], End;
68 -
69 - /*finish up remainder */
70 - LD_C:($NMPT) $A+=64 => $C0;
71 - MGSQ_R:($NMPT) $C0 => $R0;
72 - ST_R:($NMPT) $R0 => $B+=32;
73 -
74 - End::
75 -
76 - }
77 - #/
78 -
79 - }
```

```
 1 - /*Code to notify 68020 of task completion.  This code is never actually
 2 - called from anywhere.  Instead, its address is used as the return
 3 - address in the call frame that the 68020 sets up when invoking another
 4 - routine.  When the routine completes and returns, it will execute this
 5 - code.  This method allows all routines to be called without having
 6 - them terminate the task until final completion.
 7 - */
 8 -
 9 - /*status bit value to indicate finished */
10 - #define FINISHED 2
11 -
12 - zsp325()
13 - {
14 - /#
15 - SUBROUTINE FINISH()
16 - {
17 - /*get value for status bits */
18 - LDR #FINISHED => $X;
19 -
20 - /*make sure all operations are complete */
21 - SYNC:[AS,CU,EU,MU];
22 -
23 - /*write to global status latch */
24 - STR $X => 0x40000;
25 -
26 - /*halt */
27 - HLT;
28 - }
29 - #/
30 - }
```

```
  1 - /*Routine to perform polar to rectangular conversion on a complex vector.
  2 - Uses separate sine and cosine tables.  Could use one table for both,
  3 - but that would require extra time.  Only operates on angles in the first
  4 - quadrant since those are the only ones produced by the rectangular to
  5 - polar conversion.  The table size will determine the accuracy of the
  6 - conversion.  The error will be less than 100% * pi / (4 * table size).
  7 -
  8 - The vector length is passed in the parameter Length.  The parameter
  9 - In_Data points to the start of the vector to be converted.  The result
 10 - is placed at Out_Data.  This algorithm can be performed in place if
 11 - desired.
 12 -
 13 - This routine uses software pipelining to maximize throughput.  This
 14 - should cause the bus to be busy most of the time.  If two chips are
 15 - performing this at the same time, there will not be enough bandwidth.
 16 - Benchmarking will need to be used to determine whether this is faster
 17 - than a version which does not attempt pipelining but uses larger blocks.
 18 - */
 19 -
 20 - /*need trig functions for tables */
 21 - #include <math.h>
 22 -
 23 - /*size of sine and cosine tables */
 24 - #define TAB_SIZE 128
 25 -
 26 - /*size of increment between table entries */
 27 - #define INCREMENT (asin(1.0)/(TAB_SIZE-1))
 28 -
 29 - /*assembly generation function */
 30 - zsp325()
 31 - {
 32 - int index;
 33 -
 34 - /*Generate trig function tables.  */
 35 - /#
 36 - SinTab::
 37 - #/
 38 - for (index = 0; index < TAB_SIZE; index++)
 39 - {
 40 - /#
 41 - .DATA { (IEEE_Float(sin(index*INCREMENT))) };
 42 - #/
 43 - }
 44 - /#
 45 - CosTab::
 46 - #/
 47 - for (index = 0; index < TAB_SIZE; index++)
 48 - {
 49 - /#
 50 - .DATA { (IEEE_Float(cos(index*INCREMENT))) };
 51 - #/
 52 - }
 53 -
 54 - /#
 55 - SUBROUTINE POL2RECT(zr325int Length, zr325ref In_Data, zr325ref Out_Data)
 56 - {
 57 -
 58 - /*use both RAM banks to optimize throughput */
 59 - /*Note: chosen interleaving pattern assumes LUT instruction
 60 - makes no use of EU since it is a data movement instruction.
 61 - Also assumes that arithmetic operations that use external
 62 - operands can't be overlapped with move instructions, though
 63 - this isn't clear.
 64 - Benchmark might be needed to check the interleaving pattern.
 65 - */
 66 -
 67 - /*set up two RAM sections, swapped by $LC, round to nearest */
 68 - SET [ =!NMS, =XOR, =ROUND ];
 69 -
 70 - /*load pointers to data, shifting $A to angle, compensate pre-inc */
 71 - ISETR In_Data => $A;
 72 - LDR Out_Data => $B;
 73 - SUBR [$B, $A], #64;
 74 -
 75 - /*initialize loop count to number of 32s, skip loop if none */
 76 - SHRSETR:[SHIFT=5] Length => $LC;
 77 - JMPC [ZR], Do_Rest;
 78 -
 79 - /*start up conversion with first RAM bank */
 80 - /*load angle into imaginary part */
 81 - LD I:(32) $A+=64:(2,1) => $I0;
 82 - /*multiply by factor to get table offset */
 83 - MULT (R,R):(32) $C0, #(IEEE_Float(1.0/INCREMENT)) => $I0;
 84 - /*convert to integer to get integer part right justified */
 85 - FPINT_R:(32) $I0 => $I0;
 86 -
 87 - /*if no more to do, skip rest of loop */
 88 - JMPC:[IE,DL] [LZ], Do_Store;
 89 -
 90 - /*loop with software pipelining */
 91 - Loop::
 92 - /*load and start next vector */
 93 - LD I:(32) $A+=64:(2,1) => $I0;
 94 - MULT (R,R):(32) $C0, #(IEEE_Float(1.0/INCREMENT)) => $I0;
 95 - /*do bus operation for previous during execution of current */
 96 - LUT_R:(32) CosTab, $I1 => $R1;
 97 - /*do next operation on current vector */
 98 - FPINT_R:(32) $I0 => $I0;
 99 - /*finish and store previous vector */
100 - LUT_R:(32) SinTab, $I1 => $I1;
101 - /*assume external operand fetch monopolizes bus unit */
102 - MULT (R,R):(32) $C1, $A-1:(2,1) => $C1;
103 - ST C:(32) $C1 => $B+=64;
104 - /*decrement count (switches banks) and loop immediately if not done */
105 - JMPC:[DL,IE] [!LZ], Loop;
106 -
107 - Do_Store::
108 - /*finish up last RAM bank */
109 - /*look up cosine of angle in table */
```

```
110 - LUT_R:(32) CosTab, $I1 => $R1;
111 - /*look up sine of angle in table */
112 - LUT_R:(32) SinTab, $I1 => $I1;
113 - /*multiply cosine and sine by magnitude to get real and imaginary */
114 - MULT_(R,R):(32) $C1, $A-1:(2,1) => $C1;
115 - /*store resulting complex number in rectangular coordinates */
116 - ST_C:(32) $C1 => $B+=64;
117 -
118 - Do_Rest::
119 - /*handle any remainder left after blocks of 32 */
120 -
121 - /*shift remainder into $NMPT, use [TC] to zero high bit (32s) */
122 - SHLSETR:[SHIFT=18,TC] Length => $PR;
123 - JMPC [ZR], End;
124 -
125 - /*finish remainder */
126 - /*load angle into imaginary part */
127 - LD_I:($NMPT) $A+=64:(2,1) => $I0;
128 - /*multiply by factor to get table offset */
129 - MULT_(R,R):($NMPT) $C0, #(IEEE_Float(1.0/INCREMENT)) => $I0;
130 - /*convert to integer to get integer part right justified */
131 - FPINT_R:($NMPT) $I0 => $I0;
132 - /*look up cosine of angle in table */
133 - LUT_R:($NMPT) CosTab, $I0 => $R0;
134 - /*look up sine of angle in table */
135 - LUT_R:($NMPT) SinTab, $I0 => $I0;
136 - /*multiply cosine and sine by magnitude to get real and imaginary */
137 - MULT_(R,R):($NMPT) $C0, $A-1:(2,1) => $C0;
138 - /*store resulting complex number in rectangular coordinates */
139 - ST_C:($NMPT) $C0 => $B+=64;
140 -
141 - End::
142 -
143 - }
144 - #/
145 - }
```

```
/*Routine to perform polar to rectangular conversion on a complex vector.
Uses separate sine and cosine tables.  Could use one table for both, but
that would require extra time.  Only operates on angles in the first
quadrant since those are the only ones produced by the rectangular to
polar conversion.  Other angles will produce unexpected results.  The
table size will determine the accuracy of the conversion.  The error
will be less than 100% * pi / (4 * table size).

Length of the vector to be converted is passed in Length.  In_Data
points to the start of the input vector.  Output is placed at location
Out_Data.  Conversion can be performed in place if desired.

This version assumes performance is bounded by local bus bandwidth and
therefore doesn't attempt software pipelining alternating RAM banks.
Instead it uses the entire RAM at once to minimize bus traffic for
instruction fetching.  This also makes the code more readable.  Testing
will be needed to see which method is faster.  Using half of RAM and
loading magnitude in other half before MULT might save more bandwidth.
*/

/*need trig functions for tables */
#include <math.h>

/*size of sine and cosine tables */
#define TAB_SIZE 128

/*size of increment between table entries */
#define INCREMENT (asin(1.0)/(TAB_SIZE-1))

/*assembly generation function */
zsp325()
{
int index;

/*Generate trig function tables. */
/#
SinTab::
#/
for (index = 0; index < TAB_SIZE; index++)
{
/#
.DATA { (IEEE_Float(sin(index*INCREMENT))) };
#/
}
/#
CosTab::
#/
for (index = 0; index < TAB_SIZE; index++)
{
/#
.DATA { (IEEE_Float(cos(index*INCREMENT))) };
#/
}


/#
SUBROUTINE POL2RECT(zr325int Length, zr325ref In_Data, zr325ref Out_Data)
{

/*set up one RAM section, set rounding to nearest */
SET [ =RMS, =!XOR, =ROUND ];

/*load pointers to data, compensate for pre-increment */
/*increment $A at load so it points to angle part */
ISETR In_Data => $A;
LDR Out_Data => $B;
SUBR [$A, $B], #128;

/*initialize loop count to number of 64s, skip loop if none */
SHRSETR:[SHIFT=6] Length => $LC;
JMPC [ZR], Do_Rest;

Loop::
/*load angle into imaginary part */
LD I:(64) $A+=128:(2,1) => $I;
/*multiply by factor to get table offset */
MULT_(R,R):(64) $C, #(IEEE_Float(1.0/INCREMENT)) => $I;
/*convert to integer to get integer part right justified */
FPINT_R:(64) $I => $I;
/*look up cosine of angle in table */
LUT_R:(64) CosTab, $I => $R;
/*look up sine of angle in table */
LUT_R:(64) SinTab, $I => $I;
/*multiply cosine and sine by magnitude to get real and imaginary */
MULT_(R,R):(64) $C, $A-1:(2,1) => $C;
/*store resulting complex number in rectangular coordinates */
ST C:(64) $C => $B+=128;
/*decrement $LC, loop immediately on not zero */
JMPC:[DL,IE] [!LZ], Loop;

Do_Rest::
/*handle remainder left after blocks of 64 */

/*shift remainder into $MMPT, skip if none */
SHLSETR:[SHIFT=18] Length => $PR;
JMPC [ZR], End;

/*finish remainder */
/*load angle into imaginary part */
LD I:($MMPT) $A+=128:(2,1) => $I;
/*multiply by factor to get table offset */
MULT_(R,R):($MMPT) $C, #(IEEE_Float(1.0/INCREMENT)) => $I;
/*convert to integer to get integer part right justified */
FPINT_R:($MMPT) $I => $I;
/*look up cosine of angle in table */
LUT_R:($MMPT) CosTab, $I => $R;
/*look up sine of angle in table */
LUT_R:($MMPT) SinTab, $I => $I;
/*multiply cosine and sine by magnitude to get real and imaginary */
```

```
110 - MULT_(R,R):($NMPT) $C, $A-1:(2,1) => $C;
111 - /*store resulting complex number in rectangular coordinates */
112 - ST_C:($NMPT) $C => $B+=128;
113 -
114 - End::
115 -
116 - }
117 - #/
118 - }
```

```
/*Routines to compute 16x16 2D complex FFT using four VSP chips.

Operation requires two phases of operation, one to calculate row
FFTs, the other to calculate column FFTs.  Using multiple VSP chips
requires synchronization between phases so that data can be exchanged.
These routines do not include the sychronization.  The routines for
each phase can be called from another routine which provides it between
calls, or the 68020 can invoke the first phase and wait for it to
finish before invoking the second.

Each VSP chip could calculate its four rows or columns in one instruction,
but using two RAM sections allows more concurrency.  Each chip should
be passed data pointers to row or column (CHIP * 4) with CHIP equalling
0, 1, 2, or 3, depending on the chip.

The parameter In_Data points to the input vector.  The output vector
is placed at Out_Data.  The operation can be performed in place if
desired.  Both input and output vectors are in normal order.

To get an inverse FFT, just change the subroutine name and change the
FFT instructions to IFFT instructions.

To use real data, either set the imaginary parts to zero to get a complex
vector, or change LD_C to LD_(R,0) to use a real vector.  With a real
vector, this operation cannot be performed in place, since the output
data would overwrite unread input data.

*/

zsp325()
{
}#
/*FFT for rows, four 16 point FFTs on sequential data */
SUBROUTINE FFT16ROW(zr325ref In_Data, zr325ref Out_Data)
{
/*set up two RAM sections, no need for exchange */
SET [ =!NMS, =!XOR ];

/*set up pointers for later offset, $A gets In_Data */
LDR Out_Data => [$B, $A];

/*load two rows into section 0 */
LD_C:(32) $A => $C0;

/*FFT as two 16 element FFTs */
FFT_C:(16,2):[FPS:8,LPS:1] $C0;

/*load remainder of entries into section 1  /
LD_C:(32) $A+64 => $C1;

/*FFT as two 16 element FFTs */
FFT_C:(16,2):[FPS:8,LPS:1] $C1;

/*store first result, row bit-reversed */
ST_C:(32) $C0 => $B:(16,16~);

/*store second result, row bit-reversed */
ST_C:(32) $C0 => $B+64:(16,16~);

}

/*FFT for columns, four 16 point FFTs on interleaved data */
SUBROUTINE FFT16COL(zr325ref In_Data, zr325ref Out_Data)
{
/*set up two RAM sections, no need for exchange */
SET [ =!NMS, =!XOR ];

/*set up pointers for later offset, $A gets In_Data */
LDR Out_Data => [$B, $A];

/*load two columns interleaved into section 0 */
LD_C:(32) $A:(16,2) => $C0;

/*FFT first set as two 16 element FFTs */
FFT_C:(32):[FPS:16,LPS:2] $C0;

/*load remainder of entries into section 1 */
LD_C:(32) $A+2:(16,2) => $C1;

/*FFT as two 16 element FFTs */
FFT_C:(32):[FPS:16,LPS:2] $C1;

/*store first result, columns bit-reversed */
ST_C:(32) $C0 => $B:(16~,2);

/*store second result, columns bit-reversed */
ST_C:(32) $C0 => $B+2:(16~,2);

}
#/
}
```

```
 1 - /*Test program for Zoran interrupts.  Status bits follow interrupt bit.
 2 - */
 3 -
 4 - /*absolute base addresses from memory map */
 5 - #define PRAM 0x00000
 6 - #define FOUR_PORT 0x20000
 7 - #define STATUS_LATCH 0x40000
 8 -
 9 - zsp325()
10 - {
11 - /#
12 -
13 - INTERRUPT SUBROUTINE SET_HALT()
14 - {
15 - /*write 1s to status latch and wait */
16 - LDR #3 => $X;
17 - STR $X => STATUS_LATCH;
18 - Poll::
19 - JMPC [IEI], Poll;
20 -
21 - /*after resume, clear status bits */
22 - LDR #0 => $X;
23 - STR $X => STATUS_LATCH;
24 - }
25 -
26 - .EXTERN _SubEntry_SET_HALT;
27 -
28 - SUBROUTINE MAIN()
29 - {
30 - /*set interrupt vector (happens to be 0, but why not) */
31 - LDR &_SubEntry_SET_HALT => $IP;
32 -
33 - /*write 0 to status latch */
34 - LDR #0 => $X;
35 - STR $X => STATUS_LATCH;
36 -
37 - /*infinite loop decrementing $LC from 0 */
38 - MOVR $X => $LC;
39 - Loop::
40 - JMP:[DL] Loop;
41 -
42 - }
43 -
44 - #/
45 - }
```

```
  1 - /*Routines to compute 32x32 2D complex FFT using four VSP chips.
  2 -
  3 - Operation requires two phases of operation, one to calculate row
  4 - FFTs, the other to calculate column FFTs.  Using multiple VSP chips
  5 - requires synchronization between phases so that data can be exchanged.
  6 - These routines do not include the sychronization.  The routines for
  7 - each phase can be called from another routine which provides it between
  8 - calls, or the 68020 can invoke the first phase and wait for it to
  9 - finish before invoking the second.
 10 -
 11 - Each chip should be passed pointers to row or column (CHIP * 8) with
 12 - CHIP equalling 0, 1, 2, or 3, depending on the chip.  It will handle
 13 - the 8 rows or columns starting at that point.  Adding a parameter to
 14 - give the number of rows or columns to do would allow the same routine
 15 - to be used by 1 or 2 chips without needing to make multiple calls.
 16 -
 17 - The parameter In_Data points to the input vector.  The output vector
 18 - is placed at Out_Data.  The operation can be performed in place if
 19 - desired.  Both input and output vectors are in normal order.
 20 -
 21 - To get an inverse FFT, just change the subroutine name and change the
 22 - FFT instructions to IFFT instructions.
 23 -
 24 - To use real data, either set the imaginary parts to zero to get a complex
 25 - vector, or change LD_C to LD_(R,0) to use a real vector.  With a real
 26 - vector, this operation cannot be performed in place, since the output
 27 - data would overwrite unread input data.
 28 -
 29 - */
 30 -
 31 - zsp325()
 32 - {
 33 - }#
 34 - /*FFT for rows, eight 32 point FFTs on sequential data */
 35 - SUBROUTINE FFT32ROW(zr325ref In_Data, zr325ref Out_Data)
 36 - {
 37 -
 38 - /*set up two RAM sections, swapped by $LC */
 39 - SET [ =!NMS, =XOR ];
 40 -
 41 - /*set pointers to input and output, compensate for increment */
 42 - /*note: depending on parameter order to get In_Data into $A */
 43 - LDR Out_Data => [$B, $A];
 44 - SUBR $B, #64;
 45 -
 46 - /*initialize loop count */
 47 - LDR #7 => $LC ;
 48 -
 49 - /*start up FFT with first RAM bank */
 50 - LD_C:(32) $A => $C1;
 51 - FFT_C:(32) $C1, $ROM=0:0;
 52 -
 53 - /*loop 7 times, XOR with $LC alternates RAM */
 54 - LD_C:(32) $A+=64 => $C0;
 55 - FFT_C:(32) $C0, $ROM=0:0;
 56 - ST_C:(32) $C1 => $B+=64:(32,1)~;
 57 - LOOP:[DL:1] [!LZ], #3;
 58 -
 59 - /* save last RAM bank */
 60 - ST_C:(32) $C1 => $B+=64:(32,1)~;
 61 -
 62 - }
 63 -
 64 - /*FFT for columns, eight 32 point FFTs on interleaved data */
 65 - SUBROUTINE FFT32COL(zr325ref In_Data, zr325ref Out_Data)
 66 - {
 67 - /*set up two RAM sections, swapped by $LC */
 68 - SET [ =!NMS, =XOR ];
 69 -
 70 - /*set pointers to data, compensate for first increment */
 71 - /*note: depending on parameter order to get In_Data into $A */
 72 - LDR Out_Data => [$B, $A];
 73 - SUBR $B, #2;
 74 -
 75 - /*initialize loop count */
 76 - LDR #7 => $LC ;
 77 -
 78 - /*start up FFT with first RAM bank */
 79 - LD_C:(32) $A:(32,1) => $C1;
 80 - FFT_C:(32) $C1, $ROM=0:0;
 81 -
 82 - /*loop 7 times, XOR with $LC alternates RAM */
 83 - LD_C:(32) $A+=2:(32,1) => $C0;
 84 - FFT_C:(32) $C0, $ROM=0:0;
 85 - ST_C:(32) $C1 => $B+=2:(32,1)~;
 86 - LOOP:[IE:1,DL:1] [!LZ], #3;
 87 -
 88 - /* save last RAM bank */
 89 - ST_C:(32) $C1 => $B+=2:(32,1)~;
 90 -
 91 - }
 92 - #/
 93 - }
```

```
/*Program to perform complex correlation between two complex vectors with
up to 32 elements in the shorter one and up to 1024 elements in the output
using a single zoran processor.  Due to requirements of the instruction
used, the longer complex vector must be padded at both ends with (shorter
length - 1) complex zero elements.  These are needed for when the shorter
vector extends beyond the end of the longer during the operation.  If the
vectors are the same length, either may be considered the longer one.

The length of the short vector is passed in the parameter Coef_Length.
The length of the desired output vector (typically equal to the sum of
the lengths of the input vectors, minus one) is passed in Out_Length.
The short input vector is pointed to by Coefficients.  The parameter
In_Data points at the first zero pad of the longer input vector.
The output is placed at Out_Data.  The output data could be stored in
the place of the first input vector if desired.  Typical call to perform
a full autocorrelation in place with a 32 (padded to 94) element vector:
CALL CCORR(32, 63, &in, &(in+31), &in)
The (in+31) skips the padding at the front of the vector.

Note: if this routine will always be used for two equal length vectors,
only one length parameter is needed.  The other can be computed from it
with some extra overhead.  On the other hand, if this routine will be
used repeatedly for the same length, sending a precomputed $PR value
instead of a length would reduce overhead slightly.
*/

zsp325()
{
/#
SUBROUTINE CCORR(zr325int Coef_Length,
zr325int Out_Length,
zr325ref Coefficients,
zr325ref In_Data,
zr325ref Out_Data)
{
/*set up mode properly, one RAM section, 24 bit integers */
SET [ =NMS, =IXOR, =IFMT ];

/*set $SAR to put output in correct place */
LDR Out_Data => $SAR;

/*load vector lengths into parameter register
$NMPT = Coef_Length, $REPEAT = Out_Length
*/
SHLSETR:[SHIFT=18] Coef_Length => $PR;
ADDR $PR, Out_Length;

/*load complex conjugate of coefficients */
LD_*C:($NMPT) *Coefficients => $C0;

/*correlate with input sequence  */
FIR_C:($NMPT,$REPEAT) $C0, *In_Data;
}
#/
}
```

```
 1 -
 2 - /*Routine to compute a 1K complex FFT using four VSP chips.
 3 -
 4 - Operation requires two phases of operation, one to calculate column
 5 - FFTs, the other to calculate row FFTs with twiddle factors.  The
 6 - column phase can be performed by calling the FFT32COL routine just
 7 - as for a 32x32 2D FFT.  The routine for the row phase differs between
 8 - chips because the twiddle factors required are different.  This
 9 - program can generate all four routines by running it with different
10 - settings for the macro CHIP.
11 -
12 - Using multiple VSP chips requires synchronization between phases so
13 - that data can be exchanged.  This can be provided by a VSP routine
14 - that synchronizes between calling FFT32COL and FFT1Kn, or the 68020
15 - can invoke the first phase and wait for it to finish before invoking
16 - the second.
17 -
18 - Each chip should be passed an input pointer to row (CHIP * 8) with
19 - CHIP equalling 0, 1, 2, or 3, depending on the chip.  The output
20 - pointer should be to column (CHIP * 8) since the results must be
21 - transposed to convert column and row bit-reversals into an overall
22 - bit-reversal.  Each chip handles the 8 rows (turning into columns)
23 - starting at that point.  Adding a parameter to give the number of
24 - rows or columns to do would allow the same routine to be used by 1
25 - or 2 chips without needing to make multiple calls.
26 -
27 - The parameter In_Data points to the input vector.  The output vector
28 - is placed at Out_Data.  The operation cannot be performed in place
29 - because of the needed transpose.  The column pass can be performed
30 - in place to avoid needing a buffer area for the intermediate results.
31 -
32 - To get an inverse FFT, just change the subroutine name and change the
33 - FFT instructions to IFFT instructions.
34 -
35 - */
36 -
37 - /*chip number */
38 - #define CHIP 0
39 -
40 - /*function name for this chip, change for each */
41 - #define FUNCNAME FFT1K0
42 -
43 -
44 -
45 - zsp325()
46 - {
47 - /#
48 - /*FFT for rows, eight 32 point FFTs with twiddle factors */
49 - SUBROUTINE FUNCNAME(zr325ref In_Data, zr325ref Out_Data)
50 - {
51 -
52 - /*set up two RAM sections, swapped by $LC */
53 - SET [ =!NMS, =XOR ];
54 -
55 - /*set pointers to input and output, compensate for increment */
56 - /*note: depending on parameter order to get In_Data into $A */
57 - LDR Out_Data => [$B, $A];
58 - SUBR $B, #2;
59 -
60 - /*initialize loop count */
61 - LDR #7 => $LC ;
62 -
63 - /*start up FFT with first RAM bank */
64 - LD_C:(32) $A => $C1;
65 - /*Increase initial twiddle factor in RBA by 8 rows per chip */
66 - FFT_C:(32) $C1, $ROM=(CHIP*8*16):0;
67 -
68 - /*loop 7 times, XOR with $LC alternates RAM */
69   LD_C:(32) $A+=64 => $C0;
70 - /*use RBA, increasing it for each set of 32 */
71 - /*increment of 16 puts it at 1 on last pass */
72 - FFT_C:(32) $C0, $ROM+=16:0;
73 - ST_C:(32) $C1 => $B+=2:(32,1)~;
74 - LOOP:[DL:1] [!LZ], #3;
75 -
76 - /* save last RAM bank */
77 - ST_C:(32) $C1 => $B+=2:(32,1)~;
78 -
79 - }
80 - #/
81 - }
```

```
/*Program to perform real correlation between two real vectors with up to
64 elements in the shorter one and up to 1024 elements in the output
using a single zoran processor.  Due to requirements of the instruction
used, the longer real vector must be padded at both ends with (shorter
length - 1) real zero elements.  These are needed for when the shorter
vector extends beyond the end of the longer during the operation.  If the
vectors are the same length, either may be considered the longer one.

The length of the short vector is passed in the parameter Coef_Length.
The length of the desired output vector (typically equal to the sum of
the lengths of the input vectors, minus one) is passed in Out_Length.
Coefficients points to the short input vector.  In_Data points to
the first zero pad in the longer input vector.  The output is placed at
Out_Data.  The output data could be stored in the place of the first
input vector if desired.  Typical call to perform a full autocorrelation
in place with a 64 (padded to 190) element vector:
CALL CCORR(64, 127, &in, &(in+63), &in)
The (in+63) skips the padding at the front of the vector.

Note: if this routine will always be used for two equal length vectors,
only one length parameter is needed.  The other can be computed from it
with some extra overhead.  On the other hand, if this routine will be
used repeatedly for the same length, sending a precomputed $PR value
instead of a length would reduce overhead slightly.
*/

zsp325()
{
}#
SUBROUTINE RCORR(zr325int Coef_Length,
zr325int Out_Length,
zr325ref Coefficients,
zr325ref In_Data,
zr325ref Out_Data)
{
/*set up mode properly, one RAM section, 24 bit integers */
SET [ =NMS, =!XOR, =!FMT ];

/*set $SAR to put output in correct place */
LDR Out_Data => $SAR;

/*to get real coefficients in zig-zag order, need to load half
as many (rounded up) "complex" coefficients
*/
SHLSETR:[SHIFT=17] Coef_Length => $PR;
ADDR $PR, #0x020000;

/*load coefficients in zig-zag real order */
LD_C:($NMPT) *Coefficients => $C0;

/*load vector lengths into parameter register
$NMPT = Coef_Length, $REPEAT = Out_Length
*/
SHLSETR:[SHIFT=18] Coef_Length => $PR;
ADDR $PR, Out_Length;

/*correlate with input sequence  */
FIR_R:($NMPT,$REPEAT) $Z0, *In_Data;
}
#/
}
```

```
  1 - /*Routine to set up reciprocal table and one to generate inline code
  2 - to compute the reciprocals for a vector.  The algorithm is to perform
  3 - a table lookup to get a starting estimate and then perform Newton-Raphson
  4 - iterations until accuracy is 24 bits.  This requires that
  5 - TAB_BITS * (1 << NUM_ITER) >= 24.
  6 -
  7 - Might be better to split reciptab into a zsp325 routine to create table
  8 - and link in after assembly.  The reciprocal function would still be
  9 - included by the using routine.  This would prevent including table
 10 - more than once if it is used by multiple other routines.
 11 - */
 12 -
 13 - /*define number of bits of accuracy in table, table size, and iterations */
 14 - #define TAB_BITS 6
 15 - #define TAB_SIZE (1 << (TAB_BITS-1))
 16 - #define NUM_ITER 2
 17 -
 18 - /*Function to create reciprocal table for initial estimate.  Must be
 19 - called once if reciprocals are to be used.
 20 - */
 21 - void reciptab()
 22 - {
 23 - long i;
 24 - union
 25 - {
 26 - float flt;
 27 - long int;
 28 - } max, min;
 29 -
 30 - /*generate label for start of table */
 31 - /#
 32 - RecipTab::
 33 - #/
 34 -
 35 - /*generate the table entries */
 36 - for (i = 0; i < TAB_SIZE; i++)
 37 - {
 38 - /* calculate max and min values that will use this entry */
 39 - min.int = (127L << 23) + (i << (24 - TAB_BITS));
 40 - max.int = (127L << 23) + ((i+1) << (24 - TAB_BITS));
 41 -
 42 - /*use midpoint between their reciprocals to minimize error */
 43 - /#
 44 - .DATA { IEEE_Float(0.5 / max.flt + 0.5 / min.flt) } ;
 45 - #/
 46 - }
 47 - }
 48 -
 49 -
 50 - /*Function to produce inline assembly to calculate the reciprocals for
 51 - a vector in internal RAM.  The internal RAM must be set up to have two
 52 - banks and the input vector must be in R0.  This limits the input vector
 53 - length to 32 or less.  The result vector ends up in R0.  All internal
 54 - RAM banks are overwritten with intermediate results.
 55 -
 56 - This function is essentially a macro.  It is called from within a
 57 - zsp325() function and generates assembly code.  It does not produce
 58 - any calls that execute at run time.  The function reciptab must also
 59 - have been called by the zsp325() function or there will be an error
 60 - during assembly.
 61 - */
 62 - void recip(int length)
 63 - {
 64 - int i;
 65 -
 66 - /#
 67 - /*split into exponent and mantissa, negate exponent, trap zero */
 68 - SPLIT_R:((length)):[DV] $R0 => $C1;
 69 -
 70 - /*look up initial estimate of reciprocal of mantissa */
 71 - LUT_R:((length)):[SHIFT=(24-TAB_BITS)] RecipTab, $I1 => $I0;
 72 -
 73 - /*change sign of estimate to match initial input sign */
 74 - SIGN_R:((length)) $R0, $I0 => $R0;
 75 - #/
 76 -
 77 - /*generate Newton-Raphson iterations inline */
 78 - for (i = 0; i < NUM_ITER; i++)
 79 - {
 80 - /#
 81 - /*new estimate = estimate * (2.0 - estimate * input) */
 82 - SBM_R:((length)) $R0, $I1, #2.0 => $I0;
 83 - MULT_R:((length)) $R0, $I0 => $R0;
 84 - #/
 85 - }
 86 -
 87 - /#
 88 - /*recombine resulting mantissa with exponent */
 89 - JOIN_R:((length)) $R1, $R0 => $R0;
 90 - #/
 91 - }
```

```
 1 - /*Test program to see if Zorans work */
 2 -
 3 - /*absolute base addresses from memory map */
 4 - #define PRAM 0x00000
 5 - #define FOUR_PORT 0x20000
 6 - #define STATUS_LATCH 0x40000
 7 -
 8 - zsp325()
 9 - {
10 - int i;
11 - float x;
12 -
13 - /*put a vector of (1.0, x) at PRAM + 0x400 */
14 - /#
15 - .ORG(PRAM + 0x400)
16 - #/
17 - for (i = 0, x = 0.0; i < 16; i++, x += 1.0)
18 - {
19 - /#
20 - .DATA{ 1.0, IEEE_Float(x) };
21 - #/
22 - }
23 -
24 - /*put a vector of (x, 1.0) at FOUR_PORT */
25 - /#
26 - .ORGFOUR_PORT
27 - #/
28 - for (i = 0, x = 0.0; i < 16; i++, x += 1.0)
29 - {
30 - /#
31 - .DATA{ IEEE_Float(x), 1.0 };
32 - #/
33 - }
34 -
35 - /#
36 - .ORG
37 - SUBROUTINE MAIN()
38 - {
39 - /*write 0 to status latch */
40 - LDR #0 => $X;
41 - STR $X => STATUS_LATCH;
42 -
43 - /*add two complex vectors and store */
44 - LD_C:(16) (PRAM + 0x400) => $C0;
45 - ADD_C:(16) FOUR_PORT, $C0 => $C0;
46 - ST_C:(16) $C0 => FOUR_PORT;
47 -
48 - /*make sure we are finished, then write 1s to status latch */
49 - SYNC:[CU,EU,MU];
50 - LDR #3 => $X;
51 - STR $X => STATUS_LATCH;
52 - }
53 - #/
54 - }
```

```
1 - /*Test program to make Zoran status bits follow 68020 bits */
2 -
3 - /*absolute base addresses from memory map */
4 - #define PRAM 0x00000
5 - #define FOUR_PORT 0x20000
6 - #define STATUS_LATCH 0x40000
7 -
8 - zsp325()
9 - {
10 -
11 - /#
12 - SUBROUTINE MAIN()
13 - {
14 - Top::
15 - LDR STATUS_LATCH => $LC;
16 - STR $LC => STATUS_LATCH;
17 - Loop::
18 - XORR:[TR] STATUS_LATCH, $LC => $X;
19 - ANDR #3, $X;
20 - JMPC:[IE:0] [ZR], Loop;
21 -
22 - JMP Top;
23 - }
24 - #/
25 - }
```

```
  1 - /*Code to start all VSP chips simultaneously.  The start address of the
  2 - code to be executed at the signal should be the first value on the
  3 - stack.
  4 - */
  5 -
  6 - /*absolute base addresses from memory map */
  7 - #define PRAM 0x00000
  8 - #define FOUR_PORT 0x20000
  9 - #define STATUS_LATCH 0x40000
 10 -
 11 - /*status bit value to indicate start */
 12 - #define START 2
 13 -
 14 - zsp325()
 15 - {
 16 - /#
 17 - SUBROUTINE START()
 18 - {
 19 - /*get mask for status bit */
 20 - LDR #START => $X;
 21 -
 22 - Poll::
 23 - ANDR:[TR] STATUS_LATCH, $X;
 24 - JMPC [ZR], Poll;
 25 -
 26 - }
 27 - #/
 28 - }
```

```
 1 - /*    Alternate routines to compute 32x32 2D complex FFT using four VSP chips.
 2 -
 3 -       The pipe_p2r routine produces incorrect results when the IE (immediate
 4 -       execution) qualifier is used on its software pipeline's loop instruction.
 5 -       Whatever mechanism causes this doesn't seem to affect the FFT routines
 6 -       that use the same qualifier.  However, if for some reason it does so,
 7 -       the routines can be rewritten to avoid using the qualifier.  Just taking
 8 -       the qualifier out of the existing code will reduce performance by around
 9 -       15%.  This is because the existing loop overlaps the FFT instruction
10 -       with the following store, the loop instruction itself, and the load in
11 -       the next loop iteration.  Removing the IE qualifier causes the loop
12 -       instruction to wait until the FFT instruction is complete and therefore
13 -       prevents overlap of the FFT instruction with the loop instruction and
14 -       more importantly, with the load in the next iteration.  By moving the
15 -       "kernel" of the software pipeline down one instruction, the load moves
16 -       past the loop instruction into the current iteration.  This allows the
17 -       load to overlap the FFT instruction even though the loop instruction
18 -       cannot.  Moving the kernel down one instruction requires alterations to
19 -       the preamble and postamble of the loop.  Since these alterations cause
20 -       the combination of the preamble and postamble to execute two iterations
21 -       instead of one, the loop count must be decreased by two instead of one.
22 -       This alternative version of the 32x32 FFT can be used as an example of
23 -       the modifications that are needed.
24 -
25 - */
26 -
27 - zsp325()
28 - {
29 -       /*
30 -       /*      FFT for rows, eight 32 point FFTs on sequential data */
31 -       SUBROUTINE FFT32ROW(zr325ref In_Data, zr325ref Out_Data)
32 -       {
33 -
34 -               /*      set up two RAM sections, swapped by $LC */
35 -               SET [ =!BMS, =XOR ];
36 -
37 -               /*      set pointers to input and output, compensate for increment */
38 -               /*      note: depending on parameter order to get In_Data into $A */
39 -               LDR Out_Data => [$B, $A];
40 -               SUBR $B, #64;
41 -
42 -               /*      initialize loop count */
43 -               LDR #6 => $LC;
44 -
45 -               /*      preamble */
46 -               LD_C:(32) $A => $C1;
47 -               FFT_C:(32) $C1;
48 -               LD_C:(32) $A+=64 => $C0;
49 -
50 -               /*      loop 6 times, XOR with $LC alternates RAM */
```

```
51 -              FFT_C:(32) $C0;
52 -              ST_C:(32) $C1 => $B+=64:(1,32~);
53 -              LD_C:(32) $A+=64 => $C1;
54 -              LOOP:[DL] [!LZ], #3;
55 -
56 -              /* postamble */
57 -              FFT_C:(32) $C0;
58 -              ST_C:(32) $C1 => $B+=64:(1,32~);
59 -              ST_C:(32) $C0 => $B+=64:(1,32~);
60 -
61 -       }
62 -
63 -       /*     FFT for columns, eight 32 point FFTs on interleaved data */
64 -       SUBROUTINE FFT32COL(zr325ref In_Data, zr325ref Out_Data)
65 -       {
66 -              /*      set up two RAM sections, swapped by $LC */
67 -              SET [ =!RMS, =XOR ];
68 -
69 -              /*      set pointers to data, compensate for first increment */
70 -              /*      note: depending on parameter order to get In_Data into $A */
71 -              LDR Out_Data => [$B, $A];
72 -              SUBR $B, #2;
73 -
74 -              /*      initialize loop count */
75 -              LDR #6 => $LC ;
76 -
77 -              /*      preamble */
78 -              LD_C:(32) $A:(32,1) => $C1;
79 -              FFT_C:(32) $C1;
80 -              LD_C:(32) $A+=2:(32,1) => $C0;
81 -
82 -              /*      loop 6 times, XOR with $LC alternates RAM */
83 -              FFT_C:(32) $C0;
84 -              ST_C:(32) $C1 => $B+=2:(32,1)~;
85 -              LD_C:(32) $A+=2:(32,1) => $C1;
86 -              LOOP:[DL] [!LZ], #3;
87 -
88 -              /*      postamble */
89 -              FFT_C:(32) $C0;
90 -              ST_C:(32) $C1 => $B+=2:(32,1)~;
91 -              ST_C:(32) $C0 => $B+=2:(32,1)~;
92 -
93 -       }
94 -       */
95 - }
```

```
 1 - /*   VPH code for convolution of a complex sequence of up to 32 points with
 2 -      another longer complex sequence, producing up to 1024 outputs.  This
 3 -      size can be done with a single FIR instruction.  This code can be
 4 -      called repeatedly on a single processor to handle convolutions where
 5 -      more than 1024 output points are required as long as the shorter
 6 -      sequence is still no more than 32 points.  However, a different routine
 7 -      designed for a longer convolution would be more efficient.  This
 8 -      same code can be used on multiple VSP chips simultaneously to give
 9 -      a considerable speed increase.  There may be no benefit to executing
10 -      on more than one VSP chip per bus since the FIR instruction may not
11 -      give up the bus between output points.
12 -
13 -      To get a full convolution of the input requires padding both ends of
14 -      the longer input sequence with a number of complex zeroes equal to the
15 -      length of the shorter sequence minus one.  This is required in order
16 -      to explicitly provide the zeroes that are assumed to be multiplied by
17 -      elements of the shorter sequence that extend beyond the ends of the
18 -      longer one during the convolution process.  The length of the output
19 -      sequence should be equal to the sum of the lengths of the (unpadded)
20 -      input sequences minus one.  If a circular convolution is desired
21 -      instead of a linear one, the zero padding should be replaced with
22 -      points from the other end of the input sequence.
23 -
24 -      The shorter input length is passed in Coef_Length.  The output length
25 -      (equal to input length before padding plus coefficient length minus one)
26 -      is passed as Out_Length.  Coefficients points to the shorter sequence.
27 -      In_Data points to the start of the longer sequence (possibly a zero
28 -      pad).  The output is placed at Out_Data.  Typical call:
29 -      CALL CCONV(4, 1024, &Coef, &In, &Out)
30 -
31 -      The convolution can be performed in place with careful choices of
32 -      parameter values.  If the convolution requires multiple calls on a
33 -      single VSP chip, the output must begin at the first location of the
34 -      long input.  This avoids overwriting inputs that will be needed for
35 -      the next call.  However, if multiple chips are being used, the output
36 -      must overwrite the last input used in its computation.  This works
37 -      because the VSP chip has already read the input into internal RAM
38 -      for further use.  It is necessary because that input is the first
39 -      one which will not be needed by the chip working on the previous
40 -      portion of the convolution.  Some further care is needed in the
41 -      initial startup of in-place multiple chip convolution to ensure that
42 -      a chip does not write over any input values before the subsequent
43 -      chip reads them in.  A multiple call, multiple chip convolution
44 -      cannot be done in place because the constraints are contradictory.
45 -      However, such a large data set would not fit into shared memory.
46 -
47 -      Splittin  up a convolution between NUM_CHIPS chips would require
48 -      somethin  ike the following invocation for chip ranging from zero
49 -      to (NUM_C  >S - 1):
50 -
```

```
51 -       CALL CCONV(COEF_LEN, OUT_SIZE(chip), &Coef,
52 -                                &(In + 2*DATA_OFFSET(chip)), &(Out + 2*DATA_OFFSET(chip)));
53 -
54 -       with the definitions
55 -
56 -       #define OUT_LEN (IN_LEN + COEF_LEN - 1)
57 -       #define DATA_OFFSET(CHIP) (((CHIP) * OUT_LEN) / NUM_CHIPS)
58 -       #define OUT_SIZE(CHIP) (DATA_OFFSET(CHIP+1) - DATA_OFFSET(CHIP))
59 -
60 -       DATA_OFFSET is doubled when used with pointer parameters because
61 -       each complex element requires two machine words.
62 - */
63 -
64 - zsp325()
65 - {
66 -       /*
67 -       SUBROUTINE CCONV(       zr325int Coef_Length,
68 -                                       zr325int Out_Length,
69 -                                       zr325ref Coefficients,
70 -                                       zr325ref In_Data,
71 -                                       zr325ref Out_Data)
72 -       {
73 -               /*      set up mode properly, one RAM bank, 24 bit integers */
74 -               SET [ =BMS, =!XOR , =!FMT ];
75 -
76 -               /*      set $SAR to put output in correct place */
77 -               LDR Out_Data => $SAR;
78 -
79 -               /*      now set up lengths for LD and FIR instructions */
80 -               SHLSETR:[SHIFT=16] Coef_Length => $PR;
81 -               ADDR $PR, Out_Length;
82 -
83 -               /*      load coefficients in reverse order */
84 -               SHLSETR:[SHIFT=1] Coef_Length => $A;
85 -               ADDR $A, Coefficients;
86 -               SUBR $A, #2;
87 -               LD_C:($NMPT) $A:(-1,1) => $C0;
88 -
89 -               /*      convolve with input sequence  */
90 -               FIR_C:($NMPT, $REPEAT) $C0, *In_Data;
91 -
92 -       }
93 -       #/
94 - }
```

```
 1 - /*    Program to perform complex correlation between two complex vectors with
 2 -       up to 32 elements in the shorter one and up to 1024 elements in the output
 3 -       using a single zoran processor.  Due to requirements of the instruction
 4 -       used, the longer complex vector must be padded at both ends with (shorter
 5 -       length - 1) complex zero elements.  These are needed for when the shorter
 6 -       vector extends beyond the end of the longer during the operation.  If the
 7 -       vectors are the same length, either may be considered the longer one.
 8 -
 9 -       The length of the short vector is passed in the parameter Coef_Length.
10 -       The length of the desired output vector (typically equal to the sum of
11 -       the lengths of the input vectors, minus one) is passed in Out_Length.
12 -       The short input vector is pointed to by Coefficients.  The parameter
13 -       In_Data points at the first zero pad of the longer input vector.
14 -       The output is placed at Out_Data.  The output data could be stored in
15 -       the place of the first input vector if desired.  Typical call to perform
16 -       a full autocorrelation in place with a 32 (padded to 94) element vector:
17 -       CALL CCORR(32, 63, &in, &(in+31), &in)
18 -       The (in+31) skips the padding at the front of the vector.
19 -
20 -       Note: if this routine will always be used for two equal length vectors,
21 -       only one length parameter is needed.  The other can be computed from it
22 -       with some extra overhead.  On the other hand, if this routine will be
23 -       used repeatedly for the same length, sending a precomputed $PR value
24 -       instead of a length would reduce overhead slightly.
25 - */
26 -
27 - zsp325()
28 - {
29 -       /#
30 -       SUBROUTINE CCORR(       zr325int Coef_Length,
31 -                                       zr325int Out_Length,
32 -                                       zr325ref Coefficients,
33 -                                       zr325ref In_Data,
34 -                                       zr325ref Out_Data)
35 -       {
36 -               /*      set up mode properly, one RAM section, 24 bit integers */
37 -               SET [ =NMS, =!XOR, =!FMT ];
38 -
39 -               /*      set $SAR to put output in correct place */
40 -               LDR Out_Data => $SAR;
41 -
42 -               /*      load vector lengths into parameter register
43 -                       $NMPT = Coef_Length, $REPEAT = Out_Length
44 -               */
45 -               SHLSETR:[SHIFT=16] Coef_Length => $PR;
46 -               ADDR $PR, Out_Length;
47 -
48 -               /*      load complex conjugate of coefficients */
49 -               LD_*C:($NMPT) *Coefficients => $C0;
50 -
```

