The SPARCengine™ 1E CPU Card User's Manual

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-8137-02
Revision A of April 10, 1990

Contents

Figures

Tables

Preface

The SPARCengine 1E CPU User's Manual provides detailed information about the two CPU cards in the SPARCengine 1E family.

When you have read this User's Manual and the associated required reference material, you will be able to communicate with the SPARCengine 1E CPU card in many system configurations.

Using this Manual

The following sections provide information that will help you use this manual.

Audience

The audience of this manual includes computer hardware engineers, system programmers, computer technicians, and others interested in interfacing one or more SPARCengine 1E cards into a VME backplane, and how to communicate with other cards in the SPARCengine 1E card family through the VME, SBus and P2 Bus ports. All readers should have an understanding of electronic hardware, operating system software interfacing with hardware, and related computer concepts.

An understanding of the required communications standards and concepts between cards and with peripheral computer devices is also required. Given the defined audience background, the Hardware Reference Manual will provide the necessary information needed to incorporate the SPARCengine 1E family of cards into many system configurations.

Organization

The organization of this User's Manual is designed to be of immediate use to you in a fashion that is clear, precise, and organized. There are four main groups of information in the manual:

Major Sections of the SPARCengine 1E CPU User's Manual	Chapters of each Major Section	Titles of each Chapter
Bringing Up the SPARCengine 1E for the First Time	Chapter 1 Chapter 2 Chapter 3	Unpacking the CPU Card Description of the CPU Card Bringing Up the CPU Card for the First Time
The Theory and Operation of the SPARCengine 1E CPU Card	Chapter 4 Chapter 5 Chapter 6 Chapter 7 Chapter 8 Chapter 9 Chapter 10 Chapter 11 Chapter 12	CPU Card Overview Control Space Device Space Memory Management Unit Boot PROM Cache Diagnostics System Reset & Reset Switch Multiprocessing Capabilities
The Input/Output Interfaces of the SPARCengine 1E CPU Card	Chapter 13 Chapter 14 Chapter 15 Chapter 16 Chapter 17 Chapter 18 Chapter 19 Chapter 20	VMEbus Interface SCSI Interface Ethernet Interface SBus Interface P2 Interface Serial A Interface Serial B Interface Keyboard/Mouse Interface
The Appendix	Appendix A Appendix B Appendix C Appendix D	Environmental Tests and Results Getting Help Schematic Diagrams & Assembly Drawings Manual Updates

For information about other cards in the SPARCengine 1E card family, refer to the additional manuals of the SPARCengine 1E User's Manuals.

Fonts in Text

In this manual, typographic fonts are used to make things a little clearer. The most common fonts are Roman, *italic*, and **bold**. We use them as follows:

Roman

Roman font is the standard for normal text, just as it appears here.

Italic

Italic font is used for four types of information:

Reference Manual titles,

notes,

figure and table titles,

or it represents a variable for which you or the computer must substitute some real value. For example:

This field contains the value *nn*

Bold

Bold font is used to indicate three types of information:

chapter names when referencing from one chapter to another within the manual,

headers in tables,

or when something deserves more attention than the surrounding text.

Textual Conventions

A '0x' before a number indicates that the number is hexadecimal. For example, 0x16 indicates hexadecimal value '16'.

In discussions of the state of a signal or a bit, the bit might be *active high* or *active low*. It if is active high, it is true when it is *active*, *HIGH*, or *set*, and it is false if it is *inactive*, *low*, or *reset*. Most bits are active high.

Bits that are active low are identified as such in text, and are marked with an asterisk (*).

Required Reference Material

On the next page is a table describing the necessary information required for understanding certain components of the SPARCengine 1E CPU card. Make sure that you have copies of this documentation before beginning study of this manual.

Where Found	Manual Title
Supplied with SunOS Documentation	***PROM User's Manual,*** Sun Part No. 800-1736-xx
	Release Manual for SunOS 4.0.3e Sun Part No. 800-1835-xx
	The SPARC Architecture Manual Sun Part No. 800-1399-xx
Supplied with SPARCengine 1E Hardware Documentation	***The SPARCengine 1E Color & Monochrome Video Cards User's Manual,*** Sun Part No. 800-8139-xx
	SBus Developer's Kit, Sun Part No. 825-1219-xx (NOTE: See chapter entitled **Boot PROM**)
Documentation You Can Obtain Outside of Sun	***AmZ8030/AmZ8530 Serial Communications Controller (SCC) Technical Manual,*** Advanced Micro Devices, Inc., 1982.
	Advanced Micro Devices Am7990 Local Area Network Controller (LANCE) Technical Manual, Advanced Micro Devices, Inc., 1986.
	Advanced Micro Devices Am7992B Serial Interface Adapter (SIA) Technical Manual, Advanced Micro Devices, Inc., October, 1985.
	The Mostek MK48T02-15 Data Sheet.
	ANSI Specification 802.3, 1986, (Ethernet Specification) also known as: ***ISO/Draft International Standard 8802/3*** ***Federal Information Processing Standard (FIPS) 107***
	VMEbus Specification Revision C.1, October, 1985. Also known as: ***IEC 821 BUS and*** ***IEEE P1014/D1.2***
	EIA Standard -- RS-232-C, Electronic Industries Association, August, 1969.
	EIA Standard -- RS-422-A, Electronic Industries Association, December, 1978
	EIA Standard -- RS-423-A, Electronic Industries Association, December, 1978.
	EIA Standard -- RS-449-A, Electronic Industries Association, December, 1977.

Revision History

Dash	Revision	Date	Comments
01	01	August 22, 1989	Beta 1st Draft
01	50	November 6, 1989	Beta Release
01	51	November 30, 1989	TOI Release
02	01	February 19, 1990	FCS 1st Draft
02	01	April 4, 1990	FCS 2nd Draft
02	A	April 10, 1990	FCS Release

1

Unpacking the CPU Card

Your SPARCengine 1E CPU Card is delivered in a protective box. The box contains the card wrapped in a static envelope, and a electrostatic protection (ESD) kit.

1. Disassemble the box, removing the card in the static envelope and the static electricity kit.

2. Disassemble the ESD kit, and follow the instructions supplied with the kit to attach the ESD device.

3. Remove the static envelope from the card.

2

Description of the CPU Card

The SPARCengine 1E CPU card is a RISC/UNIX Eurocard with I/O of VME, Ethernet, SBus, Serial 232/423/422, and keyboard/mouse interface. The SBus is usually occupied with a SPARCengine 1E video card when the CPU is used with a monitor.

There are two SPARCengine 1E CPU cards, differentiated only by the inclusion or exclusion of the Floating Point Unit.

2.1. Main Features

- Low power CMOS ASICs for high reliability.
- RISC Central Processing Unit
- Floating Point Unit.
- Wide range of I/O, including IEEE/ANSI-standard 32-bit VMEbus.
- 4MB Parity Memory on-board.
- ECC Memory via backplane P2Bus.
- Color/Monochrome Video SBus cards.
- Compatible with all SPARC-based workstations & servers.
- SunOS 4.0.3 compatible.
- Cache.
- Multiple CPU cards in a single backplane.

2.2. Card Specifications

Category	Specification
Integer Performance	12.5 MIPS @ 20MHz
Cache Memory	64KB write-through, virtual
Floating-Point Unit	1.4 MFLOPS Double Precision (Optional)
On-board Byte Parity Memory	4MB, 100-nsec DRAM SIPs
Memory Management	Sun-4 MMU ASIC
EPROM	256KB EPROM (2x27010)
Realtime Timer/Counters	Two 21-bit, 1-usec. resolution
TOD Clock Configuration Parameters	M48T02 2KB NVRAM
Multiprocessing Support through the VME Interface	32 mailbox interrupt locations RMW Multiport Shared RAM
VMEbus	IEEE/ANSI standard 1014, A32; D32
SBus Connectors	1 locations Master/slave, 32-bit
SCSI Bus	NCR 53C90, Mini-connector
Ethernet	AMD 7990; DB15 connector
Serial Port A	RS423/232C; sync/async
Serial Port B	RS422/232C; sync/differential
Keyboard/Mouse	X920B Sun Type-4 standard, DB-15
Board Size	6.29" x 9.18" (160 x 233 mm)
Power Dissipation	25 Watts

2.3. Card Landmarks

On the following page you can find a component-side drawing of the SPARCengine 1E CPU card, with various call-outs defining the card landmarks.