```
51 -                /*      correlate with input sequence  */
52 -                FIR_C:($NMPT,$REPEAT) $C0, *In_Data;
53 -        }
54 -        #/
55 - }
```

```
 1 -
 2 - /*    Routine to compute a 1K complex FFT using four VSP chips.
 3 -
 4 -       Operation requires two phases of operation, one to calculate column
 5 -       FFTs, the other to calculate row FFTs with twiddle factors.  The
 6 -       column phase can be performed by calling the FFT32COL routine just
 7 -       as for a 32x32 2D FFT.  The routine for the row phase differs between
 8 -       chips because the twiddle factors required are different.  This
 9 -       program can generate all four routines by running it with different
10 -       settings for the macro CHIP.
11 -
12 -       Using multiple VSP chips requires synchronization between phases so
13 -       that data can be exchanged.  This can be provided by a VSP routine
14 -       that synchronizes between calling FFT32COL and FFT1Kn, or the 68020
15 -       can invoke the first phase and wait for it to finish before invoking
16 -       the second.
17 -
18 -       Each chip should be passed an input pointer to row (CHIP * 8) with
19 -       CHIP equalling 0, 1, 2, or 3, depending on the chip.  The output
20 -       pointer should be to column (CHIP * 8) since the results must be
21 -       transposed to convert column and row bit-reversals into an overall
22 -       bit-reversal.  Each chip handles the 8 rows (turning into columns)
23 -       starting at that point.  Adding a parameter to give the number of
24 -       rows or columns to do would allow the same routine to be used by 1
25 -       or 2 chips without needing to make multiple calls.
26 -
27 -       The parameter In_Data points to the input vector.  The output vector
28 -       is placed at Out_Data.  The operation cannot be performed in place
29 -       because of the needed transpose.  The column pass can be performed
30 -       in place to avoid needing a buffer area for the intermediate results.
31 -
32 -       To get an inverse FFT, just change the subroutine name and change the
33 -       FFT instructions to IFFT instructions.
34 -
35 - */
36 -
37 - /*    chip number */
38 - #define CHIP 0
39 -
40 - /*    function name for this chip, change for each */
41 - #define FUNCNAME FFT1K0
42 -
43 -
44 -
45 - zsp325()
46 - {
47 -       /*
48 -       /*      FFT for rows, eight 32 point FFTs with twiddle factors */
49 -       SUBROUTINE FUNCNAME(zr325ref In_Data, zr325ref Out_Data)
50 -       {
```

```
51 -
52 -                /*      set up two RAM sections, swapped by $LC */
53 -                SET [ =!NMS, =XOR ];
54 -
55 -                /*      set pointers to input and output, compensate for increment */
56 -                /*      note: depending on parameter order to get In_Data into $A */
57 -                LDR Out_Data => [$B, $A];
58 -                SUBR $B, #2;
59 -
60 -                /*      initialize loop count */
61 -                LDR #7 => $LC ;
62 -
63 -                /*      start up FFT with first RAM bank */
64 -                LD_C:(32) $A => $C1;
65 -                /*      increase initial twiddle factor in RBA by 8 rows per chip */
66 -                FFT_C:(32) $C1, $ROM=(CHIP*8*16):0;
67 -
68 -                /*      loop 7 times, XOR with $LC alternates RAM */
69 -                LD_C:(32) $A+=64 => $C0;
70 -                /*      use RBA, increasing it for each set of 32 */
71 -                /*      increment of 16 puts it at 1 on last pass */
72 -                FFT_C:(32) $C0, $ROM+=16:0;
73 -                ST_C:(32) $C1 => $B+=2:(32,1)~;
74 -                LOOP:[DL:1] [!LZ], #3;
75 -
76 -                /* save last RAM bank */
77 -                ST_C:(32) $C1 => $B+=2:(32,1)~;
78 -
79 -        }
80 -        #/
81 - }
```

```
 1 -
 2 - /*    Routine to compute a 1K complex FFT using four VSP chips.
 3 -
 4 -       Operation requires two phases of operation, one to calculate column
 5 -       FFTs, the other to calculate row FFTs with twiddle factors.  The
 6 -       column phase can be performed by calling the FFT32COL routine just
 7 -       as for a 32x32 2D FFT.  The routine for the row phase differs between
 8 -       chips because the twiddle factors required are different.  This
 9 -       program can generate all four routines by running it with different
10 -       settings for the macro CHIP.
11 -
12 -       Using multiple VSP chips requires synchronization between phases so
13 -       that data can be exchanged.  This can be provided by a VSP routine
14 -       that synchronizes between calling FFT32COL and FFT1Kn, or the 68020
15 -       can invoke the first phase and wait for it to finish before invoking
16 -       the second.
17 -
18 -       Each chip should be passed an input pointer to row (CHIP * 8) with
19 -       CHIP equalling 0, 1, 2, or 3, depending on the chip.  The output
20 -       pointer should be to column (CHIP * 8) since the results must be
21 -       transposed to convert column and row bit-reversals into an overall
22 -       bit-reversal.  Each chip handles the 8 rows (turning into columns)
23 -       starting at that point.  Adding a parameter to give the number of
24 -       rows or columns to do would allow the same routine to be used by 1
25 -       or 2 chips without needing to make multiple calls.
26 -
27 -       The parameter In_Data points to the input vector.  The output vector
28 -       is placed at Out_Data.  The operation cannot be performed in place
29 -       because of the needed transpose.  The column pass can be performed
30 -       in place to avoid needing a buffer area for the intermediate results.
31 -
32 -       To get an inverse FFT, just change the subroutine name and change the
33 -       FFT instructions to IFFT instructions.
34 -
35 - */
36 -
37 - /*    chip number */
38 - #define CHIP 0
39 -
40 - /*    function name for this chip, change for each */
41 - #define FUNCNAME FFT1K0
42 -
43 -
44 -
45 - zap325()
46 - {
47 -       /*
48 -       /*      FFT for rows, eight 32 point FFTs with twiddle factors */
49 -       SUBROUTINE FUNCNAME(zr325ref In_Data, zr325ref Out_Data)
50 -       {
```

```
51 -
52 -              /*      set up two RAM sections, swapped by $LC */
53 -              SET [ =!NMS, =XOR ];
54 -
55 -              /*      set pointers to input and output, compensate for increment */
56 -              /*      note: depending on parameter order to get In_Data into $A */
57 -              LDR Out_Data => [$B, $A];
58 -              SUBR $B, #2;
59 -
60 -              /*      initialize loop count */
61 -              LDR #7 => $LC ;
62 -
63 -              /*      start up FFT with first RAM bank */
64 -              LD_C:(32) $A => $C1;
65 -              /*      increase initial twiddle factor in RBA by 8 rows per chip */
66 -              FFT_C:(32) $C1, $ROM=(CHIP*8*16):0;
67 -
68 -              /*      loop 7 times, XOR with $LC alternates RAM */
69 -              LD_C:(32) $A+=64 => $C0;
70 -              /*      use RBA, increasing it for each set of 32 */
71 -              /*      increment of 16 puts it at 1 on last pass */
72 -              FFT_C:(32) $C0, $ROM+=16:0;
73 -              ST_C:(32) $C1 => $B+=2:(32,1)~;
74 -              LOOP:[IE,DL] [!LZ], #3;
75 -
76 -              /* save last RAM bank */
77 -              ST_C:(32) $C1 => $B+=2:(32,1)~;
78 -
79 -      }
80 -      #/
81 - }
```

```
 1 -
 2 - /*    Routine to compute a 1K complex FFT using four VSP chips.
 3 -
 4 -       Operation requires two phases of operation, one to calculate column
 5 -       FFTs, the other to calculate row FFTs with twiddle factors.  The
 6 -       column phase can be performed by calling the FFT32COL routine just
 7 -       as for a 32x32 2D FFT.  The routine for the row phase differs between
 8 -       chips because the twiddle factors required are different.  This
 9 -       program can generate all four routines by running it with different
10 -       settings for the macro CHIP.
11 -
12 -       Using multiple VSP chips requires synchronization between phases so
13 -       that data can be exchanged.  This can be provided by a VSP routine
14 -       that synchronizes between calling FFT32COL and FFT1Kn, or the 68020
15 -       can invoke the first phase and wait for it to finish before invoking
16 -       the second.
17 -
18 -       Each chip should be passed an input pointer to row (CHIP * 8) with
19 -       CHIP equalling 0, 1, 2, or 3, depending on the chip.  The output
20 -       pointer should be to column (CHIP * 8) since the results must be
21 -       transposed to convert column and row bit-reversals into an overall
22 -       bit-reversal.  Each chip handles the 8 rows (turning into columns)
23 -       starting at that point.  Adding a parameter to give the number of
24 -       rows or columns to do would allow the same routine to be used by 1
25 -       or 2 chips without needing to make multiple calls.
26 -
27 -       The parameter In_Data points to the input vector.  The output vector
28 -       is placed at Out_Data.  The operation cannot be performed in place
29 -       because of the needed transpose.  The column pass can be performed
30 -       in place to avoid needing a buffer area for the intermediate results.
31 -
32 -       To get an inverse FFT, just change the subroutine name and change the
33 -       FFT instructions to IFFT instructions.
34 -
35 - */
36 -
37 - /*    chip number */
38 - #define CHIP 1
39 -
40 - /*    function name for this chip, change for each */
41 - #define FUNCNAME FFT1K1
42 -
43 -
44 -
45 - zsp325()
46 - {
47 -       /*
48 -       /*      FFT for rows, eight 32 point FFTs with twiddle factors */
49 -       SUBROUTINE FUNCNAME(zr325ref In_Data, zr325ref Out_Data)
50 -       {
```

```
51 -
52 -              /*      set up two RAM sections, swapped by $LC */
53 -              SET [ =!RMS, =XOR ];
54 -
55 -              /*      set pointers to input and output, compensate for increment */
56 -              /*      note: depending on parameter order to get In_Data into $A */
57 -              LDR Out_Data => [$B, $A];
58 -              SUBR $B, #2;
59 -
60 -              /*      initialize loop count */
61 -              LDR #7 => $LC ;
62 -
63 -              /*      start up FFT with first RAM bank */
64 -              LD_C:(32) $A => $C1;
65 -              /*      increase initial twiddle factor in RBA by 8 rows per chip */
66 -              FFT_C:(32) $C1, $ROM=(CHIP*8*16):0;
67 -
68 -              /*      loop 7 times, XOR with $LC alternates RAM */
69 -              LD_C:(32) $A+=64 => $C0;
70 -              /*      use RBA, increasing it for each set of 32 */
71 -              /*      increment of 16 puts it at 1 on last pass */
72 -              FFT_C:(32) $C0, $ROM+=16:0;
73 -              ST_C:(32) $C1 => $B+=2:(32,1)~;
74 -              LOOP:[IE,DL] [!LZ], #3;
75 -
76 -              /* save last RAM bank */
77 -              ST_C:(32) $C1 => $B+=2:(32,1)~;
78 -
79 -      }
80 -      #/
81 - }
```

```
 1 -
 2 - /*    Routine to compute a 1K complex FFT using four VSP chips.
 3 -
 4 -       Operation requires two phases of operation, one to calculate column
 5 -       FFTs, the other to calculate row FFTs with twiddle factors.  The
 6 -       column phase can be performed by calling the FFT32COL routine just
 7 -       as for a 32x32 2D FFT.  The routine for the row phase differs between
 8 -       chips because the twiddle factors required are different.  This
 9 -       program can generate all four routines by running it with different
10 -       settings for the macro CHIP.
11 -
12 -       Using multiple VSP chips requires synchronization between phases so
13 -       that data can be exchanged.  This can be provided by a VSP routine
14 -       that synchronizes between calling FFT32COL and FFT1Kn, or the 68020
15 -       can invoke the first phase and wait for it to finish before invoking
16 -       the second.
17 -
18 -       Each chip should be passed an input pointer to row (CHIP * 8) with
19 -       CHIP equalling 0, 1, 2, or 3, depending on the chip.  The output
20 -       pointer should be to column (CHIP * 8) since the results must be
21 -       transposed to convert column and row bit-reversals into an overall
22 -       bit-reversal.  Each chip handles the 8 rows (turning into columns)
23 -       starting at that point.  Adding a parameter to give the number of
24 -       rows or columns to do would allow the same routine to be used by 1
25 -       or 2 chips without needing to make multiple calls.
26 -
27 -       The parameter In_Data points to the input vector.  The output vector
28 -       is placed at Out_Data.  The operation cannot be performed in place
29 -       because of the needed transpose.  The column pass can be performed
30 -       in place to avoid needing a buffer area for the intermediate results.
31 -
32 -       To get an inverse FFT, just change the subroutine name and change the
33 -       FFT instructions to IFFT instructions.
34 -
35 - */
36 -
37 - /*    chip number */
38 - #define CHIP 2
39 -
40 - /*    function name for this chip, change for each */
41 - #define FUNCNAME FFT1K2
42 -
43 -
44 -
45 - zsp325()
46 - {
47 -       /*
48 -       /*      FFT for rows, eight 32 point FFTs with twiddle factors */
49 -       SUBROUTINE FUNCNAME(zr325ref In_Data, zr325ref Out_Data)
50 -       {
```

```
51 -
52 -                /*      set up two RAM sections, swapped by $LC */
53 -                SET [ =!RMS, =XOR ];
54 -
55 -                /*      set pointers to input and output, compensate for increment */
56 -                /*      note: depending on parameter order to get In_Data into $A */
57 -                LDR Out_Data => [$B, $A];
58 -                SUBR $B, #2;
59 -
60 -                /*      initialize loop count */
61 -                LDR #7 => $LC ;
62 -
63 -                /*      start up FFT with first RAM bank */
64 -                LD_C:(32) $A => $C1;
65 -                /*      increase initial twiddle factor in RBA by 8 rows per chip */
66 -                FFT_C:(32) $C1, $ROM=(CHIP*8*16):0;
67 -
68 -                /*      loop 7 times, XOR with $LC alternates RAM */
69 -                LD_C:(32) $A+=64 => $C0;
70 -                /*      use RBA, increasing it for each set of 32 */
71 -                /*      increment of 16 puts it at 1 on last pass */
72 -                FFT_C:(32) $C0, $ROM+=16:0;
73 -                ST_C:(32) $C1 => $B+=2:(32,1)~;
74 -                LOOP:[IE,DL] [!LZ], #3;
75 -
76 -                /* save last RAM bank */
77 -                ST_C:(32) $C1 => $B+=2:(32,1)~;
78 -
79 -        }
80 -        #/
81 - }
```

```
 1 -
 2 - /*    Routine to compute a 1K complex FFT using four VSP chips.
 3 -
 4 -       Operation requires two phases of operation, one to calculate column
 5 -       FFTs, the other to calculate row FFTs with twiddle factors.  The
 6 -       column phase can be performed by calling the FFT32COL routine just
 7 -       as for a 32x32 2D FFT.  The routine for the row phase differs between
 8 -       chips because the twiddle factors required are different.  This
 9 -       program can generate all four routines by running it with different
10 -       settings for the macro CHIP.
11 -
12 -       Using multiple VSP chips requires synchronization between phases so
13 -       that data can be exchanged.  This can be provided by a VSP routine
14 -       that synchronizes between calling FFT32COL and FFT1Kn, or the 68020
15 -       can invoke the first phase and wait for it to finish before invoking
16 -       the second.
17 -
18 -       Each chip should be passed an input pointer to row (CHIP * 8) with
19 -       CHIP equalling 0, 1, 2, or 3, depending on the chip.  The output
20 -       pointer should be to column (CHIP * 8) since the results must be
21 -       transposed to convert column and row bit-reversals into an overall
22 -       bit-reversal.  Each chip handles the 8 rows (turning into columns)
23 -       starting at that point.  Adding a parameter to give the number of
24 -       rows or columns to do would allow the same routine to be used by 1
25 -       or 2 chips without needing to make multiple calls.
26 -
27 -       The parameter In_Data points to the input vector.  The output vector
28 -       is placed at Out_Data.  The operation cannot be performed in place
29 -       because of the needed transpose.  The column pass can be performed
30 -       in place to avoid needing a buffer area for the intermediate results.
31 -
32 -       To get an inverse FFT, just change the subroutine name and change the
33 -       FFT instructions to IFFT instructions.
34 -
35 - */
36 -
37 - /*    chip number */
38 - #define CHIP 3
39 -
40 - /*    function name for this chip, change for each */
41 - #define FUNCNAME FFT1K3
42 -
43 -
44 -
45 - zsp325()
46 - {
47 -       /*
48 -       /*      FFT for rows, eight 32 point FFTs with twiddle factors */
49 -       SUBROUTINE FUNCNAME(zr325ref In_Data, zr325ref Out_Data)
50 -       {
```

```
51 -
52 -                /*      set up two RAM sections, swapped by $LC */
53 -                SET [ =!NMS, =XOR ];
54 -
55 -                /*      set pointers to input and output, compensate for increment */
56 -                /*      note: depending on parameter order to get In_Data into $A */
57 -                LDR Out_Data => [$B, $A];
58 -                SUBR $B, #2;
59 -
60 -                /*      initialize loop count */
61 -                LDR #7 => $LC ;
62 -
63 -                /*      start up FFT with first RAM bank */
64 -                LD_C:(32) $A => $C1;
65 -                /*      increase initial twiddle factor in RBA by 8 rows per chip */
66 -                FFT_C:(32) $C1, $ROM=(CHIP*8*16):0;
67 -
68 -                /*      loop 7 times, XOR with $LC alternates RAM */
69 -                LD_C:(32) $A+=64 => $C0;
70 -                /*      use RBA, increasing it for each set of 32 */
71 -                /*      increment of 16 puts it at 1 on last pass */
72 -                FFT_C:(32) $C0, $ROM+=16:0;
73 -                ST_C:(32) $C1 => $B+=2:(32,1)~;
74 -                LOOP:[IE,DL] [!LE], #3;
75 -
76 -                /* save last RAM bank */
77 -                ST_C:(32) $C1 => $B+=2:(32,1)~;
78 -
79 -        }
80 -        #/
81 - }
```

```
 1 - /*    Program to compute 8x8 2D complex FFT using one VSP chip.
 2 -
 3 -       The parameter In_Data points to the input vector.  The output vector
 4 -       is placed at Out_Data.  The operation can be performed in place if
 5 -       desired.  Both input and output vectors are in normal order.
 6 -
 7 -       To get an inverse FFT, just change the subroutine name and change the
 8 -       FFT instructions to IFFT instructions.
 9 -
10 -       To use real data, change LD_C to LD_(R,0).
11 -
12 -       Might be able to squeeze a little more speed out by starting with
13 -       two RAM sections, load first, FFT first rows, load second, FFT second
14 -       rows, switch to one RAM section, FFT columns, store.
15 -
16 - */
17 -
18 - zsp325()
19 - {
20 -       /*
21 -       SUBROUTINE FFT2D8(zr325ref In_Data, zr325ref Out_Data)
22 -       {
23 -              /*      set up one RAM section */
24 -              SET [ =MM8, =!XOR ];
25 -
26 -              /*      load all 64 entries, with rows bit reversed */
27 -              LD_C:(64) *In_Data:(8,8~) => $0;
28 -
29 -              /*      FFT the rows, result in normal order */
30 -              FFT_C:(8,8):[FPS:1,LPS:4] $0~, $ROM=0:512;
31 -
32 -              /*      FFT the columns, result in bit reversed order */
33 -              FFT_C:(64):[FPS:32,LPS:8] $0;
34 -
35 -              /*      store result, bit reversing columns into normal order */
36 -              ST_C:(64) $0 => *Out_Data:(8~,8);
37 -       }
38 -       */
39 - }
```

```
 1 - /*    Routines to compute 16x16 2D complex FFT using four VSP chips.
 2 -
 3 -       Operation requires two phases of operation, one to calculate row
 4 -       FFTs, the other to calculate column FFTs.  Using multiple VSP chips
 5 -       requires synchronization between phases so that data can be exchanged.
 6 -       These routines do not include the sychronization.  The routines for
 7 -       each phase can be called from another routine which provides it between
 8 -       calls, or the 68020 can invoke the first phase and wait for it to
 9 -       finish before invoking the second.
10 -
11 -       Each VSP chip could calculate its four rows or columns in one instruction,
12 -       but using two RAM sections allows more concurrency.  Each chip should
13 -       be passed data pointers to row or column (CHIP * 4) with CHIP equalling
14 -       0, 1, 2, or 3, depending on the chip.
15 -
16 -       The parameter In_Data points to the input vector.  The output vector
17 -       is placed at Out_Data.  The operation can be performed in place if
18 -       desired.  Both input and output vectors are in normal order.
19 -
20 -       To get an inverse FFT, just change the subroutine name and change the
21 -       FFT instructions to IFFT instructions.
22 -
23 -       To use real data, either set the imaginary parts to zero to get a complex
24 -       vector, or change LD_C to LD_(R,0) to use a real vector.  With a real
25 -       vector, this operation cannot be performed in place, since the output
26 -       data would overwrite unread input data.
27 -
28 - */
29 -
30 - zsp325()
31 - {
32 -       /*
33 -       /*      FFT for rows, four 16 point FFTs on sequential data */
34 -       SUBROUTINE FFT16ROW(zr325ref In_Data, zr325ref Out_Data)
35 -       {
36 -               /*      set up two RAM sections, no need for exchange */
37 -               SET [ =!RMS, =!XCH ];
38 -
39 -               /*      set up pointers for later offset, $A gets In_Data */
40 -               LDR Out_Data => [$B, $A];
41 -
42 -               /*      load two rows into section 0 */
43 -               LD_C:(32) $A => $C0;
44 -
45 -               /*      FFT as two 16 element FFTs */
46 -               FFT_C:(16,2):[FPS:8,LPS:1] $C0;
47 -
48 -               /*      load remainder of entries into section 1 */
49 -               LD_C:(32) $A+64 => $C1;
50 -
```

```
51 -                /*      FFT as two 16 element FFTs */
52 -                FFT_C:(16,2):[FPS:8,LPS:1] $C1;
53 -
54 -                /*      store first result, row bit-reversed */
55 -                ST_C:(32) $C0 => $B:(16,16~);
56 -
57 -                /*      store second result, row bit-reversed */
58 -                ST_C:(32) $C1 => $B+64:(16,16~);
59 -
60 -        }
61 -
62 -        /*      FFT for columns, four 16 point FFTs on interleaved data */
63 -        SUBROUTINE FFT16COL(zr325ref In_Data, zr325ref Out_Data)
64 -        {
65 -                /*      set up two RAM sections, no need for exchange */
66 -                SET [ =!NMS, =!XOR ];
67 -
68 -                /*      set up pointers for later offset, $A gets In_Data */
69 -                LDR Out_Data => [$B, $A];
70 -
71 -                /*      load two columns interleaved into section 0 */
72 -                LD_C:(32) $A:(16,2) => $C0;
73 -
74 -                /*      FFT first set as two 16 element FFTs */
75 -                FFT_C:(32):[FPS:16,LPS:2] $C0;
76 -
77 -                /*      load remainder of entries into section 1 */
78 -                LD_C:(32) $A+4:(16,2) => $C1;
79 -
80 -                /*      FFT as two 16 element FFTs */
81 -                FFT_C:(32):[FPS:16,LPS:2] $C1;
82 -
83 -                /*      store first result, columns bit-reversed */
84 -                ST_C:(32) $C0 => $B:(16~,2);
85 -
86 -                /*      store second result, columns bit-reversed */
87 -                ST_C:(32) $C1 => $B+4:(16~,2);
88 -
89 -        }
90 -        #/
91 - }
```

```
 1 -
 2 - /*    Routines to compute 32x32 2D complex FFT using four VSP chips.
 3 -
 4 -       Operation requires two phases of operation, one to calculate row
 5 -       FFTs, the other to calculate column FFTs.  Using multiple VSP chips
 6 -       requires synchronization between phases so that data can be exchanged.
 7 -       These routines do not include the sychronization.  The routines for
 8 -       each phase can be called from another routine which provides it between
 9 -       calls, or the 68020 can invoke the first phase and wait for it to
10 -       finish before invoking the second.
11 -
12 -       Each chip should be passed pointers to row or column (CHIP * 8) with
13 -       CHIP equalling 0, 1, 2, or 3, depending on the chip.  It will handle
14 -       the 8 rows or columns starting at that point.  Adding a parameter to
15 -       give the number of rows or columns to do would allow the same routine
16 -       to be used by 1 or 2 chips without needing to make multiple calls.
17 -
18 -       The parameter In_Data points to the input vector.  The output vector
19 -       is placed at Out_Data.  The operation can be performed in place if
20 -       desired.  Both input and output vectors are in normal order.
21 -
22 -       To get an inverse FFT, just change the subroutine name and change the
23 -       FFT instructions to IFFT instructions.
24 -
25 -       To use real data, either set the imaginary parts to zero to get a complex
26 -       vector, or change LD_C to LD_(R,0) to use a real vector.  With a real
27 -       vector, this operation cannot be performed in place, since the output
28 -       data would overwrite unread input data.
29 -
30 - */
31 -
32 -
33 - zap325()
34 - {
35 -       /*
36 -       /*      FFT for rows, eight 32 point FFTs on sequential data */
37 -       SUBROUTINE FFT32ROW(zr325ref In_Data, zr325ref Out_Data)
38 -       {
39 -
40 -              /*      set up two RAM sections, swapped by $LC */
41 -              SET [ =!NMS, =XOR ];
42 -
43 -              /*      set pointers to input and output, compensate for increment */
44 -              /*      note: depending on parameter order to get In_Data into $A */
45 -              LDR Out_Data => [$B, $A];
46 -              SUBR $B, #64;
47 -
48 -              /*      initialize loop count */
49 -              LDR #7 => $LC;
50 -
```

```
51 -                /*      start up FFT with first RAM bank */
52 -                LD_C:(32) $A => $C1;
53 -                FFT_C:(32) $C1;
54 -
55 -                /*      loop 7 times, XOR with $LC alternates RAM */
56 -                LD_C:(32) $A+=64 => $C0;
57 -                FFT_C:(32) $C0;
58 -                ST_C:(32) $C1 => $B+=64:(1,32~);
59 -                LOOP:[IE,DL] [!LZ], #3;
60 -
61 -                /* save last RAM bank */
62 -                ST_C:(32) $C1 => $B+=64:(1,32~);
63 -
64 -        }
65 -
66 -        /*      FFT for columns, eight 32 point FFTs on interleaved data */
67 -        SUBROUTINE FFT32COL(zr325ref In_Data, zr325ref Out_Data)
68 -        {
69 -                /*      set up two RAM sections, swapped by $LC */
70 -                SET [ =!MMS, =XOR ];
71 -
72 -                /*      set pointers to data, compensate for first increment */
73 -                /*      note: depending on parameter order to get In_Data into $A */
74 -                LDR Out_Data => [$B, $A];
75 -                SUBR $B, #2;
76 -
77 -                /*      initialize loop count */
78 -                LDR #7 => $LC ;
79 -
80 -                /*      start up FFT with first RAM bank */
81 -                LD_C:(32) $A:(32,1) => $C1;
82 -                FFT_C:(32) $C1;
83 -
84 -                /*      loop 7 times, XOR with $LC alternates RAM */
85 -                LD_C:(32) $A+=2:(32,1) => $C0;
86 -                FFT_C:(32) $C0;
87 -                ST_C:(32) $C1 => $B+=2:(32,1)~;
88 -                LOOP:[IE,DL] [!LZ], #3;
89 -
90 -                /* save last RAM bank */
91 -                ST_C:(32) $C1 => $B+=2:(32,1)~;
92 -
93 -        }
94 -        #/
95 - }
```

```
 1 - /*    Code to notify 68020 of task completion.  This code is never actually
 2 -       called from anywhere.  Instead, its address is used as the return
 3 -       address in the call frame that the 68020 sets up when invoking another
 4 -       routine.  When the routine completes and returns, it will execute this
 5 -       code.  This method allows all routines to be called without having
 6 -       them terminate the task until final completion.
 7 - */
 8 -
 9 - /*    status bit value to indicate finished */
10 - #define FINISHED 2
11 -
12 - zsp325()
13 - {
14 -       /#
15 -       SUBROUTINE FINISH()
16 -       {
17 -              /*      get value for status bits */
18 -              LDR #FINISHED => $X;
19 -
20 -              /*      make sure all operations are complete */
21 -              SYNC:[AS,CU,EU,MU];
22 -
23 -              /*      write to global status latch */
24 -              STR $X => 0x40000;
25 -
26 -              /*      halt */
27 -              HLT;
28 -       }
29 -       #/
30 - }
```

```
 1 - /*    Routine to find peak values in a real matrix.  By varying parameters, it
 2 -       can produce a vector of the max value in each row or column or the max
 3 -       value in the entire matrix.  The calculation can be divided between
 4 -       multiple VSP chips by giving each one a contiguous subset of the problem.
 5 -       The maximum amplitude of a complex matrix can be found by first computing
 6 -       the power (magnitude squared) and finding the maximum of that.  If the
 7 -       magnitude itself is required, it is probably still faster to find the
 8 -       peaks first and then compute the magnitude for only those points rather
 9 -       than computing all the magnitudes and finding the peaks.
10 -
11 -       The routine has a large number of parameters to allow it to be used in a
12 -       flexible manner.  The parameter Number gives the number of separate
13 -       vectors (rows or columns) to find the maximum for.  The parameter Length
14 -       gives the length of each vector (row or column).  Length must be no more
15 -       than 1024 for this routine, though a slight modification would allow up
16 -       to 64K.  The input parameter Spacing gives the distance between starting
17 -       elements of consecutive vectors.  The input parameter Interleave gives
18 -       the distance between consecutive elements within a vector.  Due to some
19 -       constraints on the $MBS_MSS register, bit 24 must also be set in the
20 -       parameter.  Such a machine word can only be created at assembly time.
21 -       It can be created directly by using a parameter ARG(value) with the
22 -       macro definition
23 -
24 -       #define ARG(X) (0x1000000 | (X))
25 -
26 -       or by using a parameter that points to such a value created at assembly
27 -       time.  As a slight compensation, a value other than 1 can be placed in
28 -       the field from bit 24 to 30.  This value will be used as the $MBS value
29 -       while the rest of the Interleave value is used as $MSS.  This allows
30 -       for each vector to be addressed more generally.  If the $MBS_MSS register
31 -       already contains an appropriate value, it can be passed.  The parameter
32 -       In_Data points to the start of the first input vector.  The output will
33 -       be placed at Out_Data.  The output will consist of a vector of length
34 -       Number of pairs of maximum values and the index between 0 and Length of
35 -       where that value appeared.
36 -
37 -
38 -       To find the maximum row values for a ROWxCOL matrix using N VSP chips
39 -       PEAK2D(COL/N, ROW, ROW, ARG(1), &(In+CHIP*ROW*COL/N), &(Out+CHIP*2*COL/N))
40 -
41 -       To find the maximum column values for a ROWxCOL matrix using N VSP chips
42 -       PEAK2D(ROW/N, COL, 1, ARG(ROW), &(In+CHIP*ROW/N), &(Out+CHIP*2*COL/N))
43 -
44 -       assuming that ROW and COL are evenly divisible by N.  Using more than
45 -       one chip on each local bus will probably not improve performance because
46 -       the operation is bus-bandwidth bound.  When using two chips, CHIP should
47 -       be set to 0 or 1 in the above formulas.
48 -
49 -       To find the overall maximum, treat as one long row using 1 VSP chip
50 -       PEAK2D(1, ROW*COL, any_value, ARG(1), &In, &Out)
```

```
51 -
52 -       To find minimum values, just change the MAX instruction to MIN.
53 -
54 -
55 -       Note: It is technically possible to accomplish the setting of the upper
56 -       bits of $MBS_MSS at execution time with sufficient ingenuity.  It requires
57 -       using (slow) floating point operations to manipulate the higher bits.  A
58 -       lookup table is another possibility.
59 -
60 - */
61 -
62 - zsp325()
63 - {
64 -       /#
65 -       SUBROUTINE PEAK2D(      zr325int Number,
66 -                                               zr325int Length,
67 -                                               zr325int Spacing,
68 -                                               zr325val Interleave,
69 -                                               zr325ref In_Data,
70 -                                               zr325ref Out_Data)
71 -       {
72 -
73 -               /*      set up automatic save to $SAR */
74 -               SET [ =SAR ];
75 -
76 -               /*      set up parameters in correct registers
77 -                       note: LDRs depend on parameter order to put In_Data into
78 -                             $A, Interleave into $MBS_MSS and Number into $LC.
79 -               */
80 -               LDR Out_Data => [$SAR, $A, $MBS_MSS];
81 -               LDR Length => [$PR, $LC];
82 -
83 -
84 -               /*      loop Number times, handling Length each time, addressing properly */
85 -               MAX_R:($MMPT,$REPEAT) $A:($MSS,$MBS) => $MMMX;
86 -               ADDR $A, Spacing;
87 -               LOOP:[IE,DL] [!LZ], #2;
88 -
89 -       }
90 -       #/
91 -
92 - }
```

```
 1 - /*    Routine to perform polar to rectangular conversion on a complex vector.
 2 -       Uses separate sine and cosine tables.  Could use one table for both,
 3 -       but that would require extra time.  Only operates on angles in the first
 4 -       quadrant since those are the only ones produced by the rectangular to
 5 -       polar conversion.  The table size will determine the accuracy of the
 6 -       conversion.  The error will be less than 100% * pi / (4 * table size).
 7 -
 8 -       The vector length is passed in the parameter Length.  The parameter
 9 -       In_Data points to the start of the vector to be converted.  The result
10 -       is placed at Out_Data.  This algorithm can be performed in place if
11 -       desired.
12 -
13 -       This routine uses software pipelining to maximize throughput.  This
14 -       should cause the bus to be busy most of the time.  If two chips are
15 -       performing this at the same time, there will not be enough bandwidth.
16 -       Benchmarking will need to be used to determine whether this is faster
17 -       than a version which does not attempt pipelining but uses larger blocks.
18 -
19 -       Note:  changing JMPC instructions to use IE qualifier causes incorrect
20 -       results.  It works correctly for other routines.  Not using IE slows
21 -       this routine down.  Moving the software pipeline loop kernel so that
22 -       the JMPC doesn't block a subsequent instruction that could be executed
23 -       concurrently with the previous one would regain most of the speed.
24 -
25 - */
26 -
27 - /*    need trig functions for tables */
28 - #include <math.h>
29 -
30 - /*    size of sine and cosine tables */
31 - #define TAB_SIZE 128
32 -
33 - /*    size of increment between table entries */
34 - #define INCREMENT (asin(1.0)/(TAB_SIZE-1))
35 -
36 - /*    assembly generation function */
37 - zsp325()
38 - {
39 -       int index;
40 -
41 -       /*      Generate trig function tables.  */
42 -       /#
43 -       SinTab::
44 -       #/
45 -       for (index = 0; index < TAB_SIZE; index++)
46 -       {
47 -               /#
48 -                       .DATA { (IEEE_Float(sin(index*INCREMENT))) };
49 -               #/
50 -       }
```

```
 51 -       /#
 52 -       CosTab::
 53 -       #/
 54 -       for (index = 0; index < TAB_SIZE; index++)
 55 -       {
 56 -              /#
 57 -                     .DATA { (IEEE_Float(cos(index*INCREMENT))) };
 58 -              #/
 59 -       }
 60 -
 61 -       /#
 62 -       SUBROUTINE POL2RECT(zr325int Length, zr325ref In_Data, zr325ref Out_Data)
 63 -       {
 64 -
 65 -              /*      use both RAM banks to optimize throughput */
 66 -              /*      Note: chosen interleaving pattern assumes LUT instruction
 67 -                     makes no use of EU since it is a data movement instruction.
 68 -                     Also assumes that arithmetic operations that use external
 69 -                     operands can't be overlapped with move instructions, though
 70 -                     this isn't clear.
 71 -                     Benchmark might be needed to check the interleaving pattern.
 72 -              */
 73 -
 74 -              /*      set up two RAM sections, swapped by $LC, round to nearest */
 75 -              SET [ =!NMS, =XOR, =ROUND ];
 76 -
 77 -              /*      load pointers to data, shifting $A to angle, compensate pre-inc */
 78 -              ISETR In_Data => $A;
 79 -              LDR Out_Data => $B;
 80 -              SUBR $B, #64;
 81 -
 82 -              /*      initialize loop count to number of 32s, skip loop if none */
 83 -              SHRSETR:[SHIFT=5] Length => $LC;
 84 -              JMPC [ZR], Do_Rest;
 85 -
 86 -              /*      start up conversion with first RAM bank */
 87 -              /*      load angle into imaginary part */
 88 -              LD_I:(32) $A:(2,1) => $IO;
 89 -              /*      multiply by factor to get table offset */
 90 -              MULT_(R,R):(32) $C0, #(IEEE_Float(1.0/INCREMENT)) => $IO;
 91 -              /*      convert to integer to get integer part right justified */
 92 -              FPINT_R:(32) $IO => $IO;
 93 -
 94 -              /*      if no more to do, skip rest of loop */
 95 -              JMPC:[DL] [LZ], Do_Store;
 96 -
 97 -              /*      loop with software pipelining */
 98 - Loop::
 99 -              /*      load and start next vector */
100 -              LD_I:(32) $A+=64:(2,1) => $IO;
```

```
101 -               MULT_(R,R):(32) $C0, #(IEEE_Float(1.0/INCREMENT)) => $I0;
102 -               /*      do bus operation for previous during execution of current */
103 -               LUT_R:(32) CosTab, $I1 => $R1;
104 -               /*      do next operation on current vector */
105 -               FPINT_R:(32) $I0 => $I0;
106 -               /*      finish and store previous vector */
107 -               LUT_R:(32) SinTab, $I1 => $I1;
108 -               /*      assume external operand fetch monopolizes bus unit */
109 -               MULT_(R,R):(32) $C1, $A-65:(2,1) => $C1;
110 -               ST_C:(32) $C1 => $B+=64;
111 -               /*      decrement count (switches banks) and loop immediately if not done */
112 -               JMPC:[DL] [!LZ], Loop;
113 -
114 - Do_Store::
115 -               /*      finish up last RAM bank */
116 -               /*      look up cosine of angle in table */
117 -               LUT_R:(32) CosTab, $I1 => $R1;
118 -               /*      look up sine of angle in table */
119 -               LUT_R:(32) SinTab, $I1 => $I1;
120 -               /*      multiply cosine and sine by magnitude to get real and imaginary */
121 -               MULT_(R,R):(32) $C1, $A-1:(2,1) => $C1;
122 -               /*      store resulting complex number in rectangular coordinates */
123 -               ST_C:(32) $C1 => $B+=64;
124 -
125 - Do_Rest::
126 -               /*      handle any remainder left after blocks of 32 */
127 -
128 -               /*      shift remainder into $NMPT, use [TC] to zero high bit (32s) */
129 -               SHLSETR:[SHIFT=18,TC] Length => $PR;
130 -               JMPC [ZR], End;
131 -
132 -               /*      finish remainder */
133 -               /*      load angle into imaginary part */
134 -               LD_I:($NMPT) $A+=64:(2,1) => $I0;
135 -               /*      multiply by factor to get table offset */
136 -               MULT_(R,R):($NMPT) $C0, #(IEEE_Float(1.0/INCREMENT)) => $I0;
137 -               /*      convert to integer to get integer part right justified */
138 -               FPINT_R:($NMPT) $I0 => $I0;
139 -               /*      look up cosine of angle in table */
140 -               LUT_R:($NMPT) CosTab, $I0 => $R0;
141 -               /*      look up sine of angle in table */
142 -               LUT_R:($NMPT) SinTab, $I0 => $I0;
143 -               /*      multiply cosine and sine by magnitude to get real and imaginary */
144 -               MULT_(R,R):($NMPT) $C0, $A-1:(2,1) => $C0;
145 -               /*      store resulting complex number in rectangular coordinates */
146 -               ST_C:($NMPT) $C0 => $B+=64;
147 -
148 - End::
149 -
150 -       }
```

```
151 -       #/
152 - }
```

```
 1 - /*    Routine to perform polar to rectangular conversion on a complex vector.
 2 -       Uses separate sine and cosine tables.  Could use one table for both, but
 3 -       that would require extra time.  Only operates on angles in the first
 4 -       quadrant since those are the only ones produced by the rectangular to
 5 -       polar conversion.  Other angles will produce unexpected results.  The
 6 -       table size will determine the accuracy of the conversion.  The error
 7 -       will be less than 100% * pi / (4 * table size).
 8 -
 9 -       Length of the vector to be converted is passed in Length.  In_Data
10 -       points to the start of the input vector.  Output is placed at location
11 -       Out_Data.  Conversion can be performed in place if desired.
12 -
13 -       This version assumes performance is bounded by local bus bandwidth and
14 -       therefore doesn't attempt software pipelining alternating RAM banks.
15 -       Instead it uses the entire RAM at once to minimize bus traffic for
16 -       instruction fetching.  This also makes the code more readable.  Testing
17 -       will be needed to see which method is faster.  Using half of RAM and
18 -       loading magnitude in other half before MULT might save more bandwidth.
19 - */
20 -
21 - /*    need trig functions for tables */
22 - #include <math.h>
23 -
24 - /*    size of sine and cosine tables */
25 - #define TAB_SIZE 128
26 -
27 - /*    size of increment between table entries */
28 - #define INCREMENT (asin(1.0)/(TAB_SIZE-1))
29 -
30 - /*    assembly generation function */
31 - zsp325()
32 - {
33 -       int index;
34 -
35 -       /*      Generate trig function tables. */
36 -       /#
37 -       SinTab::
38 -       #/
39 -       for (index = 0; index < TAB_SIZE; index++)
40 -       {
41 -               /#
42 -                       .DATA { (IEEE_Float(sin(index*INCREMENT))) };
43 -               #/
44 -       }
45 -       /#
46 -       CosTab::
47 -       #/
48 -       for (index = 0; index < TAB_SIZE; index++)
49 -       {
50 -               /#
```

```
51 -                        .DATA { (IEEE_Float(cos(index*INCREMENT))) };
52 -                #/
53 -        }
54 -
55 -
56 -        /#
57 -        SUBROUTINE POL2RECT(zr325int Length, zr325ref In_Data, zr325ref Out_Data)
58 -        {
59 -
60 -                /*      set up one RAM section, set rounding to nearest */
61 -                SET [ =RMS, =ROUND ];
62 -
63 -                /*      load pointers to data, compensate for pre-increment */
64 -                /*      increment $A at load so it points to angle part */
65 -                ISETR In_Data => $A;
66 -                LDR Out_Data => $B;
67 -                SUBR [$A, $B], #128;
68 -
69 -                /*      initialize loop count to number of 64s, skip loop if none */
70 -                SHRSETR:[SHIFT=6] Length => $LC;
71 -                JMPC [ER], Do_Rest;
72 -
73 - Loop::
74 -                /*      load angle into imaginary part */
75 -                LD_I:(64) $A+=128:(2,1) => $I;
76 -                /*      multiply by factor to get table offset */
77 -                MULT_(R,R):(64) $C, #(IEEE_Float(1.0/INCREMENT)) => $I;
78 -                /*      convert to integer to get integer part right justified */
79 -                FPINT_R:(64) $I => $I;
80 -                /*      look up cosine of angle in table */
81 -                LUT_R:(64) CosTab, $I => $R;
82 -                /*      look up sine of angle in table */
83 -                LUT_R:(64) SinTab, $I => $I;
84 -                /*      multiply cosine and sine by magnitude to get real and imaginary */
85 -                MULT_(R,R):(64) $C, $A-1:(2,1) => $C;
86 -                /*      store resulting complex number in rectangular coordinates */
87 -                ST_C:(64) $C => $B+=128;
88 -                /*      decrement $LC, loop immediately on not zero */
89 -                JMPC:[DL,IE] [!LZ], Loop;
90 -
91 - Do_Rest::
92 -                /*      handle remainder left after blocks of 64 */
93 -
94 -                /*      shift remainder into $MMPT, skip if none */
95 -                SHLSETR:[SHIFT=18] Length => $PR;
96 -                JMPC [ER], End;
97 -
98 -                /*      finish remainder */
99 -                /*      load angle into imaginary part */
100 -               LD_I:($MMPT) $A+=128:(2,1) => $I;
```

```
101 -               /*      multiply by factor to get table offset */
102 -              MULT_(R,R):($NMPT) $C, #(IEEE_Float(1.0/INCREMENT)) => $I;
103 -               /*      convert to integer to get integer part right justified */
104 -              FPINT_R:($NMPT) $I => $I;
105 -               /*      look up cosine of angle in table */
106 -              LUT_R:($NMPT) CosTab, $I => $R;
107 -               /*      look up sine of angle in table */
108 -              LUT_R:($NMPT) SinTab, $I => $I;
109 -               /*      multiply cosine and sine by magnitude to get real and imaginary */
110 -              MULT_(R,R):($NMPT) $C, $A-1:(2,1) => $C;
111 -               /*      store resulting complex number in rectangular coordinates */
112 -              ST_C:($NMPT) $C => $B+=128;
113 -
114 - End::
115 -
116 -       }
117 -       #/
118 - }
```

```
 1 - /*    Routine to compute magnitude squared for a complex vector.  If the vector
 2 -       is the FFT of a signal, this is the power spectrum of the signal.  This
 3 -       routine is faster than the rectangular to polar conversion and should be
 4 -       used if the magnitude squared is as useful as the magnitude.  For example,
 5 -       the point of maximum magnitude is also the point of maximum power.
 6 -
 7 -       This routine can be performed in place, producing an output vector half
 8 -       the length of the input.  This would leave gaps if multiple VSP chips
 9 -       were being used.  If the calculation is not performed in place, or gaps
10 -       are acceptable, there is no problem using multiple chips to calculate
11 -       parts of the output vectors.
12 -
13 -       Note: this routine is I/O bound even on a single VSP.  With two sharing
14 -       a bus, it will be even worse.  If it is being used immediately after an
15 -       FFT operation, it would be more efficient to perform the magnitude
16 -       squared operation as the last step of an FFT routine before storing the
17 -       result.  This would save a store and reload.
18 -
19 -       The input parameter Length contains the number of elements in the
20 -       input vector.  The parameter In_Data points to the start of the
21 -       input vector.  The output will be placed at Out_Data.
22 -
23 - */
24 -
25 - zsp325()
26 - {
27 -       /*
28 -       SUBROUTINE POWER(zr325int Length, zr325ref In_Data, zr325ref Out_Data)
29 -       {
30 -
31 -               /*      use both RAM banks to improve throughput */
32 -
33 -               /*      set up two RAM sections, swapped by $LC */
34 -               SET [ =!RMS, =XOR ];
35 -
36 -               /*      set up pointers to data areas, compensate $B for pre-increment */
37 -               /*      Note: Load depends on parameter order to get In_Data into $A */
38 -               LDR Out_Data => [$B, $A] ;
39 -               SUBR $B, #32;
40 -
41 -               /*      initialize loop count to number of 32s, skip loop if none */
42 -               SHRSETR:[SHIFT=5] Length => $LC;
43 -               JMPC [ZR], Do_Rest;
44 -
45 -               /*      start up with first RAM bank */
46 -               LD_C:(32) $A => $C0;
47 -               MOSQ_R:(32) $C0 => $R0;
48 -
49 -               /*      if no more to do, skip rest of loop */
50 -               JMPC:[DL] [LZ], Do_Store;
```

```
51 -
52 -                /*      loop with software pipeline, XOR with $LC alternates RAM */
53 -                LD_C:(32) $A+=64 => $C0;
54 -                MGSQ_R:(32) $C0 => $R0;
55 -                ST_R:(32) $R1 => $B+=32;
56 -                LOOP:[DL] [!LZ], @3;
57 -
58 - Do_Store::
59 -                /* save last RAM bank */
60 -                ST_R:(32) $R1 => $B+=32;
61 -
62 - Do_Rest::
63 -                /*      handle remainder left after blocks of 32 */
64 -
65 -                /*      shift remainder into $NMPT, use [TC] to zero high bit (32s) */
66 -                SHLSETR:[SHIFT=18,TC] Length => $PR;
67 -                JMPC [ZR], End;
68 -
69 -                /*      finish up remainder */
70 -                LD_C:($NMPT) $A+=64 => $C0;
71 -                MGSQ_R:($NMPT) $C0 => $R0;
72 -                ST_R:($NMPT) $R0 => $B+=32;
73 -
74 - End::
75 -
76 -        }
77 -        @/
78 -
79 - )
```

```
 1 - /*    VPH code for convolution of a real sequence of up to 64 points with
 2 -       another longer real sequence, producing up to 1024 outputs.  This
 3 -       size can be done with a single FIR instruction.  This code can be
 4 -       called repeatedly on a single processor to handle convolutions where
 5 -       more than 1024 output points are required as long as the shorter
 6 -       sequence is still less than 64 points.  However, a different routine
 7 -       designed for a longer convolution would be more efficient.  This
 8 -       same code can be used on multiple VSP chips simultaneously to give
 9 -       a considerable speed increase.  There may be no benefit to executing
10 -       on more than one VSP chip per bus because the FIR instruction may not
11 -       give up the bus between output points.
12 -
13 -       To get a full convolution of the input requires padding both ends of
14 -       the longer input sequence with a number of zeroes equal to the length
15 -       of the shorter sequence minus one.  This is required in order to
16 -       explicitly provide the zeroes that are assumed to be multiplied by
17 -       elements of the shorter sequence that extend beyond the ends of the
18 -       longer one during the convolution process.  The length of the output
19 -       sequence should be equal to the sum of the lengths of the (unpadded)
20 -       input sequences minus one.  If a circular convolution is desired
21 -       instead of a linear one, the zero padding should be replaced with
22 -       points from the other end of the input sequence.
23 -
24 -       The shorter input length is passed in Coef_Length.  The output length
25 -       (equal to input length before padding plus coefficient length minus one)
26 -       is passed as Out_Length.  Coefficients points to the shorter sequence
27 -       (typically FIR filter coefficients).  In_Data points to the start
28 -       of the longer sequence (possibly a zero pad).  The output is placed
29 -       at Out_Data.  Typical call for a four tap filter:
30 -       CALL RCONV(4, 1024, &Coef, &In, &Out)
31 -
32 -       The convolution can be performed in place with careful choices of
33 -       parameter values.  If the convolution requires multiple calls on a
34 -       single VSP chip, the output must begin at the first location of the
35 -       long input.  This avoids overwriting inputs that will be needed for
36 -       the next call.  However, if multiple chips are being used, the output
37 -       must overwrite the last input used in its computation.  This works
38 -       because the VSP chip has already read the input into internal RAM
39 -       for further use.  It is necessary because that input is the first
40 -       one which will not be needed by the chip working on the previous
41 -       portion of the convolution.  Some further care is needed in the
42 -       initial startup of in-place multiple chip convolution to ensure that
43 -       a chip does not write over any input values before the subsequent
44 -       chip reads them in.  A multiple call, multiple chip convolution
45 -       cannot be done in place because the constraints are contradictory.
46 -       However, such a large data set would not fit into shared memory.
47 -
48 -       Splitting up a convolution between NUM_CHIPS chips would require
49 -       something like the following invocation for chip ranging from zero
50 -       to (NUM_CHIPS - 1):
```

```
51 -
52 -       CALL RCONV(COEF_LEN, OUT_SIZE(chip), &Coef,
53 -                                &(In + DATA_OFFSET(chip)), &(Out + DATA_OFFSET(chip)));
54 -
55 -       with the definitions
56 -
57 -       #define OUT_LEN (IN_LEN + COEF_LEN - 1)
58 -       #define DATA_OFFSET(CHIP) (((CHIP) * OUT_LEN) / NUM_CHIPS)
59 -       #define OUT_SIZE(CHIP) (DATA_OFFSET(CHIP+1) - DATA_OFFSET(CHIP))
60 -
61 -       Note that since all this routine does is to load various values into
62 -       internal registers and RAM and then execute a single instruction, it
63 -       might be faster for the 68020 to load the values directly and execute
64 -       the FIR instruction in slave mode.  The same applies to the complex
65 -       convolution and the correlations.
66 - */
67 -
68 - zsp325()
69 - {
70 -       /*
71 -       SUBROUTINE RCONV(       zr325int Coef_Length,
72 -                                               zr325int Out_Length,
73 -                                               zr325ref Coefficients,
74 -                                               zr325ref In_Data,
75 -                                               zr325ref Out_Data)
76 -       {
77 -               /*      set up mode properly, one RAM bank, 24 bit integers */
78 -               SET [ =BMS, =!XOR , =!FMT ];
79 -
80 -               /*      set $SAR to put output in correct place */
81 -               LDR Out_Data => $SAR;
82 -
83 -               /*      to get real coefficients in zig-zag order, need to load half
84 -                       as many (rounded up) "complex" coefficients
85 -               */
86 -               SHLSETR:[SHIFT=17] Coef_Length => $PR;
87 -               ADDR $PR, #0x020000;
88 -
89 -               /*      load coefficients in reverse zig-zag real order */
90 -               LDR Coefficients => $A;
91 -               ADDR $A, Coef_Length;
92 -               SUBR $A, #2;
93 -               LD_(I,R):($MMPT) $A:(-1,1) => $C0;
94 -
95 -               /*      now set up actual lengths for FIR instruction */
96 -               SHLSETR:[SHIFT=18] Coef_Length => $PR;
97 -               ADDR $PR, Out_Length;
98 -
99 -               /*      convolve with input sequence  */
100 -              FIR_R:($MMPT, $REPEAT) $Z0, *In_Data;
```

```
101 -
102 -        }
103 -        #/
104 - }
```

```
 1 - /*    Program to perform real correlation between two real vectors with up to
 2 -       64 elements in the shorter one and up to 1024 elements in the output
 3 -       using a single zoran processor.  Due to requirements of the instruction
 4 -       used, the longer real vector must be padded at both ends with (shorter
 5 -       length - 1) real zero elements.  These are needed for when the shorter
 6 -       vector extends beyond the end of the longer during the operation.  If the
 7 -       vectors are the same length, either may be considered the longer one.
 8 -
 9 -       The length of the short vector is passed in the parameter Coef_Length.
10 -       The length of the desired output vector (typically equal to the sum of
11 -       the lengths of the input vectors, minus one) is passed in Out_Length.
12 -       Coefficients points to the short input vector.  In_Data points to
13 -       the first zero pad in the longer input vector.  The output is placed at
14 -       Out_Data.  The output data could be stored in the place of the first
15 -       input vector if desired.  Typical call to perform a full autocorrelation
16 -       in place with a 64 (padded to 190) element vector:
17 -       CALL CCORR(64, 127, &in, &(in+63), &in)
18 -       The (in+63) skips the padding at the front of the vector.
19 -
20 -       Note: if this routine will always be used for two equal length vectors,
21 -       only one length parameter is needed.  The other can be computed from it
22 -       with some extra overhead.  On the other hand, if this routine will be
23 -       used repeatedly for the same length, sending a precomputed $PR value
24 -       instead of a length would reduce overhead slightly.
25 - */
26 -
27 - zsp325()
28 - {
29 -       /#
30 -       SUBROUTINE RCORR(       zr325int Coef_Length,
31 -                                       zr325int Out_Length,
32 -                                       zr325ref Coefficients,
33 -                                       zr325ref In_Data,
34 -                                       zr325ref Out_Data)
35 -       {
36 -               /*      set up mode properly, one RAM section, 24 bit integers */
37 -               SET [ =MMS, =!XOR, =!FMT ];
38 -
39 -               /*      set $SAR to put output in correct place */
40 -               LDR Out_Data => $SAR;
41 -
42 -               /*      to get real coefficients in zig-zag order, need to load half
43 -                       as many (rounded up) "complex" coefficients
44 -               */
45 -               SHLSETR:[SHIFT=17] Coef_Length => $PR;
46 -               ADDR $PR, #0x020000;
47 -
48 -               /*      load coefficients in zig-zag real order */
49 -               LD_C:($MMPT) *Coefficients => $C0;
50 -
```

```
51 -                /*      load vector lengths into parameter register
52 -                        $NMPT = Coef_Length, $REPEAT = Out_Length
53 -                */
54 -                SHLSETR:[SHIFT=16] Coef_Length => $PR;
55 -                ADDR $PR, Out_Length;
56 -
57 -                /*      correlate with input sequence  */
58 -                FIR_R:($NMPT,$REPEAT) $Z0, *In_Data;
59 -        }
60 -        #/
61 - }
```

```
 1 - /*    Routine to set up reciprocal table and one to generate inline code
 2 -       to compute the reciprocals for a vector.  The algorithm is to perform
 3 -       a table lookup to get a starting estimate and then perform Newton-Raphson
 4 -       iterations until accuracy is 24 bits.  This requires that
 5 -       TAB_BITS * (1 << NUM_ITER) >= 24.
 6 -
 7 -       Might be better to split reciptab into a zsp325 routine to create table
 8 -       and link in after assembly.  The reciprocal function would still be
 9 -       included by the using routine.  This would prevent including table
10 -       more than once if it is used by multiple other routines.
11 - */
12 -
13 - /*    define number of bits of accuracy in table, table size, and iterations */
14 - #define TAB_BITS 6
15 - #define TAB_SIZE (1 << (TAB_BITS-1))
16 - #define NUM_ITER 2
17 -
18 - /*    Function to create reciprocal table for initial estimate.  Must be
19 -       called once if reciprocals are to be used.
20 - */
21 - void reciptab()
22 - {
23 -       long i;
24 -       union
25 -       {
26 -               float flt;
27 -               long bits;
28 -       } max, min;
29 -
30 -       /*      generate label for start of table */
31 -       /#
32 - RecipTab::
33 -       #/
34 -
35 -       /*      generate the table entries */
36 -       for (i = 0; i < TAB_SIZE; i++)
37 -       {
38 -               /* calculate max and min values that will use this entry */
39 -               min.bits = (127L << 23) + (i << (24 - TAB_BITS));
40 -               max.bits = (127L << 23) + ((i+1) << (24 - TAB_BITS));
41 -
42 -               /*      use midpoint between their reciprocals to minimize error */
43 -               /#
44 -               .DATA { IEEE_Float(0.5 / max.flt + 0.5 / min.flt) } ;
45 -               #/
46 -       }
47 - }
48 -
49 -
50 - /*    Function to produce inline assembly to calculate the reciprocals for
```

```
51 -        a vector in internal RAM.  The internal RAM must be set up to have two
52 -        banks and the input vector must be in R0.  This limits the input vector
53 -        length to 32 or less.  The result vector ends up in R0.  All internal
54 -        RAM banks are overwritten with intermediate results.
55 -
56 -        This function is essentially a macro.  It is called from within a
57 -        zsp325() function and generates assembly code.  It does not produce
58 -        any calls that execute at run time.  The function reciptab must also
59 -        have been called by the zsp325() function or there will be an error
60 -        during assembly.
61 - */
62 - void recip(int length)
63 - {
64 -        int i;
65 -
66 -        /#
67 -        /*      split into exponent and mantissa, negate exponent, trap zero */
68 -        SPLIT_R:((length)):[DV] $R0 => $C1;
69 -
70 -        /*      look up initial estimate of reciprocal of mantissa */
71 -        LUT_R:((length)):[SHIFT=(24-TAB_BITS)] RecipTab, $I1 => $I0;
72 -
73 -        /*      change sign of estimate to match initial input sign */
74 -        SIGN_R:((length)) $R0, $I0 => $R0;
75 -        #/
76 -
77 -        /*      generate Newton-Raphson iterations inline */
78 -        for (i = 0; i < NUM_ITER; i++)
79 -        {
80 -                /#
81 -                /*      new estimate = estimate * (2.0 - estimate * input) */
82 -                SBM_R:((length)) $R0, $I1, #2.0 => $I0;
83 -                MULT_R:((length)) $R0, $I0 => $R0;
84 -                #/
85 -        }
86 -
87 -        /#
88 -        /*      recombine resulting mantissa with exponent */
89 -        JOIN_R:((length)) $R1, $R0 => $R0;
90 -        #/
91 - }
```

```
 1 - /*    Routine to perform rectangular to polar conversion on a complex vector.
 2 -       Uses a Cordic-like algorithm for magnitude and an arctangent lookup
 3 -       table for angle in radians.  Maximum error in magnitude is 2% for
 4 -       three iterations, which can easily be reduced to a value as low as
 5 -       0.0002% by increasing the number of iterations to eight.  Maximum error
 6 -       in angle is 2.33% of a quadrant (0.0366 radians) for 5 bits from each
 7 -       mantissa, which requires a table of 1K entries for first quadrant angles
 8 -       only.  The table size must be quadrupled for each doubling in precision,
 9 -       so this approach is not practical for high precision.
10 -
11 -       This program computes only first quadrant angles.  Other angles are
12 -       moved into the first quadrant by taking the absolute value of both
13 -       components.  This means that the angle will be correct for the first
14 -       quadrant, equal to pi minus the true angle in the second quadrant,
15 -       equal to the true angle minus pi in the third quadrant and equal to
16 -       minus the true angle in the fourth quadrant.  These angles are the
17 -       absolute values of the angles between the complex numbers and the
18 -       nearest real axis.  If full angles are needed, the table can just be
19 -       quadrupled to handle sign bits in the index.
20 -
21 -       The vector length is passed in the parameter Length.  The parameter
22 -       In_Data points to the vector to be converted.  The output is placed
23 -       at Out_Data.  The conversion can be performed in place if desired.
24 -
25 - */
26 -
27 - /*    need arctangent function for table */
28 - #include <math.h>
29 -
30 - /*    number of bits from each mantissa to be used in arctangent table lookup */
31 - #define TAB_BITS 5
32 -
33 - /*    number of Cordic iterations for magnitude calculations */
34 - #define MAG_ITER 3
35 -
36 - /*    function to return arctangent table value for index number */
37 - /*    only handles first quadrant angles, but could be modified for all four */
38 - float tabentry(int i)
39 - {
40 -       int fbits[2];
41 -       int part;
42 -       int index;
43 -
44 -       /*      determine numbers that would have produced the given index */
45 -       for (part = 0; part <= 1; part++)
46 -       {
47 -               /*      extract interleaved mantissa bits from index */
48 -               fbits[part] = 0;
49 -               for (index = 0; index < TAB_BITS; index++)
50 -               {
```

```
51 -                        fbits[part] |= (1 << index) & (i >> index + part);
52 -                }
53 -        }
54 -
55 -        /*      return middle angle of the possible range */
56 -        return (atan2((double) fbits[0] + 1, (double) fbits[1]) +
57 -                        atan2((double) fbits[0], (double) fbits[1] + 1)) / 2.0;
58 - }
59 -
60 -
61 - /*     actual assembly generation function */
62 - zap325()
63 - {
64 -        int index;
65 -
66 -        /*      Generate arctangent table.  Because of normalization, only first
67 -                entry and last three quarters of table are actually used.
68 -        */
69 -        /#
70 -        AtanTab::
71 -        #/
72 -        for (index = 0; index < (1 << TAB_BITS*2); index++)
73 -        {
74 -                /#
75 -                        .DATA { (IEEE_Float(tabentry(index))) };
76 -                #/
77 -        }
78 -
79 -        /#
80 -        SUBROUTINE RECT2POL(zr325int Length, zr325ref In_Data, zr325ref Out_Data)
81 -        {
82 -
83 -                /*      set up two RAM sections, swapping on each loop iteration */
84 -                SET [ =!MMS, =XOR ];
85 -
86 -                /*      load data pointers, parameter order gets In_Data into $A */
87 -                LDR Out_Data => [$B, $A];
88 -
89 -                /*      initialize loop count to number of 32s, skip loop if none */
90 -                SHRSETR:[SHIFT=5] Length => $LC;
91 -                JMPC [ZR], Do_Rest;
92 -
93 -                /*      first part of loop to fill software pipeline */
94 -
95 -                /*      load to bank 1, take absolute value to put in first quadrant */
96 -                LD_||:(32) $A => $C1;
97 -                /*      align mantissas and interleave to create atan index in $ID */
98 -                ALIGN:(32) $R1, $I1 => $IO;
99 -                /*      do cordic iterations to get magnitude in $R1, takes a while */
100 -               MAG:(32,MAG_ITER) $C1;
```

```
101 -               /*      look up arctangent in table, overlaps with MAG */
102 -               /*      extra +1 is because of the sign bits technically included */
103 -               LUT:(32):[SHIFT=(23 - 2*(TAB_BITS+1))] AtanTab, $I0 => $I0;
104 -               /*      store angle, overlaps with MAG */
105 -               ST_I:(32) $I0 => $B+=1:(2,1);
106 -
107 -               /*      decrement $LC, end loop if done */
108 -               JMPC:[DL] [LZ], Do_Store;
109 -
110 -               /*      software pipelined loop, allows next load to overlap MAG */
111 - Loop::
112 -               LD_||:(32) $A+=64 => $C1;
113 -               /*      store magnitude from previous vector */
114 -               ST_R:(32) $R0 => $B-1:(2,1);
115 -               ALIGN:(32) $R1, $I1 => $I0;
116 -               MAG:(32,MAG_ITER) $C1;
117 -               LUT:(32):[SHIFT=(23 - 2*(TAB_BITS+1))] AtanTab, $I0 => $I0;
118 -               ST_I:(32) $I0 => $B+=64:(2,1);
119 -               /*      decrement counter and branch to top if not done */
120 -               JMPC:[DL] [!LZ], Loop;
121 -
122 - Do_Store::
123 -               /*      rest of loop to empty software pipeline */
124 -               /*      store magnitude from last vector */
125 -               ST_R:(32) $R0 => $B-1:(2,1);
126 -
127 - Do_Rest::
128 -               /*      handle remainder left after blocks of 32 */
129 -
130 -               /*      shift remainder into $NMPT, use [TC] to zero high bit */
131 -               SHLSETR:[SHIFT=16,TC] Length => $PR;
132 -               JMPC [ER], End;
133 -
134 -               /*      need MAG_ITER in $REPEAT to use $PR with MAG */
135 -               ADDR $PR, #MAG_ITER;
136 -
137 -               /*      finish up remainder */
138 -               LD_||:($NMPT) $A+=64 => $C1;
139 -               ALIGN:($NMPT) $R1, $I1 => $I0;
140 -               MAG:($NMPT,$REPEAT) $C1;
141 -               LUT:($NMPT):[SHIFT=(23 - 2*(TAB_BITS+1))] AtanTab, $I0 => $I0;
142 -               ST_I:($NMPT) $I0 => $B+=64:(2,1);
143 -               ST_R:($NMPT) $R1 => $B-1:(2,1);
144 -
145 - End::
146 -
147 -       }
148 -       #/
149 - }
```

```
 1 - /*    Test program to see if Zorans work */
 2 -
 3 - /*    absolute base addresses from memory map */
 4 - #define PRAM 0x00000
 5 - #define FOUR_PORT 0x20000
 6 - #define STATUS_LATCH 0x40000
 7 -
 8 - #include "recip.asm"
 9 -
10 - zsp325()
11 - {
12 -       int i;
13 -       float x;
14 -
15 -       /*      set up reciprocal starting table */
16 -       reciptab();
17 -
18 - /#
19 -       .ORG 0
20 - SUBROUTINE MAIN()
21 - {
22 -       /*      set up two RAM sections */
23 -       SET [=!NMS, =!XDR];
24 -       LD_R:(16) FOUR_PORT => $R0;
25 - #/
26 -       recip(16);
27 - /#
28 -       ST_R:(16) $R0 => FOUR_PORT;
29 - }
30 - #/
31 - )
```

```
 1 - /*    Initialization for synthetic stack frames for FFT2D16.
 2 -       rstackN and cstackN are starting $SP values for chip N.
 3 -       One free spot left in stacks for interrupt.
 4 - */
 5 -
 6 - /*    absolute base addresses from memory map */
 7 - #define FRAM 0x00000
 8 - #define FOUR_PORT 0x20000
 9 - #define STATUS_LATCH 0x40000
10 -
11 - #define COLUMN(N) (FOUR_PORT + (N) * 2)
12 - #define ROW(N) (FOUR_PORT + (N) * 32)
13 -
14 - zsp325()
15 - {
16 - /#
17 -
18 - SUBROUTINE FINISH();
19 - SUBROUTINE FFT16COL(zr325ref In_Data, zr325ref Out_Data);
20 - SUBROUTINE FFT16ROW(zr325ref In_Data, zr325ref Out_Data);
21 -
22 - .EXTERN _SubEntry_FINISH
23 - .EXTERN _SubEntry_FFT16COL
24 - .EXTERN _SubEntry_FFT16ROW
25 -
26 -
27 - STACKS::
28 -       .DATA { 0 };
29 - cstack0::
30 -       .DATA { 0 };
31 -       .DATA { &_SubEntry_FFT16COL };
32 -       .DATA { &_SubEntry_FINISH };
33 -       .DATA { COLUMN(0) };
34 -       .DATA { COLUMN(0) };
35 - cstack1::
36 -       .DATA { 0 };
37 -       .DATA { &_SubEntry_FFT16COL };
38 -       .DATA { &_SubEntry_FINISH };
39 -       .DATA { COLUMN(4) };
40 -       .DATA { COLUMN(4) };
41 - cstack2::
42 -       .DATA { 0 };
43 -       .DATA { &_SubEntry_FFT16COL };
44 -       .DATA { &_SubEntry_FINISH };
45 -       .DATA { COLUMN(8) };
46 -       .DATA { COLUMN(8) };
47 - cstack3::
48 -       .DATA { 0 };
49 -       .DATA { &_SubEntry_FFT16COL };
50 -       .DATA { &_SubEntry_FINISH };
```

```
51 -        .DATA { COLUMN(12) };
52 -        .DATA { COLUMN(12) };
53 - rstack0::
54 -        .DATA { 0 };
55 -        .DATA { &_SubEntry_FFT16ROW };
56 -        .DATA { &_SubEntry_FINISH };
57 -        .DATA { ROW(0) };
58 -        .DATA { ROW(0) };
59 - rstack1::
60 -        .DATA { 0 };
61 -        .DATA { &_SubEntry_FFT16ROW };
62 -        .DATA { &_SubEntry_FINISH };
63 -        .DATA { ROW(4) };
64 -        .DATA { ROW(4) };
65 - rstack2::
66 -        .DATA { 0 };
67 -        .DATA { &_SubEntry_FFT16ROW };
68 -        .DATA { &_SubEntry_FINISH };
69 -        .DATA { ROW(8) };
70 -        .DATA { ROW(8) };
71 - rstack3::
72 -        .DATA { 0 };
73 -        .DATA { &_SubEntry_FFT16ROW };
74 -        .DATA { &_SubEntry_FINISH };
75 -        .DATA { ROW(12) };
76 -        .DATA { ROW(12) };
77 - #/
78 - }
```

```
 1 - /*    Initialization for synthetic stack frames for FFT2D32
 2 -       rstackN and cstackN are starting $SP values for chip N.
 3 -       One free spot left in stacks for interrupt.
 4 - */
 5 -
 6 - /*    absolute base addresses from memory map */
 7 - #define PRAM 0x00000
 8 - #define FOUR_PORT 0x20000
 9 - #define STATUS_LATCH 0x40000
10 -
11 - #define COLUMN(N) (FOUR_PORT + (N) * 2)
12 - #define ROW(N) (FOUR_PORT + (N) * 64)
13 -
14 - zsp325()
15 - {
16 - /#
17 -
18 - SUBROUTINE FINISH();
19 - SUBROUTINE FFT32COL(zr325ref In_Data, zr325ref Out_Data);
20 - SUBROUTINE FFT32ROW(zr325ref In_Data, zr325ref Out_Data);
21 -
22 - .EXTERN _SubEntry_FINISH
23 - .EXTERN _SubEntry_FFT32COL
24 - .EXTERN _SubEntry_FFT32ROW
25 -
26 -
27 - STACKS::
28 -       .DATA { 0 };
29 - cstack0::
30 -       .DATA { 0 };
31 -       .DATA { &_SubEntry_FFT32COL };
32 -       .DATA { &_SubEntry_FINISH };
33 -       .DATA { COLUMN(0) };
34 -       .DATA { COLUMN(0) };
35 - cstack1::
36 -       .DATA { 0 };
37 -       .DATA { &_SubEntry_FFT32COL };
38 -       .DATA { &_SubEntry_FINISH };
39 -       .DATA { COLUMN(8) };
40 -       .DATA { COLUMN(8) };
41 - cstack2::
42 -       .DATA { 0 };
43 -       .DATA { &_SubEntry_FFT32COL };
44 -       .DATA { &_SubEntry_FINISH };
45 -       .DATA { COLUMN(16) };
46 -       .DATA { COLUMN(16) };
47 - cstack3::
48 -       .DATA { 0 };
49 -       .DATA { &_SubEntry_FFT32COL };
50 -       .DATA { &_SubEntry_FINISH };
```

```
51 -        .DATA { COLUMN(24) };
52 -        .DATA { COLUMN(24) };
53 - rstack0::
54 -        .DATA { 0 };
55 -        .DATA { &_SubEntry_FFT32ROW };
56 -        .DATA { &_SubEntry_FINISH };
57 -        .DATA { ROW(0) };
58 -        .DATA { ROW(0) };
59 - rstack1::
60 -        .DATA { 0 };
61 -        .DATA { &_SubEntry_FFT32ROW };
62 -        .DATA { &_SubEntry_FINISH };
63 -        .DATA { ROW(8) };
64 -        .DATA { ROW(8) };
65 - rstack2::
66 -        .DATA { 0 };
67 -        .DATA { &_SubEntry_FFT32ROW };
68 -        .DATA { &_SubEntry_FINISH };
69 -        .DATA { ROW(16) };
70 -        .DATA { ROW(16) };
71 - rstack3::
72 -        .DATA { 0 };
73 -        .DATA { &_SubEntry_FFT32ROW };
74 -        .DATA { &_SubEntry_FINISH };
75 -        .DATA { ROW(24) };
76 -        .DATA { ROW(24) };
77 - #/
78 - }
```

```
 1 - /*    Initialization for synthetic stack frames for FFT1K
 2 -       rstackN and cstackN are starting $SP values for chip N.
 3 -       One free spot left in stacks for interrupt.
 4 - */
 5 -
 6 - /*    absolute base addresses from memory map */
 7 - #define PRAM 0x00000
 8 - #define FOUR_PORT 0x20000
 9 - #define STATUS_LATCH 0x40000
10 -
11 - #define COLUMN(N) (FOUR_PORT + (N) * 2)
12 - #define ROW(N) (FOUR_PORT + (N) * 64)
13 - #define OUT_OFFSET 0x800
14 -
15 - zap325()
16 - {
17 - /#
18 -
19 - SUBROUTINE FINISH();
20 - SUBROUTINE FFT32COL(zr325ref In_Data, zr325ref Out_Data);
21 - SUBROUTINE FFT1K0(zr325ref In_Data, zr325ref Out_Data);
22 - SUBROUTINE FFT1K1(zr325ref In_Data, zr325ref Out_Data);
23 - SUBROUTINE FFT1K2(zr325ref In_Data, zr325ref Out_Data);
24 - SUBROUTINE FFT1K3(zr325ref In_Data, zr325ref Out_Data);
25 -
26 - .EXTERN _SubEntry_FINISH
27 - .EXTERN _SubEntry_FFT32COL
28 - .EXTERN _SubEntry_FFT1K0
29 - .EXTERN _SubEntry_FFT1K1
30 - .EXTERN _SubEntry_FFT1K2
31 - .EXTERN _SubEntry_FFT1K3
32 -
33 - STACKS::
34 -       .DATA { 0 };
35 - cstack0::
36 -       .DATA { 0 };
37 -       .DATA { &_SubEntry_FFT32COL };
38 -       .DATA { &_SubEntry_FINISH };
39 -       .DATA { COLUMN(0) };
40 -       .DATA { COLUMN(0) };
41 - cstack1::
42 -       .DATA { 0 };
43 -       .DATA { &_SubEntry_FFT32COL };
44 -       .DATA { &_SubEntry_FINISH };
45 -       .DATA { COLUMN(8) };
46 -       .DATA { COLUMN(8) };
47 - cstack2::
48 -       .DATA { 0 };
49 -       .DATA { &_SubEntry_FFT32COL };
50 -       .DATA { &_SubEntry_FINISH };
```

```
51 -        .DATA { COLUMN(16) };
52 -        .DATA { COLUMN(16) };
53 - cstack3::
54 -         DATA { 0 };
55 -        .DATA { &_SubEntry_FFT32COL };
56 -        .DATA { &_SubEntry_FINISH };
57 -        .DATA { COLUMN(24) };
58 -        .DATA { COLUMN(24) };
59 - rstack0::
60 -        .DATA { 0 };
61 -        .DATA { &_SubEntry_FFT1K0 };
62 -        .DATA { &_SubEntry_FINISH };
63 -        .DATA { (COLUMN(0) + OUT_OFFSET) };
64 -        .DATA { ROW(0) };
65 - rstack1::
66 -        .DATA { 0 };
67 -        .DATA { &_SubEntry_FFT1K1 };
68 -        .DATA { &_SubEntry_FINISH };
69 -        .DATA { (COLUMN(8) + OUT_OFFSET) };
70 -        .DATA { ROW(8) };
71 - rstack2::
72 -        .DATA { 0 };
73 -        .DATA { &_SubEntry_FFT1K2 };
74 -        .DATA { &_SubEntry_FINISH };
75 -        .DATA { (COLUMN(16) + OUT_OFFSET) };
76 -        .DATA { ROW(16) };
77 - rstack3::
78 -        .DATA { 0 };
79 -        .DATA { &_SubEntry_FFT1K3 };
80 -        .DATA { &_SubEntry_FINISH };
81 -        .DATA { (COLUMN(24) + OUT_OFFSET) };
82 -        .DATA { ROW(24) };
83 - #/
84 - }
```

```
 1 - /*    Initialization for synthetic stack frames for RCONV.
 2 -       stack is starting $SP value.
 3 - */
 4 -
 5 - /*    absolute base addresses from memory map */
 6 - #define PRAM 0x00000
 7 - #define FOUR_PORT 0x20000
 8 - #define STATUS_LATCH 0x40000
 9 -
10 - #define COEF_LEN 8
11 - #define OUT_LEN 25
12 -
13 - zsp325()
14 - {
15 - /#
16 -
17 - SUBROUTINE FINISH();
18 - SUBROUTINE RCONV(     zr325int Coef_Length,
19 -                                zr325int Out_Length,
20 -                                zr325ref Coefficients,
21 -                                zr325ref In_Data,
22 -                                zr325ref Out_Data);
23 -
24 - .EXTERN _SubEntry_FINISH
25 - .EXTERN _SubEntry_RCONV
26 -
27 -
28 - /*    define stack with parameters in reverse order */
29 - STACK8::
30 -       .DATA { 0 };
31 - stack::
32 -       .DATA { 0 };
33 -       .DATA { &_SubEntry_RCONV };
34 -       .DATA { &_SubEntry_FINISH };
35 -       .DATA { (FOUR_PORT + 0x100) };
36 -       .DATA { (FOUR_PORT + COEF_LEN) };
37 -       .DATA { FOUR_PORT };
38 -       .DATA { OUT_LEN };
39 -       .DATA { COEF_LEN };
40 - #/
41 - }
```

```
 1 - /*    Initialization for synthetic stack frame for CCONV.
 2 -       stack is starting $SP value.
 3 - */
 4 -
 5 - /*    absolute base addresses from memory map */
 6 - #define PRAM 0x00000
 7 - #define FOUR_PORT 0x20000
 8 - #define STATUS_LATCH 0x40000
 9 -
10 - #define COEF_LEN 4
11 - #define OUT_LEN 13
12 -
13 - zsp325()
14 - {
15 - /#
16 -
17 - SUBROUTINE FINISH();
18 - SUBROUTINE CCONV(     zr325int Coef_Length,
19 -                                  zr325int Out_Length,
20 -                                  zr325ref Coefficients,
21 -                                  zr325ref In_Data,
22 -                                  zr325ref Out_Data);
23 -
24 - .EXTERN _SubEntry_FINISH
25 - .EXTERN _SubEntry_CCONV
26 -
27 -
28 - /*    define stack with parameters in reverse order */
29 - STACK$::
30 -       .DATA { 0 };
31 - stack::
32 -       .DATA { 0 };
33 -       .DATA { &_SubEntry_CCONV };
34 -       .DATA { &_SubEntry_FINISH };
35 -       .DATA { (FOUR_PORT + 0x100) };
36 -       .DATA { (FOUR_PORT + COEF_LEN*2) };
37 -       .DATA { FOUR_PORT };
38 -       .DATA { OUT_LEN };
39 -       .DATA { COEF_LEN };
40 - #/
41 - }
```

```
 1 - /*    Initialization for synthetic stack frames for RCORR.
 2 -       stack is starting $SP value.
 3 - */
 4 -
 5 - /*    absolute base addresses from memory map */
 6 - #define PRAM 0x00000
 7 - #define FOUR_PORT 0x20000
 8 - #define STATUS_LATCH 0x40000
 9 -
10 - #define COEF_LEN 8
11 - #define OUT_LEN 25
12 -
13 - zsp325()
14 - {
15 - /#
16 -
17 - SUBROUTINE FINISH();
18 - SUBROUTINE RCORR(     zr325int Coef_Length,
19 -                                zr325int Out_Length,
20 -                                zr325ref Coefficients,
21 -                                zr325ref In_Data,
22 -                                zr325ref Out_Data);
23 -
24 - .EXTERN _SubEntry_FINISH
25 - .EXTERN _SubEntry_RCORR
26 -
27 -
28 - /*    define stack with parameters in reverse order */
29 - STACK5::
30 -       .DATA { 0 };
31 - stack::
32 -       .DATA { 0 };
33 -       .DATA { &_SubEntry_RCORR };
34 -       .DATA { &_SubEntry_FINISH };
35 -       .DATA { (FOUR_PORT + 0x100) };
36 -       .DATA { (FOUR_PORT + COEF_LEN) };
37 -       .DATA { FOUR_PORT };
38 -       .DATA { OUT_LEN };
39 -       .DATA { COEF_LEN };
40 - #/
41 - }
```

```
 1 - /*    Initialization for synthetic stack frame for CCORR.
 2 -       stack is starting $SP value.
 3 - */
 4 -
 5 - /*    absolute base addresses from memory map */
 6 - #define PRAM 0x00000
 7 - #define FOUR_PORT 0x20000
 8 - #define STATUS_LATCH 0x40000
 9 -
10 - #define COEF_LEN 4
11 - #define OUT_LEN 13
12 -
13 - zsp325()
14 - {
15 - /#
16 -
17 - SUBROUTINE FINISH();
18 - SUBROUTINE CCORR(     sr325int Coef_Length,
19 -                                sr325int Out_Length,
20 -                                sr325ref Coefficients,
21 -                                sr325ref In_Data,
22 -                                sr325ref Out_Data);
23 -
24 - .EXTERN _SubEntry_FINISH
25 - .EXTERN _SubEntry_CCORR
26 -
27 -
28 - /*    define stack with parameters in reverse order */
29 - STACKS::
30 -       .DATA { 0 };
31 - stack::
32 -       .DATA { 0 };
33 -       .DATA { &_SubEntry_CCORR };
34 -       .DATA { &_SubEntry_FINISH };
35 -       .DATA { (FOUR_PORT + 0x100) };
36 -       .DATA { (FOUR_PORT + COEF_LEN*2) };
37 -       .DATA { FOUR_PORT };
38 -       .DATA { OUT_LEN };
39 -       .DATA { COEF_LEN };
40 - #/
41 - }
```

```
 1 - /*    Initialization for synthetic stack frame for POL2RECT.
 2 -       stack is starting $SP value.
 3 - */
 4 -
 5 - /*    absolute base addresses from memory map */
 6 - #define PRAM 0x00000
 7 - #define FOUR_PORT 0x20000
 8 - #define STATUS_LATCH 0x40000
 9 -
10 - #define LENGTH 200
11 -
12 - zsp325()
13 - {
14 - /#
15 -
16 - SUBROUTINE POL2RECT(zr325int Length, zr325ref In_Data, zr325ref Out_Data);
17 - SUBROUTINE FINISH();
18 -
19 - .EXTERN _SubEntry_POL2RECT
20 - .EXTERN _SubEntry_FINISH
21 -
22 -
23 - /*    define stack with parameters in reverse order */
24 - STACK8::
25 -       .DATA { 0 };
26 - stack::
27 -       .DATA { 0 };
28 -       .DATA { &_SubEntry_POL2RECT };
29 -       .DATA { &_SubEntry_FINISH };
30 -       .DATA { FOUR_PORT };
31 -       .DATA { FOUR_PORT };
32 -       .DATA { LENGTH };
33 - #/
34 - }
35 -
```

```
 1 - /*    Initialization for synthetic stack frame for POL2RECT.
 2 -       stack is starting $SP value.
 3 - */
 4 -
 5 - /*    absolute base addresses from memory map */
 6 - #define PRAM 0x00000
 7 - #define FOUR_PORT 0x20000
 8 - #define STATUS_LATCH 0x40000
 9 -
10 - #define LENGTH 200
11 -
12 - zsp325()
13 - {
14 - /*
15 -
16 - SUBROUTINE RECT2POL(zr325int Length, zr325ref In_Data, zr325ref Out_Data);
17 - SUBROUTINE FINISH();
18 -
19 - .EXTERN _SubEntry_RECT2POL
20 - .EXTERN _SubEntry_FINISH
21 -
22 -
23 - /*    define stack with parameters in reverse order */
24 - STACK9::
25 -       .DATA ( 0 );
26 - stack::
27 -       .DATA ( 0 );
28 -       .DATA ( &_SubEntry_RECT2POL );
29 -       .DATA ( &_SubEntry_FINISH );
30 -       .DATA ( FOUR_PORT );
31 -       .DATA ( FOUR_PORT );
32 -       .DATA ( LENGTH );
33 - */
34 - }
```

```
 1 - /*    Initialization for synthetic stack frames for FFT1K benchmark
 2 -       stackN is starting $SP value for chip N.
 3 -       Execution sequence is to start at STARTUP, return to do columns,
 4 -       return to synchronize for second wave, pop parameters, return
 5 -       to do rows, then return to finish.
 6 -       One free spot left in stacks for interrupt.
 7 - */
 8 -
 9 - /*    absolute base addresses from memory map */
10 - #define PRAM 0x00000
11 - #define FOUR_PORT 0x20000
12 - #define STATUS_LATCH 0x40000
13 -
14 - #define COLUMN(N) (FOUR_PORT + (N) * 2)
15 - #define ROW(N) (FOUR_PORT + (N) * 64)
16 - #define OUT_OFFSET 0x800
17 -
18 - zsp325()
19 - {
20 - /*
21 -
22 - SUBROUTINE FINISH();
23 - SUBROUTINE SYNCHRONIZE();
24 - SUBROUTINE FFT32COL(zr325ref In_Data, zr325ref Out_Data);
25 - SUBROUTINE FFT1K0(zr325ref In_Data, zr325ref Out_Data);
26 - SUBROUTINE FFT1K1(zr325ref In_Data, zr325ref Out_Data);
27 - SUBROUTINE FFT1K2(zr325ref In_Data, zr325ref Out_Data);
28 - SUBROUTINE FFT1K3(zr325ref In_Data, zr325ref Out_Data);
29 -
30 - .EXTERN _SubEntry_FINISH
31 - .EXTERN _SubEntry_SYNCHRONIZE
32 - .EXTERN _SubEntry_FFT32COL
33 - .EXTERN _SubEntry_FFT1K0
34 - .EXTERN _SubEntry_FFT1K1
35 - .EXTERN _SubEntry_FFT1K2
36 - .EXTERN _SubEntry_FFT1K3
37 -
38 - STACKS::
39 -       .DATA { 0 };
40 - stack0::
41 -       .DATA { 0 };
42 -       .DATA { &_SubEntry_FFT32COL };
43 -       .DATA { &_SubEntry_SYNCHRONIZE };
44 -       .DATA { COLUMN(0) };
45 -       .DATA { COLUMN(0) };
46 -       .DATA { &_SubEntry_FFT1K0 };
47 -       .DATA { &_SubEntry_FINISH };
48 -       .DATA { (COLUMN(0) + OUT_OFFSET) };
49 -       .DATA { ROW(0) };
50 - stack1::
```

```
51 -        .DATA { 0 };
52 -        .DATA { &_SubEntry_FFT32COL };
53 -        .DATA { &_SubEntry_SYNCHRONIZE };
54 -        .DATA { COLUMN(8) };
55 -        .DATA { COLUMN(8) };
56 -        .DATA { &_SubEntry_FFT1K1 };
57 -        .DATA { &_SubEntry_FINISH };
58 -        .DATA { (COLUMN(8) + OUT_OFFSET) };
59 -        .DATA { ROW(8) };
60 - stack2::
61 -        .DATA { 0 };
62 -        .DATA { &_SubEntry_FFT32COL };
63 -        .DATA { &_SubEntry_SYNCHRONIZE };
64 -        .DATA { COLUMN(16) };
65 -        .DATA { COLUMN(16) };
66 -        .DATA { &_SubEntry_FFT1K2 };
67 -        .DATA { &_SubEntry_FINISH };
68 -        .DATA { (COLUMN(16) + OUT_OFFSET) };
69 -        .DATA { ROW(16) };
70 - stack3::
71 -        .DATA { 0 };
72 -        .DATA { &_SubEntry_FFT32COL };
73 -        .DATA { &_SubEntry_SYNCHRONIZE };
74 -        .DATA { COLUMN(24) };
75 -        .DATA { COLUMN(24) };
76 -        .DATA { &_SubEntry_FFT1K3 };
77 -        .DATA { &_SubEntry_FINISH };
78 -        .DATA { (COLUMN(24) + OUT_OFFSET) };
79 -        .DATA { ROW(24) };
80 - #/
81 - }
```

```
 1 - /*    Code to start all VSP chips simultaneously.  The start address of the
 2 -       code to be executed at the signal should be the first value on the
 3 -       stack.  Placed at absolute location 0 to simplify startup.
 4 - */
 5 -
 6 - /*    absolute base addresses from memory map */
 7 - #define PRAM 0x00000
 8 - #define FOUR_PORT 0x20000
 9 - #define STATUS_LATCH 0x40000
10 -
11 - /*    status bit value to indicate start */
12 - #define START 2
13 -
14 - ssp325()
15 - {
16 - /#
17 - .ORG 0
18 - SUBROUTINE STARTUP()
19 - {
20 -       /*      reset status bits */
21 -       LDR #0 => $X;
22 -       STR $X => STATUS_LATCH;
23 -
24 -       /*      get mask for start bit */
25 -       LDR #START => $X;
26 -
27 - Poll::
28 -       ANDR:[TR] STATUS_LATCH, $X;
29 -       LOOP [ZR], #1;
30 -
31 - }
32 - #/
33 - }
```

```
 1 - /*    Test program to make Zoran status bits follow 68020 bits */
 2 -
 3 - /*    absolute base addresses from memory map */
 4 - #define FRAM 0x00000
 5 - #define FOUR_PORT 0x20000
 6 - #define STATUS_LATCH 0x40000
 7 -
 8 - zsp325()
 9 - {
10 -
11 - /#
12 - SUBROUTINE MAIN()
13 - {
14 - Top::
15 -       LDR STATUS_LATCH => $LC;
16 -       STR $LC => STATUS_LATCH;
17 - Loop::
18 -       XORR:[TR] STATUS_LATCH, $LC => $X;
19 -       ANDR #3, $X;
20 -       JMPC [ER], Loop;
21 -
22 -       JMP Top;
23 - }
24 - #/
25 - }
```