Figure 2-1 *CPU Card Landmarks -- Major Chips*

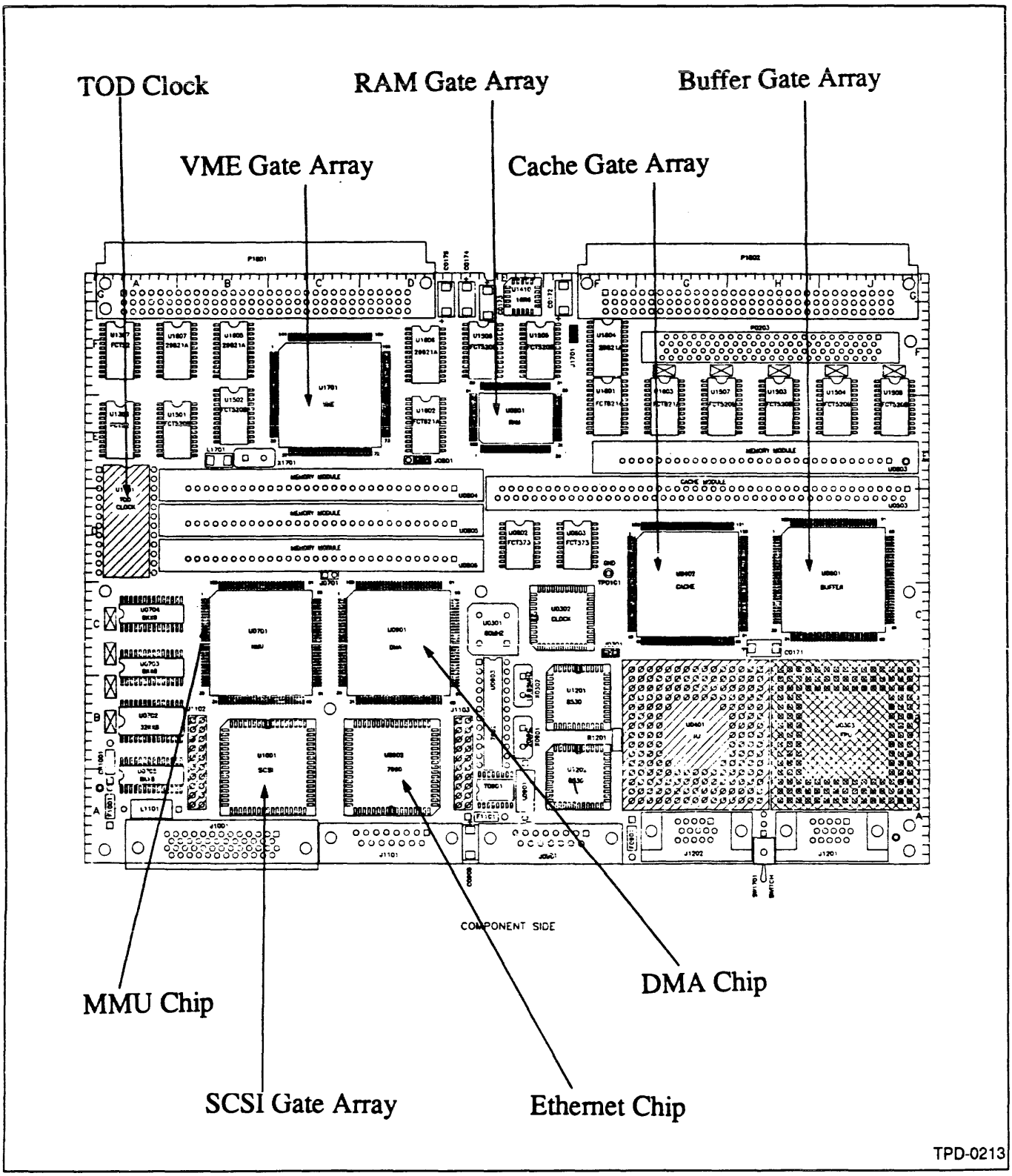


Figure 2-2 *CPU Card Landmarks -- I/O Connectors*

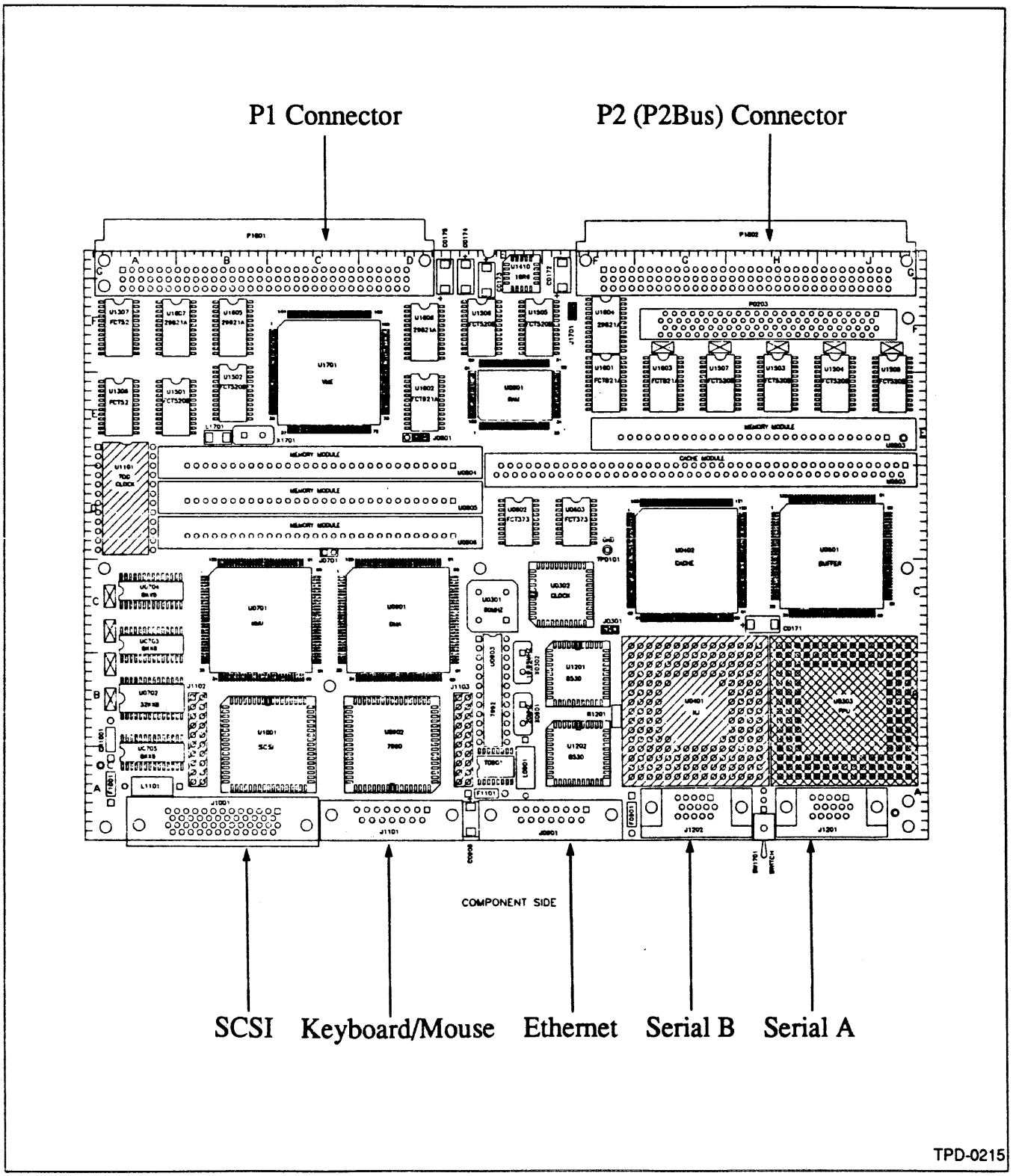
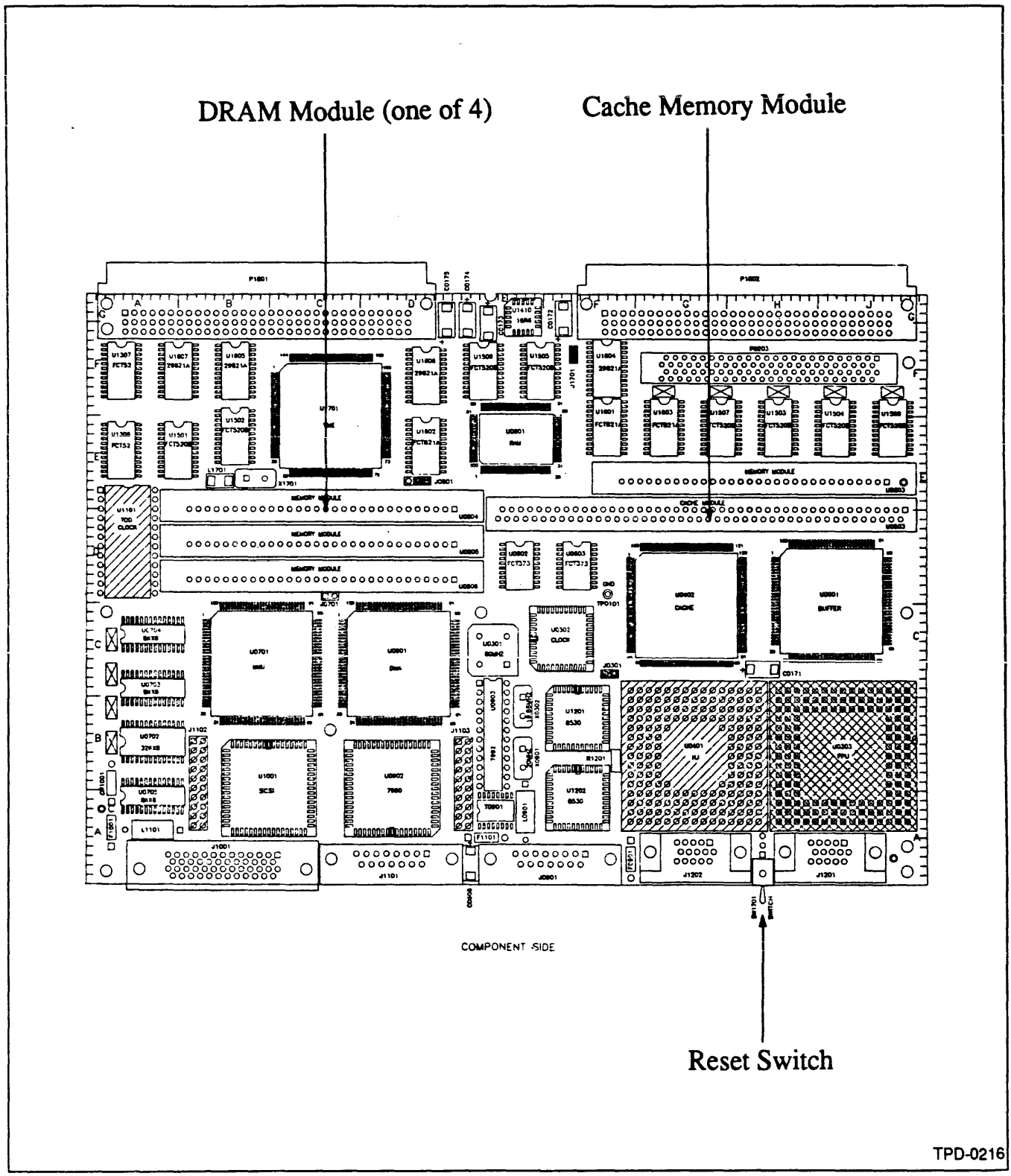


Figure 2-3 *CPU Card Landmarks -- Memory Modules*



3

Bringing Up the SPARCengine 1E CPU for the First Time

3.1. Required Reference Material

VMEbus Specification Revision C.1, 1987.

Open Boot PROM Toolkit User's Manual, contained in the ***SBus Developer's Kit***, Sun Part No. 825-1219-xx.

PROM User's Manual, Sun Part No. 800-1736-xx.

Here are the instructions and considerations for powering up your new SPARCengine 1E for the first time. The information included here is reduced to the basics, and is meant for you to verify that your SPARCengine 1E is operational and is ready for further activities. The configurations included in this section are ONLY for first-time power-up, and are not an exhaustive explanation of the various configurations that can be achieved with the SPARCengine 1E CPU card.

3.2. Backplane Definition

The SPARCengine 1E is a double-height board requiring both a J1 and a J2 backplane (or a combination J1/J2 backplane)

The user-defined J2/P2 pin assignments are made according to Sun's SPARCengine 1E P2 Bus private specification. (For more information on the P2 Bus specifications, see the chapter in this manual entitled **P2 Bus Interface**.)

The SPARCengine 1E is designed to fully comply with the specifications of the VMEbus. (For more information on the VMEbus specifications, see sections 7.4, 7.5 and 7.6 of the ***VMEbus Specification Revision C.1***.)

3.3. Board Jumpers

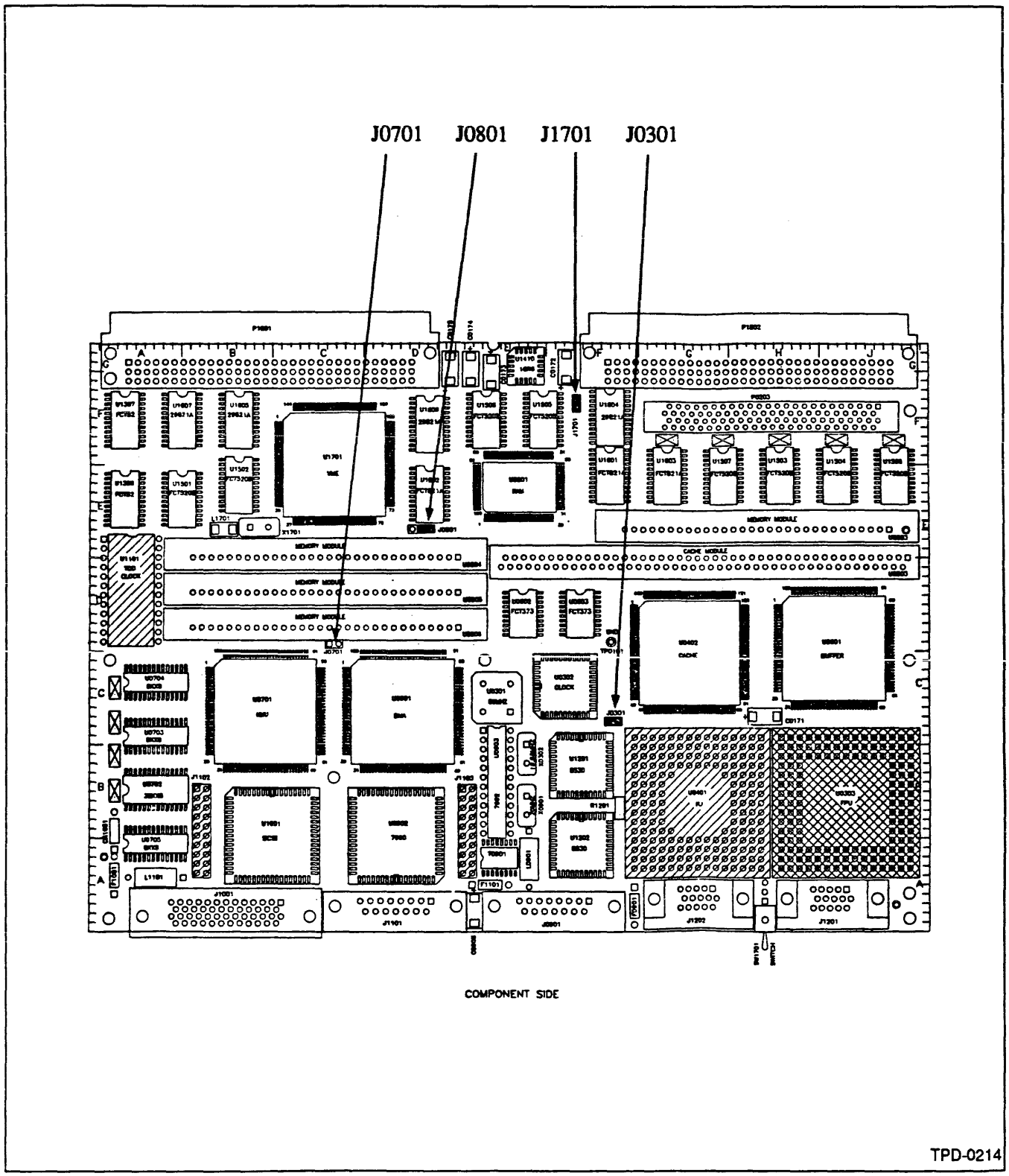
Below is a chart defining how the jumpers on the CPU card have been set at the Sun factory. Check the jumpers to ensure that the jumpers are set in the fashion described below.