```
 1 - /*    Code to synchronize VSPs between waves of FFT.  Also needs to pop the
 2 -       parameters of the first wave before returning.
 3 - */
 4 -
 5 - /*    absolute base addresses from memory map */
 6 - #define PRAM 0x00000
 7 - #define FOUR_PORT 0x20000
 8 - #define STATUS_LATCH 0x40000
 9 -
10 - /*    status bit value to indicate wave sync */
11 - #define WAVE 1
12 -
13 - /*    define number of parameters we supposedly sent */
14 - #define NUM_PARAM 2
15 -
16 - zsp325()
17 - {
18 - /#
19 - .STACKACCESS
20 -
21 - SUBROUTINE SYNCHRONIZE()
22 - {
23 -       /*      set status bit */
24 -       LDR #WAVE => $X;
25 -       STR $X => STATUS_LATCH;
26 -
27 -       /*      get rid of synthetic parameters */
28 -       ADDR #NUM_PARAM, $SP;
29 -
30 -
31 -       /*      wait for sync response */
32 - Poll::
33 -       ANDR:[TR] STATUS_LATCH, $X;
34 -       LOOP [ZR], #1;
35 -
36 - }
37 - #/
38 - }
```

```
 1 - /*    Test program to see if Zorans work */
 2 -
 3 - /*    absolute base addresses from memory map */
 4 - #define PRAM 0x00000
 5 - #define FOUR_PORT 0x20000
 6 - #define STATUS_LATCH 0x40000
 7 -
 8 - zsp325()
 9 - {
10 -       int i;
11 -       float x;
12 -
13 -       /*      put a vector of (1.0, x) at PRAM + 0x400 */
14 - /#
15 -       .ORG    (PRAM + 0x400)
16 - #/
17 -       for (i = 0, x = 0.0; i < 16; i++, x += 1.0)
18 -       {
19 -       /#
20 -               .DATA   { 1.0, IEEE_Float(x) };
21 -       #/
22 -       }
23 -
24 -       /*      put a vector of (x, 1.0) at FOUR_PORT */
25 - /#
26 -       .ORG    FOUR_PORT
27 - #/
28 -       for (i = 0, x = 0.0; i < 16; i++, x += 1.0)
29 -       {
30 -       /#
31 -               .DATA   { IEEE_Float(x), 1.0 };
32 -       #/
33 -       }
34 -
35 - /#
36 -       .ORG
37 - SUBROUTINE MAIN()
38 - {
39 -       /*      write 0 to status latch */
40 -       LDR #0 => $X;
41 -       STR $X => STATUS_LATCH;
42 -
43 -       /*      add two complex vectors and store */
44 -       LD_C:(16) (PRAM + 0x400) => $C0;
45 -       ADD_C:(16) FOUR_PORT, $C0 => $C0;
46 -       ST_C:(16) $C0 => FOUR_PORT;
47 -
48 -       /*      make sure we are finished, then write 1s to status latch */
49 -       SYNC:[CU,EU,MU];
50 -       LDR #3 => $X;
```

```
51 -        STR $X => STATUS_LATCH;
52 - }
53 - #/
54 - }
```

```
 1 - /*    Test program for Zoran interrupts.  Main routine is an infinite loop
 2 -       that decrements $LC (starting at 0 and wrapping around in 16 bits).
 3 -       Interrupt routine sets status and halts.  After restart, it clears
 4 -       status again and returns to infinite loop.
 5 - */
 6 -
 7 - /*    absolute base addresses from memory map */
 8 - #define PRAM 0x00000
 9 - #define FOUR_PORT 0x20000
10 - #define STATUS_LATCH 0x40000
11 -
12 - zsp325()
13 - {
14 - /#
15 -
16 - INTERRUPT SUBROUTINE SET_HALT()
17 - {
18 -
19 -       /*      write 1s to status latch and wait for !IEI */
20 -       LDR #3 => $X;
21 -       STR $X => STATUS_LATCH;
22 - InfLoop::
23 -       ANDR:[TR] $IP, #0x001000;
24 -       JMPC [!ZR], InfLoop;
25 -
26 -       /*      after resume, clear status bits */
27 -       LDR #0 => $X;
28 -       STR $X => STATUS_LATCH;
29 - }
30 -
31 - .EXTERN _SubEntry_SET_HALT;
32 -
33 - SUBROUTINE MAIN()
34 - {
35 -       /*      set interrupt vector (happens to be 0, but why not) */
36 -       LDR &_SubEntry_SET_HALT => $IP;
37 -
38 -       /*      write 0 to status latch */
39 -       LDR #0 => $X;
40 -       STR $X => STATUS_LATCH;
41 -
42 -       /*      infinite loop decrementing $LC from 0 */
43 -       MOVR $X => $LC;
44 - Loop::
45 -       JMP:[DL] Loop;
46 -
47 - }
48 -
49 - #/
50 - }
```

APPENDIX C

PC INTERFACE PROGRAMS

```
/*****************************************************************************
* Module:Test Module --- Test I/O boards.
*
* Author:John Stevens
*
* Copyright:Copyright 1988-1992 by Space Tech Corporation,
*Ft Collins, Colorado, USA.  All Rights Reserved.
*
* Status:This program is the sole property of Space Tech Corporation
*and is covered under non-disclosure agreements.  This program
*is PROPRIETARY / CONFIDENTIAL / TRADE SECRET, and disclosure
*of the contents of this document shall constitute violation
*of signed agreements and will result in severe penalties.
*****************************************************************************/

/*****************************************************************************
*Modified by Steven Sharp to add file upload/download capability.
*****************************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<ctype.h>
#include<dos.h>
#include<string.h>
#include<time.h>

typedefunsignedintUINT;

/*Status register bit defines.*/
#definePARITY_ERR0x0004
#defineWR_FIFO_EMPTY0x0008
#defineWR_FIFO_FULL0x0010
#defineWR_FIFO_ALMOST_EMPTY0x0020
#defineWR_FIFO_ALMOST_FULL0x0040

#defineRD_FIFO_EMPTY0x0080
#defineRD_FIFO_FULL0x0100
#defineRD_FIFO_ALMOST_EMPTY0x0200
#defineRD_FIFO_ALMOST_FULL0x0400

/*File format flags*/
#define INVALID 0
#define REX_FORMAT 1
#define S_FORMAT 2

/*Maximum filename length; must be less than 255 */
#define NAME_SIZE 50

staticUINTBase = 0x340;
staticUINTPort;

/*---------------------------------------------------------------------------
| Routine:ShowStat() --- Show the value of the status register.
|
| Inputs:Base- Base I/O address of board to check.
---------------------------------------------------------------------------*/

static
voidShowStat(intBase)
{
registerinti;
registerintj;
autointbit;

staticchar*StatReg[] =
{
"Parity Error            - ",
"Write FIFO Empty        - ",
"Write FIFO Full         - ",
"Write FIFO Almost Empty - ",
"Write FIFO Almost Full  - ",
"Read FIFO Empty         - ",
"Read FIFO Full          - ",
"Read FIFO Almost Empty  - ",
"Read FIFO Almost Full   - "
};

/*Print status header.*/
gotoxy(1, 1);
textcolor( LIGHTBLUE );
textbackground( LIGHTGRAY );
cprintf(" I/O Board Status \r\n");
textcolor( LIGHTGRAY );
textbackground( BLACK );

/*Read the status port, display values.*/
j = inport(Base + 2);

/*Print values.*/
for (i = 8, bit = RD_FIFO_ALMOST_FULL; i >= 0; i--)
{
/*Print status values.*/
cprintf( StatReg[i] );
if (j & bit)
{
textcolor( LIGHTGRAY );
textbackground( BLACK );
cprintf("False\r\n");
}
else
{
textcolor( BLACK );
textbackground( LIGHTGRAY );
cprintf("True \r\n");
}
bit >>= 1;

/*Reset to normal colors.*/
```

```
110 - textcolor( LIGHTGRAY );
111 - textbackground( BLACK );
112 - }
113 -
114 - /*Show value of status bits in status register.*/
115 - cprintf("Status Bit Values       - 0x%x\r\n", j & 0x3);
116 - }
117 -
118 - /*-------------------------------------------------------------------------
119 - | Routine:GetHexNo() --- Get a hexadecimal number.
120 - |
121 - | Return:Returns an integer number.
122 - -------------------------------------------------------------------------*/
123 -
124 - static
125 - unsigned longGetHexNo(intlimit)
126 - {
127 - registerinti;
128 - registerintc;
129 - auto unsigned longret;
130 - autocharBf[9];
131 -
132 - /*Get characters from the keyboard.*/
133 - for (i = 0; ; )
134 - {
135 - /*Get hex characters.*/
136 - do
137 - {
138 - c = getch();
139 - c = tolower( c );
140 - if (! (c >= 'a' && c <= 'f') &&
141 - ! (c >= '0' && c <= '9') &&
142 - c != '\b' && c != '\r')
143 - putch( '\x07' );
144 - } while (! (c >= 'a' && c <= 'f') &&
145 -  ! (c >= '0' && c <= '9') &&
146 -  c != '\b' && c != '\r');
147 -
148 - /*Break out on carriage return.*/
149 - if (c == '\r')
150 - break;
151 -
152 - /*Character.*/
153 - switch ( c )
154 - {
155 - case '\b':
156 - if (i > 0)
157 - {
158 - Bf[--i] = '\0';
159 - cputs( "\b \b" );
160 - }
161 - break;
162 - default:
163 - if (i < limit)
164 - {
165 - Bf[i++] = (char) c;
166 - Bf[i] = '\0';
167 - putch( c );
168 - }
169 - break;
170 - }
171 - }
172 -
173 - /*Return number.*/
174 - sscanf(Bf, "%lx", &ret);
175 - return( ret );
176 - }
177 -
178 - /*-------------------------------------------------------------------------
179 - | Routine:PrtVal() --- Print an integer value in hexadecimal, decimal
180 - |and binary.
181 - |
182 - | Inputs:Val- Value to print.
183 - -------------------------------------------------------------------------*/
184 -
185 - static
186 - voidPrtVal(UINTVal)
187 - {
188 - registerinti;
189 - registerUINTbit;
190 - autocharStr[20];
191 -
192 - Str[19] = '\0';
193 - for (i = 18, bit = 1; i >= 0; i--)
194 - if ((i + 1) % 5 == 0)
195 - Str[i] = ' ';
196 - else
197 - {
198 - if (Val & bit)
199 - Str[i] = '1';
200 - else
201 - Str[i] = '0';
202 - bit <<= 1;
203 - }
204 -
205 - cprintf("0x%04x  %5u  %s", Val, Val, Str);
206 - }
207 -
208 - /*-------------------------------------------------------------------------
209 - | Routine:InitDsp() --- Initialize the display.
210 - |
211 - | Inputs:Port- Current port value for reads and writes.
212 - -------------------------------------------------------------------------*/
213 -
214 - static
215 - voidInitDsp(UINTPort)
216 - {
217 - /*Print current base number.*/
218 - gotoxy(1, 14);
```

```
219 - clreol();
220 - textcolor( LIGHTBLUE );
221 - textbackground( LIGHTGRAY );
222 - cputs( "Base  Value" );
223 - textcolor( LIGHTGRAY );
224 - textbackground( BLACK );
225 - cputs(" : ");
226 - PrtVal( Base );
227 -
228 - /*Print current port number.*/
229 - gotoxy(1, 15);
230 - clreol();
231 - textcolor( LIGHTBLUE );
232 - textbackground( LIGHTGRAY );
233 - cputs( "Port  Value" );
234 - textcolor( LIGHTGRAY );
235 - textbackground( BLACK );
236 - cputs(" : ");
237 - PrtVal( Port );
238 - }
239 -
240 - /*-------------------------------------------------------------------------
241 - | Routine:CheckFile() --- Check file type and possibly count length
242 - |
243 - | Inputs:Filename- Filename for file to check
244 - |
245 - | Outputs:Size- Pointer to UINT to store length in words
246 - |
247 - | Returns:Type of file, HEX_FORMAT or S_FORMAT
248 - -------------------------------------------------------------------------*/
249 -
250 - static
251 - intCheckFile( char *Filename, UINT *Size )
252 - {
253 - FILE*InFile;
254 - intc;
255 - unsigned long nibbles;
256 -
257 - /*open file */
258 - InFile = fopen( Filename, "rt");
259 - if (!InFile)
260 - {
261 - return( INVALID );
262 - }
263 -
264 - /*find first non-white character */
265 - while ( isspace(c = getc(InFile)) )
266 - ;
267 -
268 - /*check for S format */
269 - if (c == 'S')
270 - {
271 - fclose(InFile);
272 - return( S_FORMAT );
273 - }
274 -
275 - /*otherwise hex format, count nibbles */
276 - nibbles = 0;
277 - while ( c != EOF )
278 - {
279 - if ( isxdigit(c) )
280 - nibbles++;
281 - c = getc(InFile);
282 - }
283 -
284 - /*compute size in words (rounding up) and return format */
285 - fclose(InFile);
286 - *Size = (nibbles + 3) >> 2;
287 - return( HEX_FORMAT );
288 - }
289 -
290 - /*-------------------------------------------------------------------------
291 - | Routine:Upload() --- Transfer memory from VPH to PC file
292 - |
293 - | Inputs:Start- Start address
294 - |Size- Number of words to transfer (should be even)
295 - |Filename- Name of destination file
296 - -------------------------------------------------------------------------*/
297 -
298 - static
299 - voidUpload( unsigned long Start, UINT Size, char *Filename )
300 - {
301 - FILE *Outfile;
302 - UINT Count;
303 - UINT Val;
304 -
305 - /*open output file */
306 - Outfile = fopen(Filename, "wt");
307 - if ( Outfile == NULL )
308 - {
309 - gotoxy(1, 23);
310 - clreol();
311 - cprintf("File open failed \x07");
312 - sleep(3);
313 - return;
314 - }
315 -
316 - /*send transfer command to VPH */
317 - outport(Port, 0x0021);
318 - /*send size in longwords */
319 - outport(Port, (Size >> 1));
320 - outport(Port, (int) Start);
321 - outport(Port, (int) (Start >> 16));
322 - outport(Port, 0x0000);
323 -
324 - /*upload file */
325 - for (Count = 0; Count < Size; Count++)
326 - {
327 - /*wait until read fifo not empty */
```

```
328 - while ( !( inport(Base + 2) & RD_FIFO_EMPTY) )
329 - ;
330 -
331 - /*get word and send to file */
332 - Val = inport(Port);
333 - fprintf(Outfile, "%04x", Val);
334 - if ( (Count & 0xf) == 0xf )
335 - putc('\n', Outfile);
336 - }
337 - putc('\n', Outfile);
338 - fclose(Outfile);
339 - }
340 -
341 - /*-------------------------------------------------------------------------
342 - | Routine:GetWord() --- reads hex words from input file
343 - |pads and differently than scanf for odd # of bytes
344 - |
345 - | Inputs: Infile- Input file pointer
346 - |
347 - | Returns:integer value read
348 - -------------------------------------------------------------------------*/
349 -
350 - static
351 - UINTGetWord( FILE *Infile )
352 - {
353 - autoUINTVal;
354 - autointc;
355 - staticcharBuf[5] = "0000";
356 - autointnibble;
357 -
358 - /*read 4 hex digits */
359 - for (nibble = 0; nibble < 4; nibble++)
360 - {
361 - /*ignore embedded spaces and newlines */
362 - while ( isspace(c = getc(Infile)) )
363 - ;
364 -
365 - /*store hex digits, zero pad at EOF */
366 - if (isxdigit(c))
367 - {
368 - Buf[nibble] = c;
369 - }
370 - else if (c == EOF)
371 - {
372 - Buf[nibble] = '0';
373 - }
374 - else
375 - {
376 - gotoxy(1, 23);
377 - clreol();
378 - cprintf("Unexpected character %c in file \x07", c);
379 - sleep(2);
380 - Buf[nibble] = '0';
381 - }
382 - }
383 -
384 - /*convert to integer */
385 - sscanf(Buf, "%x", &Val);
386 - return(Val);
387 - }
388 -
389 - /*-------------------------------------------------------------------------
390 - | Routine:SendBlock() --- Transfer block from PC file to VPH memory
391 - |
392 - | Inputs: Infile- Input file pointer
393 - |Start- Start address
394 - |Size- Number of words to transfer, will pad to even
395 - -------------------------------------------------------------------------*/
396 -
397 - static
398 - voidSendBlock( FILE *Infile, unsigned long Start, UINT Size )
399 - {
400 - UINTVal;
401 - UINTCount;
402 -
403 - /*send transfer command to VPH */
404 - outport(Port, 0x0020);
405 - /*send size in longwords, round up */
406 - outport(Port, (Size + 1) >> 1);
407 - outport(Port, (int) Start);
408 - outport(Port, (int) (Start >> 16));
409 - outport(Port, 0x0000);
410 -
411 - /*handle expected number of words */
412 - for (Count = 0; Count < Size; Count++)
413 - {
414 - /*read a word from file */
415 - Val = GetWord(Infile);
416 -
417 - /*wait until write fifo not full */
418 - while ( !( inport(Base + 2) & WR_FIFO_FULL) )
419 - ;
420 -
421 - /*write word */
422 - outport(Port, Val);
423 - }
424 -
425 - /*if number of words is odd, pad with 0 */
426 - if (Size & 1)
427 - {
428 - /*wait until write fifo not full */
429 - while ( !( inport(Base + 2) & WR_FIFO_FULL) )
430 - ;
431 -
432 - /*write pad word */
433 - outport(Port, 0x0000);
434 - }
435 -
436 - }
```

```
437 -
438 - /*---------------------------------------------------------------------------
439 - | Routine:Download() --- Transfer PC file to VPH memory
440 -
441 - | Inputs: Filename- Name of source file
442 - |Format- Input file format
443 - |Start- Start address
444 - |Size- Number of words to transfer
445 - ---------------------------------------------------------------------------*/
446 -
447 - static
448 - void Download( char *Filename, int Format, unsigned long Start, UINT Size)
449 - {
450 - FILE*Infile;
451 - intDone;
452 - intc;
453 -
454 - /*open input file */
455 - Infile = fopen(Filename, "rt");
456 - if ( Infile == NULL )
457 - {
458 - gotoxy(1, 23);
459 - clreol();
460 - cprintf("File open failed \x07");
461 - sleep(3);
462 - return;
463 - }
464 -
465 - /*check format and handle each */
466 - if ( Format == HEX_FORMAT )
467 - {
468 - SendBlock( Infile, Start, Size );
469 - }
470 - else
471 - {
472 - /*S format - handle each record  */
473 - Done = 0;
474 - while (!Done)
475 - {
476 - /*find next 'S' and get record type */
477 - while ( (c = getc(Infile)) != 'S' && c != EOF)
478 - ;
479 - c = getc(Infile);
480 -
481 - /*handle each type */
482 - if (c == '3')
483 - {
484 - /*get record length and compute data words */
485 - fscanf(Infile, "%2x", &Size);
486 - Size = (Size >> 1) - 2;
487 -
488 - /*get start address */
489 - fscanf(Infile, "%8lx", &Start);
490 -
491 - /*transfer data field */
492 - SendBlock( Infile, Start, Size );
493 - }
494 - else if (c == '7')
495 - {
496 - Done = 1;
497 - }
498 - else if (c == EOF)
499 - {
500 - gotoxy(1, 23);
501 - clreol();
502 - cprintf("Missing end-of-file record \x07");
503 - sleep(3);
504 - Done = 1;
505 - }
506 - else
507 - {
508 - gotoxy(1, 23);
509 - clreol();
510 - cprintf("Skipping unexpected record type %c in file \x07", c);
511 - sleep(3);
512 - }
513 - }
514 - }
515 - fclose(Infile);
516 - }
517 -
518 - /*---------------------------------------------------------------------------
519 - | Routine:main() --- Entry point and main routine for program.
520 - ---------------------------------------------------------------------------*/
521 -
522 - voidmain(intargc,
523 -  char**argv)
524 - {
525 - registerUINTVal;
526 - registerintc;
527 - auto unsigned longStart;
528 - autoUINTSize;
529 - autochar Buffer[NAME_SIZE+2] = { NAME_SIZE };
530 - autochar*Filename;
531 - autointFormat;
532 -
533 - /*Get command line parameters.*/
534 - if (argc > 1)
535 - {
536 - /*Get I/O board base address.*/
537 - if (sscanf(argv[1], "%x", &Base) != 1)
538 - {
539 - fprintf(stderr, "Error, parameter one must be a "
540 - "hexadecimal I/O space address.\n");
541 - exit( 2 );
542 - }
543 - }
544 - Port = Base;
545 -
```

```
546 - /*Set up to read command.*/
547 - clrscr();
548 - gotoxy(1, 22);
549 - cprintf("Commands: (B)ase, (P)ort, (R)ead, (W)rite, (U)pload, (D)ownload, "
550 - "(Q)uit");
551 -
552 - /*Print current port number.*/
553 - InitDsp( Port );
554 -
555 - /*Initialize the screen.*/
556 - for ( ; ; )
557 - {
558 - /*Show the status.*/
559 - ShowStat( Base );
560 -
561 - /*Get command.*/
562 - gotoxy(1, 23);
563 - clreol();
564 - cprintf("Command? ");
565 - if ((c = getche()) == 'Q' || c == 'q')
566 - break;
567 -
568 - /*Execute command.*/
569 - switch ( tolower( c ) )
570 - {
571 - case 'b':
572 - /*Get base address number.*/
573 - gotoxy(1, 23);
574 - clreol();
575 - cprintf("Enter Base Port Address > ");
576 -
577 - /*Get hex number.*/
578 - Base = GetHexNo( 3 );
579 -
580 - /*Print current port number.*/
581 - gotoxy(1, 14);
582 - clreol();
583 - textcolor( LIGHTBLUE );
584 - textbackground( LIGHTGRAY );
585 - cputs( "Base  Value" );
586 - textcolor( LIGHTGRAY );
587 - textbackground( BLACK );
588 - cputs(" : ");
589 - PrtVal( Base );
590 - break;
591 - case 'p':
592 - /*Get port number.*/
593 - gotoxy(1, 23);
594 - clreol();
595 - cprintf("Enter Port Number > ");
596 -
597 - /*Get hex number.*/
598 - Port = GetHexNo( 3 );
599 -
600 - /*Print current port number.*/
601 - gotoxy(1, 15);
602 - clreol();
603 - textcolor( LIGHTBLUE );
604 - textbackground( LIGHTGRAY );
605 - cputs( "Port  Value" );
606 - textcolor( LIGHTGRAY );
607 - textbackground( BLACK );
608 - cputs(" : ");
609 - PrtVal( Port );
610 - break;
611 - case 'r':
612 - /*Read value from port.*/
613 - Val = inport( Port );
614 -
615 - /*Print current port number.*/
616 - gotoxy(1, 17);
617 - clreol();
618 - textcolor( LIGHTBLUE );
619 - textbackground( LIGHTGRAY );
620 - cputs( "Read  Value" );
621 - textcolor( LIGHTGRAY );
622 - textbackground( BLACK );
623 - cputs(" : ");
624 - PrtVal( Val );
625 - break;
626 - case 'w':
627 - /*Get port number.*/
628 - gotoxy(1, 23);
629 - clreol();
630 - cprintf("Enter Value to Write > ");
631 -
632 - /*Get hex number.*/
633 - Val = GetHexNo( 4 );
634 - outport(Port, Val);
635 -
636 - /*Print current port number.*/
637 - gotoxy(1, 16);
638 - clreol();
639 - textcolor( LIGHTBLUE );
640 - textbackground( LIGHTGRAY );
641 - cputs( "Write Value" );
642 - textcolor( LIGHTGRAY );
643 - textbackground( BLACK );
644 - cputs( " : " );
645 - PrtVal( Val );
646 - break;
647 - case 'u':
648 - /*Get start address.*/
649 - gotoxy(1, 23);
650 - clreol();
651 - cprintf("Enter Start Address > ");
652 -
653 - /*Get hex number.*/
654 - Start = GetHexNo( 8 );
```

```
655 -
656 - /*Get size in words.*/
657 - gotoxy(1, 23);
658 - clreol();
659 - cprintf("Enter Number of Longwords > ");
660 -
661 - /*Get hex number, convert to number of words*/
662 - Size = GetHexNo( 4 ) << 1;
663 -
664 - /*Get filename.*/
665 - gotoxy(1, 23);
666 - clreol();
667 - cprintf("Enter Destination Filename > ");
668 - Filename = cgets(Buffer);
669 -
670 - /*Perform upload operation */
671 - Upload( Start, Size, Filename);
672 - break;
673 - case 'd':
674 - /*Get filename.*/
675 - gotoxy(1, 23);
676 - clreol();
677 - cprintf("Enter Source Filename > ");
678 - Filename = cgets(Buffer);
679 -
680 - /*check format, count words if hex format */
681 - Format = CheckFile( Filename, &Size );
682 -
683 - /*if hex format, request start address */
684 - if (Format == HEX_FORMAT)
685 - {
686 - }/*Get start address.*/
687 - gotoxy(1, 23);
688 - clreol();
689 - cprintf("Enter Start Address > ");
690 - Start = GetHexNo( 8 );
691 - }
692 -
693 - /*Perform download operation */
694 - Download( Filename, Format, Start, Size );
695 - break;
696 - }
697 - }
698 -
699 - clrscr();
700 - }
```

```
/*****************************************************************************
* Module:Test Module --- Test I/O boards.
*
* Author:John Stevens
*
* Copyright:Copyright 1988-1992 by Space Tech Corporation,
*Ft Collins, Colorado, USA.  All Rights Reserved.
*
* Status:This program is the sole property of Space Tech Corporation
*and is covered under non-disclosure agreements.  This program
*is PROPRIETARY / CONFIDENTIAL / TRADE SECRET, and disclosure
*of the contents of this document shall constitute violation
*of signed agreements and will result in severe penalties.
*****************************************************************************/

#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
#include<ctype.h>
#include<dos.h>
#include<string.h>
#include<time.h>

typedefunsignedintUINT;

/*Status register bit defines.*/
#definePARITY_ERR0x0004
#defineWR_FIFO_EMPTY0x0008
#defineWR_FIFO_FULL0x0010
#defineWR_FIFO_ALMOST_EMPTY0x0020
#defineWR_FIFO_ALMOST_FULL0x0040

#defineRD_FIFO_EMPTY0x0080
#defineRD_FIFO_FULL0x0100
#defineRD_FIFO_ALMOST_EMPTY0x0200
#defineRD_FIFO_ALMOST_FULL0x0400

/*File format flags*/
#define INVALID 0
#define HEX_FORMAT 1
#define S_FORMAT 2

/*Maximum filename length; must be less than 255 */
#define NAME_SIZE 50

staticUINTBase = 0x340;
staticUINTPort;

/*---------------------------------------------------------------------------
 Routine:ShowStat() --- Show the value of the status register.

 Inputs:Base- Base I/O address of board to check.
---------------------------------------------------------------------------*/

static
voidShowStat(intBase)
{
registerinti;
registerintj;
autointbit;

staticchar*StatReg[] =
{
"Parity Error            - ",
"Write FIFO Empty        - ",
"Write FIFO Full         - ",
"Write FIFO Almost Empty - ",
"Write FIFO Almost Full  - ",
"Read FIFO Empty         - ",
"Read FIFO Full          - ",
"Read FIFO Almost Empty  - ",
"Read FIFO Almost Full   - "
};

/*Print status header.*/
gotoxy(1, 1);
textcolor( LIGHTBLUE );
textbackground( LIGHTGRAY );
cprintf(" I/O Board Status \r\n");
textcolor( LIGHTGRAY );
textbackground( BLACK );

/*Read the status port, display values.*/
j = inport(Base + 2);

/*Print values.*/
for (i = 8, bit = RD_FIFO_ALMOST_FULL; i >= 0; i--)
{
/*Print status values.*/
cprintf( StatReg[i] );
if (j & bit)
{
textcolor( LIGHTGRAY );
textbackground( BLACK );
cprintf("False\r\n");
}
else
{
textcolor( BLACK );
textbackground( LIGHTGRAY );
cprintf("True \r\n");
}
bit >>= 1;

/*Reset to normal colors.*/
textcolor( LIGHTGRAY );
textbackground( BLACK );
}
```

```
110 - /*Show value of status bits in status register.*/
111 - cprintf("Status Bit Values       - 0x%x\r\n", j & 0x3);
112 - }
113 -
114 - /*---------------------------------------------------------------------------
115 - | Routine:GetHexNo() --- Get a hexadecimal number.
116 - |
117 - | Return:Returns an integer number.
118 - ---------------------------------------------------------------------------*/
119 -
120 - static
121 - unsigned longGetHexNo(intlimit)
122 - {
123 - registerinti;
124 - registerintc;
125 - auto unsigned longret;
126 - autocharBf[9];
127 -
128 - /*Get characters from the keyboard.*/
129 - for (i = 0; ; )
130 - {
131 - /*Get hex characters.*/
132 - do
133 - {
134 - c = getch();
135 - c = tolower( c );
136 - if (! (c >= 'a' && c <= 'f') &&
137 - ! (c >= '0' && c <= '9') &&
138 - c != '\b' && c != '\r')
139 - putch( '\x07' );
140 - } while (! (c >= 'a' && c <= 'f') &&
141 -  ! (c >= '0' && c <= '9') &&
142 -  c != '\b' && c != '\r');
143 -
144 - /*Break out on carriage return.*/
145 - if (c == '\r')
146 - break;
147 -
148 - /*Character.*/
149 - switch ( c )
150 - {
151 - case '\b':
152 - if (i > 0)
153 - {
154 - Bf[--i] = '\0';
155 - cputs( "\b \b" );
156 - }
157 - break;
158 - default:
159 - if (i < limit)
160 - {
161 - Bf[i++] = (char) c;
162 - Bf[i] = '\0';
163 - putch( c );
164 - }
165 - break;
166 - }
167 - }
168 -
169 - /*Return number.*/
170 - sscanf(Bf, "%lx", &ret);
171 - return( ret );
172 - }
173 -
174 - /*---------------------------------------------------------------------------
175 - | Routine:PrtVal() --- Print an integer value in hexadecimal, decimal
176 - |and binary.
177 - |
178 - | Inputs:Val- Value to print.
179 - ---------------------------------------------------------------------------*/
180 -
181 - static
182 - voidPrtVal(UINTVal)
183 - {
184 - registerinti;
185 - registerUINTbit;
186 - autocharStr[20];
187 -
188 - Str[19] = '\0';
189 - for (i = 18, bit = 1; i >= 0; i--)
190 - if ((i + 1) % 5 == 0)
191 - Str[i] = ' ';
192 - else
193 - {
194 - if (Val & bit)
195 - Str[i] = '1';
196 - else
197 - Str[i] = '0';
198 - bit <<= 1;
199 - }
200 -
201 - cprintf("0x%04x  %5u  %s", Val, Val, Str);
202 - }
203 -
204 - /*---------------------------------------------------------------------------
205 - | Routine:InitDsp() --- Initialize the display.
206 - |
207 - | Inputs:Port- Current port value for reads and writes.
208 - ---------------------------------------------------------------------------*/
209 -
210 - static
211 - voidInitDsp(UINTPort)
212 - {
213 - /*Print current base number.*/
214 - gotoxy(1, 14);
215 - clreol();
216 - textcolor( LIGHTBLUE );
217 - textbackground( LIGHTGRAY );
218 - cputs( "Base  Value" );
```

```
219 - textcolor( LIGHTGRAY );
220 - textbackground( BLACK );
221 - cputs(" : ");
222 - PrtVal( Base );
223 -
224 - /*Print current port number.*/
225 - gotoxy(1, 15);
226 - clreol();
227 - textcolor( LIGHTBLUE );
228 - textbackground( LIGHTGRAY );
229 - cputs( "Port  Value" );
230 - textcolor( LIGHTGRAY );
231 - textbackground( BLACK );
232 - cputs(" : ");
233 - PrtVal( Port );
234 - }
235 -
236 - /*---------------------------------------------------------------------------
237 - | Routine:CheckFile() --- Check file type and possibly count length
238 - |
239 - | Inputs:Filename- Filename for file to check
240 - |
241 - | Outputs:Size- Pointer to UINT to store length in
242 - |
243 - | Returns:Type of file, HEX_FORMAT or S_FORMAT
244 - ---------------------------------------------------------------------------*/
245 -
246 - static
247 - intCheckFile( char *Filename, UINT *Size )
248 - {
249 - FILE*InFile;
250 - intc;
251 - unsigned long nibbles;
252 -
253 - /*open file */
254 - InFile = fopen( Filename, "rt");
255 - if (!InFile)
256 - {
257 - return( INVALID );
258 - }
259 -
260 - /*find first non-white character */
261 - while ( isspace(c = getc(InFile)) )
262 - ;
263 -
264 - /*check for S format */
265 - if (c == 'S')
266 - {
267 - fclose(InFile);
268 - return( S_FORMAT );
269 - }
270 -
271 - /*otherwise hex format, count nibbles */
272 - nibbles = 0;
273 - while ( c != EOF )
274 - {
275 - if ( isxdigit(c) )
276 - nibbles++;
277 - c = getc(InFile);
278 - }
279 -
280 - /*compute size in words (rounding up) and return format */
281 - fclose(InFile);
282 - *Size = (nibbles + 3) >> 2;
283 - return( HEX_FORMAT );
284 - }
285 -
286 - /*---------------------------------------------------------------------------
287 - | Routine:Upload() --- Transfer memory from VPH to PC file
288 - |
289 - | Inputs:Start- Start address
290 - | Size- Number of words to transfer
291 - | Filename- Name of destination file
292 - ---------------------------------------------------------------------------*/
293 -
294 - static
295 - voidUpload( unsigned long Start, UINT Size, char *Filename )
296 - {
297 - FILE *Outfile;
298 - UINT Count;
299 - UINT Val;
300 -
301 - /*open output file */
302 - Outfile = fopen(Filename, "wt");
303 - if ( Outfile == NULL )
304 - {
305 - gotoxy(1, 23);
306 - clreol();
307 - cprintf("File open failed \x07");
308 - sleep(3);
309 - return;
310 - }
311 -
312 - /*send transfer command to VPH */
313 - outport(Port, 0x0002);
314 - outport(Port, Size);
315 - outport(Port, (int) Start);
316 - outport(Port, (int) (Start >> 16));
317 - outport(Port, 0x0000);
318 -
319 - /*upload file */
320 - for (Count = 0; Count < Size; Count++)
321 - {
322 - /*wait until read fifo not empty */
323 - while ( !( inport(Base + 2) & RD_FIFO_EMPTY) )
324 - ;
325 -
326 - /*get word and send to file */
327 - Val = inport(Port);
```

```
328 - fprintf(Outfile, "%04x", Val);
329 - if ( (Count & 0xf) == 0xf )
330 - putc('\n', Outfile);
331 - }
332 - putc('\n', Outfile);
333 - fclose(Outfile);
334 - }
335 -
336 - /*---------------------------------------------------------------------------
337 - | Routine:GetWord() --- reads hex words from input file
338 - |pads end differently than scanf for odd # of bytes
339 - |
340 - | Inputs: Infile- Input file pointer
341 - |
342 - | Returns:integer value read
343 - ---------------------------------------------------------------------------*/
344 -
345 - static
346 - UINTGetWord( FILE *Infile )
347 - {
348 - autoUINTVal;
349 - autointc;
350 - staticcharBuf[5] = "0000";
351 - autointnibble;
352 -
353 - /*read 4 hex digits */
354 - for (nibble = 0; nibble < 4; nibble++)
355 - {
356 - /*ignore embedded spaces and newlines */
357 - while ( isspace(c = getc(Infile)) )
358 - ;
359 -
360 - /*store hex digits, zero pad at EOF */
361 - if (isxdigit(c))
362 - {
363 - Buf[nibble] = c;
364 - }
365 - else if (c == EOF)
366 - {
367 - Buf[nibble] = '0';
368 - }
369 - else
370 - {
371 - gotoxy(1, 23);
372 - clreol();
373 - cprintf("Unexpected character %c in file \x07", c);
374 - sleep(2);
375 - Buf[nibble] = '0';
376 - }
377 - }
378 -
379 - /*convert to integer */
380 - sscanf(Buf, "%x", &Val);
381 - return(Val);
382 - }
383 -
384 - /*---------------------------------------------------------------------------
385 - | Routine:SendBlock() --- Transfer block from PC file to VPH memory
386 - |
387 - | Inputs: Infile- Input file pointer
388 - |Start- Start address
389 - |Size- Number of words to transfer
390 - ---------------------------------------------------------------------------*/
391 -
392 - static
393 - voidSendBlock( FILE *Infile, unsigned long Start, UINT Size )
394 - {
395 - UINTVal;
396 -
397 - /*send transfer command to VPH */
398 - outport(Port, 0x0001);
399 - outport(Port, Size);
400 - outport(Port, (int) Start);
401 - outport(Port, (int) (Start >> 16));
402 - outport(Port, 0x0000);
403 -
404 - /*handle expected number of words */
405 - while (Size--)
406 - {
407 - /*read a word from file */
408 - Val = GetWord(Infile);
409 -
410 - /*wait until write fifo not full */
411 - while ( !( inport(Base + 2) & WR_FIFO_FULL) )
412 - ;
413 -
414 - /*write word */
415 - outport(Port, Val);
416 - }
417 - }
418 -
419 - /*---------------------------------------------------------------------------
420 - | Routine:Download() --- Transfer PC file to VPH memory
421 - |
422 - | Inputs: Filename- Name of source file
423 - |Format- Input file format
424 - |Start- Start address
425 - |Size- Number of words to transfer
426 - ---------------------------------------------------------------------------*/
427 -
428 - static
429 - void Download( char *Filename, int Format, unsigned long Start, UINT Size)
430 - {
431 - FILE*Infile;
432 - intDone;
433 - intc;
434 -
435 - /*open input file */
436 - Infile = fopen(Filename, "rt");
```

```
437 - if ( Infile == NULL )
438 - {
439 - gotoxy(1, 23);
440 - clreol();
441 - cprintf("File open failed \x07");
442 - sleep(3);
443 - return;
444 - }
445 -
446 - /*check format and handle each */
447 - if ( Format == HEX_FORMAT )
448 - {
449 - SendBlock( Infile, Start, Size );
450 - }
451 - else
452 - {
453 - /*S format - handle each record  */
454 - Done = 0;
455 - while (!Done)
456 - {
457 - /*find next 'S' and get record type */
458 - while ( (c = getc(Infile)) != 'S' && c != EOF)
459 - ;
460 - c = getc(Infile);
461 -
462 - /*handle each type */
463 - if (c == '3')
464 - {
465 - /*get record length and compute data words */
466 - fscanf(Infile, "%2x", &Size);
467 - Size = (Size >> 1) - 2;
468 -
469 - /*get start address */
470 - fscanf(Infile, "%8lx", &Start);
471 -
472 - /*transfer data field */
473 - SendBlock( Infile, Start, Size );
474 - }
475 - else if (c == '7')
476 - {
477 - Done = 1;
478 - }
479 - else if (c == EOF)
480 - {
481 - gotoxy(1, 23);
482 - clreol();
483 - cprintf("Missing end-of-file record \x07");
484 - sleep(3);
485 - Done = 1;
486 - }
487 - else
488 - {
489 - gotoxy(1, 23);
490 - clreol();
491 - cprintf("Unexpected record type %c in file \x07", c);
492 - sleep(3);
493 - Done = 1;
494 - }
495 - }
496 - }
497 - fclose(Infile);
498 - }
499 -
500 - /*---------------------------------------------------------------------------
501 - | Routine:main() --- Entry point and main routine for program.
502 - ---------------------------------------------------------------------------*/
503 -
504 - voidmain(intargc,
505 -  char**argv)
506 - {
507 - registerUINTVal;
508 - registerintc;
509 - auto unsigned longStart;
510 - autoUINTSize;
511 - autochar Buffer[NAME_SIZE+2] = { NAME_SIZE };
512 - autochar*Filename;
513 - autointFormat;
514 -
515 - /*Get command line parameters.*/
516 - if (argc > 1)
517 - {
518 - /*Get I/O board base address.*/
519 - if (sscanf(argv[1], "%x", &Base) != 1)
520 - {
521 - fprintf(stderr, "Error, parameter one must be a "
522 - "hexadecimal I/O space address.\n");
523 - exit( 2 );
524 - }
525 - }
526 - Port = Base;
527 -
528 - /*Set up to read command.*/
529 - clrscr();
530 - gotoxy(1, 22);
531 - cprintf("Commands: (B)ase, (P)ort, (R)ead, (W)rite, (U)pload, (D)ownload, "
532 - "(Q)uit");
533 -
534 - /*Print current port number.*/
535 - InitDsp( Port );
536 -
537 - /*Initialize the screen.*/
538 - for ( ; ; )
539 - {
540 - /*Show the status.*/
541 - ShowStat( Base );
542 -
543 - /*Get command.*/
544 - gotoxy(1, 23);
545 - clreol();
```

```
546 - cprintf("Command? ");
547 - if ((c = getche()) == 'Q' || c == 'q')
548 - break;
549 -
550 - /*Execute command.*/
551 - switch ( tolower( c ) )
552 - {
553 - case 'b':
554 - /*Get base address number.*/
555 - gotoxy(1, 23);
556 - clreol();
557 - cprintf("Enter Base Port Address > ");
558 -
559 - /*Get hex number.*/
560 - Base = GetHexNo( 3 );
561 -
562 - /*Print current port number.*/
563 - gotoxy(1, 14);
564 - clreol();
565 - textcolor( LIGHTBLUE );
566 - textbackground( LIGHTGRAY );
567 - cputs( "Base  Value" );
568 - textcolor( LIGHTGRAY );
569 - textbackground( BLACK );
570 - cputs(" : ");
571 - PrtVal( Base );
572 - break;
573 - case 'p':
574 - /*Get port number.*/
575 - gotoxy(1, 23);
576 - clreol();
577 - cprintf("Enter Port Number > ");
578 -
579 - /*Get hex number.*/
580 - Port = GetHexNo( 3 );
581 -
582 - /*Print current port number.*/
583 - gotoxy(1, 15);
584 - clreol();
585 - textcolor( LIGHTBLUE );
586 - textbackground( LIGHTGRAY );
587 - cputs( "Port  Value" );
588 - textcolor( LIGHTGRAY );
589 - textbackground( BLACK );
590 - cputs(" : ");
591 - PrtVal( Port );
592 - break;
593 - case 'r':
594 - /*Read value from port.*/
595 - Val = inport( Port );
596 -
597 - /*Print current port number.*/
598 - gotoxy(1, 17);
599 - clreol();
600 - textcolor( LIGHTBLUE );
601 - textbackground( LIGHTGRAY );
602 - cputs( "Read  Value" );
603 - textcolor( LIGHTGRAY );
604 - textbackground( BLACK );
605 - cputs(" : ");
606 - PrtVal( Val );
607 - break;
608 - case 'w':
609 - /*Get port number.*/
610 - gotoxy(1, 23);
611 - clreol();
612 - cprintf("Enter Value to Write > ");
613 -
614 - /*Get hex number.*/
615 - Val = GetHexNo( 4 );
616 - outport(Port, Val);
617 -
618 - /*Print current port number.*/
619 - gotoxy(1, 16);
620 - clreol();
621 - textcolor( LIGHTBLUE );
622 - textbackground( LIGHTGRAY );
623 - cputs( "Write Value" );
624 - textcolor( LIGHTGRAY );
625 - textbackground( BLACK );
626 - cputs( " : " );
627 - PrtVal( Val );
628 - break;
629 - case 'u':
630 - /*Get start address.*/
631 - gotoxy(1, 23);
632 - clreol();
633 - cprintf("Enter Start Address > ");
634 -
635 - /*Get hex number.*/
636 - Start = GetHexNo( 8 );
637 -
638 - /*Get size in words.*/
639 - gotoxy(1, 23);
640 - clreol();
641 - cprintf("Enter Number of Words > ");
642 -
643 - /*Get hex number.*/
644 - Size = GetHexNo( 4 );
645 -
646 - /*Get filename.*/
647 - gotoxy(1, 23);
648 - clreol();
649 - cprintf("Enter Destination Filename > ");
650 - Filename = cgets(Buffer);
651 -
652 - /*Perform upload operation */
653 - Upload( Start, Size, Filename);
654 - break;
```

```
655 - case 'd':
656 - /*Get filename.*/
657 - gotoxy(1, 23);
658 - clreol();
659 - cprintf("Enter Source Filename > ");
660 - Filename = cgets(Buffer);
661 -
662 - /*check format, count words if hex format */
663 - Format = CheckFile( Filename, &Size );
664 -
665 - /*if hex format, request start address */
666 - if (Format == HEX_FORMAT)
667 - {
668 - /*Get start address.*/
669 - gotoxy(1, 23);
670 - clreol();
671 - cprintf("Enter Start Address > ");
672 - Start = GetHexNo( 8 );
673 - }
674 -
675 - /*Perform download operation */
676 - Download( Filename, Format, Start, Size );
677 - break;
678 - }
679 - }
680 -
681 - clrscr();
682 - }
```

PC I/O Board Driver

Installation:

To install the driver, an installable device driver entry must be placed in the config.sys file. The entry looks like:

```
Device=c:\iob.bin 340 a 1
```

where 'Device=' tells MS-DOS that what follows is the file name of an installable device driver, 'c:\iob.bin' is the disk, directory and file name of the device driver file, '340' is the base address of the I/O ports used by the PC I/O board, 'a' is the interrupt number used by the board (interrupts are not currently implemented) and '1' is the device unit.

At boot time, the device driver will display a header containing the name of the driver, some information about the device driver configuration (most of it taken straight off the device driver command line), and then print a prompt and wait for the user to press any key.

Shown below is the exact config.sys file that *I* used to install the I/O board drivers when I was testing them.