Table 3-1 *Board Jumper Table*

Reference Designator	Jumper	Available Options * indicates Sun factory setting
J0301	Clock Enable	The jumper must be used. *1. Pins 1 and 2 must be jumpered together for normal operation.
J0801	ECC/Parity	The jumper must be used. *1. Jumpering pins 1 and 2 together selects on-board parity memory Memory Select to be at the bottom of Type 0 space below off-board ECC memory. 2. Jumpering pins 2 and 3 together selects off-board ECC memory to be at the bottom of Type 0 space and disables on-board parity memory. **NOTE: See the SPARCengine 1E ECC Memory Card User's Manual to configure the jumpers resident on the ECC Memory Card.**
J1701	VME Slot 1	The jumper does not need to be used. *1. IN selects the board to be a VME Slot 1 device. 2. OUT selects the board to be a VME non-Slot 1 device. **NOTE: See the chapter of this manual called VMEbus Interface for more information.**
J0702	SBus Select 2	The jumper does not need to be used. 1. IN supports future SBus expansion. *2. OUT supports SBus cards with parity option.

NOTE: Pin 1 for jumpers is indicated by a square on the board silkscreen.

Figure 3-1 *CPU Card Jumper Location & Factory Settings*



3.4. Backplane Slot Configuration Requirements

The SPARCengine 1E can be plugged into the VME backplane in slot 1 or in a slot other than slot 1.

If a SPARCengine 1E CPU is installed in a slot other than slot 1 with empty slots between it and the slot 1 card, then each of those empty slots must be configured as described below:

IACKIN* and IACKOUT* must be jumpered together.

Pins 21 and 22 on the J1 backplane must be jumpered together.

The signals Bus-Grant(0-3) and IACK must be jumpered across empty slots, as follows:

BG0 : jumper pins 5 and 6
BG1 : jumper pins 7 and 8
BG2 : jumper pins 9 and 10
BG3 : jumper pins 11 and 12
IACK : jumper pins 21 and 22

The following chart shows how to arrange the SPARCengine 1E family of cards in the VME backplane. It assumes that the CPU will be placed in Slot 1. If the CPU is placed in a different slot, the RELATIVE card ordering in the chart should still be observed.

NOTE: In the chart below, boards are shown on the left, and the slots they go into appear in the columns to the right. The relative priority is expressed with a letter from A, B, or C; A is highest priority, B is medium, and C is lowest. A slot labelled "A" is the first choice of locations for the board; if that slot is not available, place the board in the slot labelled "B", and if that slot is not available, please it in the slot labelled "C".

Table 3-2 *Board Positioning Chart*

Board	Slot 1	Slot 2	Slot 3	Slot 4	Slot 5	Slots 6-21
1E CPU	A					
1E Video		A				
1E ECC (#1)*		A	B			
1E ECC (#2)*			A	B		
1E ECC (#3)*				A	B	
1E ECC (#4)*					A	
SCSI/Ether (3E340)						Any Free Slot
Other VME Cards						Any Free Slot

* Be sure that the P2 Bus extends to this slot.

3.5. Insertion into a Backplane

SPARCengine 1E connectors P1 and P2 respectively plug into the backplane connectors J1 and J2. All four connectors are keyed to prevent misinsertion.

Connect SERIAL A via a cable to a terminal. See section 18.7 of this manual.

3.6. Power Up

The default power-up sequence consists of a series of minimal component functional tests and initialization, followed by booting.

Turn on the power to the backplane.

In the default autoboot mode, the SPARCengine 1E CPU attempts to boot SunOS 4.0.3e from an attached SCSI disk. Because a disk is not attached, the Open PROM displays an identification banner and enters the command mode of the "PROM monitor" (a program that monitors the activity of the keyboard). The PROM displays a > prompt.

How to Talk to the SPARCengine 1E 4MB On-Board Memory

Type *n* to enter the FORTH interpreter.

FORTH displays the **ok** prompt. Basic Assembly PROM commands are available to the SPARCengine 1E CPU in this interpreter.

There is at least 4MB of memory available to the SPARCengine 1E CPU card. The first tests to validate the correct operation of the SPARCengine 1E CPU card PROM is to validate correct operation of this memory. Use one of the three commands explained below:

Memory Test Command One

Perform a LOOP/READ that validates that the on-board memory is functional:

100 50 dump

RESULTS: If the PROM and the on-board memory is functional, a memory location table is displayed. If the PROM or the on-board memory is not working, you receive one of a number of possible error messages indicating a problem with the memory.

Memory Test Command Two

Perform a LOOP/WRITE that writes a number pattern to memory, validating memory and memory response:

100 50 12345678 lfill

RESULTS: If the SPARCengine 1E is working correctly, the number pattern 12345678 is written to memory, and you will receive the **ok** prompt. If the PROM or the on-board memory is not working correctly, you receive one of a number of possible error messages indicating a problem.

Memory Test Command Three

Perform a memory test that exercises the on-board memory. This test does not reside in the PROM, and must be keyed in at the FORTH prompt. Key in the following program:

```
: memory-test ( -- )
    150 100 do                          from 0x100 to 0x150
     12345678  i l!                     longword write
     i l@ dup                           longword read(result
                                        left on stack)
     12345678 <>                        compare
     if ." obs = " .
       ." exp = " 12345678 .
       ." adr = i . cr
     ekedrop
     then
    4 +loop
;
```

When this code has been correctly entered, you can perform the memory test:

memory-test

RESULTS: If the SPARCengine 1E is working correctly, the memory test is written to memory, and you will receive the **ok** prompt. If the PROM or the on-board memory is not working correctly, you receive one of a number of possible error messages indicating a problem.

Connect the cables for the additional devices you wish to test with the SPARCengine 1E: SCSI, keyboard, Ethernet, Serial Ports A and B.

3.7. How to Talk to the SPARCengine 1E Buses

Accessing devices available on the SPARCengine's buses in general requires mapping and reading and writing the device.

The procedure below can be used to map in memory (ECC or parity). Devices are mapped in two stages:

Step One

Select unused segments, virtual address range and size. Map in the segments, e.g., seg# = 0x80, va = 0x1000000, size = 0x800000.

80 1000000 800000 map-segments

NOTE: 0xff is conventionally used as the invalid pointer. Some segments are already being used, e.g., 0x0-0xF for 4MB parity memory for ECC memory low, 0xf8-0xfb for the framebuffer, 0xfd for ROM, 0xfe for RAM [for PROM].

Step Two

Select a physical address range and space. Map in the pages, e.g., pa = 0x40000, space = vmed32a32.

40000 vmed32a32 1000000 800000 map-pages

How to Talk to the VMEbus

VME devices must be mapped in using the above procedure. (Spaces available for use are vmed32a32, vmed32a24, vmed32a16, vmed16a32, vmed16a24, vmed16a16).

Example: to map 2MB of VME D32 memory at 0x800000:

```
use seg#=0x30, VA = 0x400000
30 400000 200000 map-segments
800000 vmed32a32 400000 200000 map-pages
```

Memory can then be dumped with:

800000 100 dump

How to Talk to the SBus

SBus devices are handled using the IDPROM onboard the SBus card. The IDPROM contains a driver for the SBus card which is read at boot time and interpreted by the Open PROM. During debug of an SBus device or it's driver a device in an SBus slot may be mapped in using **map-sbus**. (See the ***Open PROM Toolkit User's Manual*** for a description of the map-sbus).

How to Talk to the P-2 Bus

P2 devices must use the procedure below (space = obmem for P2 memory; space = obio for P2 I/O).

Example: to map 2MB of ECC memory at 800000:

```
use seg#=0x30, VA = 0x400000
30 400000 200000 map-segments
800000 obmem 400000 200000 map-pages
```

3.8. Running Extended Selftests

Setting the diagnostic switch and reseting the board will cause the board to come-up into the extended selftests.

setenv diag-switch? true
reset

NOTE *Results of this test are printed only on Serial Interface A.*

3.9. Running Functional Tests

The following tests are available for testing functional units on the board. These tests are automatically run at a reset or a power-up. They can be run individually by leaving the PROM monitor and entering the PROM.

At the > prompt, enter *n*.

At the **ok** prompt, enter one of the following commands.

test-control-regs
test-net
test-cache
test-memory
watch-clock
watch-net

NOTE *Press escape to abort or stop any test.*

4

Board Overview

4.1. Required Reference Material for the SPARCengine 1E

Reference material required for a complete definition of the SPARCengine 1E:

The SPARC Architecture Manual Sun Part No. 800-1399-xx

Release Manual for SunOS 4.0.3e Sun Part No. 800-1835-xx

4.2. Introduction

The SPARCengine 1E is a RISC/UNIX Eurocard for industry-standard double-high (6U) VMEbus and rugged applications. It features 12.5 MIPS and 1.4 MFLOPS performance, 4 MB of parity memory on-card, and off-card expansion up to 64 MB of ECC memory on a private P2 Bus. Other input/output ports include one SBus expansion slot, SCSI and Ethernet ports and two serial I/O ports. The CPU card contains both a cache and a memory management unit to achieve maximum use of the SPARC architecture.

The CPU card comes with or without a floating-point unit (FPU), depending upon what is ordered by the customer.

The CPU card uses the Sun SF9010 chip set to implement the Sun-4 architecture. It features a full 32-bit virtual address and 32-bit data capability, and it uses the SPARC RISC architecture to achieve maximum speed.

The SPARCengine 1E CPU runs SunOS 4.0.3e, a variant of the standard SunOS 4.0.3 for the Sun-4. Refer to the **Release Manual for SunOS 4.0.3e** for more details about the SunOS for the SPARCengine 1E.

NOTE *SunOS 4.0.3 and SunOS 4.0.3c (SPARCstation 1 SunOS) will not operate on the SPARCengine 1E.*

4.3. CPU

The CPU is comprised of an Integer Unit (IU) that performs basic processing and an optional Floating-Point Unit (FPU) that performs floating point calculations.

The IU includes a 32-bit external bus interface with separate data and address buses, a four-stage instruction pipeline, a barrel shifter, two data aligners, and a three-port register file consisting of 120 registers. These registers are configured into overlapping sets that facilitates the passing of parameters. All instructions with the exception of loads, stores, and floating-point operations can be executed in one machine cycle.

The IU and FPU are linked through a dedicated interface that supports concurrent floating- point instruction execution.

Refer to the SPARC Reference Manual (825-1080-01) for more details.

4.4. MMU

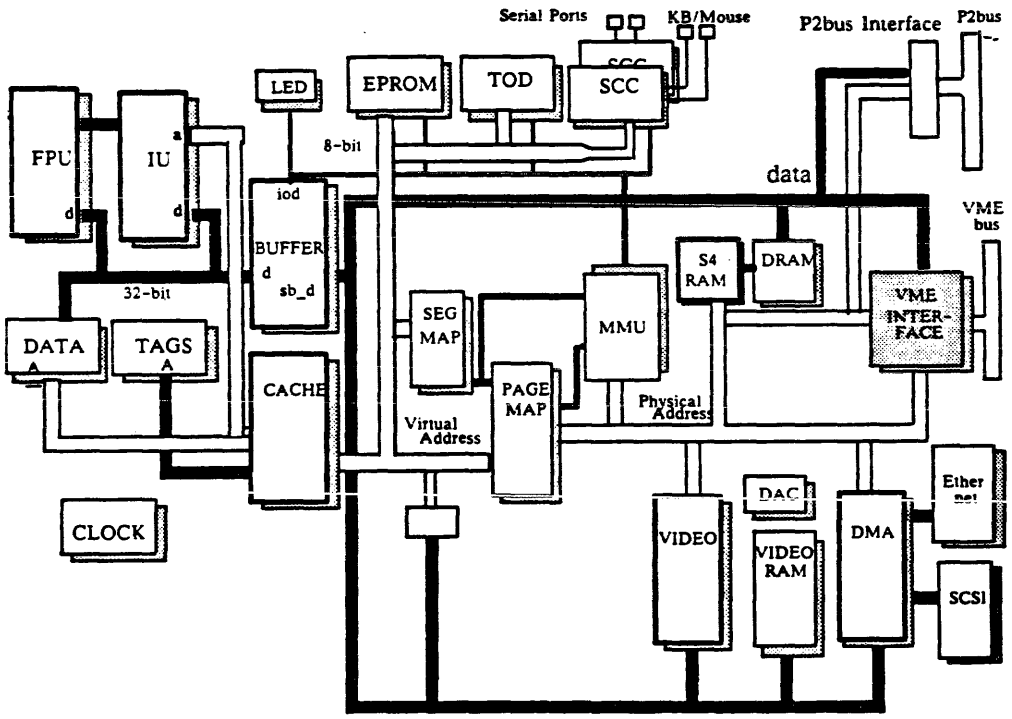
The Memory Management Unit (MMU) maps virtual addresses used by the SPARC processor into the physical addresses used by main memory.

The card architecture is divided between control space and device space. Control space contains the architectural extensions to the CPU, on the untranslated side of the MMU. Device space contains the devices on the translated side of the MMU. Control space is used for system control operations, and device space is used (mostly) for normal operation.

4.5. Cache

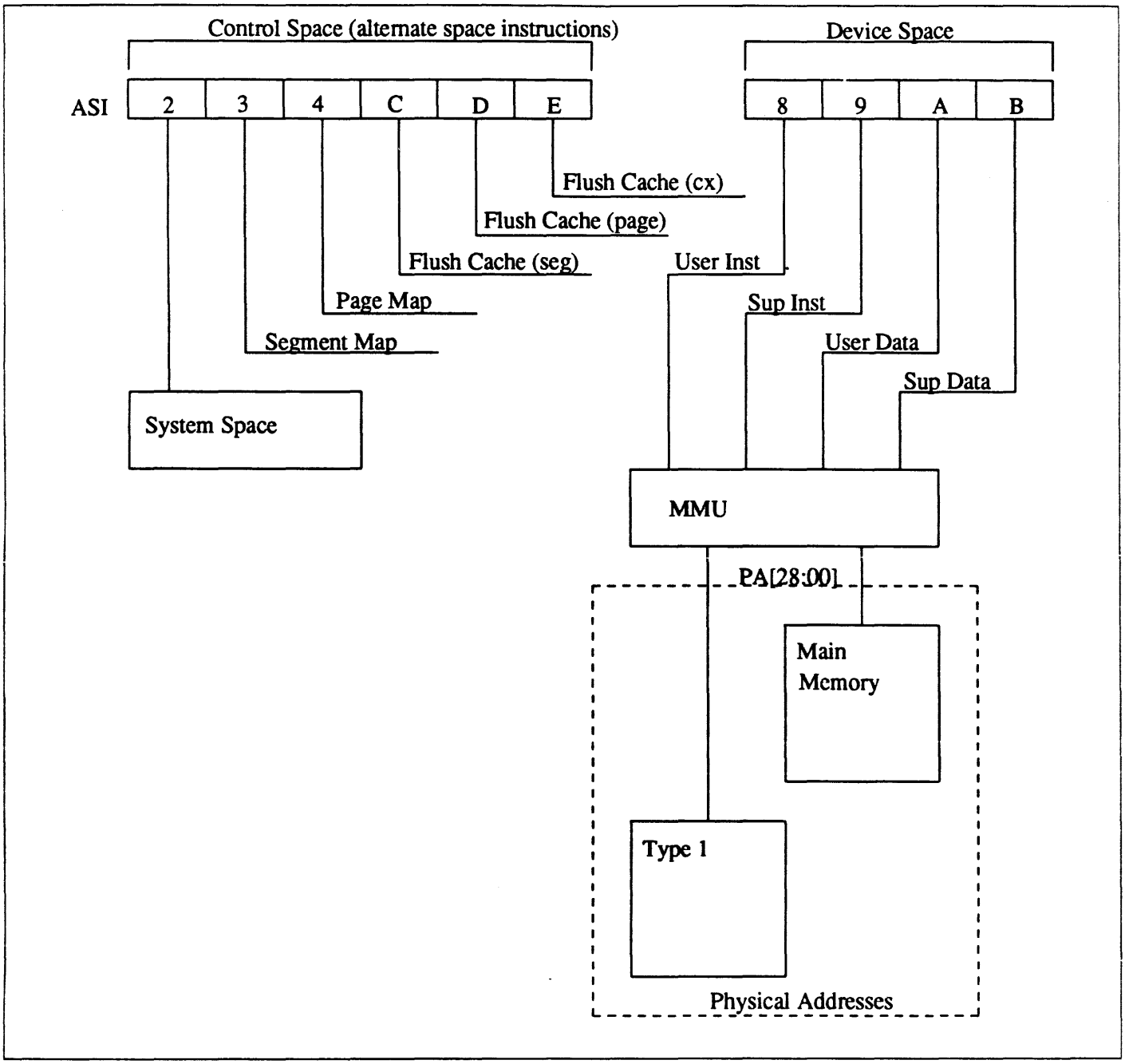
The cache is a direct-mapped virtual-address write-through cache, organized into 16-byte blocks, called lines. Each line contains 4 words of data from main memory, and has a corresponding tag field containing information about the line. Its relationship to the system appears in the following figures.

Figure 4-1 *Functional Block Diagram*



TPD-0209

Figure 4-2 *System Address Diagram*



4.6. SBus

The SBus is the basic communication mechanism between the processing core (CPU, MMU, and Cache) and the main memory and various I/O devices. The Cache serves as the system controller for the SBus. The MMU translates the virtual address output of the CPU and drives the resultant physical address onto the SBus. It also decodes this physical address into a set of select signals used to enable the main memory and I/O devices on the SBus. See the chapter on the SBus Interface for more details.

4.7. Direct Virtual Memory Access (DVMA)

The DVMA (Direct Virtual Memory Access) unit provides two channels of direct memory access between the main memory on the SBus and the SCSI and Ethernet interfaces. It also provides the path between the SBus and the SCSI and Ethernet interface devices required for the CPU to initialize and configure these interfaces. See the chapters on the SCSI Interface and the Ethernet Interface for more details.

In addition to the Ethernet and SCSI interfaces, the VME slave interface permits accesses of main memory by a VMEbus master through direct virtual memory access. See the chapter on the **VMEbus Interface** for more details.

4.8. Device Space

Device space consists of all mapped main memory and I/O devices that are accessed through translated physical addresses. These include On-Board parity memory, Local Devices (Keyboard/Mouse Port, Serial Ports, TOD Clock and NVRAM, EPROM, Counter-Timer registers, Memory Error register, and Interrupt register), SBus Slots, the VME Master interface and the P2 Bus interface. See the chapters on **Device Space, VMEbus Interface, and P2 Interface** for more details.

4.9. Address Spaces

At the top level, address spaces are identified by the address space identifier (*asi*) bits. These are part of the SPARC Architecture, and are described in the *SPARC Architecture Manual.*

The *asi* bits divide the addresses into two broad categories; control space and device space. Control space contains various unmapped system utilities; device space contains the part of the system that is accessed through the map. These spaces appear in the figure above.

Note that the CPU card only supports the *asi* values shown in the above figure.

The SPARC architecture automatically sets the *asi* bits correctly for accesses to user data, user instruction, supervisor data, and supervisor instruction spaces. To access other spaces, use the "alternate space" instructions described in the *SPARC Architecture Manual* to force the *asi* bits to the desired value.

4.10. Control Space

Control space contains the unmapped portion of the system. This contains devices used to directly control the system. Note that the MMU itself is in control space.

System Space

System space includes all accesses with *asi* = 0x2. It contains control and status registers, a serial port bypass, and cache tags. The system space devices are listed and described in the chapter *System Space.*

Page and Segment Maps (MMU Direct Access)

These address spaces enable direct access to the MMU RAMs. They are used to load the map, and are described in the chapter *Memory Management Unit.*

Cache Flush Operations

Cache flush space operations flush lines in the cache based on a matching criteria. If the line matches the criteria, the line is invalidated. Cache flush

criteria include page match, segment match, or context match. Cache flush operations are described in the chapter *Cache*.

Cache Data Space
Cache data space accesses provide direct access to the cache RAMs. A[16:0] provide the address.

4.11. Error Registers

The CPU card provides two types of error registers. They are:

- The **bus error register** is in system space. After a memory error exception, these registers identify the cause and location of the error. This register is described in the chapter *System Space*.
- The **memory error register** is in device space. This provides information about memory parity errors. It is described in the chapter *Device Space*.

4.12. Memory Maps for the SPARCengine 1E

These memory maps are presented from the Open PROM point of view at boot time. The diagrams are not to scale. Device space is represented by two memory maps.