```
files=40
buffers=10
break=on
lastdrive=z
Device=c:\himem.sys
Device=c:\emm386.sys 2000
device=c:\windows\smartdrv.sys 2048 1024
device=c:\windows\ramdrive.sys 1024 /e
Device=c:\dnadrive\station.sys units=4
Device=c:\dnadrive\spool.sys
Device=c:\iob.bin 340 a 1
Device=c:\iob.bin 360 b 2
shell=c:\etc\init.exe -R c:
```

Use:

This is a REAL MS-DOS driver, which means that you can use it just like any other character device. For example, to send a file to the VPH, type

```
copy vphfile.dat iob1
```

and MS-DOS will send the file (if it is an even number of bytes) to the VPH.

It is important to remember that the device driver trys to mimic a character device driver, but the PC I/O interface board is a word device. This means that if you send q * w + r (where w is 2, r is zero or one) bytes to the device, only q * w bytes will be received at the other end. The device driver will report that it sent all q * w + r bytes, but the last byte will be waiting in a buffer in the device driver, and will not actually be sent until at least one more byte is written to the device driver to make up a full word.

Since the driver is a real MS-DOS device driver, it can be accessed just like any other file or device from any programming language that supports file I/O.

A list of the device driver functions that this device supports is:

0-Initialization. This function is NEVER accessed by the user.
4-Read. Read data from the device.
6-Input Status. Determine if there is any data to read.
7-Input Flush. Throws away any data in the input buffer.
8-Write. Write data to the device.
10-Output Status. Determines whether the output buffer is empty.
16-Output Until Busy. Output until device output buffers are full.
This is synonymous with the Write function for this device.
19-Generic IO Control. Send commands to the device. The device currently only supports one command; reset.

Debugging:

The PC I/O Board driver is written in pure assembly language, and is NOT debugable by any of STC's inhouse software debuggers. There are two ways to track down bugs; code inspection and documentation review, and checkpoint dumps. The first is the recomended way, the second is useful when the programmer becomes to lazy or frustrated to use the first.

A check point dump consists of allocating a big enough buffer in memory to store the relevant information, and inserting code into the part of the driver to debug to write the information into the buffer. The buffers can be read or written from the application level, but NOT from within MS-DOS or the device driver (MS-DOS is not reentrant and there is only ONE request packet for all device drivers in the system. Which means you can set a breakpoint in the device driver code, but when the break occurs the data in the request packet will be for the last I/O call made by the debugger, NOT your device driver).

```
;*****************************************************************************
; Module:Structure and Constant definitions.
;
; Author:John W. M. Stevens
;*****************************************************************************

; I/O board status register bits.
SR_STAT_0equ0001h
SR_STAT_1equ0002h
SR_PARITYequ0004h
SR_WRT_EMPTYequ0008h
SR_WRT_FULLequ0010h
SR_WRT_ALMOST_EMPTYequ0020h
SR_WRT_ALMOST_FULLequ0040h
SR_RD_EMPTYequ0080h
SR_RD_FULLequ0100h
SR_RD_ALMOST_EMPTYequ0200h
SR_RD_ALMOST_FULLequ0400h

; I/O board control register bits.
CR_BASE_VALUEequ048ch
CR_SET_MASKequ0200h
CR_RESETequ0010h
CR_ENABLE_INTSequ0800h

; I/O board interrupt mask value.
IR_RD_FULLequ0100h
IR_RD_EMPTYequ0080h
IR_WR_FULLequ0010h
IR_WR_EMPTYequ0008h
IR_STAT_0equ0001h
IR_STAT_1equ0002h

; Status for request header structures.
RH_OKequ01h
RH_BUSYequ03h
RH_ERRORequ81h

; Error codes.
ERR_WRITE_PROTECT_VIOLATIONequ00h
ERR_UNKNOWN_UNITequ01h
ERR_DRIVE_NOT_READYequ02h
ERR_UNKNOWN_COMMANDequ03h
ERR_CRC_ERRORequ04h
ERR_INCORRECT_LENGTHequ05h
ERR_SEEK_ERRORequ06h
ERR_UNKNOWN_MEDIAequ07h
ERR_SECTOR_NOT_FOUNDequ08h
ERR_PRINTER_OUT_OF_PAPERequ09h
ERR_WRITE_FAULTequ0ah
ERR_READ_FAULTequ0bh
ERR_GENERAL_FAILUREequ0ch
ERR_INVALID_DISK_CHANGEequ0fh

DVC_HDRstruc
dhLInkdd?; Pointer to next driver.
dhAttribdw?; Driver attributes.
dhStrategydw?; Strategy routine offset.
dhInterruptdw?; 'Interrupt' routine offset.
dhNameOrUnitsdb'????????'; Device driver name.
DVC_HDRends

REQ_PKTstruc
rhLengthdb?; Length of record in bytes.
rhUnitdb?; Not used.
rhFunctiondb?; Function number, always zero.
rhStatusdw?; Returns the status.
rhReserveddb8 dup (?); Spare space.
REQ_PKTends

INITstruc
irLengthdb?; Length of record in bytes.
irUnitdb?; Not used.
irFunctiondb?; Function number, always zero.
irStatusdw?; Returns the status.
irReserveddb8 dup (?); Spare space.
irUnitsdb?; Number of units for block device.
irEndAddressdd?; Input:End of driver.
; Output:New end of driver.
irParamAddressdd?; Pointer to config.sys device=
; command line.
irDriveNumberdb?; First drive number for block device.
irMessageFlagdw?; Error message flag.
INITends

IOCTLRWREQUESTstruc
irwrLengthdb?; Length of record in bytes.
irwrUnitdb?; Not used.
irwrFunctiondb?; Function number.
irwrStatusdw?; Returns the status.
irwrReserveddb8 dup (?); Spare space.
irwrDatadb?; Not Used.
irwrBufferdd?; Pointer to data buffer.
irwrBytesdw?; Size of read or write.
IOCTLRWREQUESTends

IOCTLREQUESTstruc
giLengthdb?; Length of record in bytes.
giUnitdb?; Not used.
giFunctiondb?; Function number.
giStatusdw?; Returns the status.
giReserved1db8 dup (?); Spare space.
giCategorydb?; Category of device driver.
giMinorCodedb?; Minor code.
giReserved2dd?; Spare space.
giIOCTLDatadd?; Pointer to IOCTL data structure.
IOCTLREQUESTends
```

```
;*****************************************************************************
; Module:PC I/O board driver.
;
; Author:John Stevens
;*****************************************************************************

includeiob.inc

_textsegment byte public 'CODE'
assumecs:_text,ds:_text,es:_text

;First things first, the device driver header structure.
DvcHdrDVC_HDR<0ffffffffh, 0c800h, Strategy, Interrupt, 'IOB?    ' >

;Debug data.
WrtParamsIOCTLRWREQUEST<>
DbgPtrdw0
DbgBfrdw256dup(?)

;Global variables.
ReqPktdd?;Pointer to request packet.
CmdLinedd?;Pointer to command line.
IOBasedw?;Base I/O port address.
IntVecNodw?;Interrupt vector number.
CRIntNodw?;Control register interrupt number.
RdFlagdb?;Flag to indicate that buffer has data.
RdBffrdb?;Read buffer for non-even read sizes.
WrtFlagdb?;Flag to indicate that buffer has data.
WrtBffrdb?;Read buffer for non-even write sizes.
DspBtrdb32dup (?)

;Information strings.
HdrMsgdb'Space Tech PC I/O Board Driver Vers. 1.0', 0dh, 0ah, '$'
DrvrNamedb'    Driver Name          : $'
Stat1db'    I/O Base port address : $'
Stat2db'    Interrupt vector     : $'
AbsAddrdb'    Driver address       : $'
PressKeydb'Press any key to continue . . .$'
CrLfdb0dh, 0ah, '$'

;Error strings.
BadIOBasedb'Bad value for I/O base address.', 0dh, 0ah, '$'
BadCmdLinedb'Bad command line structure.', 0dh, 0ah, '$'
BadIntNodb'Bad interrupt number.', 0dh, 0ah, '$'
SyntaxErrdb'Syntax Error: iob.bin <IOBase> <Interrupt Vector> '
db'<Device Number>', 0dh, 0ah, '$'

;---------------------------------------------------------------------------
;Save Operations --- Save the operations requested in a debug buffer.
;---------------------------------------------------------------------------

procDebug

;Save registers used.
pushbx
pushax

;Save command.
movax, bx

;Check for overflow.
movbx, cs:DbgPtr
cmpbx, 200h
jgeDbgOvrFlow

;Get pointer to debug buffer.
leabx, cs:DbgBfr
addbx, cs:DbgPtr

;Save value of command word.
movcs:[bx], ax
addcs:DbgPtr, 2

;Return from debug routine.
DbgOvrFlow:
popax
popbx
ret

endpDebug

;Execute commands.
Interrupt:
pushax
pushbx
pushcx
pushdx
pushdi
pushsi
pushds
pushes

;Retrieve the address of the request header packet.
movbx, cs
movds, bx
lesdi, ReqPkt

;Get function number.
movbl, es:[di].rhFunction
xorbh, bh

;This is a legal function number.
callDebug
shlbx, 1
jmpword ptr [FuncTbl + bx]

FuncTbldwInitialize;Function #00.
dwBlockDvc;Function #01.
dwBlockDvc;Function #02.
```

```
dwIOCtlRead;Function #03.
dwRead;Function #04.
dwBlockDvc;Function #05.
dwInputStatus;Function #06.
dwInputFlush;Function #07.
dwWrite;Function #08.
dwWriteVer;Function #09.
dwOutputStatus;Function #0a.
dwBlockDvc;Function #0b.
dwIOCtlWrite;Function #0c.
dwOpenDvc;Function #0d.
dwCloseDvc;Function #0e.
dwBlockDvc;Function #0f.
dwBlockDvc;Function #10.
dwBlockDvc;Function #11.
dwBlockDvc;Function #12.
dwGenIOCtl;Function #13.
dwBlockDvc;Function #14.
dwBlockDvc;Function #15.
dwBlockDvc;Function #16.
dwBlockDvc;Function #17.
dwBlockDvc;Function #18.
dwIOCtlQuery;Function #19.

;-----------------------------------------------------------------------------
;Read --- Read data from device.  This routine has two entry points.
;-----------------------------------------------------------------------------

IOCtlRead:
Read:

;Error check for possible zero byte reads.
movcx, es:[di].irwrBytes
cmpcx, 0
jnzNonZeroRead
jmpReturnOK

;Get pointer to read buffer, zero out bytes read counter.
NonZeroRead:
lesdi, es:[di].irwrBuffer
xorsi, si

;If there is a spare byte to read, get it.
cmpRdFlag, 0
jeDoRead

;Clear flag.
movRdFlag, 0

;Get byte and store in read buffer.
moval, RdBffr
moves:[di], al
incdi
incsi
deccx

;Determine if there are any complete words to read.
DoRead:
movbx, cx
shrcx, 1
jcxzReadByte

;Read the proper number of words.
movdx, IOBase
ReadWords:
;If the Read FIFO is empty, read is complete.
adddx, 2
inax, dx
testax, 080h
jzRdDone

;Read word.
subdx, 2
inax, dx

;Save word.
moves:[di], ax
adddi, 2
addsi, 2

;Check to see if enough words have been read.
subbx, 2
loopReadWords

;If there is a byte left to read, do so.
ReadByte:
testbx, 1
jzRdDone

;Check to see if Read Fifo is empty.
movdx, IOBase
adddx, 2
inax, dx
testax, 080h
jzRdDone

;Read the word, save bytes.
subdx, 2
inax, dx
moves:[di], al
incsi
incdi
movRdBffr, ah
movRdFlag, 1

;Calculate and save the number of bytes read.
RdDone:
lesdi, ReqPkt
moves:[di].irwrBytes, si
```

```
219 - jmpReturnOK
220 -
221 - ;---------------------------------------------------------------------------
222 - ;Write --- Write data to a device.
223 - ;---------------------------------------------------------------------------
224 -
225 - Write:
226 - IOCtlWrite:
227 -
228 - ;Get write count and error check for possible zero length writes.
229 - movcx, es:[di].irwrBytes
230 - cmpcx, 0
231 - jneNonZeroWrt
232 - jmpReturnOK
233 -
234 - ;Get pointer to write buffer, save write buffer size and zero out byte
235 - ;write count.
236 - NonZeroWrt:
237 - lesdi, es:[di].irwrBuffer
238 - xorsi, si
239 -
240 - ;Is there a spare byte to write?
241 - cmpWrtFlag, 0
242 - jeDoWrite
243 -
244 - ;Check to make sure that the Write FIFO is not full.
245 - movdx, IOBase
246 - adddx, 2
247 - inax, dx
248 - testax, 010h
249 - jzWrtDone
250 -
251 - ;Zero out flag.
252 - movWrtFlag, 0
253 -
254 - ;Build word to write.
255 - moval, WrtBffr
256 - movah, es:[di]
257 - incdi
258 - incsi
259 - deccx
260 -
261 - ;Write word to buffer.
262 - subdx, 2
263 - outdx, ax
264 -
265 - ;Write words to Write FIFO if there are any to write.
266 - DoWrite:
267 - movbx, cx
268 - shrcx, 1
269 - jcxzWriteByte
270 -
271 - ;Loop to write words.
272 - movdx, IOBase
273 - WrtWords:
274 - ;Check for Write FIFO being full.
275 - adddx, 2
276 - inax, dx
277 - testax, 010h
278 - jzWrtDone
279 -
280 - ;Get word to write.
281 - movax, es:[di]
282 - adddi, 2
283 - addsi, 2
284 - subbx, 2
285 -
286 - ;Write word.
287 - subdx, 2
288 - outdx, ax
289 -
290 - ;Any more words to write?
291 - loopWrtWords
292 -
293 - ;Check for a byte left to write.
294 - WriteByte:
295 - testbx, 1
296 - jzWrtDone
297 -
298 - ;Save byte and set flag.
299 - movWrtFlag, 1
300 - moval, es:[di]
301 - movWrtBffr, al
302 - incdi
303 - incsi
304 -
305 - ;Save number of bytes written.
306 - WrtDone:
307 - lesdi, ReqPkt
308 - moves:[di].irwrBytes, si
309 -
310 - ;Save the write parameters in the debug buffer.
311 - pushcx
312 - pushes
313 - pushds
314 - pushsi
315 - pushdi
316 -
317 - xorch, ch
318 - movcl, es:[di].irwrLength
319 -
320 - ldssi, cs:ReqPkt
321 - pushcs
322 - popes
323 - lesdi, WrtParams
324 - repmovsb
325 -
326 - popdi
327 - popsi
```

```
328 - popds
329 - popes
330 - popcx
331 -
332 - jmpReturnOK
333 -
334 - ;---------------------------------------------------------------------------
335 - ;InputStatus --- Determine whether there are any characters to read or
336 - ;not.
337 - ;---------------------------------------------------------------------------
338 -
339 - InputStatus:
340 -
341 - ;Check to see if there is a saved read byte.
342 - cmpRdFlag, 0
343 - jneRFNotEmpty
344 -
345 - ;Get the status of the read FIFO.
346 - movdx, IOBase
347 - adddx, 2
348 - inax, dx
349 - testax, 080h
350 - jnzRFNotEmpty
351 -
352 - ;The read FIFO is empty, return busy.
353 - RFEmpty:
354 - movah, RH_BUSY
355 - xoral, al
356 - moves:[di].rhStatus, ax
357 - jmpEndInterrupt
358 -
359 - ;The read FIFO is not empty, return that there is a character in it.
360 - RFNotEmpty:
361 - movah, RH_OK
362 - xoral, al
363 - moves:[di].rhStatus, ax
364 - jmpEndInterrupt
365 -
366 - ;---------------------------------------------------------------------------
367 - ;InputFlush --- Input flush.
368 - ;---------------------------------------------------------------------------
369 -
370 - InputFlush:
371 -
372 - ;Set flag to no saved read character.
373 - movRdFlag, 0
374 -
375 - ;Read words from the read FIFO until there are no more.
376 - movdx, IOBase
377 - InFlushLp:
378 - adddx, 2
379 - inax, dx
380 - testax, 080h
381 - jzInFlushDone
382 -
383 - subdx, 2
384 - inax, dx
385 - jmpInFlushLp
386 -
387 - ;Done with flush.
388 - InFlushDone:
389 - jmpReturnOK
390 -
391 - ;---------------------------------------------------------------------------
392 - ;OutputStatus --- Determine whether all characters have been read or not.
393 - ;---------------------------------------------------------------------------
394 -
395 - OutputStatus:
396 -
397 - ;Check to see if there is a saved write byte.
398 - cmpWrtFlag, 0
399 - jneWFNotEmpty
400 -
401 - ;Get the status of the write FIFO.
402 - movdx, IOBase
403 - adddx, 2
404 - inax, dx
405 - testax, 08h
406 - jnzWFNotEmpty
407 -
408 - ;The read FIFO is empty, return busy.
409 - WFEmpty:
410 - movah, RH_BUSY
411 - xoral, al
412 - moves:[di].rhStatus, ax
413 - jmpEndInterrupt
414 -
415 - ;The read FIFO is not empty, return that there is a character in it.
416 - WFNotEmpty:
417 - movah, RH_OK
418 - xoral, al
419 - moves:[di].rhStatus, ax
420 - jmpEndInterrupt
421 -
422 - ;---------------------------------------------------------------------------
423 - ;OpenDvc/CloseDvc --- Reset the device.
424 - ;---------------------------------------------------------------------------
425 -
426 - OpenDvc:
427 - CloseDvc:
428 - jmpReturnOK
429 -
430 - ;Establish the base address of the board.
431 - movdx, IOBase
432 - adddx, 2
433 -
434 - ;Reset the board.
435 - movax, CR_RESET
436 - outdx, ax
```

```
437 -
438 - ;Now set normal operating values.
439 - movax, CR_BASE_VALUE
440 - outdx, ax
441 -
442 - ;Clear read and write buffer flags.
443 - movRdFlag, 0
444 - movWrtFlag, 0
445 -
446 - ;Done.
447 - jmpReturnOK
448 -
449 - ;-----------------------------------------------------------------------------
450 - ;Unimplemented functions below.
451 - ;-----------------------------------------------------------------------------
452 -
453 - WriteVer:
454 - GenIOCtl:
455 - GetLogical:
456 - SetLogical:
457 - IOCtlQuery:
458 -
459 - ;Illegal or unsupported command.
460 - BlockDvc:
461 - moval, ERR_UNKNOWN_COMMAND
462 -
463 - ;Error in command.
464 - ReturnERR:
465 - lesdi, cs:ReqPkt
466 - movah, RH_ERROR
467 - moves:[di].rhStatus, ax
468 - jmpEndInterrupt
469 -
470 - ;Command completed succesfully.
471 - ReturnOK:
472 - lesdi, cs:ReqPkt
473 - movah, RH_OK
474 - xoral, al
475 - moves:[di].rhStatus, ax
476 -
477 - EndInterrupt:
478 - popes
479 - popds
480 - popsi
481 - popdi
482 - popdx
483 - popcx
484 - popbx
485 - popax
486 - retf
487 -
488 - ;Initialize the driver by saving the address of the request header
489 - ;packet.
490 - Strategy:
491 - movword ptr cs:ReqPkt, bx
492 - movword ptr cs:ReqPkt + 2, es
493 - retf
494 -
495 - ;=============================================================================
496 - ;Dump what is below this point.
497 - ;=============================================================================
498 - EndDriver:
499 -
500 - ;-----------------------------------------------------------------------------
501 - ;PrintAry --- Print the proper number of characters to the screen.
502 - ;-----------------------------------------------------------------------------
503 -
504 - procPrintAry
505 -
506 - ;Save characters.
507 - pushax
508 - pushdx
509 -
510 - ;Print characters.
511 - PrtAryLp:
512 - movdl, es:[bx]
513 - movah, 02h
514 - int21h
515 - incbx
516 - loopPrtAryLp
517 -
518 - ;Return from print loop.
519 - popdx
520 - popax
521 - ret
522 -
523 - endpPrintAry
524 -
525 - ;-----------------------------------------------------------------------------
526 - ;PrintMsg --- Print an error message to the screen.
527 - ;
528 - ;dx- Contains the offset of the error message to print.
529 - ;-----------------------------------------------------------------------------
530 -
531 - procPrintMsg
532 -
533 - ;Save registers to be used.
534 - pushds
535 - pushax
536 -
537 - ;Create pointer to string.
538 - movax, cs
539 - movds, ax
540 -
541 - ;Call MS-DOS to display string.
542 - movah, 09h
543 - int21h
544 -
545 - ;Restore registers, return.
```

```
546 - popax
547 - popds
548 - ret
549 -
550 - endpPrintMsg
551 -
552 - ;-----------------------------------------------------------------------------
553 - ;SkipWhite --- Skip white space in the string.
554 - ;
555 - ;The 20 bit pointer es:[bx] points to the string.
556 - ;-----------------------------------------------------------------------------
557 -
558 - procSkipWhite
559 -
560 - ;Skip white space.
561 - pushax
562 - SkipWhiteSpace:
563 - moval, es:[bx]
564 - cmpal, ' '
565 - jeWhtSpace
566 - cmpal, 09h
567 - jeWhtSpace
568 - jmpNotWhtSpace
569 -
570 - ;Is white space, continue.
571 - WhtSpace:
572 - incbx
573 - loopSkipWhiteSpace
574 -
575 - ;Found non-white space character, return.
576 - NotWhtSpace:
577 - popax
578 - ret
579 -
580 - endpSkipWhite
581 -
582 - ;-----------------------------------------------------------------------------
583 - ;PrtHex --- Print a hexadecimal number to the display.
584 - ;
585 - ;The number to print is stored in ax.
586 - ;-----------------------------------------------------------------------------
587 -
588 - procPrtHex
589 -
590 - ;Save registers used.
591 - pushcx
592 - pushdx
593 - pushdi
594 -
595 - ;Initialize string.
596 - movdi, offset DspStr
597 - movbyte ptr [di], '0'
598 - incdi
599 - movbyte ptr [di], 'x'
600 - incdi
601 - movbyte ptr [di], '0'
602 - incdi
603 - movbyte ptr [di], '0'
604 - incdi
605 - movbyte ptr [di], '0'
606 - incdi
607 - movbyte ptr [di], '0'
608 - incdi
609 - movbyte ptr [di], '$'
610 - decdi
611 -
612 - ;Print value to string.
613 - movcx, 4
614 - PrtHexLp:
615 - ;Get current hex character.
616 - movdl, al
617 - anddl, 0fh
618 -
619 - ;Check to see if decimal (0-9) or higher (a-f).
620 - cmpdl, 0ah
621 - jlIsDecimal
622 -
623 - ;Is higher (a-f), convert to character to display.
624 - subdl, 0ah
625 - adddl, 'a'
626 - jmpNxtChar
627 -
628 - ;Is decimal, convert to character to display.
629 - IsDecimal:
630 - adddl, '0'
631 -
632 - ;Store character.
633 - NxtChar:
634 - movbyte ptr [di], dl
635 - decdi
636 -
637 - ;Shift next character into lowest nibble.
638 - shrax, 1
639 - shrax, 1
640 - shrax, 1
641 - shrax, 1
642 - loopPrtHexLp
643 -
644 - ;Create pointer to string.
645 - movdx, offset DspStr
646 - callPrintMsg
647 -
648 - ;Restore registers and return.
649 - popdi
650 - popdx
651 - popcx
652 - ret
653 -
654 - endpPrtHex
```

```
655 -
656 - ;-----------------------------------------------------------------------------
657 - ;GetHex --- Get a hexadecimal number from a string.
658 - ;
659 - ;The 20 bit pointer es:[bx] points to the start of the hexadecimal number.
660 - ;-----------------------------------------------------------------------------
661 -
662 - procGetHex
663 -
664 - ;Set up a loop counter.
665 - pushcx
666 - xorax, ax
667 - movcx, 4
668 -
669 - GetDigits:
670 - ;Test for in 0-9.
671 - movdl, es:[bx]
672 - cmpdl, '0'
673 - jlNotDecErr
674 - cmpdl, '9'
675 - jgNotDecErr
676 -
677 - ;Is decimal, convert.
678 - subdl, '0'
679 - jmpAddHexDigit
680 -
681 - ;Test for in a-f.
682 - NotDecErr:
683 - cmpdl, 'a'
684 - jlNotLoCase
685 - cmpdl, 'f'
686 - jgNotLoCase
687 -
688 - ;Is lower case hexadecimal, convert.
689 - subdl, 'a'
690 - adddl, 0ah
691 - jmpAddHexDigit
692 -
693 - ;Test for in A-F.
694 - NotLoCase:
695 - cmpdl, 'A'
696 - jlNotHex
697 - cmpdl, 'F'
698 - jgNotHex
699 -
700 - ;Is lower case hexadecimal, convert.
701 - subdl, 'A'
702 - adddl, 0ah
703 -
704 - ;Add another hex digit.
705 - AddHexDigit:
706 - shlax, 1
707 - shlax, 1
708 - shlax, 1
709 - shlax, 1
710 - oral, dl
711 -
712 - ;Go around again.
713 - incbx
714 - loopGetDigits
715 -
716 - NotHex:
717 - popcx
718 - ret
719 -
720 - endpGetHex
721 -
722 - ;-----------------------------------------------------------------------------
723 - ;Pause --- Wait for a key to be input before continuing on.
724 - ;-----------------------------------------------------------------------------
725 -
726 - procPause
727 -
728 - ;Print prompt message.
729 - movdx, offset PressKey
730 - callPrintMsg
731 -
732 - ;Get key from keyboard.
733 - movah, 01h
734 - int21h
735 -
736 - ;Print new line and carriage return.
737 - movdx, offset CrLf
738 - callPrintMsg
739 -
740 - ret
741 -
742 - endpPause
743 -
744 - ;-----------------------------------------------------------------------------
745 - ;Initialize the driver and get parameters from the Config.sys
746 - ;command line.
747 - ;-----------------------------------------------------------------------------
748 -
749 - ;Print driver info.
750 - Initialize:
751 - ;Get a pointer to the config.sys command line.
752 - lesbx, es:[di].irParamAddress
753 - movword ptr CmdLine, bx
754 - movword ptr CmdLine + 2, es
755 -
756 - ;Search for end of line.
757 - movcx, 300h
758 - LnEndSearch:
759 - movdl, byte ptr es:[bx]
760 - cmpdl, 0dh
761 - jeFndEndLine
762 - cmpdl, 0ah
763 - jeFndEndLine
```

```
764 - incbx
765 - loopLnEndSearch
766 -
767 - ;If we got here, we could not find end of line.
768 - movdx, offset BadCmdLine
769 - callPrintMsg
770 - jmpBadInit
771 -
772 - ;Found the end of the command line parameters, establish count.
773 - FndEndLine:
774 - movcx, bx
775 - movbx, word ptr cs:CmdLine
776 - subcx, bx
777 -
778 - ;Skip program name.
779 - SkipProgName:
780 - movdl, es:[bx]
781 - cmpdl, ' '
782 - jeEndProgName
783 - cmpdl, 09h
784 - jeEndProgName
785 - incbx
786 - loopSkipProgName
787 -
788 - ;If we got here, we are at end of command line.
789 - movdx, offset SyntaxErr
790 - callPrintMsg
791 - jmpBadInit
792 -
793 - ;Skip white space and get the first parameter, which is the I/O
794 - ;base address.
795 - EndProgName:
796 - callSkipWhite
797 - callGetHex
798 - movcs:IOBase, ax
799 -
800 - ;Check for legality of IOBase address.
801 - cmpax, 3ffh
802 - jleBaseOK
803 - movdx, offset BadIOBase
804 - callPrintMsg
805 - jmpBadInit
806 -
807 - ;Get second parameter, which is the hardware interrupt vector number.
808 - BaseOK:
809 - callSkipWhite
810 - callGetHex
811 - movcs:IntVecNo, ax
812 -
813 - ;Is this interrupt vector number 10?  If so, convert to the proper
814 - ;control register value.
815 - cmpax, 10
816 - jneChkInt11
817 - movcs:CRIntNo, 0
818 - jmpGetDvcNo
819 -
820 - ;Is this interrupt vector number 11?  If so, convert to the proper
821 - ;control register value.
822 - ChkInt11:
823 - cmpax, 11
824 - jneChkInt12
825 - movcs:CRIntNo, 1000h
826 - jmpGetDvcNo
827 -
828 - ;Is this interrupt vector number 12?  If so, convert to the proper
829 - ;control register value.
830 - ChkInt12:
831 - cmpax, 12
832 - jneChkInt15
833 - movcs:CRIntNo, 2000h
834 - jmpGetDvcNo
835 -
836 - ;Is this interrupt vector number 12?  If not, print an error message and
837 - ;exit driver with a bad initialization.
838 - ChkInt15:
839 - cmpax, 15
840 - jeIsInt15
841 - movdx, offset BadIntNo
842 - callPrintMsg
843 - jmpBadInit
844 -
845 - ;This is interrupt vector number 15, convert to the proper control
846 - ;register value.
847 - IsInt15:
848 - movcs:CRIntNo, 3000h
849 -
850 - ;Get the device number.
851 - GetDvcNo:
852 - callSkipWhite
853 - callGetHex
854 -
855 - ;Modify the driver name in the header structure.
856 - leabx, DvcHdr.dhNameOrUnits
857 - addal, '0'
858 - mov[bx + 3], al
859 -
860 - ;Print driver title.
861 - movdx, offset HdrMsg
862 - callPrintMsg
863 -
864 - ;Print the description for the driver name.
865 - movdx, offset DrvrName
866 - callPrintMsg
867 -
868 - ;Print the driver name.
869 - pushes
870 - movcx, 8
871 - movbx, cs
872 - moves, bx
```

```
873 - leabx, DvcHdr.dhNameOrUnits
874 - callPrintAry
875 - popes
876 -
877 - ;Print carriage return, new line.
878 - movdx, offset CrLf
879 - callPrintMsg
880 -
881 - ;Echo the IOBase address value.
882 - movdx, offset Stat1
883 - callPrintMsg
884 - movax, IOBase
885 - callPrtHex
886 - movdx, offset CrLf
887 - callPrintMsg
888 -
889 - ;Echo the Interrupt vector value.
890 - movdx, offset Stat2
891 - callPrintMsg
892 - movax, IntVecNo
893 - callPrtHex
894 - movdx, offset CrLf
895 - callPrintMsg
896 -
897 - ;Print the absolute address of this driver.
898 - movdx, offset AbsAddr
899 - callPrintMsg
900 - movax, cs
901 - callPrtHex
902 - movdl, ':'
903 - movah, 02h
904 - int21h
905 - leabx, cs:DvcHdr
906 - movax, bx
907 - callPrtHex
908 - movdx, offset CrLf
909 - callPrintMsg
910 -
911 - ;Initialize the board.  Begin by reseting all registers.
912 - movdx, cs:IOBase
913 - adddx, 2
914 - movax, CR_RESET
915 - outdx, ax
916 -
917 - ;Now set control register with interrupt number and interrupts turned
918 - ;off.
919 - movax, CR_BASE_VALUE
920 - outdx, ax
921 -
922 - ;Succesful initialization, set proper values in packet, return.
923 - lesdi, cs:ReqPkt
924 - moves:[di].irStatus, 100h
925 - movword ptr es:[di].irEndAddress, offset EndDriver
926 - movword ptr es:[di].irEndAddress + 2, cs
927 -
928 - callPause
929 - jmpEndInterrupt
930 -
931 - ;Error in initialization, set proper values in packet, return.
932 - BadInit:
933 - lesdi, cs:ReqPkt
934 - moves:[di].irStatus, 8100h
935 - movword ptr es:[di].irEndAddress, offset EndDriver
936 - movword ptr es:[di].irEndAddress + 2, cs
937 - moves:[di].irMessageFlag, 1
938 -
939 - callPause
940 - jmpEndInterrupt
941 -
942 - _textends
943 -
944 - end
```

```
;*****************************************************************************
; Module:PC I/O board driver.
;
; Author:John Stevens
;*****************************************************************************

includeiob.inc

_textsegment byte public 'CODE'
assumecs:_text,ds:_text,es:_text

;First things first, the device driver header structure.
DvcHdrDVC_HDR<0ffffffffh, 0a040h, Strategy, Interrupt, 'IOB?    ' >

;Global variables.
ReqPktdd?;Pointer to request packet.
CmdLinedd?;Pointer to command line.
IOBasedw?;Base I/O port address.
IntVecNodw?;Interrupt vector number.
CRIntNodw?;Control register interrupt number.
RdFlagdb?;Flag to indicate that buffer has data.
RdBffrdb?;Read buffer for non-even read sizes.
WrtFlagdb?;Flag to indicate that buffer has data.
WrtBffrdb?;Read buffer for non-even write sizes.
Dup8trdb)2dup (?)

;Information strings.
HdrMsgdb'Space Tech PC I/O Board Driver Vers. 1.0', 0dh, 0ah, '$'
DrvrNamedb'    Driver Name          : $'
Stat1db'    I/O Base port address : $'
Stat2db'    Interrupt vector     : $'
AbsAddrdb'    Driver address       : $'
PressKeydb'Press any key to continue . . .$'
CrLfdb0dh, 0ah, '$'

;Error strings.
BadIOBasedb'Bad value for I/O base address.', 0dh, 0ah, '$'
BadCmdLinedb'Bad command line structure.', 0dh, 0ah, '$'
BadIntNodb'Bad interrupt number.', 0dh, 0ah, '$'
SyntaxErrdb'Syntax Error: iob.bin <IOBase> <Interrupt Vector> '
db'<Device Number>', 0dh, 0ah, '$'

;---------------------------------------------------------------------------
;Execute commands.
;---------------------------------------------------------------------------

Interrupt:
pushax
pushbx
pushcx
pushdx
pushdi
pushsi
pushds
pushes

;Retrieve the address of the request header packet.
movbx, cs
movds, bx
lesdi, ReqPkt

;Get function number.
movbl, es:[di].rhFunction
xorbh, bh

;This is a legal function number.
shlbx, 1
jmpword ptr [FuncTbl + bx]

FuncTbldwInitialize;Function #00.
dwBlockDvc;Function #01.
dwBlockDvc;Function #02.
dwIOCtlRead;Function #03.
dwRead;Function #04.
dwBlockDvc;Function #05.
dwInputStatus;Function #06.
dwInputFlush;Function #07.
dwWrite;Function #08.
dwWriteVer;Function #09.
dwOutputStatus;Function #0a.
dwBlockDvc;Function #0b.
dwIOCtlWrite;Function #0c.
dwOpenDvc;Function #0d.
dwCloseDvc;Function #0e.
dwBlockDvc;Function #0f.
dwOutBusy;Function #10.
dwBlockDvc;Function #11.
dwBlockDvc;Function #12.
dwGenIOCtl;Function #13.
dwBlockDvc;Function #14.
dwBlockDvc;Function #15.
dwBlockDvc;Function #16.
dwBlockDvc;Function #17.
dwBlockDvc;Function #18.
dwIOCtlQuery;Function #19.

;---------------------------------------------------------------------------
;Read --- Read data from device.  This routine has two entry points.
;---------------------------------------------------------------------------

Read:

;Error check for possible zero byte reads.
movcx, es:[di].irwrBytes
cmpcx, 0
jnzNonZeroRead
jmpReturnOK

;Get pointer to read buffer, zero out bytes read counter.
```

```
110 - NonZeroRead:
111 - lesdi, es:[di].irwrBuffer
112 - xorsi, si
113 -
114 - ;If there is a spare byte to read, get it.
115 - cmpRdFlag, 0
116 - jeDoRead
117 -
118 - ;Clear flag.
119 - movRdFlag, 0
120 -
121 - ;Get byte and store in read buffer.
122 - moval, RdBffr
123 - moves:[di], al
124 - incdi
125 - incsi
126 - deccx
127 -
128 - ;Determine if there are any complete words to read.
129 - DoRead:
130 - movbx, cx
131 - shrcx, 1
132 - jcxzReadByte
133 -
134 - ;Read the proper number of words.
135 - movdx, IOBase
136 - ReadWords:
137 - ;If the Read FIFO is empty, read is complete.
138 - adddx, 2
139 - inax, dx
140 - testax, SR_RD_EMPTY
141 - jzRdDone
142 -
143 - ;Read word.
144 - subdx, 2
145 - inax, dx
146 -
147 - ;Save word.
148 - moves:[di], ax
149 - adddi, 2
150 - addsi, 2
151 -
152 - ;Check to see if enough words have been read.
153 - subbx, 2
154 - loopReadWords
155 -
156 - ;If there is a byte left to read, do so.
157 - ReadByte:
158 - testbx, 1
159 - jzRdDone
160 -
161 - ;Check to see if Read Fifo is empty.
162 - movdx, IOBase
163 - adddx, 2
164 - inax, dx
165 - testax, SR_RD_EMPTY
166 - jzRdDone
167 -
168 - ;Read the word, save bytes.
169 - subdx, 2
170 - inax, dx
171 - moves:[di], al
172 - incsi
173 - incdi
174 - movRdBffr, ah
175 - movRdFlag, 1
176 -
177 - ;Calculate and save the number of bytes read.
178 - RdDone:
179 - lesdi, ReqPkt
180 - moves:[di].irwrBytes, si
181 - jmpReturnOK
182 -
183 - ;-----------------------------------------------------------------------------
184 - ;Write --- Write data to a device.
185 - ;-----------------------------------------------------------------------------
186 -
187 - Write:
188 - OutBusy:
189 -
190 - ;Get write count and error check for possible zero length writes.
191 - movcx, es:[di].irwrBytes
192 - cmpcx, 0
193 - jneNonZeroWrt
194 - jmpReturnOK
195 -
196 - ;Get pointer to write buffer, save write buffer size and zero out byte
197 - ;write count.
198 - NonZeroWrt:
199 - lesdi, es:[di].irwrBuffer
200 - xorsi, si
201 -
202 - ;Is there a spare byte to write?
203 - cmpWrtFlag, 0
204 - jeDoWrite
205 -
206 - ;Check to make sure that the Write FIFO is not full.
207 - movdx, IOBase
208 - adddx, 2
209 - inax, dx
210 - testax, SR_WRT_FULL
211 - jzWrtDone
212 -
213 - ;Zero out flag.
214 - movWrtFlag, 0
215 -
216 - ;Build word to write.
217 - moval, WrtBffr
218 - movah, es:[di]
```

```
219 - incdi
220 - incsi
221 - deccx
222 -
223 - ;Write word to buffer.
224 - subdx, 2
225 - outdx, ax
226 -
227 - ;Write words to Write FIFO if there are any to write.
228 - doWrite:
229 - movbx, cx
230 - shrcx, 1
231 - jcxzWriteByte
232 -
233 - ;Loop to write words.
234 - movdx, IOBase
235 - WrtWords:
236 - ;Check for Write FIFO being full.
237 - adddx, 2
238 - inax, dx
239 - testax, SR_WRT_FULL
240 - jzWrtDone
241 -
242 - ;Get word to write.
243 - movax, es:[di]
244 - adddi, 2
245 - addsi, 2
246 - subbx, 2
247 -
248 - ;Write word.
249 - subdx, 2
250 - outdx, ax
251 -
252 - ;Any more words to write?
253 - loopWrtWords
254 -
255 - ;Check for a byte left to write.
256 - WriteByte:
257 - testbx, 1
258 - jzWrtDone
259 -
260 - ;Save byte and set flag.
261 - movWrtFlag, 1
262 - moval, es:[di]
263 - movWrtBffr, al
264 - incdi
265 - incsi
266 -
267 - ;Save number of bytes written.
268 - WrtDone:
269 - lesdi, ReqPkt
270 - moves:[di].irwrBytes, si
271 - jmpReturnOK
272 -
273 - ;-----------------------------------------------------------------------------
274 - ;InputStatus --- Determine whether there are any characters to read or
275 - ;not.
276 - ;-----------------------------------------------------------------------------
277 -
278 - InputStatus:
279 -
280 - ;Check to see if there is a saved read byte.
281 - cmpRdFlag, 0
282 - jneRFNotEmpty
283 -
284 - ;Get the status of the read FIFO.
285 - movdx, IOBase
286 - adddx, 2
287 - inax, dx
288 - testax, SR_RD_EMPTY
289 - jneRFNotEmpty
290 -
291 - ;The read FIFO is empty, return busy.
292 - RFEmpty:
293 - movah, RH_BUSY
294 - xoral, al
295 - moves:[di].rhStatus, ax
296 - jmpEndInterrupt
297 -
298 - ;The read FIFO is not empty, return that there is a character in it.
299 - RFNotEmpty:
300 - movah, RH_OK
301 - xoral, al
302 - moves:[di].rhStatus, ax
303 - jmpEndInterrupt
304 -
305 - ;-----------------------------------------------------------------------------
306 - ;InputFlush --- Input flush.
307 - ;-----------------------------------------------------------------------------
308 -
309 - InputFlush:
310 -
311 - ;Set flag to no saved read character.
312 - movRdFlag, 0
313 -
314 - ;Read words from the read FIFO until there are no more.
315 - movdx, IOBase
316 - InFlushLp:
317 - adddx, 2
318 - inax, dx
319 - testax, SR_RD_EMPTY
320 - jzInFlushDone
321 -
322 - subdx, 2
323 - inax, dx
324 - jmpInFlushLp
325 -
326 - ;Done with flush.
327 - InFlushDone:
```

```
328 - jmpReturnOK
329 -
330 - ;---------------------------------------------------------------------------
331 - ;OutputStatus --- Determine whether all characters have been written or not.
332 - ;---------------------------------------------------------------------------
333 -
334 - OutputStatus:
335 -
336 - ;Check to see if there is a saved write byte.
337 - cmpWrtFlag, 0
338 - jneWFNotEmpty
339 -
340 - ;Get the status of the write FIFO.
341 - movdx, IOBase
342 - adddx, 2
343 - inax, dx
344 - testax, SR_WRT_EMPTY
345 - jnzWFNotEmpty
346 -
347 - ;The read FIFO is empty, return busy.
348 - WFEmpty:
349 - movah, RH_BUSY
350 - xoral, al
351 - moves:[di].rhStatus, ax
352 - jmpEndInterrupt
353 -
354 - ;The read FIFO is not empty, return that there is a character in it.
355 - WFNotEmpty:
356 - movah, RH_OK
357 - xoral, al
358 - moves:[di].rhStatus, ax
359 - jmpEndInterrupt
360 -
361 - ;---------------------------------------------------------------------------
362 - ;OpenDvc/CloseDvc --- Reset the device.
363 - ;---------------------------------------------------------------------------
364 -
365 - GenIOCtl:
366 - ;Establish the base address of the board.
367 - movdx, IOBase
368 - adddx, 2
369 -
370 - ;Reset the board.
371 - movax, CR_RESET
372 - outdx, ax
373 -
374 - ;Now set normal operating values.
375 - movax, CR_BASE_VALUE
376 - outdx, ax
377 -
378 - ;Clear read and write buffer flags.
379 - movRdFlag, 0
380 - movWrtFlag, 0
381 -
382 - ;Done.
383 - jmpReturnOK
384 -
385 - ;---------------------------------------------------------------------------
386 - ;Unimplemented functions below.
387 - ;---------------------------------------------------------------------------
388 -
389 - OpenDvc:
390 - CloseDvc:
391 - IOCtlRead:
392 - IOCtlWrite:
393 - WriteVer:
394 - GetLogical:
395 - SetLogical:
396 - IOCtlQuery:
397 -
398 - ;Illegal or unsupported command.
399 - BlockDvc:
400 - moval, ERR_UNKNOWN_COMMAND
401 -
402 - ;Error in command.
403 - ReturnERR:
404 - leadi, cs:ReqPkt
405 - movah, RH_ERROR
406 - moves:[di].rhStatus, ax
407 - jmpEndInterrupt
408 -
409 - ;Command completed succesfully.
410 - ReturnOK:
411 - leadi, cs:ReqPkt
412 - movah, RH_OK
413 - xoral, al
414 - moves:[di].rhStatus, ax
415 -
416 - EndInterrupt:
417 - popes
418 - popds
419 - popsi
420 - popdi
421 - popdx
422 - popcx
423 - popbx
424 - popax
425 - retf
426 -
427 - ;Initialise the drivar by saving the address of the request header
428 - ;packet.
429 - Strategy:
430 - movword ptr cs:ReqPkt, bx
431 - movword ptr cs:ReqPkt + 2, es
432 - retf
433 -
434 - ;===========================================================================
435 - ;Dump what is below this point.
436 - ;===========================================================================
```

```
EndDriver:

;-----------------------------------------------------------------------------
;PrintAry --- Print the proper number of characters to the screen.
;-----------------------------------------------------------------------------

procPrintAry

;Save characters.
pushax
pushdx

;Print characters.
PrtAryLp:
movdl, es:[bx]
movah, 02h
int21h
incbx
loopPrtAryLp

;Return from print loop.
popdx
popax
ret

endpPrintAry

;-----------------------------------------------------------------------------
;PrintMsg --- Print an error message to the screen.
;
;dx- Contains the offset of the error message to print.
;-----------------------------------------------------------------------------

procPrintMsg

;Save registers to be used.
pushds
pushax

;Create pointer to string.
movax, cs
movds, ax

;Call MS-DOS to display string.
movah, 09h
int21h

;Restore registers, return.
popax
popds
ret

endpPrintMsg

;-----------------------------------------------------------------------------
;SkipWhite --- Skip white space in the string.
;
;The 20 bit pointer es:[bx] points to the string.
;-----------------------------------------------------------------------------

procSkipWhite

;Skip white space.
pushax
SkipWhiteSpace:
moval, es:[bx]
cmpal, ' '
jeWhtSpace
cmpal, 09h
jeWhtSpace
jmpNotWhtSpace

;Is white space, continue.
WhtSpace:
incbx
loopSkipWhiteSpace

;Found non-white space character, return.
NotWhtSpace:
popax
ret

endpSkipWhite

;-----------------------------------------------------------------------------
;PrtHex --- Print a hexadecimal number to the display.
;
;The number to print is stored in ax.
;-----------------------------------------------------------------------------

procPrtHex

;Save registers used.
pushcx
pushdx
pushdi

;Initialize string.
movdi, offset DspStr
movbyte ptr [di], '0'
incdi
movbyte ptr [di], 'x'
incdi
movbyte ptr [di], '0'
incdi
movbyte ptr [di], '0'
incdi
movbyte ptr [di], '0'
incdi
```