1. With parity memory enabled: 4MB of on-board parity in low memory and one to four 4MB/16MB ECC cards in high memory.

2. With parity memory disabled: one to four 4MB/16MB ECC cards in low memory.

Figure 4-3 *Type 0 (obmem) Space Physical Memory Layout -- Parity Enabled*

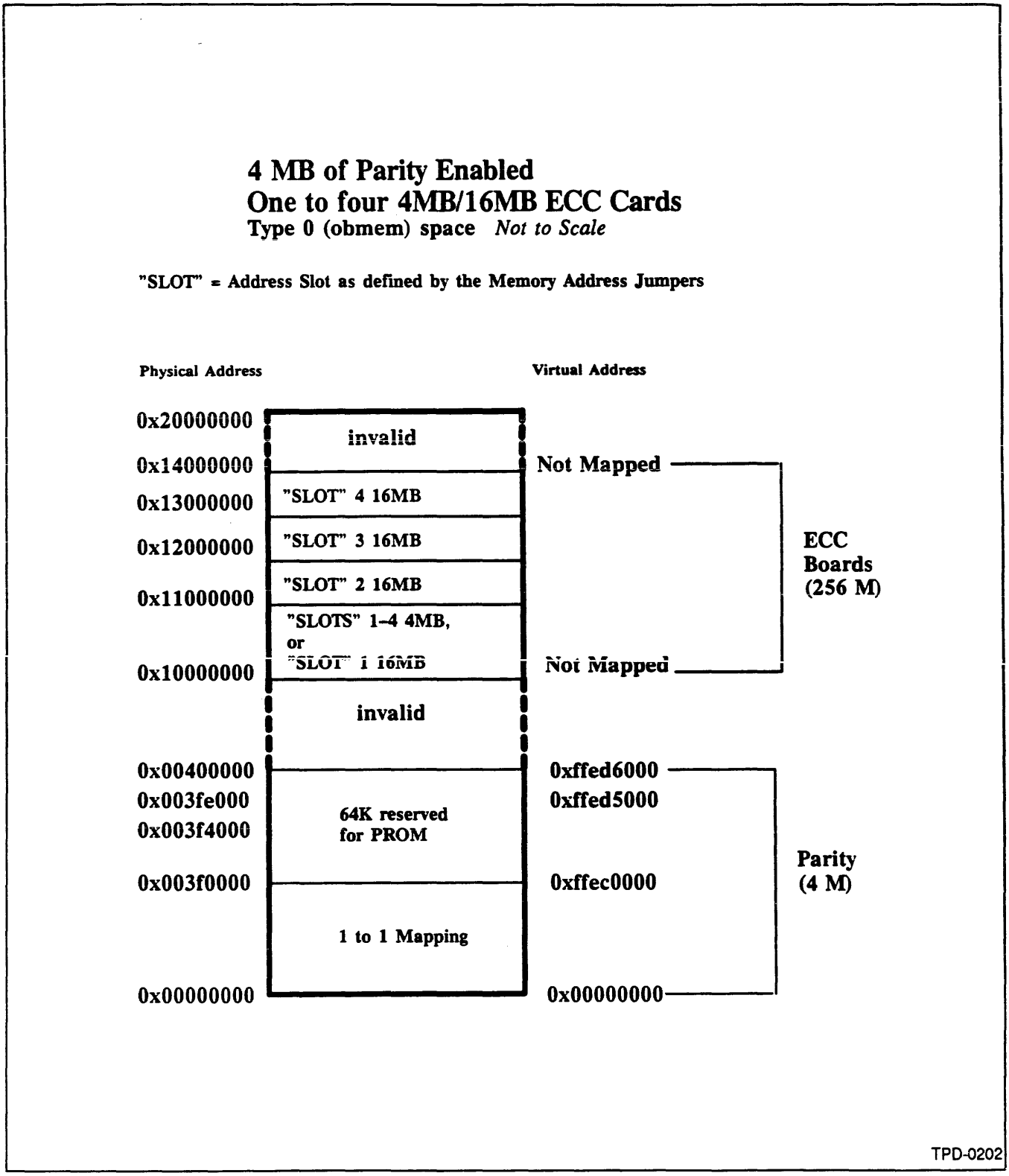


Figure 4-4 *Type 0 (obmem) Space Physical Memory Layout -- Parity Disabled*

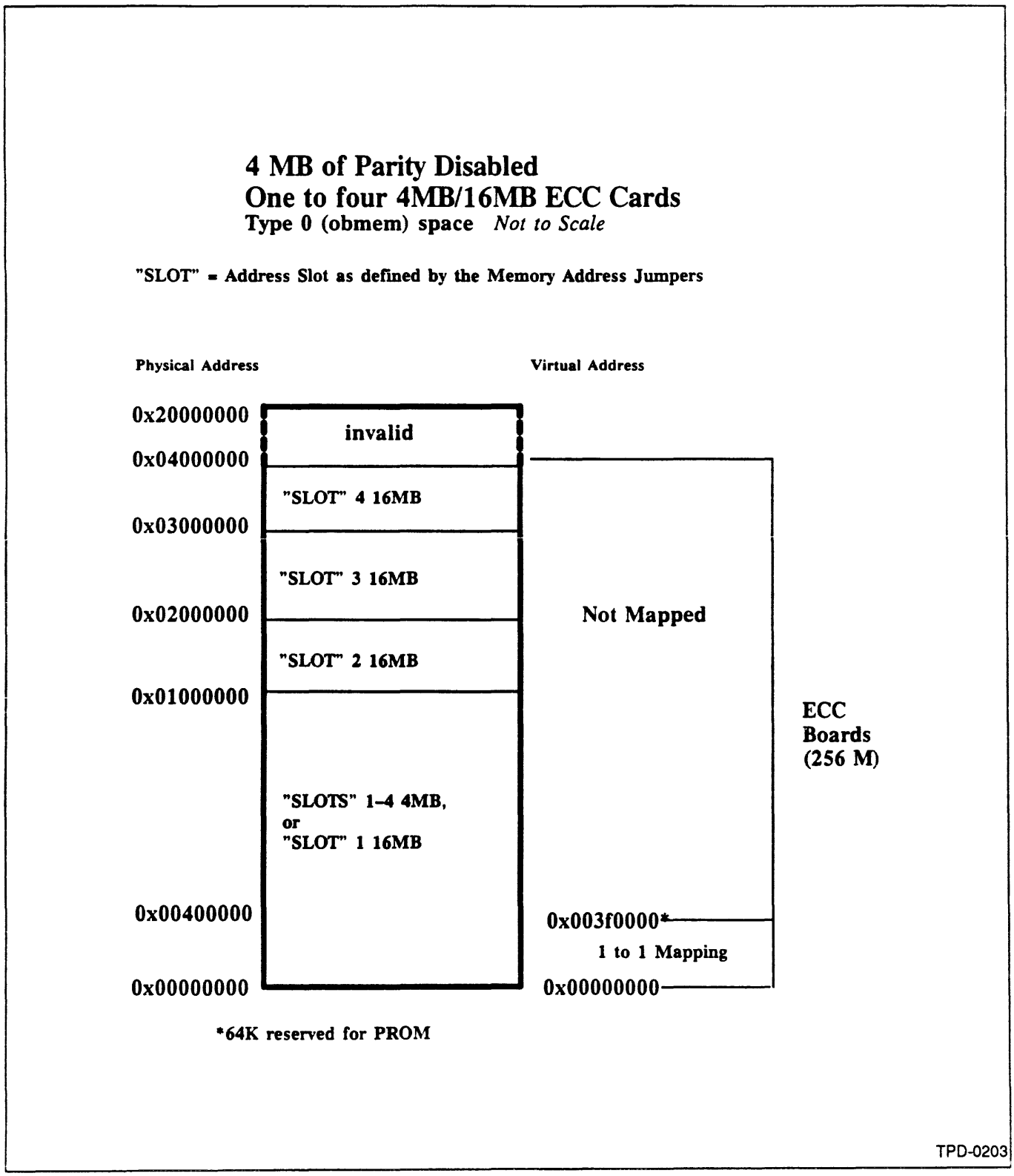


Figure 4-5 *Type 1 (obmem) Space Segment Allocation*

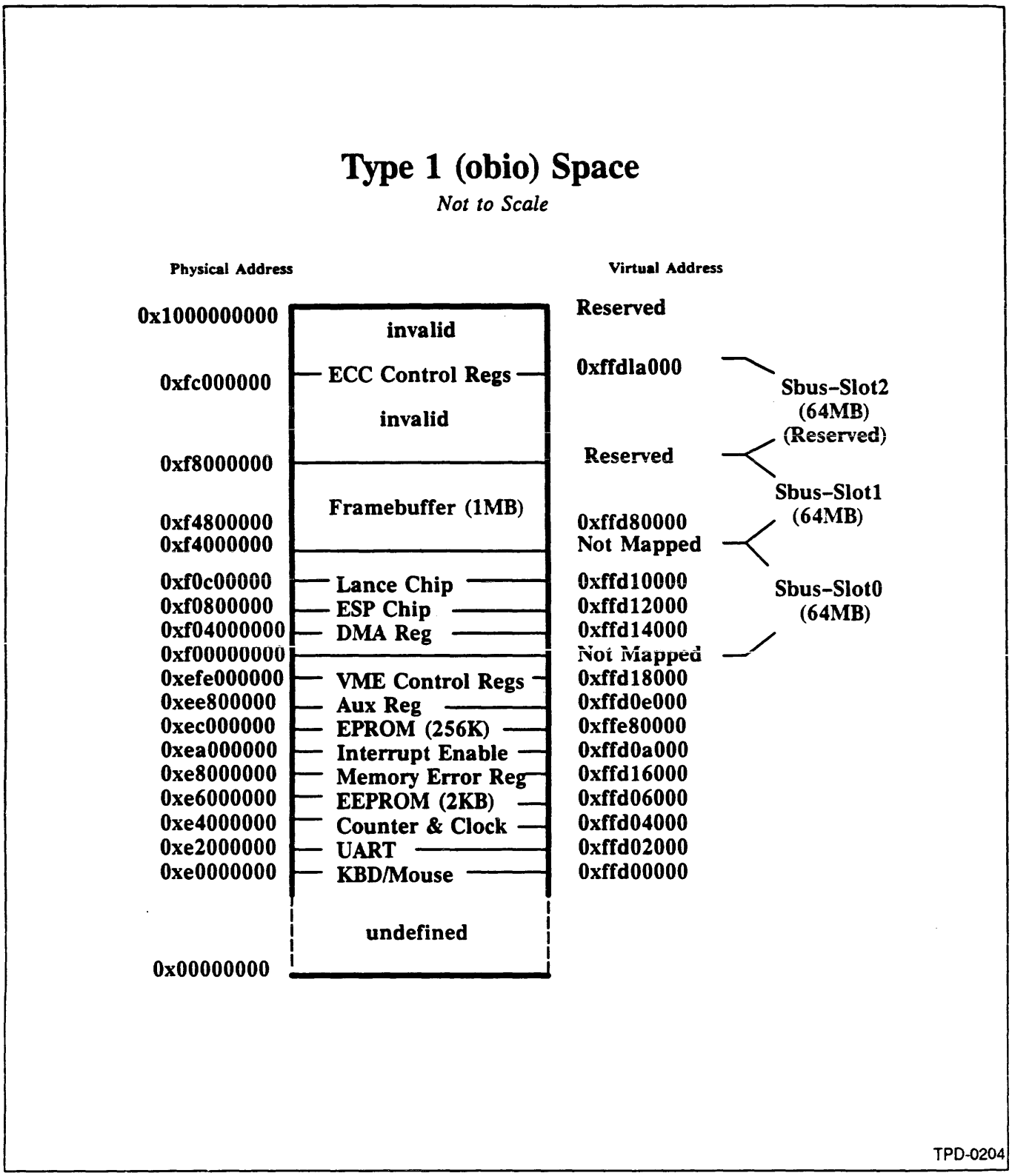


Figure 4-6 *Type 2 Space*

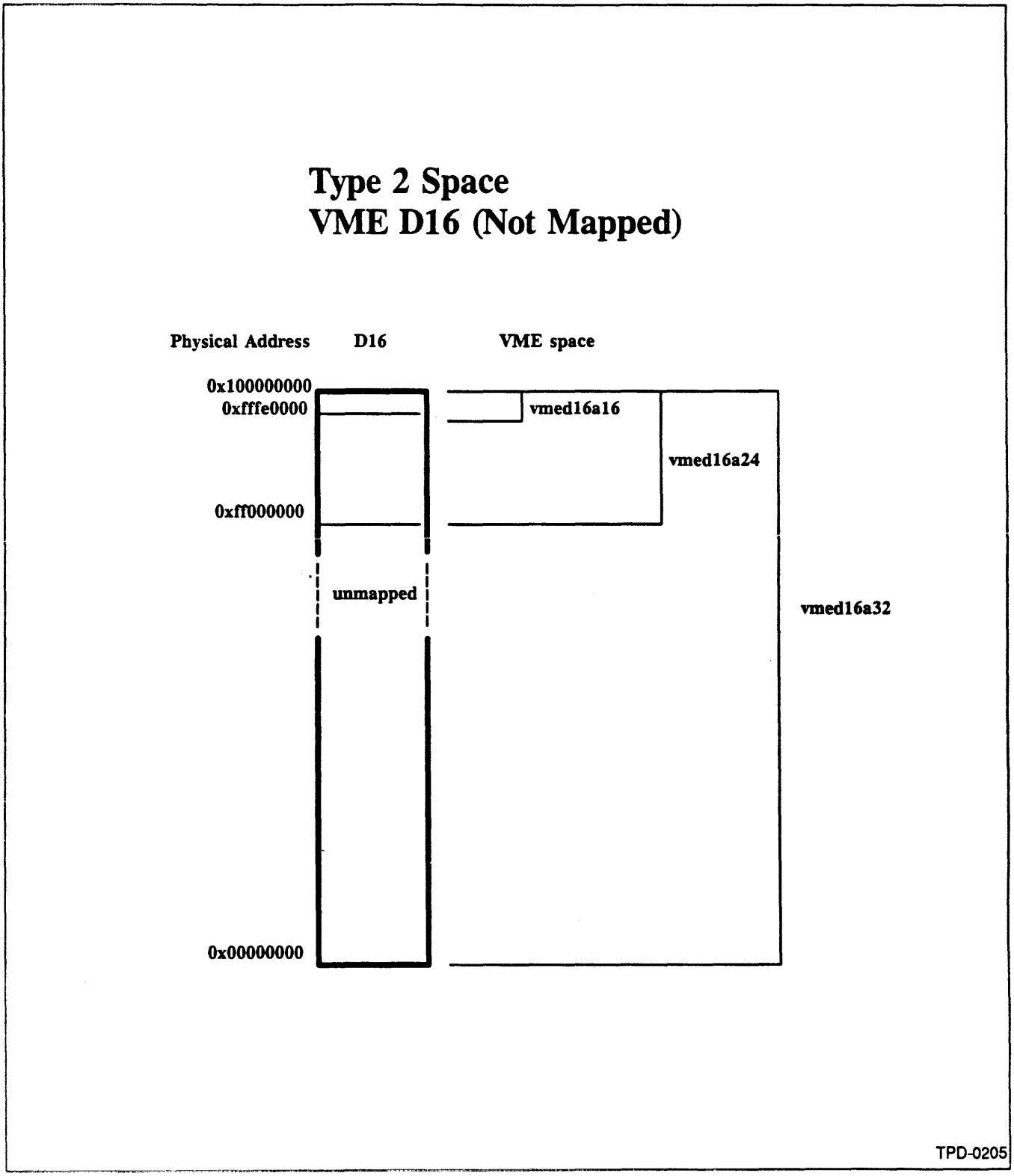


Figure 4-7 *Type 3 Space*

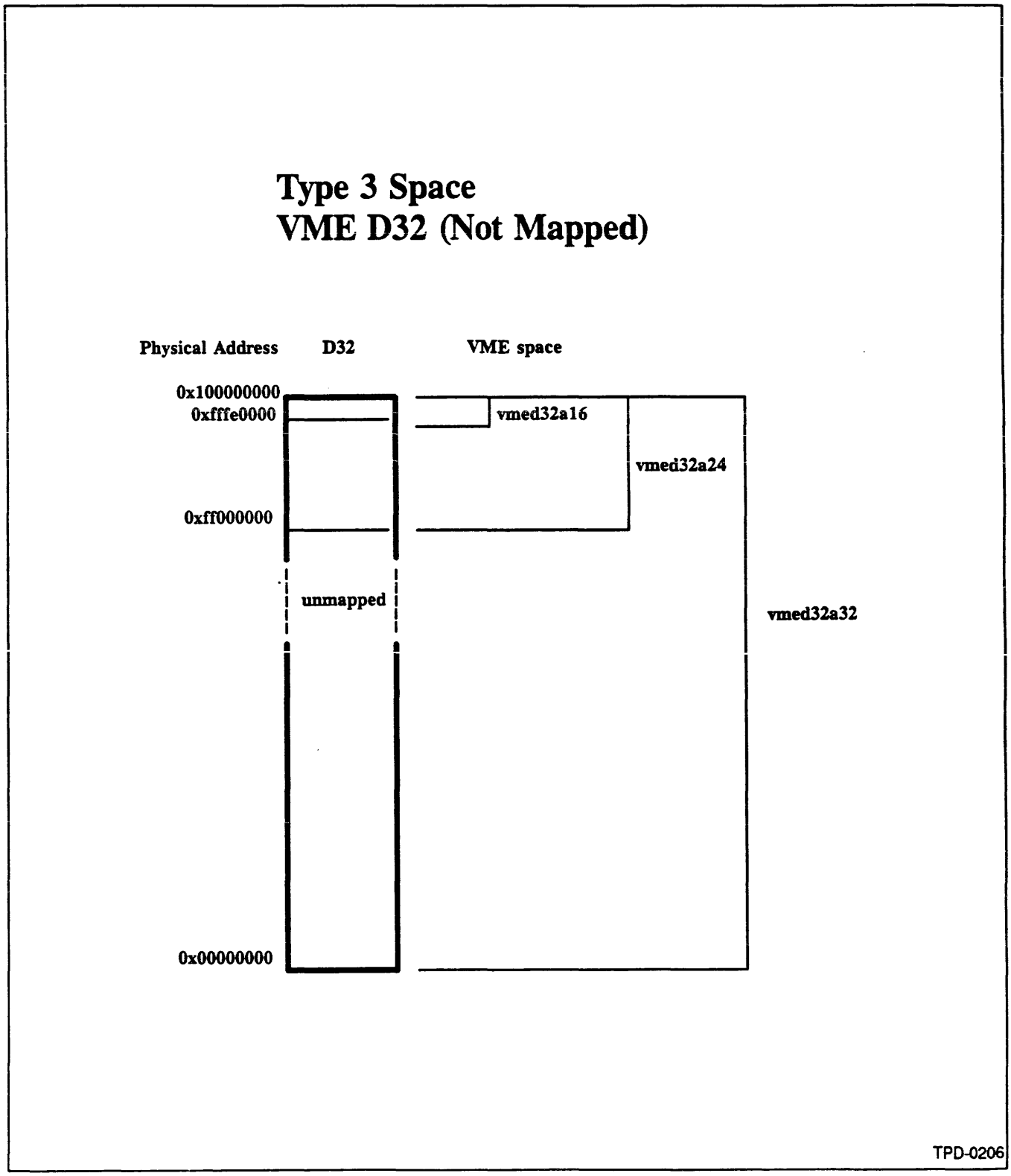


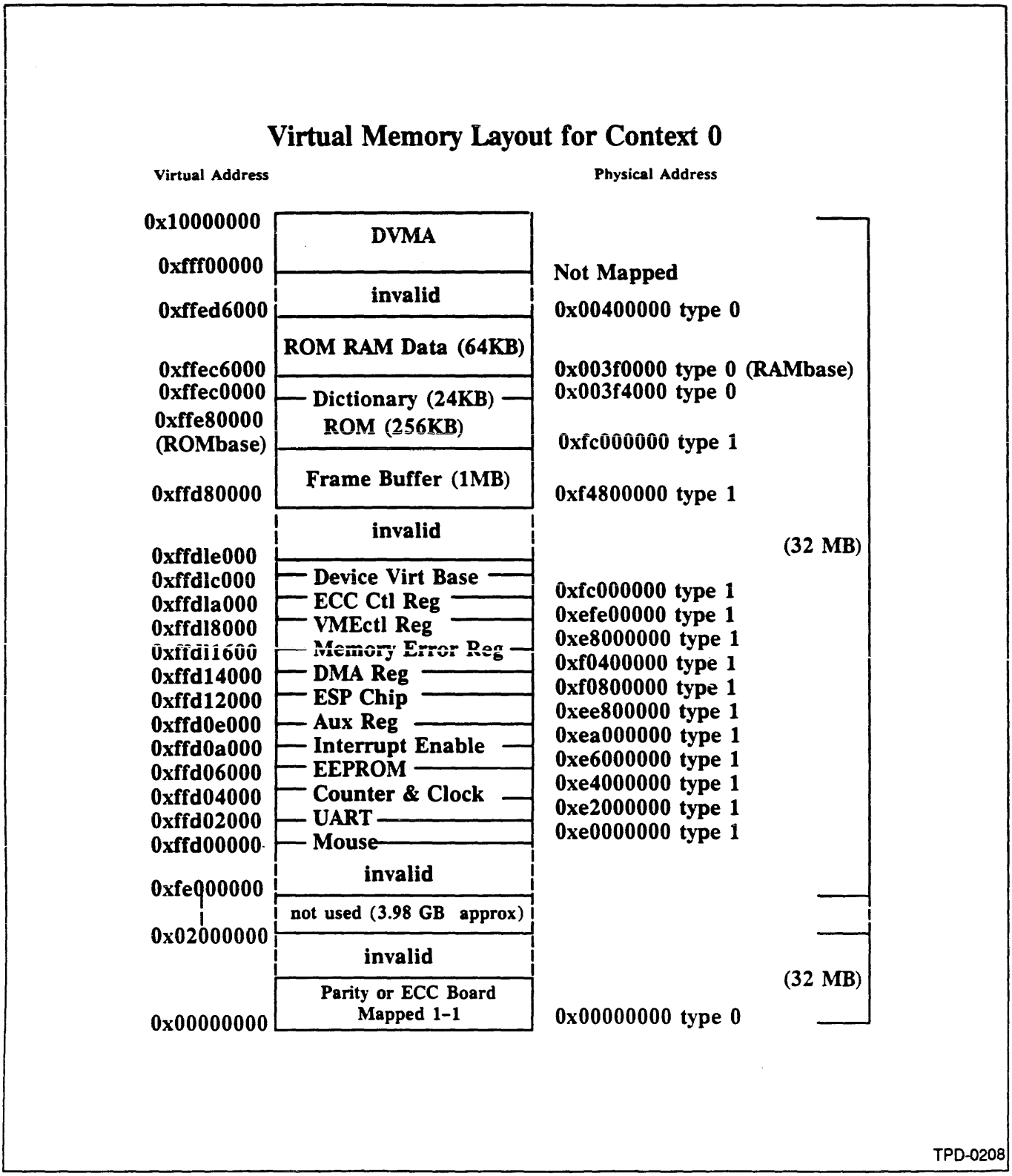
Figure 4-8 *Segment Allocation (Context 0)*

Segment Allocation (Context 0)

Start Virtual Address	Segment Number	Maps To	Start Physical Address
0x00000000	0	Parity/ECC	0x00000000
0x00040000	1	Parity/ECC	0x00040000
0x00080000	2	Parity/ECC	0x00080000
0x000c0000	3	Parity/ECC	0x000c0000
0x00100000	4	Parity/ECC	0x00100000
0x00140000	5	Parity/ECC	0x00140000
0x00180000	6	Parity/ECC	0x00180000
0x001c0000	7	Parity/ECC	0x001c0000
0x00200000	8	Parity/ECC	0x00200000
0x00240000	9	Parity/ECC	0x00240000
0x00280000	a	Parity/ECC	0x00280000
0x002c0000	b	Parity/ECC	0x002c0000
0x00300000	c	Parity/ECC	0x00300000
0x00340000	d	Parity/ECC	0x00340000
0x00380000	e	Parity/ECC	0x00380000
0x003c0000	f	Parity/ECC	0x003c0000
0x00400000	ff	Invalid	Not Mapped
.	ff	Invalid	Not Mapped
0x02000000	ff	Not Used	
.	ff	Not Used	
.	ff	Not Used	
0xfe000000	ff	Not Used	
.	ff	Invalid	Not Mapped
.	ff	Invalid	Not Mapped
0xffcc0000	ff	Invalid	Not Mapped
0xffd00000	fc	IO Segment	0xe0000000
0xffd40000	ff	Invalid	Not Mapped
0xffd80000	f8	Frame Buffer	0xf8000000
0xffdc0000	f9	Frame Buffer	0xf8040000
0xffe00000	fa	Frame Buffer	0xf8080000
0xffe40000	fb	Frame Buffer	0xf80c0000
0xffe80000	fd	ROM Segment	0xfc000000
0xffec0000	fe	RAM Segment	0x003c0000
0xfff00000	ff	DVMA Segment 1	Not Mapped
0xfff40000	ff	DVMA Segment 2	Not Mapped
0xfff80000	ff	DVMA Segment 3	Not Mapped
0xfffc0000	ff	DVAM Segment 4	Not Mapped

TPD-0207

Figure 4-9 *Virtual Memory Layout for Context 0*



5

Control Space Registers and Utilities

System space contains various control and status registers, cache tags, and a bypass port to the serial port.

The SPARC processor accesses system space with *asi* = 0x2. Address bits A[31:28] select a device within the system space, and various other address bits may select an offset within that space. Address bits outside defined fields are ignored.

NOTE ***System space accesses can only be done using the "alternate space" instructions described in the SPARC Architecture Manual. System space accesses require supervisor privilege, and are not mapped (they do not go through the MMU).***

The following table shows the system space addresses:

Table 5-1 *System Space Devices*

Device	A[31:28]	Size	Type	Offset Bits
Context Register	0x3	8-bit	R/W	
System Enable Reg	0x4	8-bit	R/W	
Bus Error Regs	0x6	8-bit	Read only	
Cache Tags	0x8	32-bit	R/W	A[15:2]
Cache Data	0x9	32-bit	R/W	A[15:2]
Serial Port Bypass	0xF	8-bit	R/W	A[2:1]

The following sections describe each:

5.1. Context Register

A context is the memory access for each process (window) running up to a total of eight processes (SunOS limitation). If the number of processes running under SunOS exceeds eight, swapping into the eight context addresses occurs.

The context register is in system space at address 0x3. It is a read/write device; bits D[2:0] identify the current MMU context (8 possible). Writing to unused bits has no effect, and reading them produces zeroes.

The context register provides a convenient and easy method of switching contexts.

Programming Example for Setting the Context Register

The following programming example is provided simply as a guide for your programming, and is not intended to be used as is.

```
/*
 * cntx.s
 * Copyright (c) 1989 by Sun Microsystems, Inc.
 */

#define  CNTXT_NUM    0x04  /* Context Register Test */
#define  CONTEXTBASE  0x30000000    /* context reg */
#define  ASI_CTL      0x2   /* control space*/
#define  quit_key     0x11  /* Cntl-q: quit this test,go to next */
#define  NCONTEXT     8     /* polaris */

/*#define FORCE_CONTEXT_ERRORS*/

     .seg   "text"
     .global context_tst

!-------------------------------------------------------------------------
! Context Register Test
!
!    Data from 0x00 to NCONTEXT-1 is written to the Context Register and
!    read back and compared.
!
!
context_tst:
     save   %sp, -MINFRAME,%sp     ! preserve calling window
     set    cntxt_tst_txt, %o0     ! test descriptor text
     call   test$
     or     %g0, CNTXT_NUM, %o1    ! set test #
     mov    %g0, %o2               ! Set initial pattern.
1:
     set    2f, %g4                ! <<<TOP OF TEST LOOP>>>
2:
     set    CONTEXTBASE, %o3       ! point to the context reg
     stba   %o2, [%o3] ASI_CTL     ! Write Context register.
     lduba  [%o3] ASI_CTL, %o1     ! Read Context register.

     cmp    %o2, %o1               ! Data read back correct?

#ifndef FORCE_CONTEXT_ERRORS
     be     3f                     ! if equal OK
#endif FORCE_CONTEXT_ERRORS

     nop
     set    cntxt_err_txt, %o0     ! error text msg addr
     call   error$                 ! Context register read data != write.
     nop
3:                                 ! data not = read data!
     call   loop$end               ! <<<BOTTOM OF TEST LOOP>>>
     nop
     cmp    %o0, quit_key          ! This test aborted?
     be     9f
     nop
     cmp    %o2, NCONTEXT-1
     beq    9f
     nop

     inc    %o2                    !Increment test pattern.
     b      2b
     nop
9:
```

```
set   CONTEXTBASE, %l4      ! pointer to context register
stba  %g0, [%l4]ASI_CTL     ! set context 0.
ret
restore
```

5.2. System Enable Register

The system enable register is an 8-bit read/write register in system space at address 0x4. It enables various system facilities, including booting. The bits have the following meanings:

NOTE *All these bits are active HIGH except EN.BOOT, which is active LOW. During reset, this register is cleared to all zeroes. This enables the boot state, and disables all others.*

Table 5-2 *System Enable Register Bits*

Bit	Name	Function
D[0]	ENA.DIAG	Reads back as 0
D[1]	Not Used	Reads back as 0
D[2]	ENA.RESET	Resets CPU and cache
D[3]	Not Used	Reads back as 0
D[4]	ENA.CACHE	Enables cache
D[5]	ENA.SDVMA	Enables system DVMA
D[6]	Not used	Reads back as 0
D[7]	ENA.BOOT_	Enable boot state

ENA.DIAG
: Reads back as 0.

ENA.RESET
: Setting this bit to 1 causes a reset. Note that control is not returned to the program that reset the bit; instead, a reset occurs.

ENA.CACHE
: This bit enables the cache when it is HIGH.

ENA.SDVMA
: This bit enables system DVMA from the system bus.

ENA.BOOT_
: This bit enables the system boot state when it is LOW. During boot state, all supervisor program fetches are from EPROM, regardless of the state of the MMU. All other types of references are unaffected, and are mapped as usual.

Three Programming Examples for the System Enable Register

The following examples demonstrate typical uses of the System Enable Register. The code fragments below: (1) turn the processor cache on; (2) turn the processor cache off, and (3) reset the CPU.