```
546 - movbyte ptr [di], '0'
547 - incdi
548 - movbyte ptr [di], '$'
549 - decdi
550 -
551 - ;Print value to string.
552 - movcx, 4
553 - PrtHexLp:
554 - ;Get current hex character.
555 - movdl, al
556 - anddl, 0fh
557 -
558 - ;Check to see if decimal (0-9) or higher (a-f).
559 - cmpdl, 0ah
560 - jlIsDecimal
561 -
562 - ;Is higher (a-f), convert to character to display.
563 - subdl, 0ah
564 - adddl, 'a'
565 - jmpNxtChar
566 -
567 - ;Is decimal, convert to character to display.
568 - IsDecimal:
569 - adddl, '0'
570 -
571 - ;Store character.
572 - NxtChar:
573 - movbyte ptr [di], dl
574 - decdi
575 -
576 - ;Shift next character into lowest nibble.
577 - shrax, 1
578 - shrax, 1
579 - shrax, 1
580 - shrax, 1
581 - loopPrtHexLp
582 -
583 - ;Create pointer to string.
584 - movdx, offset DspStr
585 - callPrintMsg
586 -
587 - ;Restore registers and return.
588 - popdi
589 - popdx
590 - popcx
591 - ret
592 -
593 - endpPrtHex
594 -
595 - ;-----------------------------------------------------------------------------
596 - ;GetHex --- Get a hexadecimal number from a string.
597 - ;
598 - ;The 20 bit pointer es:[bx] points to the start of the hexadecimal number.
599 - ;-----------------------------------------------------------------------------
600 -
601 - procGetHex
602 -
603 - ;Set up a loop counter.
604 - pushcx
605 - xorax, ax
606 - movcx, 4
607 -
608 - GetDigits:
609 - ;Test for in 0-9.
610 - movdl, es:[bx]
611 - cmpdl, '0'
612 - jlNotDecErr
613 - cmpdl, '9'
614 - jgNotDecErr
615 -
616 - ;Is decimal, convert.
617 - subdl, '0'
618 - jmpAddHexDigit
619 -
620 - ;Test for in a-f.
621 - NotDecErr:
622 - cmpdl, 'a'
623 - jlNotLoCase
624 - cmpdl, 'z'
625 - jgNotLoCase
626 -
627 - ;Is lower case hexadecimal, convert.
628 - subdl, 'a'
629 - adddl, 0ah
630 - jmpAddHexDigit
631 -
632 - ;Test for in A-F.
633 - NotLoCase:
634 - cmpdl, 'A'
635 - jlNotHex
636 - cmpdl, 'F'
637 - jgNotHex
638 -
639 - ;Is lower case hexadecimal, convert.
640 - subdl, 'A'
641 - adddl, 0ah
642 -
643 - ;Add another hex digit.
644 - AddHexDigit:
645 - shlax, 1
646 - shlax, 1
647 - shlax, 1
648 - shlax, 1
649 - oral, dl
650 -
651 - ;Go around again.
652 - incbx
653 - loopGetDigits
654 -
```

```
655 - NotHex:
656 - popcx
657 - ret
658 -
659 - endpGetHex
660 -
661 - ;-----------------------------------------------------------------------------
662 - ;Pause --- Wait for a key to be input before continuing on.
663 - ;-----------------------------------------------------------------------------
664 -
665 - procPause
666 -
667 - ;Print prompt message.
668 - movdx, offset PressKey
669 - callPrintMsg
670 -
671 - ;Get key from keyboard.
672 - movah, 01h
673 - int21h
674 -
675 - ;Print new line and carriage return.
676 - movdx, offset CrLf
677 - callPrintMsg
678 -
679 - ret
680 -
681 - endpPause
682 -
683 - ;-----------------------------------------------------------------------------
684 - ;Initialize the driver and get parameters from the Config.sys
685 - ;command line.
686 - ;-----------------------------------------------------------------------------
687 -
688 - ;Print driver info.
689 - Initialize:
690 - ;Get a pointer to the config.sys command line.
691 - lesbx, es:[di].irParamAddress
692 - movword ptr CmdLine, bx
693 - movword ptr CmdLine + 2, es
694 -
695 - ;Search for end of line.
696 - movcx, 300h
697 - LnEndSearch:
698 - movdl, byte ptr es:[bx]
699 - cmpdl, 0dh
700 - jeFndEndLine
701 - cmpdl, 0ah
702 - jeFndEndLine
703 - incbx
704 - loopLnEndSearch
705 -
706 - ;If we got here, we could not find end of line.
707 - movdx, offset BadCmdLine
708 - callPrintMsg
709 - jmpBadInit
710 -
711 - ;Found the end of the command line parameters, establish count.
712 - FndEndLine:
713 - movcx, bx
714 - movbx, word ptr cs:CmdLine
715 - subcx, bx
716 -
717 - ;Skip program name.
718 - SkipProgName:
719 - movdl, es:[bx]
720 - cmpdl, ' '
721 - jeEndProgName
722 - cmpdl, 09h
723 - jeEndProgName
724 - incbx
725 - loopSkipProgName
726 -
727 - ;If we got here, we are at end of command line.
728 - movdx, offset SyntaxErr
729 - callPrintMsg
730 - jmpBadInit
731 -
732 - ;Skip white space and get the first parameter, which is the I/O
733 - ;base address.
734 - EndProgName:
735 - callSkipWhite
736 - callGetHex
737 - movcs:IOBase, ax
738 -
739 - ;Check for legality of IOBase address.
740 - cmpax, 3ffh
741 - jleBaseOK
742 - movdx, offset BadIOBase
743 - callPrintMsg
744 - jmpBadInit
745 -
746 - ;Get second parameter, which is the hardware interrupt vector number.
747 - BaseOK:
748 - callSkipWhite
749 - callGetHex
750 - movcs:IntVecNo, ax
751 -
752 - ;Is this interrupt vector number 10?  If so, convert to the proper
753 - ;control register value.
754 - cmpax, 10
755 - jneChkInt11
756 - movcs:CRIntNo, 0
757 - jmpGotDvcNo
758 -
759 - ;Is this interrupt vector number 11?  If so, convert to the proper
760 - ;control register value.
761 - ChkInt11:
762 - cmpax, 11
763 - jneChkInt12
```

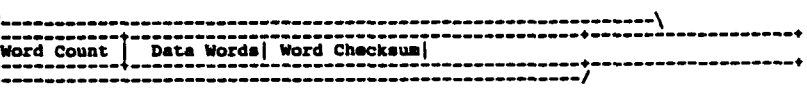
```
764 - movcs:CRIntNo, 1000h
765 - jmpGetDvcNo
766 -
767 - ;Is this interrupt vector number 12?  If so, convert to the proper
768 - ;control register value.
769 - ChkInt12:
770 - cmpax, 12
771 - jneChkInt15
772 - movcs:CRIntNo, 2000h
773 - jmpGetDvcNo
774 -
775 - ;Is this interrupt vector number 12?  If not, print an error message and
776 - ;exit driver with a bad initialization.
777 - ChkInt15:
778 - cmpax, 15
779 - jeIsInt15
780 - movdx, offset BadIntNo
781 - callPrintMsg
782 - jmpBadInit
783 -
784 - ;This is interrupt vector number 15, convert to the proper control
785 - ;register value.
786 - IsInt15:
787 - movcs:CRIntNo, 3000h
788 -
789 - ;Get the device number.
790 - GetDvcNo:
791 - callSkipWhite
792 - callGetHex
793 -
794 - ;Modify the driver name in the header structure.
795 - leabx, DvcHdr.dhNameOrUnits
796 - addal, '0'
797 - mov[bx + 3], al
798 -
799 - ;Print driver title.
800 - movdx, offset HdrMsg
801 - callPrintMsg
802 -
803 - ;Print the description for the driver name.
804 - movdx, offset DrvrName
805 - callPrintMsg
806 -
807 - ;Print the driver name.
808 - pushes
809 - movcx, 8
810 - movbx, cs
811 - moves, bx
812 - leabx, DvcHdr.dhNameOrUnits
813 - callPrintAry
814 - popes
815 -
816 - ;Print carriage return, new line.
817 - movdx, offset CrLf
818 - callPrintMsg
819 -
820 - ;Echo the IOBase address value.
821 - movdx, offset Stat1
822 - callPrintMsg
823 - movax, IOBase
824 - callPrtHex
825 - movdx, offset CrLf
826 - callPrintMsg
827 -
828 - ;Echo the Interrupt vector value.
829 - movdx, offset Stat2
830 - callPrintMsg
831 - movax, IntVecNo
832 - callPrtHex
833 - movdx, offset CrLf
834 - callPrintMsg
835 -
836 - ;Print the absolute address of this driver.
837 - movdx, offset AbsAddr
838 - callPrintMsg
839 - movax, cs
840 - callPrtHex
841 - movdl, ':'
842 - movah, 02h
843 - int21h
844 - leabx, cs:DvcHdr
845 - movax, bx
846 - callPrtHex
847 - movdx, offset CrLf
848 - callPrintMsg
849 -
850 - ;Initialize the board.  Begin by reseting all registers.
851 - movdx, cs:IOBase
852 - adddx, 2
853 - movax, CR_RESET
854 - outdx, ax
855 -
856 - ;Now set control register with interrupt number and interrupts turned
857 - ;off.
858 - movax, CR_BASE_VALUE
859 - outdx, ax
860 -
861 - ;Successful initialization, set proper values in packet, return.
862 - lesdi, cs:ReqPkt
863 - moves:[di].irStatus, 100h
864 - movword ptr es:[di].irEndAddress, offset EndDriver
865 - movword ptr es:[di].irEndAddress + 2, cs
866 -
867 - callPause
868 - jmpEndInterrupt
869 -
870 - ;Error in initialization, set proper values in packet, return.
871 - BadInit:
872 - lesdi, cs:ReqPkt
```

```
873 - moves:[di].irStatus, 8100h
874 - movword ptr es:[di].irEndAddress, offset EndDriver
875 - movword ptr es:[di].irEndAddress + 2, cs
876 - movbs:[di].irMessageFlag, 1
877 -
878 - callPause
879 - jmpEndInterrupt
880 -
881 - _textends
882 -
883 - end
```

PC I/O Interface Board Driver

Packet Structure

Word count includes data packet and check sum.



Check sum is 2's complement of the summation of the word count and all words in the data packet.

Word Count:Is 16 bits, little endian unsigned integer that represents the number of data words and the check sum word.

Data Words:Any number of data words (an even number of bytes).

Word Checksum:The 2's complement negation of the word count and all words in the data packet.

Communications Protocol

When designing a driver to work with the MS-DOS PC I/O board device driver, it is important to realize that the MS-DOS driver looks like a character device to the application.

This means that the driver will packetize writes to it and send them to the VPH/CPH using the following protocol.

Since the MS-DOS device driver has no buffering (except what is on the I/O board) and uses no interrupts, packets to the MS-DOS device driver must be less than or equal to the size of the FIFO's.

SenderReciever

1)Write all of packet to the Write FIFO.

2)Set STAT_0 bit in control register.

3)Wait for STAT_1 bit in status register to go High.

1)Interrupt (or poll for) the STAT_0 bit going high in the status register.

2)Read packet word count.

3)While packet not complete do Read data from Read FIFO. done

4)Set STAT_1 bit in control register.

5)Wait for STAT_0 in status register to be cleared.

4)Clear STAT_0 bit in control register.

5)Wait for STAT_1 bit in status register to be cleared.

6)Clear STAT_1 bit in control register.

Instructions for Using the MS-DOS PC I/O Board Driver

To use the device:

1)Open the device as you would any other device.

3)Read or write to driver.

4)When done, close the device.

```
Yet Another PC I/O Board Driver Design Document
-----------------------------------------------

Sofware structure of MS-DOS PC device driver:

Write Packet
------------

Set the write size to be the size of the write buffer.
Clear the write size value in the request header.
While the write size is greater than zero
do
If the write size is greater than or equal to 8188 bytes
then
Packet size is 8188 bytes.
else
Packet size is write size.
endif

Divide the packet size in bytes by two to get size in words.

Write the word size of the packet to the write FIFO.
Initialize the check sum with the packet size.
Write the packet to the Write FIFO and sum the words to
calculate the checksum.
Write the checksum to the write FIFO.

Set STAT0.
Loop
Read status register.
while STAT1 bit is clear.
Clear STAT0.
Loop
Read status register
while STAT1 is set.

Subtract twice the packet size from the write size.
done

Read Packet
-----------

If there is a partial packet still waiting in the read FIFO
then
Set the read size equal to the remaining packet size.
else If STAT0 is clear
then
Set the busy bit in the request header and return.
else
Read the packet size from the read FIFO.
Save the packet size in the packet size buffer.
Save the packet size as the initial checksum value.
Set the read size equal to the packet size.
endif

Divide the read buffer size by two to get the word count.
If the read size > read buffer size
then
Set the read size to the read buffer size.
endif

Read words from the read FIFO, summing them for the checksum.

If the number of words read is less than the packet size
then
Set the read size.
Return.
else If the entire packet has been read
then
Read the checksum word from the read FIFO and add to the checksum.

Set STAT1 in the control register.
Loop
Read status register.
while STAT0 is set.
Clear STAT1.

If the checksum is non-zero
then
Return a check sum error.
else
Set the packet size buffer to zero.
Return success.
endif
endif
```

APPENDIX D

MICROINSTRUCTION FORMAT

ADDRESS GENERATO

ADDRESS REGISTER FILE

PORT A WRITE SOURCE				PORT A WRITE ADDRESS				PORT B WRITE SOURCE				PORT B WRITE ADDRESS				SHIFT CTRL		PORT A READ ADDRESS				PORT B READ ADDRESS			
SEL3	SEL2	SEL1	SEL0	VRA5	VRA4	VRA3	VRA2	VRA1	VRA0	SEL3	SEL2	SEL1	SEL0	VRB5	VRB4	VRB3	VRB2	VRB1	VRB0	SM1	SM0	RDA5	RDA4	RDA3	RDA2	RDA1	RDA0	RDB5	RDB4	RDB3	RDB2	RDB1	RDB0
383	382	381	380	379	378	377	376	375	374	373	372	371	370	369	368	319	318	317	316	315	314	323	322	313	312	311	310	309	308	307	306	305	304	303	302

PORT A WRITE SOURCE				PORT A WRITE ADDRESS				PORT B WRITE SOURCE				PORT B WRITE ADDRESS				SHIFT CTRL		PORT A READ ADDRESS				PORT B READ ADDRESS			
SEL3	SEL2	SEL1	SEL0	VRA5	VRA4	VRA3	VRA2	VRA1	VRA0	SEL3	SEL2	SEL1	SEL0	VRB5	VRB4	VRB3	VRB2	VRB1	VRB0	SM1	SM0	RDA5	RDA4	RDA3	RDA2	RDA1	RDA0	RDB5	RDB4	RDB3	RDB2	RDB1	RDB0
383	382	381	380	379	378	377	376	375	374	373	372	371	370	369	368	319	318	317	316	315	314	323	322	313	312	311	310	309	308	307	306	305	304	303	302

MICROSEQUENCE

BRANCH ADDRESS																	
BA15	BA14	BA13	BA12	BA11	BA10	BA09	BA08	BA07	BA06	BA05	BA04	BA03	BA02	BA01	BA00	IA7	
367	366	365	364	363	362	361	360	359	358	357	356	355	354	353	352	351	

ADDRESS PORTS

PORT C					PORT D				BANK ADDRESS				
SEL3	SEL2	SEL1	SEL0	PSEL	SEL3	SEL2	SEL1	SEL0	PSEL	SEL3	SEL2	SEL1	SEL0	PSEL
301	300	299	298	297	296	295	294	293	292	291	290	289	288	255

PORT A					PORT B				PORT E				
SEL3	SEL2	SEL1	SEL0	PSEL	SEL3	SEL2	SEL1	SEL0	PSEL	SEL3	SEL2	SEL1	SEL0	PSEL
301	300	299	298	297	296	295	294	293	292	291	290	289	288	255

ADDRESS RAM #2

ADDRESS					DATA					
SEL3	SEL2	SEL1	SEL0	PSEL	SEL3	SEL2	SEL1	SEL0	PSEL	R/W
278	277	276	275	274	273	272	271	270	269	268

ADDRESS					DATA					
SEL3	SEL2	SEL1	SEL0	PSEL	SEL3	SEL2	SEL1	SEL0	PSEL	R/W
278	277	276	275	274	273	272	271	270	269	268

COMPARATOR

ADDRESS X SELECT					ADDRESS Y SELECT				
SEL3	SEL2	SEL1	SEL0	PSEL	SEL3	SEL2	SEL1	SEL0	PSEL
267	266	265	264	263	262	261	260	259	258

TWO DIMENSIONAL COUNTER #4

INPUT SOURCE					ROW COUNTER			COLUMN COUNTER			REGS	
SEL3	SEL2	SEL1	SEL0	PSEL	EN	LD	U/D	EN	LD	U/D	MX1	MX2
253	252	251	250	249	248	247	246	245	244	243	242	241

INPUT SOURCE					ROW COUNTER			COLUMN COUNTER			REGS	
SEL3	SEL2	SEL1	SEL0	PSEL	EN	LD	U/D	EN	LD	U/D	MX1	MX2
253	252	251	250	249	248	247	246	245	244	243	242	241

TWO DIMENSIONAL COUNTER #5

INPUT SOURCE					R	C						
SEL3	SEL2	SEL1	SEL0	PSEL	EN							
236	235	234	233	232	231							

INPUT SOURCE					R	C						
SEL3	SEL2	SEL1	SEL0	PSEL	EN							
236	235	234	233	232	231							

PROCESSOR BO

MULTIPLIER #1

X SOURCE					X CTRL			Y SOURCE					Y CTRL		INSTRUCTION							Z PORT		
SEL3	SEL2	SEL1	SEL0	PSEL	XSEL	MS/LS	XEN	SEL3	SEL2	SEL1	SEL0	PSEL	MS/LS	YEN	I7	I6	I5	I4	I3	I2	I1	I0	ZEN	MSWSEL	MSWEN
191	190	189	188	187	186	185	184	183	182	181	180	179	178	177	176	175	174	173	172	171	170	169	168	167	166

X SOURCE					X CTRL			Y SOURCE					Y CTRL		INSTRUCTION							Z PORT		
SEL3	SEL2	SEL1	SEL0	PSEL	XSEL	MS/LS	XEN	SEL3	SEL2	SEL1	SEL0	PSEL	MS/LS	YEN	I7	I6	I5	I4	I3	I2	I1	I0	ZEN	MSWSEL	MSWEN
191	190	189	188	187	186	185	184	183	182	181	180	179	178	177	176	175	174	173	172	171	170	169	168	167	166

MULTIPLIER #2

X SOURCE					XCTRL			Y SOURCE					Y CTRL		INSTRUCTION							Z PORT		
SEL3	SEL2	SEL1	SEL0	PSEL	XSEL	MS/LS	XEN	SEL3	SEL2	SEL1	SEL0	PSEL	MS/LS	YEN	I7	I6	I5	I4	I3	I2	I1	I0	ZEN	MSWSEL	MSWEN
165	164	163	162	161	160	159	158	157	156	155	154	153	152	151	150	149	148	147	146	145	144	143	142	141	140

X SOURCE					X CTRL			Y SOUR					Y CTRL		INSTRUCTION							Z PORT		
SEL3	SEL2	SEL1	SEL0	PSEL	XSEL	MS/LS	XEN	SEL3	SEL2	SEL1	SEL0	PSEL	MS/LS	YEN	I7	I6	I5	I4	I3	I2	I1	I0	ZEN	MSWSEL	MSWEN
165	164	163	162	161	160	159	158	157	156	155	154	153	152	151	150	149	148	147	146	145	144	143	142	141	140

DATA REGISTER FILE (EN = 1)
36-BIT IMMEDIATE DATA FIELDS (EN = 0)

PORT A WRITE SOURCE				PORT A WRITE ADDRESS					PORT B WRITE SOURCE				PORT B WRITE ADDRESS					SHIFT CTRL		PORT A READ ADDRESS					PORT B READ ADDRESS				
SEL3	SEL2	SEL1	SEL0	VRA5	VRA4	VRA3	VRA2	VRA1	VRA0	SEL3	SEL2	SEL1	SEL0	VRB5	VRB4	VRB3	VRB2	VRB1	VRB0	SM1	SM0	RDA5	RDA4	RDA3	RDA2	RDA1	RDA0	RDB5	RDB4	RDB3	RDB2	RDB1	RDB0
85	84	83	82	81	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50

PORT A WRITE SOURCE				PORT A WRITE ADDRESS					PORT B WRITE SOURCE				PORT B WRITE ADDRESS					SHIFT CTRL		PORT A READ ADDRESS					PORT B READ ADDRESS				
SEL3	SEL2	SEL1	SEL0	VRA5	VRA4	VRA3	VRA2	VRA1	VRA0	SEL3	SEL2	SEL1	SEL0	VRB5	VRB4	VRB3	VRB2	VRB1	VRB0	SM1	SM0	RDA5	RDA4	RDA3	RDA2	RDA1	RDA0	RDB5	RDB4	RDB3	RDB2	RDB1	RDB0
85	84	83	82	81	80	79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50

IMMEDIATE DATA	MEMORY SOURCE											
EN	PORT C (REAL)											
49	SEL3	SEL2	SEL1	SEL0	PSEL	MSWSEL						
48	47	46	45	44	43							

IMMEDIATE DATA	MEMORY SOURCE											
EN	PORT C (REAL)											
49	SEL3	SEL2	SEL1	SEL0	PSEL	MSWSEL						
48	47	46	45	44	43							

NERATOR BOARD

Group	Field (upper row)	Upper row	Field (lower row)	Lower row	Bit
MICROSEQUENCER	BRANCH ADDRESS	BA13			365
		BA12			364
		BA11			363
		BA10			362
		BA09			361
		BA08			360
		BA07			359
		BA06			358
		BA05			357
		BA04			356
		BA03			355
		BA02			354
		BA01			353
		BA00			352
	INSTRUCTION	IA7	CONDITION SELECT	SEL6	351
		IA6		SEL5	350
		IA5		SEL4	349
		IA4		SEL3	348
		IA3		SEL2	347
		IA2		SEL1	346
		IA1		SEL0	345
		IA0		POL	344
MEMORY CONTROL	CACHE WRITE	[illegible]	CACHE READ	RDA	343
		[illegible]		RDB	342
	AUX WRITE	[illegible]	AUX READ	[illegible]	341
		[illegible]		[illegible]	340
IMMEDIATE DATA FIELDS					339
					338
					337
					336
					208
					207
					206
					205
					257
					256
					223
					222
					240
					239
					238
					237
?		MX2		MX2	326
		MX1		MX1	325
		MX0		MX0	324
ADDRESS RAM #1	ADDRESS	SEL3	ADDRESS	SEL3	321
		SEL2		SEL2	320
		SEL1		SEL1	287
		SEL0		SEL0	286
		PSEL		PSEL	285
	DATA	SEL3	DATA	SEL3	284
		SEL2		SEL2	283
		SEL1		SEL1	282
		SEL0		SEL0	281
		PSEL		PSEL	280
		R/W		R/W	279

Group	Field (upper row)	Upper row	Field (lower row)	Lower row	Bit
DIMENSIONAL OUNTER #4		PSEL		PSEL	249
	ROW COUNTER	EN	ROW COUNTER	EN	248
		LD		LD	247
		U/D		U/D	246
	COLUMN COUNTER	EN	COLUMN COUNTER	EN	245
		LD		LD	244
		U/D		U/D	243
	REGS	MX1	REGS	MX1	242
		MX2		MX2	241
TWO DIMENSIONAL COUNTER #3	INPUT SOURCE	SEL3	INPUT SOURCE	SEL3	236
		SEL2		SEL2	235
		SEL1		SEL1	234
		SEL0		SEL0	233
		PSEL		PSEL	232
	ROW COUNTER	EN	ROW COUNTER	EN	231
		LD		LD	230
		U/D		U/D	229
	COLUMN COUNTER	EN	COLUMN COUNTER	EN	228
		LD		LD	227
		U/D		U/D	226
	REGS	MX1	REGS	MX1	225
		MX2		MX2	224
TWO DIMENSIONAL COUNTER #2	INPUT SOURCE	SEL3	INPUT SOURCE	SEL3	221
		SEL2		SEL2	220
		SEL1		SEL1	219
		SEL0		SEL0	218
		PSEL		PSEL	217
	ROW COUNTER	EN	ROW COUNTER	EN	216
		LD		LD	215
		U/D		U/D	214
	COLUMN COUNTER	EN	COLUMN COUNTER	EN	213
		LD		LD	212
		U/D		U/D	211
	REGS	MX1	REGS	MX1	210
		MX2		MX2	209
TWO DIMENSIONAL COUNTER #1	INPUT SOURCE	SEL3	INPUT SOURCE	SEL3	204
		SEL2		SEL2	203
		SEL1		SEL1	202
		SEL0		SEL0	201
		PSEL		PSEL	200
	ROW COUNTER	EN	ROW COUNTER	EN	199
		LD		LD	198
		U/D		U/D	197
	COLUMN COUNTER	EN	COLUMN COUNTER	EN	196
		LD		LD	195
		U/D		U/D	194
	REGS	MX1	REGS	MX1	193
		MX2		MX2	192
UNDEFINED					335
					334
					333
					332
					331
					330
		[illegible]		[illegible]	329
		[illegible]		[illegible]	328
		[illegible]		[illegible]	327
		[illegible]		[illegible]	254

SSOR BOARD

Group	Field (upper row)	Upper row	Field (lower row)	Lower row	Bit
		I1		I1	144
		I0		I0	143
	Z PORT	ZEN	Z PORT	ZEN	142
		MSVSEL		MSVSEL	141
		MSVEN		MSVEN	140
ALU #1	X SOURCE	SEL3	X SOURCE	SEL3	139
		SEL2		SEL2	138
		SEL1		SEL1	137
		SEL0		SEL0	136
		PSEL		PSEL	135
	X CTRL	XSEL	X CTRL	XSEL	134
		MS/LS		MS/LS	133
		XEN		XEN	132
	Y SOURCE	SEL3	Y SOURCE	SEL3	131
		SEL2		SEL2	130
		SEL1		SEL1	129
		SEL0		SEL0	128
		PSEL		PSEL	127
	Y CTRL	MS/LS	Y CTRL	MS/LS	126
		YEN		YEN	125
	YSEL	YSEL		YSEL	124
	INSTRUCTION	I7	INSTRUCTION	I7	123
		I6		I6	122
		I5		I5	121
		I4		I4	120
		I3		I3	119
		I2		I2	118
		I1		I1	117
		I0		I0	116
	Z PORT	ZEN	Z PORT	ZEN	115
		MSVSEL		MSVSEL	114
		MSVEN		MSVEN	113
ALU #2	X SOURCE	SEL3	X SOURCE	SEL3	112
		SEL2		SEL2	111
		SEL1		SEL1	110
		SEL0		SEL0	109
		PSEL		PSEL	108
	X CTRL	XSEL	X CTRL	XSEL	107
		MS/LS		MS/LS	106
		XEN		XEN	105
	Y SOURCE	SEL3	Y SOURCE	SEL3	104
		SEL2		SEL2	103
		SEL1		SEL1	102
		SEL0		SEL0	101
		PSEL		PSEL	100
	Y CTRL	MS/LS	Y CTRL	MS/LS	99
		YEN		YEN	98
	YSEL	YSEL		YSEL	97
	INSTRUCTION	I7	INSTRUCTION	I7	96
		I6		I6	95
		I5		I5	94
		I4		I4	93
		I3		I3	92
		I2		I2	91
		I1		I1	90
		I0		I0	89
	Z PORT	ZEN	Z PORT	ZEN	88
		MSVSEL		MSVSEL	87
		MSVEN		MSVEN	86

Group	Field (upper row)	Upper row	Field (lower row)	Lower row	Bit
= 0)	A ADDRESS	RDA2	A ADDRESS	RDA2	58
		RDA1		RDA1	57
		RDA0		RDA0	56
	PORT B READ ADDRESS	RDB5	PORT B READ ADDRESS	RDB5	55
		RDB4		RDB4	54
		RDB3		RDB3	53
		RDB2		RDB2	52
		RDB1		RDB1	51
		RDB0		RDB0	50
IMMEDIATE DATA		EN		EN	49
MEMORY WRITE SOURCE SELECT	PORT C (REAL)	SEL3	PORT C (REAL)	SEL3	48
		SEL2		SEL2	47
		SEL1		SEL1	46
		SEL0		SEL0	45
		PSEL		PSEL	44
		MSVSEL		MSVSEL	43
	PORT D (IMAGINARY)	SEL3	PORT D (IMAGINARY)	SEL3	42
		SEL2		SEL2	41
		SEL1		SEL1	40
		SEL0		SEL0	39
		PSEL		PSEL	38
		MSVSEL		MSVSEL	37
ADDDRESS PORTS	PORT C	SEL3	PORT A	SEL3	36
		SEL2		SEL2	35
		SEL1		SEL1	34
		SEL0		SEL0	33
		PSEL		PSEL	32
		MS/LS		MS/LS	31
	PORT D	SEL3	PORT B	SEL3	30
		SEL2		SEL2	29
		SEL1		SEL1	28
		SEL0		SEL0	27
		PSEL		PSEL	26
		MS/LS		MS/LS	25
I/O PORT	REAL	SEL3	REAL	SEL3	24
		SEL2		SEL2	23
		SEL1		SEL1	22
		SEL0		SEL0	21
		PSEL		PSEL	20
		MS/LS		MS/LS	19
	IMAGINARY	SEL3	IMAGINARY	SEL3	18
		SEL2		SEL2	17
		SEL1		SEL1	16
		SEL0		SEL0	15
		PSEL		PSEL	14
		MS/LS		MS/LS	13
					12
					11
					10
					9
					8
					7
					6
					5
					4
					3
					2
					1
					0

APPENDIX E

CPH DEFINITION FILE

```
1 -         /* PRELIMINARY MICRO ASM DEFINITION FOR CPH */
2 - WIDTH  = 768
3 - PHASES = 1
4 - DEFBIT = 0
5 -
6 - /*      ====================================
7 -         = MICROPROGRAM SEQUENCER (FIELD SEQ) =
8 -         ====================================
9 - */
10 -
11 -    SEQ[24]
12 -    {
13 - DEFAULT = 0X00007F
14 -        /* BRANCH ADDRESS */
15 -        BRA[16]
16 -        {
17 -            DEFAULT = 0X0000
18 - LABEL
19 -        }
20 -
21 -        /* INSTRUCTION */
22 -        INS[8]
23 -        {
24 -            DEFAULT     = 0X7F /* CONTINUE */
25 -            CONT        = 0X7F /* CONTINUE */
26 -            LDLC        = 0X7E /* LOAD LOOP COUNTER */
27 -            LDSP        = 0X7D /* LOAD STACK POINTER */
28 -            LDSRP       = 0X7C /* LOAD SUBROUTINE RAM POINTER */
29 -            LDSUBR      = 0X7E /* LOAD SUBROUTINE RAM */
30 -            SIM         = 0X7A /* SET INTERRUPT MASK BITS */
31 -            RIM         = 0X79 /* RESET INTERRUPT MASK BITS */
32 -            RINT        = 0X78 /* RESETS INTERRUPTS */
33 -            JI          = 0X77 /* JUMP IMMEDIATE */
34 -            JIC         = 0X76 /* CONDITIONAL JUMP IMMEDIATE */
35 -            JR          = 0X75 /* JUMP RELATIVE */
36 -            JRC         = 0X74 /* CONDITIONAL JUMP RELATIVE */
37 -            LI          = 0X73 /* LOOP IMMEDIATE */
38 -            LR          = 0X72 /* LOOP RELATIVE */
39 -            LS          = 0X71 /* LOOP TOP OF STACK */
40 -            TWBI        = 0X70 /* IMMEDIATE THREE WAY BRANCH */
41 -            TWBR        = 0X6F /* RELATIVE THREE WAY BRANCH */
42 -            CALL        = 0X6E /* CALL SUBROUTINE */
43 -            CALLC       = 0X6D /* CONDITIONAL CALL SUBROUTINE */
44 -            RET         = 0X6C /* RETURN FROM SUBROUTINE */
45 -            RETC        = 0X6B /* CONDITIONAL RETURN FROM SUBROUTINE */
46 -            PUSH        = 0X6A /* PUSH STACK */
47 -            PUSHC       = 0X69 /* CONDITIONAL PUSH STACK */
48 -            PLDLC       = 0X68 /* PUSH STACK AND LOAD COUNTER */
49 -            PLDLCC      = 0X67 /* PUSH STACK AND CONDITIONALY LOAD COUNTER */
50 -            POP         = 0X66 /* POP STACK */
51 -            POPC        = 0X65 /* CONDITIONAL POP STACK */
52 -            EI          = 0X64 /* ENABLE ALL UNMASKED INTERRUPTS */
53 -            DI          = 0X63 /* DISABLE ALL INTERRUPTS */
54 -        }
55 -    }
56 -
57 - /*      ====================================
58 -         = CONDITION CODE SELECT (FIELD CCS) =
59 -         ====================================
60 - */
61 -
62 -    CCS[8]
63 -    {
64 - DEFAULT = 0B11111111
65 -        /* SELECT */
66 -        SEL[7]
67 -        {
68 -            DEFAULT     = 0B1111111
69 -
70 -                /* FLAGS FOR MULTIPLIER #1 */
71 -            M1INT       = 0B0010000 /* INTERUPT */
72 -            M1PE        = 0B0010001 /* PARITY ERROR */
73 -            M1N         = 0B0010010 /* NEGATIVE */
74 -            M1ZR        = 0B0010011 /* ZERO */
75 -            M1OV        = 0B0010100 /* OVERFLOW */
76 -            M1UF        = 0B0010101 /* UNDERFLOW */
77 -            M1INX       = 0B0010110 /* INEXACT */
78 -            M1INV       = 0B0010111 /* INVALID OPERATION */
79 -            M1NAN       = 0B0011000 /* NOT A NUMBER */
80 -            M1RND       = 0B0011001 /* ROUND UP */
81 -            M1DEN       = 0B0011010 /* DENORMALIZED */
82 -            M1DIVZ      = 0B0011011 /* DIVIDE BY ZERO */
83 -
84 -                /* FLAGS FOR MULTIPLIER #2 */
85 -            M2INT       = 0B0011100 /* INTERUPT */
86 -            M2PE        = 0B0011101 /* PARITY ERROR */
87 -            M2N         = 0B0011110 /* NEGATIVE */
88 -            M2ZR        = 0B0011111 /* ZERO */
89 -            M2OV        = 0B0100000 /* OVERFLOW */
90 -            M2UF        = 0B0100001 /* UNDERFLOW */
91 -            M2INX       = 0B0100010 /* INEXACT */
92 -            M2INV       = 0B0100011 /* INVALID OPERATION */
93 -            M2NAN       = 0B0100100 /* NOT A NUMBER */
94 -            M2RND       = 0B0100101 /* ROUND UP */
95 -            M2DEN       = 0B0100110 /* DENORMALIZED */
96 -            M2DIVZ      = 0B0100111 /* DIVIDE BY ZERO */
97 -
98 -                /* FLAGS FOR ALU #1 */
99 -            A1INT       = 0B0101000 /* INTERUPT */
100 -           A1PE        = 0B0101001 /* PARITY ERROR */
101 -           A1N         = 0B0101010 /* NEGATIVE */
102 -           A1ZR        = 0B0101011 /* ZERO */
103 -           A1OV        = 0B0101100 /* OVERFLOW */
104 -           A1UF        = 0B0101101 /* UNDERFLOW */
105 -           A1INX       = 0B0101110 /* INEXACT */
106 -           A1INV       = 0B0101111 /* INVALID OPERATION */
107 -           A1NAN       = 0B0110000 /* NOT A NUMBER */
108 -           A1RND       = 0B0110001 /* ROUND UP */
109 -           A1DEN       = 0B0110010 /* DENORMALIZED */
```

```
110 -               A1CRY       = 0B0110011 /* CARRY OUT */
111 -
112 -                   /* FLAGS FOR ALU #2 */
113 -               A2INT       = 0B0110100 /* INTERUPT */
114 -               A2PE        = 0B0110101 /* PARITY ERROR */
115 -               A2N         = 0B0110110 /* NEGATIVE */
116 -               A2ZR        = 0B0110111 /* ZERO */
117 -               A2OV        = 0B0111000 /* OVERFLOW */
118 -               A2UF        = 0B0111001 /* UNDERFLOW */
119 -               A2INX       = 0B0111010 /* INEXACT */
120 -               A2INV       = 0B0111011 /* INVALID OPERATION */
121 -               A2NAN       = 0B0111100 /* NOT A NUMBER */
122 -               A2RND       = 0B0111101 /* ROUND UP */
123 -               A2DEN       = 0B0111110 /* DENORMALIZED */
124 -               A2CRY       = 0B0111111 /* CARRY OUT */
125 -           }
126 -           /* POLARITY */
127 -           PSEL[1]
128 -           {
129 -               DEFAULT     = 0B0
130 -           }
131 -       }
132 -
133 - /*      ===================================
134 -         = IMMEDIATE ADDRESS (FIELD IMMADD) =
135 -         ===================================
136 - */
137 -
138 -       IMMADD[16]
139 -       {
140 -           DEFAULT = 0X0000
141 - /*must have a subfield for normal syntax.  IMMADD = value fails. */
142 - IMMVALUE[16]
143 - {
144 -           DEFAULT = 0X0000
145 - }
146 -       }
147 -
148 - /*      ================================================
149 -         = TWO DIMENSIONAL COUNTERS (FIELDS CNT1-CNT4) =
150 -         ================================================
151 - */
152 -
153 -       CNT1[13]
154 -       CNT2[13]
155 -       CNT3[13]
156 -       CNT4[13]
157 -       {
158 -       DEFAULT = 0B0000011011011
159 -           /* INPUT PORT SOURCE SELECT */
160 -           SEL[4]
161 -           {
162 -               DEFAULT     = 0X0 /* HOLD (REGISTER SELECTS ITS SELF) */
163 -               HOLD        = 0X0 /* HOLD (REGISTER SELECTS ITS SELF) */
164 -               CNT1        = 0X3 /* SELECT COUNTER #1 ADDRESS OUTPUT */
165 -               CNT2        = 0X5 /* SELECT COUNTER #2 ADDRESS OUTPUT */
166 -               CNT3        = 0X8 /* SELECT COUNTER #3 ADDRESS OUTPUT */
167 -               CNT4        = 0XA /* SELECT COUNTER #4 ADDRESS OUTPUT */
168 -               RC1         = 0X2 /* SELECT COUNTER #1 ROW/COL OUTPUT */
169 -               RC2         = 0X4 /* SELECT COUNTER #2 ROW/COL OUTPUT */
170 -               RC3         = 0X7 /* SELECT COUNTER #3 ROW/COL OUTPUT */
171 -               RC4         = 0X9 /* SELECT COUNTER #4 ROW/COL OUTPUT */
172 -               IMM         = 0X6 /* SELECT IMMEDIATE ADDRESS */
173 -               IOPORT      = 0X1 /* SELECT IO ADDRESS PORT */
174 -               RAM1        = 0XB /* SELECT ADDRESS RAM #1 */
175 -               RAM2        = 0XC /* SELECT ADDRESS RAM #2 */
176 -               REGA        = 0XD /* SELECT REGISTER FILE PORT A */
177 -               REGB        = 0XE /* SELECT REGISTER FILE PORT B */
178 -               FFT         = 0XF /* SELECT FFT ADDRESS SEQUENCER */
179 -           }
180 -
181 -           /* INPUT PORT PHASE SELECT */
182 -           PSEL[1]
183 -           {
184 -               DEFAULT     = 0B0 /* PHASE 0 */
185 -               P0          = 0B0 /* PHASE 0 */
186 -               P1          = 0B1 /* PHASE 1 */
187 -           }
188 -
189 -           /* ROW AND COLUMN COUNTERS */
190 -           ROW[3]
191 -           COL[3]
192 -           {
193 -               DEFAULT     = 0B110 /* NO OPERATION */
194 -               NOP         = 0B110 /* NO OPERATION */
195 -               CLEAR       = 0B000 /* RESET COUNTER */
196 -               INC         = 0B010 /* INCREMENT */
197 -               DEC         = 0B011 /* DECREMENT */
198 -               LOAD        = 0B100 /* LOAD */
199 -           }
200 -
201 -           /* DIMENSION AND OFFSET REGISTERS */
202 -           REGS[2]
203 -           {
204 -               DEFAULT     = 0B11 /* NO OPERATION */
205 -               NOP         = 0B11 /* NO OPERATION */
206 -               DIM         = 0B01 /* LOAD DIMENSION REGISTER */
207 -               OFF         = 0B10 /* LOAD OFFSET REGISTER */
208 -               DIMOFF      = 0B00 /* LOAD BOTH REGISTERS */
209 -           }
210 -       }
211 -
212 - /*      ======================================
213 -         = ADDRESS REGISTER FILE (FIELD AREG) =
214 -         ======================================
215 - */
216 -
217 -       AREG[36]
218 -       {
```

global variables to pass data back and forth. Should be compiled with the standard C compiler and linked with our object code. Under

```
219 - DEFAULT = 0X080100000
220 -         /* PORT A INPUT SOURCE */
221 -         ASEL[5]
222 -         {
223 -             DEFAULT     = 0B00001 /* NO OPERATION */
224 -             NOP         = 0B00001 /* NO OPERATION */
225 -             CLEAR       = 0B00000 /* CLEAR REGISTER (ALL ZEROS) */
226 -             CNT1        = 0B00110 /* SELECT COUNTER #1 ADDRESS OUTPUT */
227 -             CNT2        = 0B01010 /* SELECT COUNTER #2 ADDRESS OUTPUT */
228 -             CNT3        = 0B10000 /* SELECT COUNTER #3 ADDRESS OUTPUT */
229 -             CNT4        = 0B10100 /* SELECT COUNTER #4 ADDRESS OUTPUT */
230 -             RC1         = 0B00100 /* SELECT COUNTER #1 ROW/COL OUTPUT */
231 -             RC2         = 0B01000 /* SELECT COUNTER #2 ROW/COL OUTPUT */
232 -             RC3         = 0B01110 /* SELECT COUNTER #3 ROW/COL OUTPUT */
233 -             RC4         = 0B10010 /* SELECT COUNTER #4 ROW/COL OUTPUT */
234 -             IMM         = 0B01100 /* SELECT IMMEDIATE ADDRESS */
235 -             IOPORT      = 0B00010 /* SELECT IO ADDRESS PORT */
236 -             RAM1        = 0B10110 /* SELECT ADDRESS RAM #1 */
237 -             RAM2        = 0B11000 /* SELECT ADDRESS RAM #2 */
238 -             REGA        = 0B11010 /* SELECT REGISTER FILE PORT A */
239 -             REGB        = 0B11100 /* SELECT REGISTER FILE PORT B */
240 -             SET         = 0B11110 /* SET REGISTER (ALL ONES) */
241 -         }
242 -
243 -         /* PORT A WRITE ADDRESS */
244 -         WRA[6]
245 -         {
246 -             DEFAULT     = 0B000000 /* REGISTER 0 */
247 -         }
248 -
249 -         /* PORT B INPUT SOURCE */
250 -         BSEL[5]
251 -         {
252 -             DEFAULT     = 0B00001 /* NO OPERATION */
253 -             NOP         = 0B00001 /* NO OPERATION */
254 -             CLEAR       = 0B00000 /* CLEAR REGISTER (ALL ZEROS) */
255 -             CNT1        = 0B00110 /* SELECT COUNTER #1 ADDRESS OUTPUT */
256 -             CNT2        = 0B01010 /* SELECT COUNTER #2 ADDRESS OUTPUT */
257 -             CNT3        = 0B10000 /* SELECT COUNTER #3 ADDRESS OUTPUT */
258 -             CNT4        = 0B10100 /* SELECT COUNTER #4 ADDRESS OUTPUT */
259 -             RC1         = 0B00100 /* SELECT COUNTER #1 ROW/COL OUTPUT */
260 -             RC2         = 0B01000 /* SELECT COUNTER #2 ROW/COL OUTPUT */
261 -             RC3         = 0B01110 /* SELECT COUNTER #3 ROW/COL OUTPUT */
262 -             RC4         = 0B10010 /* SELECT COUNTER #4 ROW/COL OUTPUT */
263 -             IMM         = 0B01100 /* SELECT IMMEDIATE ADDRESS */
264 -             IOPORT      = 0B00010 /* SELECT IO ADDRESS PORT */
265 -             RAM1        = 0B10110 /* SELECT ADDRESS RAM #1 */
266 -             RAM2        = 0B11000 /* SELECT ADDRESS RAM #2 */
267 -             REGA        = 0B11010 /* SELECT REGISTER FILE PORT A */
268 -             REGB        = 0B11100 /* SELECT REGISTER FILE PORT B */
269 -             SET         = 0B11110 /* SET REGISTER (ALL ONES) */
270 -         }
271 -
272 -         /* PORT B WRITE ADDRESS */
273 -         WRB[6]
274 -         {
275 -             DEFAULT     = 0B000000 /* REGISTER 0 */
276 -         }
277 -
278 -         /* SHIFT MODE CONTROL */
279 -         SMODE[2]
280 -         {
281 -             DEFAULT     = 0B00 /* NORMAL (REGISTER FILE MODE) */
282 -             REG         = 0B00 /* NORMAL (REGISTER FILE MODE) */
283 -             R8X8         = 0B01 /* 8 BY 8 SHIFT REGISTER MODE */
284 -             R4X16        = 0B10 /* 4 BY 16 SHIFT REGISTER MODE */
285 -             R2X32        = 0B11 /* 2 BY 32 SHIFT REGISTER MODE */
286 -         }
287 -
288 -         /* PORT A READ ADDRESS */
289 -         RDA[6]
290 -         {
291 -             DEFAULT     = 0B000000 /* REGISTER 0 */
292 -         }
293 -
294 -         /* PORT B READ ADDRESS */
295 -         RDB[6]
296 -         {
297 -             DEFAULT     = 0B000000 /* REGISTER 0 */
298 -         }
299 -     }
300 -
301 - /*      ================================          ================================
302 -         = ADDRESS RAM ONE (FIELD RAM1) =  -AND-  = ADDRESS RAM TWO (FIELD RAM2) =
303 -         ================================          ================================
304 - */
305 -
306 -     RAM1[11]
307 -     RAM2[11]
308 -     {
309 - DEFAULT = 0B000000111101
310 -         /* CROSSBAR REGISTER SOURCE SELECT */
311 -         ADD[4]
312 -         {
313 -             DEFAULT     = 0X0 /* HOLD (REGISTER SELECTS ITS SELF) */
314 -             HOLD        = 0X0 /* HOLD (REGISTER SELECTS ITS SELF) */
315 -             CNT1        = 0X3 /* SELECT COUNTER #1 ADDRESS OUTPUT */
316 -             CNT2        = 0X5 /* SELECT COUNTER #2 ADDRESS OUTPUT */
317 -             CNT3        = 0X8 /* SELECT COUNTER #3 ADDRESS OUTPUT */
318 -             CNT4        = 0XA /* SELECT COUNTER #4 ADDRESS OUTPUT */
319 -             RC1         = 0X2 /* SELECT COUNTER #1 ROW/COL OUTPUT */
320 -             RC2         = 0X4 /* SELECT COUNTER #2 ROW/COL OUTPUT */
321 -             RC3         = 0X7 /* SELECT COUNTER #3 ROW/COL OUTPUT */
322 -             RC4         = 0X9 /* SELECT COUNTER #4 ROW/COL OUTPUT */
323 -             IMM         = 0X6 /* SELECT IMMEDIATE ADDRESS */
324 -             IOPORT      = 0X1 /* SELECT IO ADDRESS PORT */
325 -             RAM1        = 0XB /* SELECT ADDRESS RAM #1 */
326 -             RAM2        = 0XC /* SELECT ADDRESS RAM #2 */
327 -             REGA        = 0XD /* SELECT REGISTER FILE PORT A */
```

```
328 -               REGB        = 0XE /* SELECT REGISTER FILE PORT B */
329 -               DIS         = 0XF /* DISABLE PORT (REGISTER HOLDS) */
330 -           }
331 -
332 -           /* ADDRESS PORT PHASE SELECT */
333 -           ACTRL[1]
334 -           {
335 -               DEFAULT     = 0B0 /* PHASE 0 */
336 -               P0          = 0B0 /* PHASE 0 */
337 -               P1          = 0B1 /* PHASE 1 */
338 -           }
339 -
340 -           /* CROSSBAR REGISTER SOURCE SELECT */
341 -           DATA[4]
342 -           {
343 -               DEFAULT     = 0XF /* DISABLE PORT (REGISTER HOLDS) */
344 -               HOLD        = 0X0 /* HOLD (REGISTER SELECTS ITS SELF) */
345 -               CNT1        = 0X3 /* SELECT COUNTER #1 ADDRESS OUTPUT */
346 -               CNT2        = 0X5 /* SELECT COUNTER #2 ADDRESS OUTPUT */
347 -               CNT3        = 0X8 /* SELECT COUNTER #3 ADDRESS OUTPUT */
348 -               CNT4        = 0XA /* SELECT COUNTER #4 ADDRESS OUTPUT */
349 -               RC1         = 0X2 /* SELECT COUNTER #1 ROW/COL OUTPUT */
350 -               RC2         = 0X4 /* SELECT COUNTER #2 ROW/COL OUTPUT */
351 -               RC3         = 0X7 /* SELECT COUNTER #3 ROW/COL OUTPUT */
352 -               RC4         = 0X9 /* SELECT COUNTER #4 ROW/COL OUTPUT */
353 -               IMM         = 0X6 /* SELECT IMMEDIATE ADDRESS */
354 -               IOPORT      = 0X1 /* SELECT IO ADDRESS PORT */
355 -               RAM1        = 0XB /* SELECT ADDRESS RAM #1 */
356 -               RAM2        = 0XC /* SELECT ADDRESS RAM #2 */
357 -               REGA        = 0XD /* SELECT REGISTER FILE PORT A */
358 -               REGB        = 0XE /* SELECT REGISTER FILE PORT B */
359 -               DIS         = 0XF /* DISABLE PORT (REGISTER HOLDS) */
360 -           }
361 -
362 -           /* DATA PORT CONTROL */
363 -           DCTRL[2]
364 -           {
365 -               DEFAULT     = 0B01 /* READ RAM */
366 -               READ        = 0B01 /* READ RAM */
367 -               WRP0        = 0B00 /* WRITE RAM FROM PHASE 0 */
368 -               WRP1        = 0B10 /* WRITE RAM FROM PHASE 1 */
369 -           }
370 -       }
371 -
372 - /*      ========================================
373 -         = ADDRESS PORTS A-E (FIELDS ADDRA-ADDRE) =
374 -         ========================================
375 - */
376 -       BANK[5]
377 -       ADDRA[5]
378 -       ADDRB[5]
379 -       ADDRC[5]
380 -       ADDRD[5]
381 -       ADDRE[5]
382 -       {
383 - DEFAULT = 0B000000
384 -           /* CROSSBAR REGISTER SOURCE SELECT */
385 -           SEL[4]
386 -           {
387 -               DEFAULT     = 0X0 /* HOLD (REGISTER SELECTS ITS SELF) */
388 -               HOLD        = 0X0 /* HOLD (REGISTER SELECTS ITS SELF) */
389 -               CNT1        = 0X3 /* SELECT COUNTER #1 ADDRESS OUTPUT */
390 -               CNT2        = 0X5 /* SELECT COUNTER #2 ADDRESS OUTPUT */
391 -               CNT3        = 0X8 /* SELECT COUNTER #3 ADDRESS OUTPUT */
392 -               CNT4        = 0XA /* SELECT COUNTER #4 ADDRESS OUTPUT */
393 -               RC1         = 0X2 /* SELECT COUNTER #1 ROW/COL OUTPUT */
394 -               RC2         = 0X4 /* SELECT COUNTER #2 ROW/COL OUTPUT */
395 -               RC3         = 0X7 /* SELECT COUNTER #3 ROW/COL OUTPUT */
396 -               RC4         = 0X9 /* SELECT COUNTER #4 ROW/COL OUTPUT */
397 -               IMM         = 0X6 /* SELECT IMMEDIATE ADDRESS */
398 -               IOPORT      = 0X1 /* SELECT IO ADDRESS PORT */
399 -               RAM1        = 0XB /* SELECT ADDRESS RAM #1 */
400 -               RAM2        = 0XC /* SELECT ADDRESS RAM #2 */
401 -               REGA        = 0XD /* SELECT REGISTER FILE PORT A */
402 -               REGB        = 0XE /* SELECT REGISTER FILE PORT B */
403 -               DIS         = 0XF /* DISABLE PORT (REGISTER HOLDS) */
404 -           }
405 -
406 -           /* ADDRESS PORT PHASE SELECT */
407 -           PSEL[1]
408 -           {
409 -               DEFAULT     = 0B0 /* PHASE 0 */
410 -               P0          = 0B0 /* PHASE 0 */
411 -               P1          = 0B1 /* PHASE 1 */
412 -           }
413 -       }
414 -
415 - /*      ======================================
416 -         = CACHE MEMORY CONTROL (FIELD CACHE) =
417 -         ======================================
418 - */
419 -
420 -       CACHE[4]
421 -       {
422 - DEFAULT = 0B1111
423 -           /* CACHE WRITE INSTRUCTIONS */
424 -           WRITE[2]
425 -           {
426 -               DEFAULT         = 0B11 /* NO OPERATION */
427 -               WRR             = 0B01 /* WRITE REAL */
428 -               WRI             = 0B10 /* WRITE IMAGINARY */
429 -               WRRI            = 0B00 /* WRITE REAL AND IMAGINARY */
430 -           }
431 -
432 -           /* CACHE READ INSTRUCTIONS */
433 -           READ[2]
434 -           {
435 -               DEFAULT         = 0B11 /* NO OPERATION */
436 -               RDA             = 0B01 /* READ BANK A */
```

```
437 -             RDB             = 0B10 /* READ BANK B */
438 -             RDAB            = 0B00 /* READ BANKS A AND B */
439 -         }
440 -     }
441 -
442 - /*      ======================================
443 -         = AUXILIARY MEMORY CONTROL (FIELD AUX) =
444 -         ======================================
445 - */
446 -
447 -     AUX[4]
448 -     {
449 - DEFAULT = 0B1111
450 -         /* AUXILIARY WRITE INSTRUCTIONS */
451 -         WRITE[2]
452 -         {
453 -             DEFAULT         = 0B11 /* NO OPERATION */
454 -             WRR             = 0B01 /* WRITE REAL */
455 -             WRI             = 0B10 /* WRITE IMAGINARY */
456 -             WRRI            = 0B00 /* WRITE REAL AND IMAGINARY */
457 -         }
458 -
459 -         /* AUXILIARY READ INSTRUCTIONS */
460 -         READ[2]
461 -         {
462 -             DEFAULT         = 0B11 /* NO OPERATION */
463 -             RDR             = 0B01 /* READ REAL */
464 -             RDI             = 0B10 /* READ IMAGINARY */
465 -             RDRI            = 0B00 /* READ REAL AND IMAGINARY */
466 -         }
467 -     }
468 -
469 - /*      ============================          ============================
470 -         = MULTIPLIER ONE (FIELD M1) =  -AND-  = MULTIPLIER TWO (FIELD M2) =
471 -         ============================          ============================
472 - */
473 -
474 -     M1[26]
475 -     M2[26]
476 -     {
477 - DEFAULT = 0B00000001000000101011000100
478 -         /* CROSSBAR REGISTER SOURCE SELECT */
479 -         XSEL[4]
480 -         {
481 -             DEFAULT     = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
482 -             HOLD        = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
483 -             M1          = 0XA /* SELECT MULTIPLIER #1 */
484 -             M2          = 0X9 /* SELECT MULTIPLIER #2 */
485 -             A1          = 0X1 /* SELECT ALU #1 */
486 -             A2          = 0X2 /* SELECT ALU #2 */
487 -             AR          = 0X3 /* SELECT CACHE PORT A REAL */
488 -             AI          = 0X4 /* SELECT CACHE PORT A IMAGINARY */
489 -             BR          = 0X5 /* SELECT CACHE PORT B REAL */
490 -             BI          = 0X6 /* SELECT CACHE PORT B IMAGINARY */
491 -             AUXR        = 0X7 /* SELECT AUXILIARY PORT REAL */
492 -             AUXI        = 0X8 /* SELECT AUXILIARY PORT IMAGINARY */
493 -             IOR         = 0XB /* SELECT I/O PORT REAL */
494 -             IOI         = 0XC /* SELECT I/O PORT IMAGINARY */
495 -             REGA        = 0XD /* SELECT REGISTER FILE PORT A */
496 -             REGB        = 0XE /* SELECT REGISTER FILE PORT B */
497 -             DIS         = 0XF /* DISABLE PORT (REGISTER HOLDS) */
498 -         }
499 -
500 -         /* PORT X CONTROL */
501 -         XCTRL[4]
502 -         {
503 -             DEFAULT     = 0B0001 /* HOLD */
504 -             HOLD        = 0B0001 /* HOLD */
505 -             P0MS        = 0B0011 /* PHASE 0 MOST SIGNIFICANT */
506 -             P1MS        = 0B1011 /* PHASE 1 MOST SIGNIFICANT */
507 -             P0LS        = 0B0000 /* PHASE 0 LEAST SIGNIFICANT */
508 -             P1LS        = 0B1000 /* PHASE 1 LEAST SIGNIFICANT */
509 -             TPORT       = 0B0100 /* T PORT */
510 -         }
511 -
512 -         /* CROSSBAR REGISTER SOURCE SELECT */
513 -         YSEL[4]
514 -         {
515 -             DEFAULT     = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
516 -             HOLD        = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
517 -             M1          = 0XA /* SELECT MULTIPLIER #1 */
518 -             M2          = 0X9 /* SELECT MULTIPLIER #2 */
519 -             A1          = 0X1 /* SELECT ALU #1 */
520 -             A2          = 0X2 /* SELECT ALU #2 */
521 -             AR          = 0X3 /* SELECT CACHE PORT A REAL */
522 -             AI          = 0X4 /* SELECT CACHE PORT A IMAGINARY */
523 -             BR          = 0X5 /* SELECT CACHE PORT B REAL */
524 -             BI          = 0X6 /* SELECT CACHE PORT B IMAGINARY */
525 -             AUXR        = 0X7 /* SELECT AUXILIARY PORT REAL */
526 -             AUXI        = 0X8 /* SELECT AUXILIARY PORT IMAGINARY */
527 -             IOR         = 0XB /* SELECT I/O PORT REAL */
528 -             IOI         = 0XC /* SELECT I/O PORT IMAGINARY */
529 -             REGA        = 0XD /* SELECT REGISTER FILE PORT A */
530 -             REGB        = 0XE /* SELECT REGISTER FILE PORT B */
531 -             DIS         = 0XF /* DISABLE PORT (REGISTER HOLDS) */
532 -         }
533 -
534 -         /* PORT Y CONTROL */
535 -         YCTRL[3]
536 -         {
537 -             DEFAULT     = 0B001 /* HOLD */
538 -             HOLD        = 0B001 /* HOLD */
539 -             P0MS        = 0B011 /* PHASE 0 MOST SIGNIFICANT */
540 -             P1MS        = 0B111 /* PHASE 1 MOST SIGNIFICANT */
541 -             P0LS        = 0B000 /* PHASE 0 LEAST SIGNIFICANT */
542 -             P1LS        = 0B100 /* PHASE 1 LEAST SIGNIFICANT */
543 -         }
544 -
545 -         /* INSTRUCTIONS FOR MULTIPLIERS */
```

```
546 -           INS[8]
547 -           {
548 -               DEFAULT     = 0B01011000  /* NO OPERATION */
549 -               NOP         = 0B01011000  /* NO OPERATION */
550 -
551 -               /* FLOATING POINT ARITHMETIC INSTRUCTIONS */
552 -
553 -               /* FLOATING POINT DIVISION */
554 -
555 -               DIV         = 0B00000000   /*     X/Y */
556 -               DDIV        = 0B00000001  /* DP: X/Y */
557 -
558 -               /* FLOATING POINT SQUARE ROOT */
559 -
560 -               SQRTX       = 0B00000010  /*     SQUARE ROOT X */
561 -               DSQRTX      = 0B00000011  /* DP: SQUARE ROOT X */
562 -
563 -               /* FLOATING POINT MULTIPLICATION WITH WRAPPED OPERANDS */
564 -
565 -               MULTWX      = 0B00000100  /*     WRAPPED X*Y */
566 -               DMULTWX     = 0B00000101  /* DP: WRAPPED X*Y */
567 -               MULTWY      = 0B00000110  /*     X*WRAPPED Y */
568 -               DMULTWY     = 0B00000111  /* DP: X*WRAPPED Y */
569 -
570 -               /* FLOATING POINT MULT WITH ABSOLUTE VALUE CAPABILITY */
571 -
572 -               MULT        = 0B00001000  /*     X*Y   */
573 -               DMULT       = 0B00001001  /* DP: X*Y   */
574 -               MULTAY      = 0B00001010  /*     X*|Y| */
575 -               DMULTAY     = 0B00001011  /* DP: X*|Y| */
576 -               MULTAX      = 0B00001100  /*     |X|*Y */
577 -               DMULTAX     = 0B00001101  /* DP: |X|*Y */
578 -               MULTA       = 0B00001110  /*     |X*Y| */
579 -               DMULTA      = 0B00001111  /* DP: |X*Y| */
580 -
581 -               /* FLOATING POINT SUPPORT INSTRUCTIONS */
582 -
583 -               /* X INPUT RETURNED UNMODIFIED */
584 -
585 -               PASSXM      = 0B00010000  /*     X */
586 -               DPASSXM     = 0B00010001  /* DP: X */
587 -
588 -               /* REGISTER ACCESS INSTRUCTIONS */
589 -
590 -               FREGMR      = 0B01011010  /* FMPY FLAG REGISTER READ */
591 -               FREGMW      = 0B01011011  /* FMPY FLAG REGISTER WRITE */
592 -               IREGMR      = 0B01011100  /* FMPY INT REGISTER READ */
593 -               IREGMW      = 0B01011101  /* FMPY INT REGISTER WRITE */
594 -               MREGMR      = 0B01011110  /* FMPY MODE REGISTER READ */
595 -               MREGMW      = 0B01011111  /* FMPY MODE REGISTER WRITE */
596 -
597 -               /* INTEGER ARITHMETIC INSTRUCTIONS */
598 -
599 -               /* INTEGER MULTIPLICATION INSTRUCTIONS */
600 -
601 -               IMULT       = 0B11111000  /* UNSIGNED X * UNSIGNED Y */
602 -               IMULTSX     = 0B11111001  /* SIGNED X * UNSIGNED Y */
603 -               IMULTSY     = 0B11111010  /* UNSIGNED X * SIGNED Y */
604 -               IMULTS      = 0B11111011  /* SIGNED X * SIGNED Y */
605 -               IMULTH      = 0B11111100  /* UNSIGNED X * UNSIGNED Y */
606 -               IMULTHSX    = 0B11111101  /* SIGNED X * UNSIGNED Y */
607 -               IMULTHSY    = 0B11111110  /* UNSIGNED X * SIGNED Y */
608 -               IMULTHS     = 0B11111111  /* SIGNED X * SIGNED Y */
609 -           }
610 -
611 -           /* T OUTPUT PORT CONTROL */
612 -           ZEN[1]
613 -           {
614 -               DEFAULT = 0B1 /* HOLD Z REGISTER */
615 -               HOLD    = 0B1 /* HOLD Z REGISTER */
616 -               EN      = 0B0 /* ENABLE Z REGISTER */
617 -           }
618 -
619 -           TSEL[2]
620 -           {
621 -               DEFAULT = 0B00 /* OUTPUT LS */
622 -               LS      = 0B00 /* OUTPUT LS */
623 -               MS      = 0B11 /* OUTPUT MS */
624 -               LSMS    = 0B01 /* OUTPUT LS TO CROSSBAR MS */
625 -               MSLS    = 0B10 /* OUTPUT MS TO CROSSBAR LS */
626 -           }
627 -       }
628 -
629 - /*      ======================          ======================
630 -       = ALU ONE (FIELD A1) =   -AND-  = ALU TWO (FIELD A2) =
631 -       ======================          ======================
632 - */
633 -       A1[27]
634 -       A2[27]
635 -       {
636 - DEFAULT = 0B000000001000000010011011000100
637 -           /* CROSSBAR REGISTER SOURCE SELECT */
638 -           XSEL[4]
639 -           {
640 -               DEFAULT     = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
641 -               HOLD        = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
642 -               M1          = 0XA /* SELECT MULTIPLIER #1 */
643 -               M2          = 0X9 /* SELECT MULTIPLIER #2 */
644 -               A1          = 0X1 /* SELECT ALU #1 */
645 -               A2          = 0X2 /* SELECT ALU #2 */
646 -               AR          = 0X3 /* SELECT CACHE PORT A REAL */
647 -               AI          = 0X4 /* SELECT CACHE PORT A IMAGINARY */
648 -               BR          = 0X5 /* SELECT CACHE PORT B REAL */
649 -               BI          = 0X6 /* SELECT CACHE PORT B IMAGINARY */
650 -               AUXR        = 0X7 /* SELECT AUXILIARY PORT REAL */
651 -               AUXI        = 0X8 /* SELECT AUXILIARY PORT IMAGINARY */
652 -               IOR         = 0XB /* SELECT I/O PORT REAL */
653 -               IOI         = 0XC /* SELECT I/O PORT IMAGINARY */
654 -               REGA        = 0XD /* SELECT REGISTER FILE PORT A */
```

```
655 -               REGB        = 0XE /* SELECT REGISTER FILE PORT B */
656 -               DIS         = 0XF /* DISABLE PORT (REGISTER HOLDS) */
657 -           }
658 -
659 -           /* PORT X CONTROL */
660 -           XCTRL[4]
661 -           {
662 -               DEFAULT     = 0B0001 /* HOLD */
663 -               HOLD        = 0B0001 /* HOLD */
664 -               P0MS        = 0B0011 /* PHASE 0 MOST SIGNIFICANT */
665 -               P1MS        = 0B1011 /* PHASE 1 MOST SIGNIFICANT */
666 -               P0LS        = 0B0000 /* PHASE 0 LEAST SIGNIFICANT */
667 -               P1LS        = 0B1000 /* PHASE 1 LEAST SIGNIFICANT */
668 -               TPORT       = 0B0100 /* T PORT */
669 -           }
670 -
671 -           /* CROSSBAR REGISTER SOURCE SELECT */
672 -           YSEL[4]
673 -           {
674 -               DEFAULT     = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
675 -               HOLD        = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
676 -               M1          = 0XA /* SELECT MULTIPLIER #1 */
677 -               M2          = 0X9 /* SELECT MULTIPLIER #2 */
678 -               A1          = 0X1 /* SELECT ALU #1 */
679 -               A2          = 0X2 /* SELECT ALU #2 */
680 -               AR          = 0X3 /* SELECT CACHE PORT A REAL */
681 -               AI          = 0X4 /* SELECT CACHE PORT A IMAGINARY */
682 -               BR          = 0X5 /* SELECT CACHE PORT B REAL */
683 -               BI          = 0X6 /* SELECT CACHE PORT B IMAGINARY */
684 -               AUXR        = 0X7 /* SELECT AUXILIARY PORT REAL */
685 -               AUXI        = 0X8 /* SELECT AUXILIARY PORT IMAGINARY */
686 -               IOR         = 0XB /* SELECT I/O PORT REAL */
687 -               IOI         = 0XC /* SELECT I/O PORT IMAGINARY */
688 -               REGA        = 0XD /* SELECT REGISTER FILE PORT A */
689 -               REGB        = 0XE /* SELECT REGISTER FILE PORT B */
690 -               DIS         = 0XF /* DISABLE PORT (REGISTER HOLDS) */
691 -           }
692 -
693 -           /* PORT Y CONTROL */
694 -           YCTRL[3]
695 -           {
696 -               DEFAULT     = 0B001 /* HOLD */
697 -               HOLD        = 0B001 /* HOLD */
698 -               P0MS        = 0B011 /* PHASE 0 MOST SIGNIFICANT */
699 -               P1MS        = 0B111 /* PHASE 1 MOST SIGNIFICANT */
700 -               P0LS        = 0B000 /* PHASE 0 LEAST SIGNIFICANT */
701 -               P1LS        = 0B100 /* PHASE 1 LEAST SIGNIFICANT */
702 -           }
703 -
704 -           /* Y SELECT INTERNAL TO THE ALU */
705 -           IYSEL[1]
706 -           {
707 -               DEFAULT     = 0B0 /* Y REGISTER */
708 -               ZREG        = 0B1 /* Z REGISTER */
709 -           }
710 -
711 -           /* ALU INSTRUCTIONS */
712 -           INS[8]
713 -           {
714 -               DEFAULT     = 0B01011000  /* NO OPERATION */
715 -               NOP         = 0B01011000  /* NO OPERATION */
716 -
717 -               /* FLOATING POINT ARITHMETIC INSTRUCTIONS */
718 -
719 -               /* MAXIMUM/MINIMUM */
720 -
721 -               MIN         = 0B00100100 /*     FLOATING POINT MIN */
722 -               DMIN        = 0B00100101 /* DP: FLOATING POINT MIN */
723 -               MAX         = 0B00100110 /*     FLOATING POINT MAX */
724 -               DMAX        = 0B00100111 /* DP: FLOATING POINT MAX */
725 -
726 -               /* ABSOLUTE, NEGATE OR PASS X OPERAND */
727 -
728 -               ABSX        = 0B00101000 /*     |X|  */
729 -               DABSX       = 0B00101001 /* DP: |X|  */
730 -               NEGX        = 0B00101010 /*     -X   */
731 -               DNEGX       = 0B00101011 /* DP: -X   */
732 -               PASSX       = 0B00101100 /*     X    */
733 -               DPASSX      = 0B00101101 /* DP: X    */
734 -
735 -               /* ADDITION AND SUBTRACTION */
736 -
737 -               ADD         = 0B00110000 /*     X+Y  */
738 -               DADD        = 0B00110001 /* DP: X+Y  */
739 -               SUBTR       = 0B00110010 /*     X-Y  */
740 -               DSUBTR      = 0B00110011 /* DP: X-Y  */
741 -               SUBX        = 0B00110100 /*     Y-X  */
742 -               DSUBX       = 0B00110101 /* DP: Y-X  */
743 -               ADDA        = 0B00111000 /*     |X|+|Y|  */
744 -               DADDA       = 0B00111001 /* DP: |X|+|Y|  */
745 -               SUBA        = 0B00111010 /*     |X|-|Y|  */
746 -               DSUBA       = 0B00111011 /* DP: |X|-|Y|  */
747 -               SUBXA       = 0B00111100 /*     |Y|-|X|  */
748 -               DSUBXA      = 0B00111101 /* DP: |Y|-|X|  */
749 -
750 -               /* FLOATING POINT SUPPORT INSTRUCTIONS */
751 -
752 -               /* SCALE */
753 -
754 -               SCALE       = 0B00100000 /*     EXPONENT X + Y */
755 -               DSCALE      = 0B00100001 /* DP: EXPONENT X + Y */
756 -
757 -               /* MERGE (CONCATENATE) */
758 -
759 -               MERGE       = 0B00100010 /*     SIGN X,EXPONENT Y,MANTISSA X */
760 -               DMERGE      = 0B00100011 /* DP: SIGN X,EXPONENT Y,MANTISSA X */
761 -
762 -               /* NORMALIZE X */
763 -
```

```
764 -               NORMX       = 0B00101110 /* NORMALIZE X */
765 -
766 -               /* COMPARE */
767 -
768 -               CMPR        = 0B00110110 /*     X,Y */
769 -               DCMPR       = 0B00110111 /* DP: X,Y */
770 -               CMPRA       = 0B00111110 /*     |X|,|Y| */
771 -               DCMPRA      = 0B00111111 /* DP: |X|,|Y| */
772 -
773 -               /* LOGICALLY LEFT SHIFT 4 PLACES AND ADD BITS SHOWN */
774 -
775 -               PASS0       = 0B01000000 /* X * 16 + 0  */
776 -               PASS1       = 0B01000001 /* X * 16 + 1  */
777 -               PASS2       = 0B01000010 /* X * 16 + 2  */
778 -               PASS3       = 0B01000011 /* X * 16 + 3  */
779 -               PASS4       = 0B01000100 /* X * 16 + 4  */
780 -               PASS5       = 0B01000101 /* X * 16 + 5  */
781 -               PASS6       = 0B01000110 /* X * 16 + 6  */
782 -               PASS7       = 0B01000111 /* X * 16 + 7  */
783 -               PASS8       = 0B01001000 /* X * 16 + 8  */
784 -               PASS9       = 0B01001001 /* X * 16 + 9  */
785 -               PASS10      = 0B01001010 /* X * 16 + 10 */
786 -               PASS11      = 0B01001011 /* X * 16 + 11 */
787 -               PASS12      = 0B01001100 /* X * 16 + 12 */
788 -               PASS13      = 0B01001101 /* X * 16 + 13 */
789 -               PASS14      = 0B01001110 /* X * 16 + 14 */
790 -               PASS15      = 0B01001111 /* X * 16 + 15 */
791 -
792 -               /* REGISTER ACCESS INSTRUCTIONS */
793 -
794 -               SCREGR      = 0B01010000 /* SC REGISTER READ */
795 -               SCREGW      = 0B01010001 /* SC REGISTER WRITE */
796 -               FREGAR      = 0B01010010 /* FALU FLAG REGISTER READ */
797 -               FREGAW      = 0B01010011 /* FALU FLAG REGISTER WRITE */
798 -               IREGAR      = 0B01010100 /* FALU INT REGISTER READ */
799 -               IREGAW      = 0B01010101 /* FALU INT REGISTER WRITE */
800 -               MREGAR      = 0B01010110 /* FALU MODE REGISTER READ */
801 -               MREGAW      = 0B01010111 /* FALU MODE REGISTER WRITE */
802 -
803 -               /* CLEAR FLAG REGISTER */
804 -
805 -               CLRFLAG     = 0B01011001 /* CLEAR FLAG REGISTER */
806 -
807 -               /* CONVERSION INSTRUCTIONS */
808 -
809 -               /* FLOATING POINT TO INTEGER AND VISA VERSA CONVERSION */
810 -
811 -               FCUI        = 0B01100000 /* SP -> 32-BIT UNSIGNED INTEGER */
812 -               DFCUI       = 0B01100001 /* DP -> 32-BIT UNSIGNED INTEGER */
813 -               FCSI        = 0B01100010 /* SP -> 32-BIT SIGNED INTEGER */
814 -               DFCSI       = 0B01100011 /* DP -> 32-BIT SIGNED INTEGER */
815 -               UICF        = 0B01100100 /* SP <- 32-BIT UNSIGNED INTEGER */
816 -               UICDF       = 0B01100101 /* DP <- 32-BIT UNSIGNED INTEGER */
817 -               SICF        = 0B01100110 /* SP <- 32-BIT SIGNED INTEGER */
818 -               SICDF       = 0B01100111 /* DP <- 32-BIT SIGNED INTEGER */
819 -               FCLUI       = 0B01101000 /* SP -> 64-BIT UNSIGNED INTEGER */
820 -               DFCLUI      = 0B01101001 /* DP -> 64-BIT UNSIGNED INTEGER */
821 -               FCLSI       = 0B01101010 /* SP -> 64-BIT SIGNED INTEGER */
822 -               DFCLSI      = 0B01101011 /* DP -> 64-BIT SIGNED INTEGER */
823 -               LUICF       = 0B01101100 /* SP <- 64-BIT UNSIGNED INTEGER */
824 -               LUICDF      = 0B01101101 /* DP <- 64-BIT UNSIGNED INTEGER */
825 -               LSICF       = 0B01101110 /* SP <- 64-BIT SIGNED INTEGER */
826 -               LSICDF      = 0B01101111 /* DP <- 64-BIT SIGNED INTEGER */
827 -               FCUIT       = 0B01110000 /* SP -> 32-BIT UNSIGNED INTEGER (RND TO 0) */
828 -               DFCUIT      = 0B01110001 /* DP -> 32-BIT UNSIGNED INTEGER (RND TO 0) */
829 -               FCSIT       = 0B01110010 /* SP -> 32-BIT SIGNED INTEGER (RND TO 0) */
830 -               DFCSIT      = 0B01110011 /* DP -> 32-BIT SIGNED INTEGER (RND TO 0) */
831 -               FCLUIT      = 0B01111000 /* SP -> 64-BIT UNSIGNED INTEGER (RND TO 0) */
832 -               DFCLUIT     = 0B01111001 /* DP -> 64-BIT UNSIGNED INTEGER (RND TO 0) */
833 -               FCLSIT      = 0B01111010 /* SP -> 64-BIT SIGNED INTEGER (RND TO 0) */
834 -               DFCLSIT     = 0B01111011 /* DP -> 64-BIT SIGNED INTEGER (RND TO 0) */
835 -
836 -               /* CONVERT WRAPPED X INPUT TO DENORMALIZED NUMBER */
837 -
838 -               WDNM        = 0B01110100 /*     WRAPPED -> DENORM */
839 -               DWDNM       = 0B01110101 /* DP: WRAPPED -> DENORM */
840 -
841 -               /* FLOATING POINT CONVERSION */
842 -
843 -               SDF         = 0B01110110 /* SP -> DP */
844 -               DSDF        = 0B01110111 /* DP -> SP */
845 -
846 -               FFI         = 0B01111100 /* SP -> SP FORMAT INTEGER */
847 -               DFFI        = 0B01111101 /* DP -> DP FORMAT INTEGER */
848 -               FFIT        = 0B01111110 /* SP -> SP FORMAT INTEGER (RND TO 0) */
849 -               DFFIT       = 0B01111111 /* DP -> DP FORMAT INTEGER (RND TO 0) */
850 -
851 -               /* INTEGER ARITHMETIC INSTRUCTIONS */
852 -
853 -               /* INTEGER ADDITION AND SUBTRACTION */
854 -
855 -               IADD        = 0B11100000 /*     X + Y */
856 -               LIADD       = 0B10100000 /* L: X + Y */
857 -               ISUB        = 0B11100101 /*     X - Y */
858 -               LISUB       = 0B10100101 /* L: X - Y */
859 -               ISUBX       = 0B11100110 /*     Y - X */
860 -               LISUBX      = 0B10100110 /* L: Y - X */
861 -               IADDP       = 0B11100100 /*     X + Y + 1 */
862 -               LIADDP      = 0B10100100 /* L: X + Y + 1 */
863 -               ISUBM       = 0B11100001 /*     X - Y - 1 */
864 -               LISUBM      = 0B10100001 /* L: X - Y - 1 */
865 -               ISUBXM      = 0B11100010 /*     Y - X - 1 */
866 -               LISUBXM     = 0B10100010 /* L: Y - X - 1 */
867 -               IADDC       = 0B11101000 /*     X + Y + CARRY */
868 -               LIADDC      = 0B10101000 /* L: X + Y + CARRY */
869 -               ISUBC       = 0B11101001 /*     X - Y - CARRY */
870 -               LISUBC      = 0B10101001 /* L: X - Y - CARRY */
871 -               ISUBXC      = 0B11101010 /*     Y - X - CARRY */
872 -               LISUBXC     = 0B10101010 /* L: Y - X - CARRY */
```

```
873 -
874 -              /* INTEGER NEGATE,NEGATE WITH CARRY, ABSOLUTE */
875 -
876 -              INEGC         = 0B11101011 /*     -X - CARRY */
877 -              LINEGC        = 0B10101011 /* L: -X - CARRY */
878 -              IABSX         = 0B11101111 /*     |X| */
879 -              LIABSX        = 0B10101111 /* L: |X| */
880 -              INEGX         = 0B11100111 /*     -X  */
881 -              LINEGX        = 0B10100111 /* L: -X  */
882 -
883 -              /* INTEGER MAXIMUM/MINIMUM */
884 -
885 -              ISMAX         = 0B11000010 /*     SIGNED MAX */
886 -              LISMAX        = 0B10000010 /* L: SIGNED MAX */
887 -              ISMIN         = 0B11000110 /*     SIGNED MIN */
888 -              LISMIN        = 0B10000110 /* L: SIGNED MIN */
889 -              IUMAX         = 0B11001010 /*     UNSIGNED MAX */
890 -              LIUMAX        = 0B10001010 /* L: UNSIGNED MAX */
891 -              IUMIN         = 0B11001110 /*     UNSIGNED MIN */
892 -              LIUMIN        = 0B10001110 /* L: UNSIGNED MIN */
893 -
894 -              /* INTEGER BOOLEAN INSTRUCTIONS */
895 -
896 -              /* BOOLEAN LOGIC */
897 -
898 -              INAND         = 0B11010000 /*     /X OR /Y */
899 -              LINAND        = 0B10010000 /* L: /X OR /Y */
900 -              IORNX         = 0B11010001 /*     /X OR  Y */
901 -              LIORNX        = 0B10010001 /* L: /X OR  Y */
902 -              IORNY         = 0B11010010 /*      X OR /Y */
903 -              LIORNY        = 0B10010010 /* L:  X OR /Y */
904 -              IOR           = 0B11010011 /*      X OR  Y */
905 -              LIOR          = 0B10010011 /* L:  X OR  Y */
906 -              IANDNY        = 0B11010100 /*      X AND /Y */
907 -              LIANDNY       = 0B10010100 /* L:  X AND /Y */
908 -              IAND          = 0B11010101 /*      X AND  Y */
909 -              LIAND         = 0B10010101 /* L:  X AND  Y */
910 -              INOR          = 0B11010110 /*     /X AND /Y */
911 -              LINOR         = 0B10010110 /* L: /X AND /Y */
912 -              IANDNX        = 0B11010111 /*     /X AND  Y */
913 -              LIANDNX       = 0B10010111 /* L: /X AND  Y */
914 -              IXNOR         = 0B11011110 /*      X XNOR  Y */
915 -              LIXNOR        = 0B10011110 /* L:  X XNOR  Y */
916 -              IXOR          = 0B11011111 /*      X XOR  Y */
917 -              LIXOR         = 0B10011111 /* L:  X XOR  Y */
918 -              ISET          = 0B11011000 /*      Z = ALL ONES */
919 -              LISET         = 0B10011000 /* L:  Z = ALL ONES */
920 -              INOTX         = 0B11011001 /*      Z = /X */
921 -              LINOTX        = 0B10011001 /* L:  Z = /X */
922 -              IPASSY        = 0B11011010 /*      Z = Y */
923 -              LIPASSY       = 0B10011010 /* L:  Z = Y */
924 -              IPASSX        = 0B11011011 /*      Z = X */
925 -              LIPASSX       = 0B10011011 /* L:  Z = X */
926 -              ICLR          = 0B11011100 /*      Z = ALL ZEROS */
927 -              LICLR         = 0B10011100 /* L:  Z = ALL ZEROS */
928 -              INOTY         = 0B11011101 /*      Z = /Y */
929 -              LINOTY        = 0B10011101 /* L:  Z = /Y */
930 -
931 -              /* INTEGER SHIFT AND ROTATE INSTRUCTIONS */
932 -
933 -              /* INTEGER SHIFT */
934 -
935 -              LSSX          = 0B11110000 /*     LOGICAL SHIFT X W/STICKY BIT */
936 -              LLSSX         = 0B10110000 /* L: LOGICAL SHIFT X W/STICKY BIT */
937 -              LSX           = 0B11110001 /*     LOGICAL SHIFT X */
938 -              LLSX          = 0B10110001 /* L: LOGICAL SHIFT X */
939 -
940 -              /* ARITHMETIC SHIFT */
941 -
942 -              AS            = 0B11110010 /*     ARITHMETIC SHIFT X */
943 -              LAS           = 0B10110010 /* L: ARITHMETIC SHIFT X */
944 -
945 -              /* ROTATE X BY THE SIGNED TWO'S COMPLEMENT
946 -                 NUMBER IN THE SHIFT COUNT (SC) REGISTER */
947 -
948 -              ROTX          = 0B11110011 /*     ROTATE X */
949 -              LROTX         = 0B10110011 /* L: ROTATE X */
950 -
951 -              /* CONCANTENATE AND ROTATE BY THE SIGNED TWO'S COMPLEMENT
952 -                 NUMBER IN THE SHIFT COUNT (SC) REGISTER */
953 -
954 -              ROTC          = 0B11110100 /* ROTATE Y|X (CONCATENATED) */
955 -              LROTC         = 0B10110100 /* L: ROTATE Y|X (CONCATENATED) */
956 -
957 -              /* BIT-REVERSE AND CONCANTENATE WITH A NON-BIT-REVERSED X */
958 -
959 -              BITR          = 0B11110101 /* ROTATE BIT REVERSED X|X */
960 -
961 -              /* INTEGER SC REGISTER INSTRUCTIONS */
962 -
963 -              ADDSC         = 0B11110110 /* Z,SC <- X + SC */
964 -              NEGSC         = 0B11110111 /* Z,SC <-  -SC */
965 -          }
966 -          /* T OUTPUT PORT CONTROL */
967 -          ZEN[1]
968 -          {
969 -              DEFAULT = 0B1 /* HOLD Z REGISTER */
970 -              HOLD    = 0B1 /* HOLD Z REGISTER */
971 -              EN      = 0B0 /* ENABLE Z REGISTER */
972 -          }
973 -
974 -          TSEL[2]
975 -          {
976 -              DEFAULT = 0B00 /* OUTPUT LS */
977 -              LS      = 0B00 /* OUTPUT LS */
978 -              MS      = 0B11 /* OUTPUT MS */
979 -              LSMS    = 0B01 /* OUTPUT LS TO CROSSBAR MS */
980 -              MSLS    = 0B10 /* OUTPUT MS TO CROSSBAR LS */
981 -          }
```

```
    }

/*      ================================
        = IMMEDIATE DATA (FIELD IMM) =
        ================================
*/

    IMM[1]
    {
DEFAULT = 0B1
        /* ENABLE FOR IMMEDIATE DATA REGISTER */
        CTRL[1]
        {
            DEFAULT     = 0B0
            EN          = 0B0 /* ENABLE DATA REGISTER */
DIS         = 0B1
        }
    }

/*      ==================================
        = DATA REGISTER FILE (FIELD REG) =
        ==================================
*/

    REG[36]
    {
DEFAULT = 0X000000000
        /* PORT A INPUT SOURCE */
        ASEL[5]
        {
            DEFAULT     = 0B00000 /* NO OPERATION */
            NOP         = 0B00000 /* NO OPERATION */
            CLEAR       = 0B00001 /* CLEAR REGISTER (ALL ZEROS) */
            M1          = 0B10101 /* SELECT MULTIPLIER #1 */
            M2          = 0B10011 /* SELECT MULTIPLIER #2 */
            A1          = 0B00011 /* SELECT ALU #1 */
            A2          = 0B00101 /* SELECT ALU #2 */
            AR          = 0B00111 /* SELECT CACHE PORT A REAL */
            AI          = 0B01001 /* SELECT CACHE PORT A IMAGINARY */
            BR          = 0B01011 /* SELECT CACHE PORT B REAL */
            BI          = 0B01101 /* SELECT CACHE PORT B IMAGINARY */
            AUXR        = 0B01111 /* SELECT AUXILIARY PORT REAL */
            AUXI        = 0B10001 /* SELECT AUXILIARY PORT IMAGINARY */
            IOR         = 0B10111 /* SELECT I/O PORT REAL */
            IOI         = 0B11001 /* SELECT I/O PORT IMAGINARY */
            REGA        = 0B11011 /* SELECT REGISTER FILE PORT A */
            REGB        = 0B11101 /* SELECT REGISTER FILE PORT B */
            SET         = 0B11111 /* SET REGISTER (ALL ONES) */
        }

        /* PORT A WRITE ADDRESS */
        WRA[6]
        {
            DEFAULT     = 0B000000 /* REGISTER 0 */
        }

        /* PORT B INPUT SOURCE */
        BSEL[5]
        {
            DEFAULT     = 0B00000 /* NO OPERATION */
            NOP         = 0B00000 /* NO OPERATION */
            CLEAR       = 0B00001 /* CLEAR REGISTER (ALL ZEROS) */
            M1          = 0B10101 /* SELECT MULTIPLIER #1 */
            M2          = 0B10011 /* SELECT MULTIPLIER #2 */
            A1          = 0B00011 /* SELECT ALU #1 */
            A2          = 0B00101 /* SELECT ALU #2 */
            AR          = 0B00111 /* SELECT CACHE PORT A REAL */
            AI          = 0B01001 /* SELECT CACHE PORT A IMAGINARY */
            BR          = 0B01011 /* SELECT CACHE PORT B REAL */
            BI          = 0B01101 /* SELECT CACHE PORT B IMAGINARY */
            AUXR        = 0B01111 /* SELECT AUXILIARY PORT REAL */
            AUXI        = 0B10001 /* SELECT AUXILIARY PORT IMAGINARY */
            IOR         = 0B10111 /* SELECT I/O PORT REAL */
            IOI         = 0B11001 /* SELECT I/O PORT IMAGINARY */
            REGA        = 0B11011 /* SELECT REGISTER FILE PORT A */
            REGB        = 0B11101 /* SELECT REGISTER FILE PORT B */
            SET         = 0B11111 /* SET REGISTER (ALL ONES) */
        }

        /* PORT B WRITE ADDRESS */
        WRB[6]
        {
            DEFAULT     = 0B000000 /* REGISTER 0 */
        }

        /* SHIFT MODE CONTROL */
        SMODE[2]
        {
            DEFAULT     = 0B00 /* NORMAL (REGISTER FILE MODE) */
            REG         = 0B00 /* NORMAL (REGISTER FILE MODE) */
            R8X8         = 0B01 /* 8 BY 8 SHIFT REGISTER MODE */
            R4X16        = 0B10 /* 4 BY 16 SHIFT REGISTER MODE */
            R2X32        = 0B11 /* 2 BY 32 SHIFT REGISTER MODE */
        }