Turning the Processor Cache ON

```
/*
 * The #defines below are used in the code fragments which follow.
 */

        .seg    "data"
_enablereg:     .byte   0               ! software copy of the System Enable Register

#define ENA_RESET 0x04                  ! Reset CPU bit
#define ENA_CACHE 0x10                  ! Cache enable bit
#define ENA_SDVMA       0x20            ! System DVMA enable bit
#define ENA_BOOT_ 0x80                  ! Boot state enable bit

#define ENABLEREG 0x40000000    ! Address of System Enable Register in ASI_CTL space
#define ASI_CTL         0x2             ! Control Space address space indicator (ASI)

/*
 * The following code turns the cache on by setting the appropriate bit
 * in the System Enable Register.
 */

        .seg    "text"

        mov     ENA_CACHE, %o0
        mov     %psr, %o3               ! %o3 gets a copy of the Processor status register
        or      %o3, PSR_PIL, %g1       ! spl() high to protect Sys Enable Register update
        mov     %g1, %psr               ! set the new processor interrupt level
        nop ; nop                       ! psr update delay (necessary)
        set     _enablereg, %o1                 ! address of software copy of the System Enable
                                        ! Register
        ldub    [%o1], %g1              ! %g1 gets the software copy
        set     ENABLEREG, %o2                  ! %o2 gets the hardware address of Sys Enable Reg
        bset    %o0, %g1                ! set the Cache Enable bit in the software copy
        stb     %g1, [%o1]
        stba    %g1, [%o2]ASI_CTL ! set the Cache Enable bit in the Sys Enable Register
        mov     %o3, %psr               ! restore original processor status register value
        nop; nop                        ! psr update delay
```

Turning the Cache OFF

```
/*
 * The following code turns the cache off by clearing the appropriate bit
 * in the System Enable Register.
 */

        .seg    "text"

        mov     ENA_CACHE, %o0
        mov     %psr, %o3               ! %o3 gets a copy of the Processor status register
        or      %o3, PSR_PIL, %g1       ! spl() high to protect Sys Enable Register update
        mov     %g1, %psr               ! set the new processor interrupt level
        nop ; nop                       ! psr update delay (necessary)
        set     _enablereg, %o1                 ! address of software copy of the System Enable
                                        ! Register
        ldub    [%o1], %g1              ! %g1 gets the software copy
        set     ENABLEREG, %o2                  ! %o2 gets the hardware address of Sys Enable Reg
        bclr    %o0, %g1                ! clear the Cache Enable bit in the software copy
        stb     %g1, [%o1]
        stba    %g1, [%o2]ASI_CTL ! clear the Cache Enable bit in the Sys Enable Reg
        mov     %o3, %psr               ! restore original processor status register value
        nop; nop                        ! psr update delay
```

Reseting the CPU with the System Enable Register

```
/*
* The following code resets the CPU by setting the appropriate bit
* in the System Enable Register.
*/

        .seg    "text"

        mov     ENA_RESET, %o0
        mov     %psr, %o3               ! %o3 gets a copy of the Processor status register
        or      %o3, PSR_PIL, %g1       ! spl() high to protect Sys Enable Register update
        mov     %g1, %psr               ! set the new processor interrupt level
        nop ; nop                       ! psr update delay (necessary)
        set     _enablereg, %o1                 ! address of software copy of the System Enable
                                        ! Register
        ldub    [%o1], %g1              ! %g1 gets the software copy
        set     ENABLEREG, %o2                  ! %o2 gets the hardware address of Sys Enable Reg
        bset    %o0, %g1                ! set the Reset Enable bit in the software copy
        stb     %g1, [%o1]
        stba    %g1, [%o2]ASI_CTL ! set the Reset Enable bit in the Sys Enable Register
        mov     %o3, %psr               ! restore original processor status register value
        nop; nop                        ! psr update delay
```

5.3. Bus Error Registers

The SPARCengine 1E CPU card provides four bus error registers; two of these identify the type and location of synchronous bus errors, and two of them identify the type and location of asynchronous bus errors.

All of these registers are accessed by fullword loads and stores. They all respond to read or write accesses; however writing them is meaningless.

The registers are addressed as follows:

Table 5-3 *Bus Error Registers*

Address	Description
0x60000000	Synchronous error register
0x60000004	Synchronous error virtual address register
0x60000008	Asynchronous error register
0x6000000C	Asynchronous error virtual address register

Synchronous bus errors are those that occur due to the execution of an instruction; they are reported to the CPU by a trap at the end of the instruction's execution. Asynchronous bus errors are those that cannot be associated with the execution of the current instruction; they are related to such things as DVMA activity or buffered writes. These errors are reported by a level-15 interrupt.

Synchronous Errors

The synchronous error register records information about any synchronous error that occurs. It records all errors since it was last cleared, and reading it clears it. All bits are active HIGH except bit D15.

NOTE: If more than one bit is HIGH, to determine the true cause of the error, first determine the address associated with the error. For data exceptions it will be in the synchronous error virtual address register; for instruction exceptions, it will be in the saved program counter (see the SPARC Architecture Manual). Then, manually inspect the page map entry associated with that address to determine the true

cause of the error.

The bits have the following meanings:

D0 — SE_WATCHDOG
This bit indicates a restart due to an IU error.

D1 — SE_SIZERR
This bit indicates that an incorrect size transfer was attempted.

D2 — Unused
This bit reads as 0.

D3 — SE_MEMERR
This bit indicates that a memory parity error occurred.

D4 — SE_SBERR
This bit indicates that a bus error happened during an SBus master, VME master, or P2 bus access.

D5 — SE_TIMEOUT
This bit indicates an access to a non-existent device or to non-existent physical memory.

D6 — SE_PROTERR
This bit indicates that a protection error occurred. This can be caused by an attempted write to a read-only page, or by a user-mode access to a supervisor-only page.

D7 — SE_INVALID
This bit indicates that the valid bit was zero in a page map entry.

D[14:8] — Unused
These bits read as zeros.

D15 — SE_RW
When this bit is HIGH, it indicates that an error occurred during a write cycle. When it is LOW, it indicates that an error occurred during a read cycle.

The synchronous error virtual address register is latched with bits VA[31:0] when any synchronous error occurs, and it is unlatched when the synchronous error register is read.

Programming Example for Using the Synchronous Bus Error Register

The following programming example is provided simply as a guide for your programming, and is not intended to be used as is.

```
/*
 * sync.s
 * Copyright (c) 1989 by Sun Microsystems, Inc.
 */


#define     ASI_CTL 0x2             /* Control Space*/
#define     BUSERROFF3    0x60000000    /* Sync Bus Error Register */

    .seg   "text"
    .global Sync_reg
```

```
!-----------------------------------------------------------------------
! Sync Register
!
!
Sync_reg:
    save   %sp, -MINFRAME,%sp    ! preserve calling window
    set    Sync_breg, %o0        ! Address of message.
    call   print$                ! "Sync Register Read/Write Test."
    nop

    clr    %l1                   ! Clear the %l1 Register.
    set    BUSERROFF1, %l7       ! Sync Bus Error Register.
    lda    [%l7]ASI_CTL, %l2     ! Save Bus error register.
    sta    %l1, [%l7]ASI_CTL     ! write to buserr reg to clear.
    ret
    restore

    .global Sync_breg
Sync_breg:
    .asciz " 15  12Sync Registers Test."
```

Programming Example for Using the Synchronous Virtual Register

The following programming example is provided simply as a guide for your programming, and is not intended to be used as is.

```
/*
 * sync_virt.s
 * Copyright (c) 1989 by Sun Microsystems, Inc.
 */


#define      ASI_CTL       0x2          /* Control Space*/
#define      BUSERROFF2    0x60000004   /* Sync Bus Error Register */

    .seg   "text"
    .global Sync_virt_reg

!-----------------------------------------------------------------------
! Sync Virtual Address Register
!
!
Sync_virt_reg:
    save   %sp, -MINFRAME,%sp    ! preserve calling window
    set    sync_virtreg, %o0     ! Address of message.
    call   print$                ! "Sync Virtual Address Register Read/Write Test."
    nop

    clr    %l1                   ! Clear the %l1 Register.
    set    BUSERROFF2, %l7       ! Sync Bus Error Register.
    lda    [%l7]ASI_CTL, %l2     ! Save Bus error register.
    sta    %l1, [%l7]ASI_CTL     ! write to buserr reg to clear.
    ret
    restore

    .global sync_virtreg
sync_virtreg:
    .asciz " 15  12Sync Virtual Address Registers Test."
```

Asynchronous Errors

The asynchronous error register latches the status of the system when an asynchronous error occurs. It latches all errors that occurred since it was last cleared. Reading this register also clears it. All bits are active HIGH.

The bits have the following meanings:

D[3:0] — Unused
These bits read as 0's.

D4 — ASE_DVMAERR
This bit indicates that a bus error occurred during a DVMA access.

D5 — ASE_TIMEOUT
This bit indicates that a non-existent device was addressed.

D7 — ASE_INVALID
This bit indicates that the valid bit was 0 (invalid) in a page map entry.

The asynchronous error virtual address register contains the virtual address associated with the last asynchronous bus error. It is cleared when the asynchronous bus error register is read.

Programming Example for Using the Asynchronous Bus Error Register

The following programming example is provided simply as a guide for your programming, and is not intended to be used as an independent program.

```
/*
 * async.s
 * Copyright (c) 1989 by Sun Microsystems, Inc.
 */

#define     ASI_CTL     0x2         /* Control Space*/
#define     BUSERROFF3    0x60000008    /* Async Bus Error Register */

    .seg   "text"
    .global Async_reg

!-----------------------------------------------------------------------
! Async Register
!
!
Async_reg:
    save   %sp, -MINFRAME,%sp    ! preserve calling window
    set    async_reg, %o0        ! Address of message.
    call   print$                ! "Async Register Read/Write Test."
    nop

    clr    %l1                   ! Clear the %l1 Register.
    set    BUSERROFF3, %l7       ! Async Bus Error Register.
    lda    [%l7]ASI_CTL, %o2     ! Read it back.
    sta    %l1, [%l7]ASI_CTL     ! write to buserr reg to clear.
    nop
    ret
    restore

    .global async_reg
async_reg:
    .asciz "  15  12Async Registers Test."
```

Programming Example for Using the Asynchronous Virtual Register

The following programming example is provided simply as a guide for your programming, and is not intended to be used as is.

```
/*
 * async_virt.s
 * Copyright (c) 1989 by Sun Microsystems, Inc.
 */

#define     ASI_CTL     0x2         /* Control Space*/
#define     BUSERROFF4    0x6000000c    /* Async Bus Virtual Register */

    .seg   "text"
    .global Async_reg

!-----------------------------------------------------------------------
! Async Register
!
!
Async_reg:
    save   %sp, -MINFRAME,%sp    ! preserve calling window
    set    async_virtreg, %o0          ! Address of message.
    call   print$              ! "Async Register Read/Write Test."
    nop

    clr    %l1                 ! Clear the %l1 Register.
    set    BUSERROFF4, %l7       ! Async Bus Error Register.
    lda    [%l7]ASI_CTL, %o2     ! Read it back.
    sta    %l1, [%l7]ASI_CTL     ! write to buserr reg to clear.
    nop
    ret
    restore

    .global async_reg
async_virtreg:
    .asciz "  15  12Async Virtual Address Registers Test."
```

More on Timeout Errors

Accesses to non-existent devices on CPU bus cycles to control space or device space will cause timeout errors.

During system space accesses, timeout errors result from accesses to invalid codes within defined fields, or from accesses to unimplemented system space devices, where A[31:28] are not legal values.

In device space, timeout errors result from accesses to addresses outside the defined address space, or accesses to I/O devices which are legal but not present.

5.4. Direct Accesses to Cache Tags and Data

System space provides for direct access to the data in the cache, and to the cache tags. These accesses are described in the chapter *Cache*.

5.5. Serial Port Bypass

The serial port bypass is in system space at 0xF. It enables accesses to the serial port A and B UARTs without using the MMU. The port and mode selection works the same as the normal accesses described in the chapter *Device Space*.

6

Device Space

6.1. Required Reference Material

VME Specifications C.1

AMD Z8530 SCC Data Sheet.

The Mostek MK48T02-15 Data Sheet.

6.2. Main Memory

Type 0 device space accesses address main memory. Main memory consists of up to 16 MB of on-board, parity-protected DRAM and up to 256 MB of off-board ECC-protected DRAM.

6.3. On-board Memory

Address bits PA [28:0] address main memory. To access on-board main memory, PA28, PA27, PA26, PA25, and PA24 must all be 0's.

The actual memory is provided on four SIP modules. If the 1 MB SIPS are present, then the size of the on-board main memory is 4 MB (0x0000000 to 0x03FFFFF). If the 4 MB SIPs are present, on-board memory is 16 MB (0x0000000 to 0x0FFFFFF).