        /* PORT A READ ADDRESS */
        RDA[6]
        {
            DEFAULT     = 0B000000 /* REGISTER 0 */
        }

        /* PORT B READ ADDRESS */
        RDB[6]
        {
            DEFAULT     = 0B000000 /* REGISTER 0 */
        }
    }

/*      ================================
```

```
1091 -          = MEMORY WRITE PORTS (FIELD MWR) =
1092 -          ==================================
1093 - */
1094 -
1095 -      MWR[12]
1096 -      {
1097 - DEFAULT = 0B111100111100
1098 -          /* CROSSBAR REGISTER SOURCE SELECT */
1099 -          RSEL[4]
1100 -          {
1101 -              DEFAULT      = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
1102 -              HOLD         = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
1103 -              M1           = 0XA /* SELECT MULTIPLIER #1 */
1104 -              M2           = 0X9 /* SELECT MULTIPLIER #2 */
1105 -              A1           = 0X1 /* SELECT ALU #1 */
1106 -              A2           = 0X2 /* SELECT ALU #2 */
1107 -              AR           = 0X3 /* SELECT CACHE PORT A REAL */
1108 -              AI           = 0X4 /* SELECT CACHE PORT A IMAGINARY */
1109 -              BR           = 0X5 /* SELECT CACHE PORT B REAL */
1110 -              BI           = 0X6 /* SELECT CACHE PORT B IMAGINARY */
1111 -              AUXR         = 0X7 /* SELECT AUXILIARY PORT REAL */
1112 -              AUXI         = 0X8 /* SELECT AUXILIARY PORT IMAGINARY */
1113 -              IOR          = 0XB /* SELECT I/O PORT REAL */
1114 -              IOI          = 0XC /* SELECT I/O PORT IMAGINARY */
1115 -              REGA         = 0XD /* SELECT REGISTER FILE PORT A */
1116 -              REGB         = 0XE /* SELECT REGISTER FILE PORT B */
1117 -              DIS          = 0XF /* DISABLE PORT (REGISTER HOLDS) */
1118 -          }
1119 -
1120 -          /* REAL PORT OUTPUT CONTROL */
1121 -          RCTRL[2]
1122 -          {
1123 -              DEFAULT      = 0B00 /* PHASE 0 LEAST SIGNIFICANT */
1124 -              P0LS         = 0B00 /* PHASE 0 LEAST SIGNIFICANT */
1125 -              P1LS         = 0B10 /* PHASE 1 LEAST SIGNIFICANT */
1126 -              P0MS         = 0B00 /* PHASE 0 MOST SIGNIFICANT */
1127 -              P1MS         = 0B10 /* PHASE 1 MOST SIGNIFICANT */
1128 -          }
1129 -
1130 -          /* CROSSBAR REGISTER SOURCE SELECT */
1131 -          ISEL[4]
1132 -          {
1133 -              DEFAULT      = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
1134 -              HOLD         = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
1135 -              M1           = 0XA /* SELECT MULTIPLIER #1 */
1136 -              M2           = 0X9 /* SELECT MULTIPLIER #2 */
1137 -              A1           = 0X1 /* SELECT ALU #1 */
1138 -              A2           = 0X2 /* SELECT ALU #2 */
1139 -              AR           = 0X3 /* SELECT CACHE PORT A REAL */
1140 -              AI           = 0X4 /* SELECT CACHE PORT A IMAGINARY */
1141 -              BR           = 0X5 /* SELECT CACHE PORT B REAL */
1142 -              BI           = 0X6 /* SELECT CACHE PORT B IMAGINARY */
1143 -              AUXR         = 0X7 /* SELECT AUXILIARY PORT REAL */
1144 -              AUXI         = 0X8 /* SELECT AUXILIARY PORT IMAGINARY */
1145 -              IOR          = 0XB /* SELECT I/O PORT REAL */
1146 -              IOI          = 0XC /* SELECT I/O PORT IMAGINARY */
1147 -              REGA         = 0XD /* SELECT REGISTER FILE PORT A */
1148 -              REGB         = 0XE /* SELECT REGISTER FILE PORT B */
1149 -              DIS          = 0XF /* DISABLE PORT (REGISTER HOLDS) */
1150 -          }
1151 -
1152 -          /* IMAGINARY PORT OUTPUT CONTROL */
1153 -          ICTRL[2]
1154 -          {
1155 -              DEFAULT      = 0B00 /* PHASE 0 LEAST SIGNIFICANT */
1156 -              P0LS         = 0B00 /* PHASE 0 LEAST SIGNIFICANT */
1157 -              P1LS         = 0B10 /* PHASE 1 LEAST SIGNIFICANT */
1158 -              P0MS         = 0B00 /* PHASE 0 MOST SIGNIFICANT */
1159 -              P1MS         = 0B10 /* PHASE 1 MOST SIGNIFICANT */
1160 -          }
1161 -      }
1162 -
1163 - /*       ================================================
1164 -          = PROCESSOR ADDRESS PORTS (FIELDS PADDRA-PADDRD) =
1165 -          ================================================
1166 - */
1167 -
1168 -      PADDRA[6]
1169 -      PADDRB[6]
1170 -      PADDRC[6]
1171 -      PADDRD[6]
1172 -      {
1173 - DEFAULT = 0B111100
1174 -          /* CROSSBAR REGISTER SOURCE SELECT */
1175 -          SEL[4]
1176 -          {
1177 -              DEFAULT      = 0XF /* DISABLE PORT (REGISTER HOLDS) */
1178 -              HOLD         = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
1179 -              M1           = 0XA /* SELECT MULTIPLIER #1 */
1180 -              M2           = 0X9 /* SELECT MULTIPLIER #2 */
1181 -              A1           = 0X1 /* SELECT ALU #1 */
1182 -              A2           = 0X2 /* SELECT ALU #2 */
1183 -              AR           = 0X3 /* SELECT CACHE PORT A REAL */
1184 -              AI           = 0X4 /* SELECT CACHE PORT A IMAGINARY */
1185 -              BR           = 0X5 /* SELECT CACHE PORT B REAL */
1186 -              BI           = 0X6 /* SELECT CACHE PORT B IMAGINARY */
1187 -              AUXR         = 0X7 /* SELECT AUXILIARY PORT REAL */
1188 -              AUXI         = 0X8 /* SELECT AUXILIARY PORT IMAGINARY */
1189 -              IOR          = 0XB /* SELECT I/O PORT REAL */
1190 -              IOI          = 0XC /* SELECT I/O PORT IMAGINARY */
1191 -              REGA         = 0XD /* SELECT REGISTER FILE PORT A */
1192 -              REGB         = 0XE /* SELECT REGISTER FILE PORT B */
1193 -              DIS          = 0XF /* DISABLE PORT (REGISTER HOLDS) */
1194 -          }
1195 -
1196 -          /* PORT OUTPUT CONTROL */
1197 -          CTRL[2]
1198 -          {
1199 -              DEFAULT      = 0B00 /* PHASE 0 LEAST SIGNIFICANT */
```

```
1200 -              P0          = 0B00 /* PHASE 0 LEAST SIGNIFICANT */
1201 -              P1          = 0B10 /* PHASE 1 LEAST SIGNIFICANT */
1202 -          }
1203 -      }
1204 -
1205 - /*      ================================
1206 -          = I/O PORTS (FIELDS IOR & IOI) =
1207 -          ================================
1208 - */
1209 -
1210 -      IOR[6]
1211 -      IOI[6]
1212 -      {
1213 - DEFAULT = 0B111100
1214 -          /* CROSSBAR REGISTER SOURCE SELECT */
1215 -          SEL[4]
1216 -          {
1217 -              DEFAULT     = 0XF /* DISABLE PORT (REGISTER HOLDS) */
1218 -              HOLD        = 0X0 /* HOLD (REGISTER SELECTS IT'S SELF) */
1219 -              M1          = 0XA /* SELECT MULTIPLIER #1 */
1220 -              M2          = 0X9 /* SELECT MULTIPLIER #2 */
1221 -              A1          = 0X1 /* SELECT ALU #1 */
1222 -              A2          = 0X2 /* SELECT ALU #2 */
1223 -              AR          = 0X3 /* SELECT CACHE PORT A REAL */
1224 -              AI          = 0X4 /* SELECT CACHE PORT A IMAGINARY */
1225 -              BR          = 0X5 /* SELECT CACHE PORT B REAL */
1226 -              BI          = 0X6 /* SELECT CACHE PORT B IMAGINARY */
1227 -              AUXR        = 0X7 /* SELECT AUXILIARY PORT REAL */
1228 -              AUXI        = 0X8 /* SELECT AUXILIARY PORT IMAGINARY */
1229 -              IOR         = 0XB /* SELECT I/O PORT REAL */
1230 -              IOI         = 0XC /* SELECT I/O PORT IMAGINARY */
1231 -              REGA        = 0XD /* SELECT REGISTER FILE PORT A */
1232 -              REGB        = 0XE /* SELECT REGISTER FILE PORT B */
1233 -              DIS         = 0XF /* DISABLE PORT (REGISTER HOLDS) */
1234 -          }
1235 -
1236 -          /* PORT OUTPUT CONTROL */
1237 -          CTRL[2]
1238 -          {
1239 -              DEFAULT     = 0B00 /* PHASE 0 LEAST SIGNIFICANT */
1240 -              P0LS        = 0B00 /* PHASE 0 LEAST SIGNIFICANT */
1241 -              P1LS        = 0B10 /* PHASE 1 LEAST SIGNIFICANT */
1242 -              P0MS        = 0B01 /* PHASE 0 MOST SIGNIFICANT */
1243 -              P1MS        = 0B11 /* PHASE 1 MOST SIGNIFICANT */
1244 -              INLS        = 0B00 /* INPUT LEAST SIGNIFICANT */
1245 -              INMS        = 0B01 /* INPUT MOST SIGNIFICANT */
1246 -          }
1247 -      }
1248 -
1249 - /* BIT ASSIGNMENTS FOR ADDRESS GENERATOR */
1250 - ASSIGN (344:351) = SEQ.INS;
1251 - ASSIGN (728:735) = CCS;
1252 - ASSIGN (352:367) = SEQ.BRA;
1253 - ASSIGN (736:751) = SEQ.BRA;
1254 - ASSIGN (237:240,222,223,256,257,205:208,336:339) = IMMADD;
1255 - ASSIGN (621:624,606,607,640,641,589:592,720:723) = IMMADD;
1256 - ASSIGN (192:204) = CNT1;
1257 - ASSIGN (576:588) = CNT1;
1258 - ASSIGN (209:221) = CNT2;
1259 - ASSIGN (593:605) = CNT2;
1260 - ASSIGN (224:236) = CNT3;
1261 - ASSIGN (608:620) = CNT3;
1262 - ASSIGN (241:253) = CNT4;
1263 - ASSIGN (625:637) = CNT4;
1264 - ASSIGN (302:313,322,323,314:319,368:383) = AREG;
1265 - ASSIGN (686:697,706,707,698:703,752:767) = AREG;
1266 - ASSIGN (279:287,320,321) = RAM1;
1267 - ASSIGN (663:671,704,705) = RAM1;
1268 - ASSIGN (268:278) = RAM2;
1269 - ASSIGN (652:662) = RAM2;
1270 - ASSIGN (297:301)     = ADDRC;
1271 - ASSIGN (681:685)     = ADDRA;
1272 - ASSIGN (292:296)     = ADDRD;
1273 - ASSIGN (676:680)     = ADDRB;
1274 - ASSIGN (255,288:291)  = BANK;
1275 - ASSIGN (639,672:675)  = ADDRE;
1276 -
1277 - /* BIT ASSIGNMENTS FOR CACHE MEMORY */
1278 - ASSIGN (340,341,724,725)     = AUX;
1279 - ASSIGN (342,343,726,727)     = CACHE;
1280 -
1281 - /* BIT ASSIGNMENTS FOR PROCESSOR */
1282 - ASSIGN (166:191)  = M1;
1283 - ASSIGN (550:575)  = M1;
1284 - ASSIGN (140:165)  = M2;
1285 - ASSIGN (524:549)  = M2;
1286 - ASSIGN (113:139)  = A1;
1287 - ASSIGN (497:523)  = A1;
1288 - ASSIGN (86:112)   = A2;
1289 - ASSIGN (470:496)  = A2;
1290 - ASSIGN (50:85)    = REG;
1291 - ASSIGN (434:469)  = REG;
1292 - ASSIGN (49)       = IMM;
1293 - ASSIGN (433)      = IMM;
1294 - ASSIGN (37:48)    = MWR;
1295 - ASSIGN (421:432)  = MWR;
1296 - ASSIGN (31:36)    = PADDRC;
1297 - ASSIGN (415:420)  = PADDRA;
1298 - ASSIGN (25:30)    = PADDRD;
1299 - ASSIGN (409:414)  = PADDRB;
1300 - ASSIGN (19:24)    = IOR;
1301 - ASSIGN (403:408)  = IOR;
1302 - ASSIGN (13:18)    = IOI;
1303 - ASSIGN (397:402)  = IOI;
1304 -
1305 - DOMAIN MSRAM[768] (0:0XFFFF)
1306 -
```

APPENDIX F

IOP DEFINITION FILE

```
1 - /*  ============================================================
2 -     =  THIS FILE INCLUDES HARDWARE CHANGES AS OF MAY 30, 1992  =
3 -     ============================================================
4 - */
5 -
6 - /*IOP Microprogram sequencer     IOPMGA.DEF     2 June 1992 */
7 - /*BIT ASSIGNMENTS FOR MICROSEQUENCER*/
8 - WIDTH = 48
9 - PHASES = 1
10 - DEFBIT = 0
11 -
12 - /*      ============================================================
13 -        =   DATA BUS SOURCE AND DESTINATION SELECT  (FIELD CR) =
14 -        ============================================================
15 - */
16 -
17 - CR[7]
18 - { DEFAULT = 0B1000000
19 -         DEST[4]                 /*DESTINATION FOR DATA*/
20 -         { DEFAULT = 0B1000
21 -           NOWR    = 0B1000 /*NO WRITE*/
22 -           AS      = 0B1100 /*MACRO RAM ADDRESS INPUT REGISTER WRITE*/
23 -           CAS     = 0B1011 /*MACRO RAM COUNTER ADDRESS WRITE*/
24 -           CARL    = 0B1010 /*MACRO RAM COUNTER ADDRESS INPUT REGISTER WRITE*/
25 -           INCR    = 0B0000 /*INCREMENT MACRO RAM ADDRESS*/
26 -           MWR     = 0B1001 /*MACRO RAM WRITE*/
27 -           CRW     = 0B1101 /*CONTROL REGISTER WRITE*/
28 -         }
29 -         BUS[3]                /*DATA BUS SOURCE SELECT*/
30 -         { DEFAULT = 0B000
31 -           NONE    = 0B000   /*NO BUS SELECTED
32 -           MPMEM   = 0B001   /*MICROPROGRAM RAM*/
33 -           IBMPC   = 0B010   /*IBM PC INTERFACE*/
34 -           SIO     = 0B011   /*SERIAL IO (VME) INTERFACE*/
35 -           HSIO    = 0B100   /*HIGH SPEED IO INTERFACE*/
36 -           MRAM    = 0B101   /*MACRO RAM*/
37 -           BOOT    = 0B111   /*BOOT INSTRUCTION*/
38 -         }
39 - }
40 -
41 - /*      ============================================
42 -        =       REGISTER ADDRESS  (FIELD CRA)      =
43 -        ============================================
44 - */
45 -
46 - CRA[4]
47 - { DEFAULT = 0B0000
48 -         REGAD[4]
49 -         {DEFAULT = 0B0000
50 -          SOURCE  = 0X0  /*RESOURCES IN USE*/
51 -          PCSTAT  = 0X1  /*PC INTERFACE CONTROL AND STATUS*/
52 -          PCTRAN  = 0X2  /*PC TRANSMIT CONTROL*/
53 -          PCIMK   = 0X3  /*PC INTERRUPT MASK*/
54 -          SIO     = 0X4  /* SERIAL INTERFACE CONTROL*/
55 -          SIOTR   = 0X5  /*SERIAL INTERFACE TRANSMIT CONTROL*/
56 -          SIOMK   = 0X6  /*SERIAL TRANSMIT INTERRUPT MASK*/
57 -          HSIO    = 0X7  /*HIGH SPEED IO INTERFACE CONTROL*/
58 -          CNTA    = 0X8  /*A COUNTER DATA TRANSFER COUNT*/
59 -          CNTB    = 0X9  /*B COUNTER DATA TRANSFER COUNT*/
60 -          MRAMAR  = 0XA   /*MACRO RAM ADDRESS REGISTER*/
61 -          MRAMACT = 0XB   /* "    "    "      COUNTER*/
62 -          MRAMCT  = 0XC   /* "    "   COUNTER REGISTER*/
63 -          HSIOAC  = 0XD   /*CPH HSIO ADDRESS COUNTER*/
64 -          HSIOSAR = 0XE   /*CPH HSIO SYSTEM ADDRESS REGISTER*/
65 -          IOPCR   = 0XF   /*IOP CONTROL REGISTER*/
66 -         }
67 -
68 - }
69 -
70 - /*      ====================================
71 -        =   CONDITION SELECT  (FIELD CCS)   =
72 -        ====================================
73 - */
74 -
75 - CCS[6]
76 -
77 - { DEFAULT = 0B000000
78 -         CCODE[6]
79 -         {DEFAULT   = 0B000000
80 -          PCTFF    = 0B000000    /*IBM-PC TRANSMIT FULL FLAG*/
81 -          PCTAFF   = 0B000001    /*IBM-PC TRANSMIT ALMOST FULL FLAG*/
82 -          PCTAEF   = 0B000011    /*IBM-PC TRANSMIT HALF FULL FLAG*/
83 -                                 /*IBM-PC TRANSMIT ALMOST EMPTY FLAG*/
84 -
85 -          PCRFF    = 0B000101    /*IBM-PC RECEIVE FULL FLAG*/
86 -                                   /*IBM-PC RECEIVE ALMOST FULL FLAG*/
87 -                                   /*IBM-PC RECEIVE HALF FULL FLAG*/
88 -                                   /*IBM-PC RECEIVE ALMOST EMPTY FLAG*/
89 -          PCREF    = 0B010011    /*IBM-PC RECEIVE EMPTY FLAG*/
90 -          SIOTF    = 0B001010    /*SIO TRANSMIT FULL FLAG*/
91 -                                   /*SIO TRANSMIT ALMOST FULL FLAG*/
92 -                                   /*SIO TRANSMIT HALF FULL FLAG*/
93 -                                   /*SIO TRANSMIT ALMOST EMPTY FLAG*/
94 -          SIOTE    = 0B001110    /*SIO TRANSMIT EMPTY FLAG*/
95 -          SIORF    = 0B001111    /*SIO RECEIVE FULL FLAG*/
96 -                                   /*SIO RECEIVE ALMOST FULL FLAG*/
97 -                                   /*SIO RECEIVE HALF FULL FLAG*/
98 -                                   /*SIO RECEIVE ALMOST EMPTY FLAG*/
99 -          SIORE    = 0B010111    /*SIO RECEIVE EMPTY FLAG*/
100 -         ZCNTA    = 0B010100    /*COUNTER A ZERO*/
101 -         ZCNTB    = 0B010101    /*COUNTER B ZERO*/
102 -         HSIOR    = 0B011000    /*HSIO RECEIVE INTERFACE AVAILABLE*/
103 -         HSIOT    = 0B011001    /*HSIO TRANSMIT INTERFACE AVAILABLE*/
104 -         SIOR     = 0B011010    /*SIO RECEIVE INTERFACE AVAILABLE*/
105 -         SIOT     = 0B011011    /*SIO TRANSMIT INTERFACE AVAILABLE*/
106 -         PCR      = 0B011100    /*PC RECEIVE INTERFACE AVAILABLE*/
107 -         PCT      = 0B011101    /*PC TRANSMIT INTERFACE AVAILABLE*/
108 -         HSIOB    = 0B011110    /*HSIO BUS BUSY*/
109 -         SYSIO    = 0B011111    /*SYSTEM INTERRUPT 0 */
```

```
110 -          SYSI1    = 0B100000     /*SYSTEM INTERRUPT 1*/
111 -          SYSI2    = 0B100001     /*SYSTEM INTERRUPT 2*/
112 -          SYSI3    = 0B100010     /*SYSTEM INTERRUPT 3*/
113 -          SYSI4    = 0B100011     /*SYSTEM INTERRUPT 4*/
114 -          SYSI5    = 0B100100     /*SYSTEM INTERRUPT 5*/
115 -          SYSI6    = 0B100101     /*SYSTEM INTERRUPT 6*/
116 -          SYSI7    = 0B100110     /*SYSTEM INTERRUPT 7*/
117 -        }
118 - }
119 - /*       ======================================================
120 -          =  Microsequencer Instructions & Branch Address/Data  =
121 -          ======================================================
122 - */
123 -
124 - SEQ[31]/*BE CAREFUL WITH LOWER 24-BITS*/
125 -      { DEFAULT = 0X0000000
126 -
127 - INSTR[7]          /*MICROSEQUENCER INSTRUCTION*/
128 -             { DEFAULT = 0B0000000
129 -               CONT    = 0B0000000  /*CONTINUE*/
130 -               IDLE    = 0B0010000
131 -               IHC     = 0B0100101  /*ENABLE INSTRUCTION HOLD CONTROL*/
132 -               WCS     = 0B0100000  /*WRITE CONTROL STORE*/
133 -               JPCOF   = 0B0010101  /*IF FLAG, JUMP PC (SELF)*/
134 -               JPCNF   = 0B0110101  /*IF NOT FLAG, JUMP PC(SELF)*/
135 -               REL8    = 0B0100110  /*RELATIVE ADDRESS WIDTH 8, TERMINATES IHC*/
136 -
137 -             /*INTERRUPT CONTROL*/
138 -
139 -               CCIR    = 0B0010001  /*CLEAR CURRENT INTERRUPT*/
140 -               CAIR    = 0B0000001  /*CLEAR ALL INTERRUPTS*/
141 -               IRMBC   = 0B0010011  /*IR MASK BITWISE CLEAR*/
142 -               IRMBS   = 0B0010010  /*IR MASK BITWISE SET*/
143 -               DISIR   = 0B0010110  /*DISABLES INTERRUPTS*/
144 -               ENAIR   = 0B0110110  /*ENABLES INTERRUPTS*/
145 -               RDIV    = 0B0101101  /*READ IV AND INCREMENT IV POINTER*/
146 -               WRIV    = 0B0001101  /*WRITE IV AND INCREMENT IV POINTER*/
147 -               RTNIR   = 0B0000011  /*RETURN FROM INTERRUPT*/
148 -               SLRIVP  = 0B0011101  /*WRITES STACK LIMIT REGISTER AND IV POINTER*/
149 -               SLIR    = 0B0010111  /*SELECTS LATCHED INTERRUPTS*/
150 -               STIR    = 0B0110111  /*SELECTS TRANSPARENT INTERRUPTS*/
151 -
152 -      }
153 -
154 - /*       ======================================================
155 -          =  BRANCH ADDRESS OR WRITE ENABLES & DATA WRITES (FIELD REG)  =
156 -          ======================================================
157 -
158 -          ****************************************************************
159 -          * NOTE: REG bits function as follows:                          *
160 -          *         Bits 23..12 - Data, useq branch address, or          *
161 -          *                       write enables for CRA Field            *
162 -          *         Bits 11..0  - Data, useq branch address, or          *
163 -          *                       data to write for CRA Field            *
164 -          *                                                              *
165 -          *              ++++++   EXCEPTIONS    +++++++                  *
166 -          *                                                              *
167 -          *         CRA Fields 8 & 9 (CNTA & CNTB) use bits 19..0        *
168 -          *              as 20 bit counters                              *
169 -          *         CRA Field 10 (MRAMAR) uses bits 12..0 as a 13 bit    *
170 -          *              Macro RAM Address                               *
171 -          *         CRA Field 13 (HSIOAC) uses bits 23..0 as a 24 bit    *
172 -          *              CPH I/O and memory space address counter        *
173 -          *                                                              *
174 -          ****************************************************************
175 - */
176 -          REG[24]
177 -                  { DEFAULT  = 0XFFFFFF
178 -
179 -                    LABEL
180 -     ENABLES[12]
181 -                     { DEFAULT = 0B111111111111
182 -                       ALL     = 0XF00 /*Enables registers 12-19*/
183 -                       NONE    = 0XFFF /*Disables registers 12-23*/
184 -                       ENR20   = 0XBFF /*Enables register bit 20*/
185 -                       ENR19   = 0XF7F /*Enables register bit 19*/
186 -                       ENR18   = 0XFBF /*Enables register bit 18*/
187 -                       ENR17   = 0XFDF /*Enables register bit 17*/
188 -                       ENR16   = 0XFEF /*Enables register bit 16*/
189 -                       ENR15   = 0XFF7 /*Enables register bit 15*/
190 -                       ENR14   = 0XFFB /*Enables register bit 14*/
191 -                       ENR13   = 0XFFD /*Enables register bit 13*/
192 -                       ENR12   = 0XFFE /*Enables register bit 12*/
193 -                       HSIOINT = 0XFF4 /*HSIO source/data enable*/
194 -                       PCXSET  = 0XFE7 /*PCXMIT clock/data enable*/
195 -                     }
196 -
197 -                     DATA[12]
198 -                      { DEFAULT = 0B111111111111
199 -                        CLR     = 0XFFF
200 -                          /*Clears enabled data bits 0-11 */
201 -                        SET     = 0X000
202 -                          /*Sets enabled data bits 0-11 */
203 -                        PSDATA  = 0XFD7
204 -                          /*selects real 32 bit data\clk a (PCXSET CRA 2 -PCTRAN)*/
205 -                          /*selects real 32 bit data\clk a (PCXSET CRA 5 -SIOTR)*/
206 -                        SINT    = 0XFF3
207 -                          /*Enables receive FF (ENR13 CRA 4 -SIO)*/
208 -                        SIORFF  = 0XFDF
209 -                          /*Enables SIO receive FF (ENR13 CRA 6 -SIOMK)*/
210 -                        SIOTFF  = 0XFFE
211 -                          /*Enables SIO transmit FF (ENR13 CRA 6 -SIOMK)*/
212 -                        HIOPC   = 0XFDA
213 -                          /*Selects PC as HSIO source 32 bit real,
214 -                                CLK A, data (HSIOINT CRA 7 -HSIO)*/
215 -                        HIOCPH  = 0XFCA
216 -                          /*Selects CPH as HSIO source 32 bit real,
217 -                                CLK A data (HSIOINT CRA 7 -HSIO)*/
218 -                        HIOSIO  = 0XFEA
```

```
                          /*Selects SIO as HSIO source 32 bit real,
                                  CLK A data (HSIOINT CRA 7 -HSIO*/
                        INTMAP0 = 0XFF8 /*Selects interrupt table 0
                                  (ENR12 CRA 15 -IOPCR)*/
                        INT1    = 0X120 /*SLRIVP -selects IVP 1,
                                  stack of 20 hex*/

                      }
}
    }

ASSIGN (41:47) = CR;
ASSIGN (37:40) = CRA;
ASSIGN (31:36) = CCS;
ASSIGN (0:30)  = SEQ;

DOMAIN IORAM[48] (0:0XFF)
```

APPENDIX G

VME ADDRESS CONTROL

VPH HEX ADDRESS MAP

HEX ADDRESS	RESOURCE	BSx	FUNC. CODES
0 - 7FFF or FFFF	EPROM	BS0	1XX,X10,X01
4 0000 - 5 FFFF	020 SRAM	BS1	1XX,X10,X01
8 0000 - 8 3FFF	020 4-PORT SRAM	BS2	111,011,0X0,00X
C 0000 - C 0FFF	ZORAN #1	BS3	111,0X0,00X
C 1000 - C 1FFF	ZORAN #2	BS3	111,0X0,00X
10 0000 - 11 FFFF	ZORAN BUS #1 PRAM	BS4	111,0X0,00X
14 0000 - 14 0FFF	ZORAN #3	BS5	111,0X0,00X
14 1000 - 14 1FFF	ZORAN #4	BS5	111,0X0,00X
18 0000 - 19 FFFF	ZORAN BUS #2 PRAM	BS6	111,0X0,00X
1C 0000	VPH STATUS LATCH	BS7	111,0X0,00X
1C 0004	ZORAN RESET LATCH	BS7	111,0X0,00X
20 0000	CPH ADDRESS (FC=011 FOR AUGMENTED XFERS) (SELECTABLE AT CPH)	BS8	111,0X0,00X
20 0002	REQUEST (WRITE) OR RELINQUISH (READ) VME BUS (BYTE OR WORD)	BS8	111,0X0,00X
20 0004	DHB FLAG (BYTE OR WORD READ BIT D0)	BS8	111,0X0,00X
20 0006	AUGMENTED XFER ADDRESS COUNTER LOAD ADDRESS (WORD)	BS8	111,0X0,00X
24 0000	PC INTERFACE FIFO (WORD)	BS9	111,0X0,00X
24 0002	PC INTERFACE STATUS/CONTROL REGISTER (WORD)	BS9	111,0X0,00X
24 0004	PC INTERFACE INTERRUPT REGISTER (WORD)	BS9	111,0X0,00X
28 0001 - 28 001B	MVME6000 LCSR (ODD BYTES)	BS10	111,0X0,00X
28 0021 - 28 002F	MVME6000 GCSR (ODD BYTES)	BS10	111,0X0,00X
2000 0000	DSACK SRAM ENABLE		
6000 0000	DSACK SRAM DISABLE		

VPH HEX ADDRESS MAP

HEX ADDRESS	RESOURCE
0 - 7FFF or FFFF	EPROM
4 0000 - 5 FFFF	020 SRAM
8 0000 - 8 3FFF	020 4-PORT SRAM
C 0000 - C 0FFF	ZORAN #1
C 1000 - C 1FFF	ZORAN #2
10 0000 - 11 FFFF	ZORAN BUS #1 PRAM
14 0000 - 14 0FFF	ZORAN #3
14 1000 - 14 1FFF	ZORAN #4
18 0000 - 19 FFFF	ZORAN BUS #2 PRAM
1C 0000	VPH STATUS LATCH
1C 0004	ZORAN RESET LATCH
20 0000	CPH ADDRESS (FC=011 FOR AUGMENTED XFERS) (SELECTABLE AT CPH)
20 0002	REQUEST (WRITE) OR RELINQUISH (READ) VME BUS (BYTE OR WORD)
20 0004	DHB FLAG (BYTE OR WORD READ BIT D0)
20 0006	AUGMENTED XFER ADDRESS COUNTER LOAD ADDRESS (WORD)
24 0000	PC INTERFACE FIFO (WORD)
24 0002	PC INTERFACE STATUS/CONTROL REGISTER (WORD)
24 0004	PC INTERFACE INTERRUPT REGISTER (WORD)
28 0001 - 28 001B	MVME6000 LCSR (ODD BYTES)
28 0021 - 28 002F	MVME6000 GCSR (ODD BYTES)
2000 0000	DSACK SRAM ENABLE
6000 0000	DSACK SRAM DISABLE

MVME6000 Configuration Information

The Base Address (BA) of the MVME6000 in the 020 address space is BA = $0028 0000.

Power-up Sequence:

Write $0 to BA + $01 - this clears the BRDFAIL bit in the LCSR. Also selects priority arbitration.

Write $20 to BA + $05 - this tells the MVME6000 that the local processor is a 68020.

Write $6A to BA + $09 - this sets up all bus timers (see page 4-10 in MVME6000 manual).

Write $8D to BA + $0D - this configures the MVME6000 to use AM code $0D for all VMEbus master transactions.

APPENDIX H

IOP PROGRAMS

```
 1 - /* IOPBOOT.ASM       CREATED: 6/1/92
 2 -                      LAST MODIFIED:6/2/92
 3 -
 4 -         THIS SUBROUTINE IS INTENDED TO WAKE-UP THE
 5 -         THE IOP FROM THE IBMPC INTERFACE.  IT IS ASSUMED
 6 -         THAT THE BOOT STATE MACHINE PAL IS PROGRAMMED SUCH
 7 -         THAT THE IBMPC INTERFACE IS SELECTED AS THE HOST
 8 -
 9 - */
10 -
11 -
12 - PROGRAM CODESEG IORAM
13 - ORG 0
14 - START:  $SEQ CONT ;     /*NECESSARY CONT INSTRUCTION FOR USEQ*/
15 -         $SEQ CONT
16 -         $SEQ.REG ALL,CLR
17 -         $CRA SOURCE;    /*CLEAR ALL RESOURCE FLAGS*/
18 -
19 -    /*INITIALIZE INTERRUPT VECTOR POINTERS*/
20 -
21 -
22 - PROGRAM ENDS
```

```
1 - /*
2 -     ++++++++++++++++++++++++++++++++++++++++++++
3 -     +                                          +
4 -     +  THIS PROGRAM NEEDS TO BE MODIFIED TO    +
5 -     +  LOAD THE COUNTER A ZERO INTERRUPT       +
6 -     +  VECTOR TO POINT TO A SERVICE ROUTINE    +
7 -     +  THAT WILL TERMINATE THE TRANSFER        +
8 -     +                                          +
9 -     ++++++++++++++++++++++++++++++++++++++++++++
10 - */
11 -
12 - /* DNLDIOP.ASM       CREATED: 6/2/92
13 -                      LAST MODIFIED:
14 -
15 -         THIS SUBROUTINE IS INTENDED TO DOWNLOAD PC
16 -         MICRO CODE TO THE IOP MACRO RAM.  A REAL COUNTER
17 -         VALUE WILL NEED TO BE USED TO LOAD COUNTER A.  THE
18 -         MAXIMUM COUNT NUMBER IS USED HERE.
19 -
20 - */
21 -
22 - PROGRAM CODESEG IORAM
23 - ORG 0
24 - START:  $SEQ DISIR;           /*DISABLE USEQUENCER INTERRUPTS*/
25 -
26 - LOOP:   $SEQ CONT             /*IS RESOURCE AVAILABLE??*/
27 -         $CRA SOURCE
28 -         $CCS ZCNTA;          /*CHECK FOR COUNTER A = 0 */
29 -
30 -         $SEQ JPCNF, LOOP;    /*WAIT FOR COUNTER A TO COUNT TO ZERO*/
31 -
32 - LOOP1:  $SEQ CONT             /*IS RESOURCE AVAILABLE??*/
33 -         $CRA SOURCE
34 -         $CCS PCT;            /*CHECK FOR PC TRANSMIT AVAILABILITY*/
35 -
36 -         $SEQ JPCNF, LOOP1;   /*WAIT FOR PC TRANSMIT AVAILABLE*/
37 -
38 -         $SEQ CONT
39 -         $CR CRW              /*WRITE CONTROL REGISTER */
40 -         $CRA SOURCE
41 -         $SEQ.REG ENR19, SET; /*SET COUNTER A BUSY FLAG*/
42 -
43 -         $SEQ CONT
44 -         $CR CRW              /*WRITE CONTROL REGISTER */
45 -         $CRA SOURCE
46 -         $SEQ.REG ENR12, SET; /*SET IBM-PC SEND BUSY FLAG*/
47 -
48 -         $SEQ CONT
49 -         $CRA CNTA
50 -         $SEQ.REG 0B000000000001, 0B111111111111; /*LOAD COUNTER A WITH MAXIMUM*/
51 -
52 -         $SEQ CONT
53 -         $CR CRW
54 -         $CRA PCTRAN
55 -         $SEQ.REG ENR12,SET;  /*RESET PC XMIT INTERFACE*/
56 -
57 -         $SEQ CONT
58 -         $CR CRW
59 -         $CRA PCTRAN
60 -         $SEQ.REG ENR12,CLR;  /*READY PC XMIT INTERFACE*/
61 -
62 -         $SEQ CONT
63 -         $CR CRW
64 -         $CRA PCTRAN
65 -         $SEQ.REG PCXSET,PSDATA; /*SET PC XMIT FOR CLK A\32 BIT REAL DATA*/
66 -
67 -         $SEQ CONT;     /* +++ MYSTERY CODE TO LOAD COUNTER A=0 INT VECTOR +++ */
68 -
69 -         $SEQ CONT
70 -         $CR  MWR,IBMPC;  /*IBM PC SELECTED AS SOURCE-MACRO RAM DESTINATION*/
71 -
72 -
73 -         $SEQ CONT
74 -         $CR CRW
75 -         $CRA PCTRAN
76 -         $SEQ.REG ENR17, SET; /*SET PGO BIT (LOW) TO BEGIN TRANSFER*/
77 -
78 -         $SEQ ENAIR;          /*ENABLE USEQUENCER INTERRUPTS*/
79 -
80 - /*COUNTER A ZERO INTERRUPT FUNCTION:
81 -         DISABLE INTERRUPTS
82 -         CLEAR PGO BIT
83 -         RESET ALL BUSY FLAGS
84 -         RESET COUNTER INTERRUPT VECTOR
85 -         RESET ALL USED PORTS
86 -         ENABLE INTERRUPTS
87 - */
88 -
89 -
90 - PROGRAM ENDS
```

```
  1 - /*
  2 -     ++++++++++++++++++++++++++++++++++++++++++
  3 -     +                                        +
  4 -     +  THIS PROGRAM NEEDS TO BE MODIFIED TO  +
  5 -     +  LOAD THE COUNTER A ZERO INTERRUPT     +
  6 -     +  VECTOR TO POINT TO A SERVICE ROUTINE  +
  7 -     +  THAT WILL TERMINATE THE TRANSFER      +
  8 -     +                                        +
  9 -     ++++++++++++++++++++++++++++++++++++++++++
 10 - */
 11 -
 12 - /* UPLDDAT.ASM       CREATED: 6/2/92
 13 -                      LAST MODIFIED:
 14 -
 15 -         THIS SUBROUTINE IS INTENDED TO UPLOAD CACHE
 16 -         DATA TO AN IBM PC.  A REAL COUNTER
 17 -         VALUE WILL NEED TO BE USED TO LOAD COUNTER A.  THE
 18 -         MAXIMUM COUNT NUMBER IS USED HERE.
 19 -
 20 - */
 21 -
 22 - PROGRAM CODESEG IORAM
 23 - ORG 0
 24 - START:  $SEQ DISIR;           /*DISABLE USEQUENCER INTERRUPTS*/
 25 -
 26 - LOOP:   $SEQ CONT              /*IS RESOURCE AVAILABLE??*/
 27 -         $CRA SOURCE
 28 -         $CCS ZCNTA;           /*CHECK FOR COUNTER A = 0 */
 29 -
 30 -         $SEQ JPCNF, LOOP;     /*WAIT FOR COUNTER A TO COUNT TO ZERO*/
 31 -
 32 - LOOP1:  $SEQ CONT              /*IS RESOURCE AVAILABLE??*/
 33 -         $CRA SOURCE
 34 -         $CCS PCR;             /*CHECK FOR PC RECEIVE AVAILABILITY*/
 35 -
 36 -         $SEQ JPCNF, LOOP1;    /*WAIT FOR PC RECEIVE AVAILABLE*/
 37 -
 38 - LOOP2:  $SEQ CONT
 39 -         $CRA SOURCE
 40 -         $CCS HSIOT;           /*CHECK FOR HSIO TRANSMIT AVAILABILITY*/
 41 -
 42 -         $SEQ JPCNF, LOOP2;    /*WAIT FOR HSIO TRANSMIT AVAILABLE*/
 43 -
 44 -         $SEQ CONT
 45 -         $CR CRW               /*WRITE CONTROL REGISTER */
 46 -         $CRA SOURCE
 47 -         $SEQ.REG ENR19, SET; /*SET COUNTER A BUSY FLAG*/
 48 -
 49 -         $SEQ CONT
 50 -         $CR CRW               /*WRITE CONTROL REGISTER */
 51 -         $CRA SOURCE
 52 -         $SEQ.REG ENR13, SET; /*SET IBM-PC RECEIVE BUSY FLAG*/
 53 -
 54 -         $SEQ CONT
 55 -         $CR CRW               /*WRITE CONTROL REGISTER */
 56 -         $CRA SOURCE
 57 -         $SEQ.REG ENR16, SET; /*SET HSIO SEND BUSY FLAG*/
 58 -
 59 -         $SEQ CONT
 60 -         $CRA CNTA
 61 -         $SEQ.REG 0X001, 0XFFF; /*LOAD COUNTER A WITH MAXIMUM*/
 62 -
 63 -         $SEQ CONT
 64 -         $CR CRW
 65 -         $CRA PCSTAT
 66 -         $SEQ.REG ENR12,SET;  /*RESET PC RECEIVE INTERFACE*/
 67 -
 68 -         $SEQ CONT
 69 -         $CR CRW
 70 -         $CRA PCSTAT
 71 -         $SEQ.REG ENR12,CLR;  /*READY PC RECEIVE INTERFACE*/
 72 -
 73 -         $SEQ CONT
 74 -         $CR CRW
 75 -         $CRA HSIO             /*SET HSIO FOR MEMORY READ, CLK A*/
 76 -         $SEQ.REG 0XFEC, 0XF7A; /*32 BIT REAL DATA*/
 77 -
 78 -         $SEQ CONT
 79 -         $CR CRW
 80 -         $CRA PCSTAT
 81 -         $SEQ.REG ENR19,SET; /*SET PC RECEIVE TO ALLOW INTERFACE TO SEND*/
 82 -
 83 -         $SEQ CONT;     /* +++ MYSTERY CODE TO LOAD COUNTER A=0 INT VECTOR +++ */
 84 -
 85 -         $SEQ CONT
 86 -         $CR ,HSIO;            /*HSIO SELECTED AS SOURCE*/
 87 -
 88 -         $SEQ CONT
 89 -         $CR CRW
 90 -         $CRA HSIO
 91 -         $SEQ.REG ENR14, SET; /*SET HGO BIT (LOW) TO BEGIN TRANSFER*/
 92 -
 93 -         $SEQ ENAIR;           /*ENABLE USEQUENCER INTERRUPTS*/
 94 -
 95 - /*COUNTER A ZERO INTERRUPT FUNCTION:
 96 -         DISABLE INTERRUPTS
 97 -         CLEAR HGO BIT
 98 -         RESET ALL BUSY FLAGS
 99 -         RESET COUNTER INTERRUPT VECTOR
100 -         RESET ALL USED PORTS
101 -         ENABLE INTERRUPTS
102 - */
103 -
104 - PROGRAM ENDS
```

```
 1 - /*
 2 -     ++++++++++++++++++++++++++++++++++++++++++++
 3 -     +                                          +
 4 -     +  THIS PROGRAM NEEDS TO BE MODIFIED TO    +
 5 -     +  LOAD THE COUNTER A ZERO INTERRUPT       +
 6 -     +  VECTOR TO POINT TO A SERVICE ROUTINE    +
 7 -     +  THAT WILL TERMINATE THE TRANSFER        +
 8 -     +                                          +
 9 -     ++++++++++++++++++++++++++++++++++++++++++++
10 - */
11 -
12 - /* DNLDCPH.ASM       CREATED: 6/2/92
13 -                      LAST MODIFIED:
14 -
15 -          THIS SUBROUTINE IS INTENDED TO DOWNLOAD PC
16 -          MICRO CODE TO THE CHP.  A REAL COUNTER
17 -          VALUE WILL NEED TO BE USED TO LOAD COUNTER A.  THE
18 -          MAXIMUM COUNT NUMBER IS USED HERE.
19 -
20 - */
21 -
22 - PROGRAM CODESEG IORAM
23 - ORG 0
24 - START:  $SEQ DISIR;           /*DISABLE USEQUENCER INTERRUPTS*/
25 -
26 - LOOP:   $SEQ CONT             /*IS RESOURCE AVAILABLE??*/
27 -          $CRA SOURCE
28 -          $CCS ZCNTA;          /*CHECK FOR COUNTER A = 0 */
29 -
30 -          $SEQ JPCNF, LOOP;    /*WAIT FOR COUNTER A TO COUNT TO ZERO*/
31 -
32 - LOOP1:  $SEQ CONT             /*IS RESOURCE AVAILABLE??*/
33 -          $CRA SOURCE
34 -          $CCS PCT;            /*CHECK FOR PC TRANSMIT AVAILABILITY*/
35 -
36 -          $SEQ JPCNF, LOOP1;   /*WAIT FOR PC TRANSMIT AVAILABLE*/
37 -
38 - LOOP2:  $SEQ CONT
39 -          $CRA SOURCE
40 -          $CCS HSIOR;          /*CHECK FOR HSIO RECEIVE AVAILABILITY*/
41 -
42 -          $SEQ JPCNF, LOOP2;   /*WAIT FOR HSIO RECEIVE AVAILABLE*/
43 -
44 -          $SEQ CONT
45 -          $CR CRW              /*WRITE CONTROL REGISTER */
46 -          $CRA SOURCE
47 -          $SEQ.REG ENR19, SET; /*SET COUNTER A BUSY FLAG*/
48 -
49 -          $SEQ CONT
50 -          $CR CRW              /*WRITE CONTROL REGISTER */
51 -          $CRA SOURCE
52 -          $SEQ.REG ENR12, SET; /*SET IBM-PC SEND BUSY FLAG*/
53 -
54 -          $SEQ CONT
55 -          $CR CRW              /*WRITE CONTROL REGISTER */
56 -          $CRA SOURCE
57 -          $SEQ.REG ENR17, SET; /*SET HSIO RECEIVE BUSY FLAG*/
58 -
59 -          $SEQ CONT
60 -          $CRA CNTA
61 -          $SEQ.REG 0X001, 0XFFF; /*LOAD COUNTER A WITH MAXIMUM*/
62 -
63 -          $SEQ CONT
64 -          $CR CRW
65 -          $CRA PCTRAN
66 -          $SEQ.REG ENR12,SET;  /*RESET PC XMIT INTERFACE*/
67 -
68 -          $SEQ CONT
69 -          $CR CRW
70 -          $CRA PCTRAN
71 -          $SEQ.REG ENR12,CLR;  /*READY PC XMIT INTERFACE*/
72 -
73 -          $SEQ CONT
74 -          $CR CRW
75 -          $CRA PCTRAN
76 -          $SEQ.REG PCXSET,PSDATA; /*SET PC XMIT FOR CLK A\32 BIT REAL DATA*/
77 -
78 -          $SEQ CONT
79 -          $CR CRW
80 -          $CRA HSIO
81 -          $SEQ.REG ALL,CLR;    /* RESET THE HSIO INTERFACE */
82 -
83 -          $SEQ CONT
84 -          $CR CRW
85 -          $CRA HSIO              /*SET HSIO TO RECEICE PC 32 BIT REAL DATA*/
86 -          $SEQ.REG HSIOINT,HIOPC;  /*AND USE COUNTER A*/
87 -
88 -          $SEQ CONT
89 -          $CR CRW
90 -          $CRA HSIO
91 -          $SEQ.REG ENR16,SET; /*ENABLE HSIO IN I/O WRITE MODE*/
92 -
93 -          $SEQ CONT;     /* +++ MYSTERY CODE TO LOAD COUNTER A=0 INT VECTOR +++ */
94 -
95 -          $SEQ CONT
96 -          $CR ,IBMPC;           /*IBM PC SELECTED AS SOURCE*/
97 -
98 -          $SEQ CONT
99 -          $CR CRW
100 -         $CRA PCTRAN
101 -         $SEQ.REG ENR17, SET; /*SET PGO BIT (LOW) TO BEGIN TRANSFER*/
102 -
103 -         $SEQ ENAIR;          /*ENABLE USEQUENCER INTERRUPTS*/
104 -
105 - /*COUNTER A ZERO INTERRUPT FUNCTION:
106 -         DISABLE INTERRUPTS
107 -         CLEAR PGO BIT
108 -         RESET ALL BUSY FLAGS
109 -         RESET COUNTER INTERRUPT VECTOR
```

```
110 -           RESET ALL USED PORTS
111 -           ENABLE INTERRUPTS
112 - */
113 -
114 -
115 - PROGRAM ENDS
```