When 1 MB SIPs are used, the 4 MB of actual memory is 'mirrored' three more times on 4 MB boundries, as follows:

0x000000 to 0x03FFFF.....actual memory
0x040000 to 0x07FFFF.....mirror of 0x000000 to 0x03FFFF
0x080000 to 0x0BFFFF.....mirror of 0x000000 to 0x03FFFF
0x0C0000 to 0x0FFFFF.....mirror of 0x000000 to 0x03FFFF

It is important to note that the memory controller for the on-board memory will respond to addresses in the range of 0x100000 to 0x1FFFFF. When data is written to this range, the data is thrown away. When data is read from this range, the data value returned is in-determinate and a parity error is likely to occur.

6.4. Off-board Memory

Off-board main memory is on the 4E ECC Memory board(s). The 4E CPU accesses the 4E Memory via the P2 Bus. To access this memory, PA28 must be 1.

Device space contains all the devices that are accessed through the memory map. This includes main memory, various on-board devices (keyboard/mouse port, the memory error register, the interrupt register, EPROM, NVRAM, the system clock, the system bus, and I/O devices), the P2 Bus, and the SBus, which provides support for some on-board devices plus additional (SBus type) boards.

The type bits and the physical base address in the Page Table Entry (see the chapter *MMU*) select items in device space. Onboard and P2 Bus main memory is type 0; type 1 is used for onboard devices, P2 Bus devices, and the SBus slot. Additional address bits may be used as an offset within the device.

The following figure shows the mapping of virtual to physical addresses:

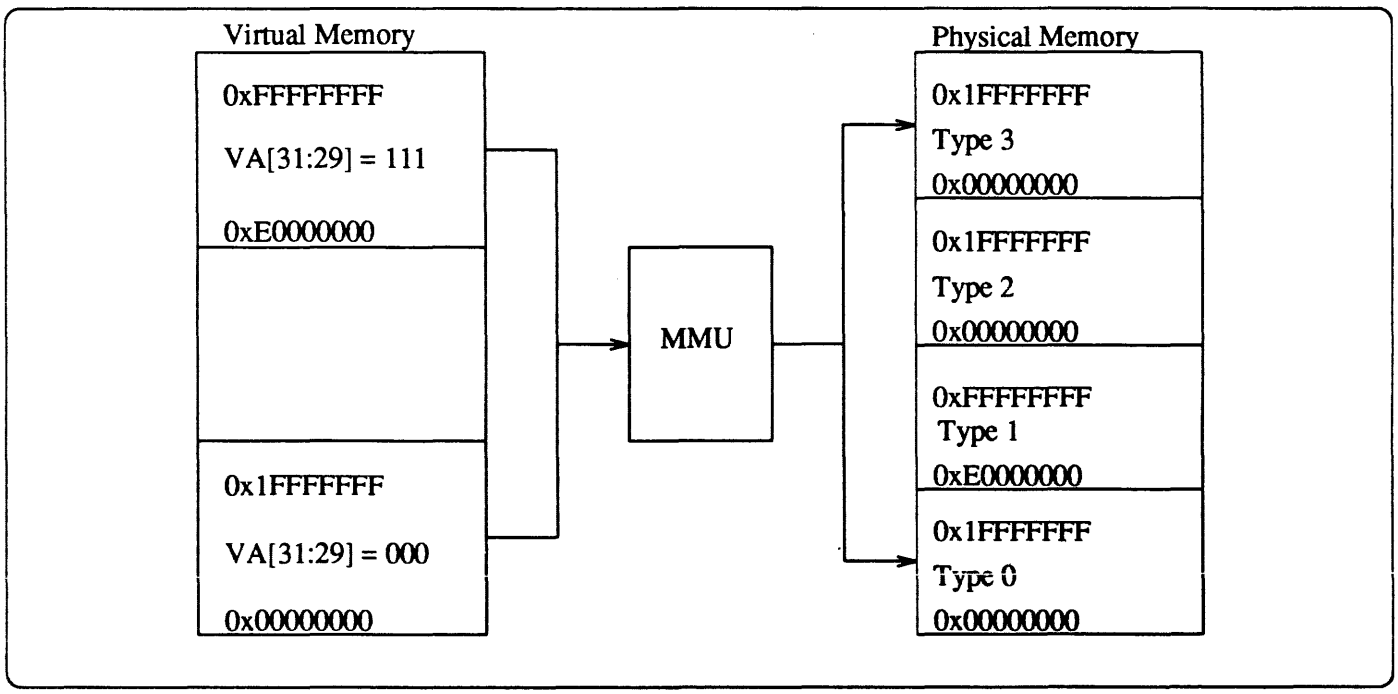


Figure 6-1 *Virtual to Physical Address Mapping*

NOTE: *The addresses listed in this chapter are* **physical addresses;** *these are the addresses that the MMU must output. The MMU inputs virtual addresses, and outputs physical addresses and type bits. The relationship between virtual addresses, and the physical addresses and type bits they produce depends on how the MMU is loaded by software.*

The type bits are described in the chapter **Memory Management Unit.**

NOTE: The MMU will output addresses in the 0-1/2 GB (29 bits) address range. To access the full VME address space, the A32MAP register on the VME gate array may be written to increment the address output by the MMU in steps of 1/2 GB. Thus, the full 4GB of VME address space may be accessed.

See the ***VME Specifications C.1*** for correct use of the A32MAP register.

The following table shows the device space physical addressing:

Table 6-1 *Device Space Addressing*

Device	Type	Physical Base Address	Address Offset Bits
Main memory, onboard	0	0x00000000	PA[25:0]
P2 Bus Memory	0	0x10000000	PA[28:0]
Keyboard/mouse port	1	0xE0000000	PA[2:1]
Serial port	1	0xE2000000	PA[2:1]
TOD clock and NVRAM	1	0xE4000000	
Counter-timer registers	1	0xE6000000	
Memory error register	1	0xE8000000	
Interrupt register	1	0xEA000000	1 byte
EPROM	1	0xEC000000	PA[17:0]
ECC G/A Register (reserved)	1	0xEE200000	
LED's (Aux I/O)	1	0xEE800000	PA[20:0]
SBus slot 0	1	0xF0000000	PA[23:0]
SBus slot 1	1	0xF4000000	PA[23:0]
SBus slot 2 (reserved)	1	0xF8000000	PA[23:0]
P2 Bus slot	1	0xFC000000	

NOTE *This table shows 32-bit addresses, even though the MMU only outputs PA[28:00]. To improve compatibility with other Sun-4 architectures, bits PA[31:29] are assumed to be all ones.*

Accesses to a non-existent device result in a timeout.

The following sections describe main memory, on board devices, and the SBus. (Note that certain on-board SBus devices are described in other chapters; see the section *SBus Interface* for details).

6.5. Local Devices

The local devices include a keyboard/mouse port, serial ports, a time-of-day clock/NVRAM, counter-timer registers, memory error registers, EPROM, and an auxillary output register

Keyboard/Mouse Port

The keyboard/mouse Serial Communications Controller (SCC) is in type 1 device space at address 0xE2000000. It is implemented with a AMD (or Zilog) Z8530 SCC. Channel A is connected to the keyboard and channel B is connected to the mouse.

Just like the serial port SCC, the clock input is independent of the CPU clock and runs at 4.9152 MHz. It interrupts the CPU on level 12. Refer to the ***AMD Z8530 SCC Data Sheet*** for a description of the SCC.

The ports consist of four byte-wide read/write ports addressed by bits PA[2:0] (PA0 is not really present, and is assumed to be a 0).

NOTE *When accessing the keyboard/mouse port, remember that your software must guarantee a recovery time of 1.6us. Do not attempt to write to the port faster than this limitation.*

The keyboard/mouse addresses are decoded as follows:

Table 6-2 *Keyboard/Mouse Addresses*

Address	Register
0xE00000000	Channel B (mouse) control
0xE00000002	Channel B (mouse) data
0xE00000004	Channel A (keyboard) control
0xE00000006	Channel A (keyboard) data

Serial Port

The serial ports are in type 1 device space at address 0xE2000000. They are implemented with a AMD (or Zilog) Z8530 SCC which features two high-speed, fully symmetrical and highly programmable serial channels with built-in baud-rate generators. The clock input is independent of the CPU clock and runs at 4.9152 MHz. It interrupts the CPU on level 12. Refer to the ***AMD Z8530 SCC Data Sheet*** for a description of the SCC.

The serial port interface consists of 4 byte-wide read/write channels selected by decoding PA[2:0] (PA0 is not really present, and is assumed to be a 0). Note that software must guarantee a recovery time of 1.6µs.

The addresses are decoded as follows:

Table 6-3 *Serial Port Addresses*

Address	Register
0xE20000000	Channel B control
0xE20000002	Channel B data
0xE20000004	Channel A control
0xE20000006	Channel A data

TOD Clock and NVRAM

The Time of Day (TOD) clock circuit contains a Mostek MK48T02-15 Zeropower/Timekeeper RAM. This chip provides 2K of RAM. The top 8 bytes are reserved for clock data, the rest is used to store system configuration information. The clock information occupies the highest 8 bytes, preceded by 20 bytes of information corresponding to the IDPROM in earlier Sun CPUs. The rest of NVRAM contains data corresponding to the EEPROM in other Sun CPUs.

The NVRAM space can be accessed by byte, halfword, or fullword loads and stores, but the TOD control register should only be written by byte stores. The time and date information is stored in 24-hour BCD format. The addresses are:

Table 6-4 *Clock Chip NVRAM Addressing*

Address	Description
0xE4000000 to 0xE40007D7	EEPROM data
0xE40007D8 to 0xE40007F7	IDPROM data
0xE40007F8	TOD control
0xE40007F9	Seconds (00 to 99)
0xE40007FA	Minutes (00 to 59)
0xE40007FB	Hours (00 to 23)
0xE40007FC	Days (00 to 07)
0xE40007FD	Date (00 to 31)
0xE40007FE	Month (01 to 12)
0xE40007FF	Year (00 to 99)

NOTE *The NVRAM space contains critical system configuration information. Altering this information could adversely affect system performance.*

The Timekeeper contains its own battery backup which has a worse case storage life of 11 years at 70'C, and a worse case consumption life of 2.8 years at 0'C.

For additional information see ***The Mostek MK48T02-15 Data Sheet.***

Counter-Timer Registers

The SPARCengine 1E CPU card provides two counter-timer register pairs located at 0xE6000000. Each counter provides a 20 bit wide writeable value which is incremented at 1 microsecond intervals. When this value reaches the value in the corresponding limit register, the circuit generates an interrupt (assuming that interrupts are enabled) at level 10 for counter/timer 0, and at level 14 for counter/timer 1. Then, the counter is reset to 1 u sec.

Reading the appropriate limit register clears the interrupt and the limit bits. Reading the counter register does not clear anything. Writing the limit register provides a value for the counter register to "match", and resets the counter register to a value equivalent to one microsecond (a 1 in bit position 10).

Setting a limit register to 0 causes the corresponding counter to freerun, which will generate interrupts when the counter overflows back to zero, approximately every 2 seconds.

Note that bits [9:0] in all counter-timer registers are unwritable, and read back as zeroes. All values are stored in bits [30:10], with bit 10 being the low-order bit. To generate an interrupt, for example every millisecond, you would load the value 1000 in a register, shift it left 10 bits, then store the result in the limit register.

Bit 31 in both the limit and the counter registers, is the "limit reached" bit. It is set when the value in the counter register reaches the value in the limit register. The limit reached bit in the counter register is a shadow of the limit reached bit in the limit register; reading the limit reached bit in the limit register resets both

registers; writing a 0 to the limit reached bit in the limit register also resets both registers. Reading the counter register does not change either.

The registers are addressed as follows:

Table 6-5 *Counter/Timer Register Addressing*

Address	Description
0xE6000000	counter 0
0xE6000004	limit 0
0xE6000008	counter 1
0xE600000C	limit 1

Programming Example for the Counter/Timer Register

The following code demonstrates how to start the counter/timer-0 to interrupt periodically with a 1/100 second interval. An interrupt handler at IU level-10 must be installed to perform the desired actions when this interrupt takes place.

A CPU scheduler interrupt could be set up in this fashion.

```
/*
 * @(#)counter.h       1.3 89/09/07 Copyr 1989 Sun Micro
 *
 * Definitions for the SPARCengine 1E on-board counter/timers
 */


#ifndef LOCORE

typedef struct counterregs
 {
  u_int       counter10;
  u_int       limit10;
  u_int       counter14;
  u_int       limit14;
 } CTR_TIMERS;

#endif !LOCORE

#define OBIO_CTR_ADDR  0xE6000000
#define CTR_ADDR  0xFFFF0000          /* Mapped virtual address for counters */

#define COUNTER          ((CTR_TIMERS *)(CTR_ADDR))

#define CTR_LIMIT_BIT    0x80000000          /* limit bit mask */
#define CTR_USEC_MASK    0x7ffffc00          /* counter/limit mask */
#define CTR_USEC_SHIFT   10                  /* counter/limit shift */

/*
 * ML byte-offsets: read CTR_ADDR+CTR_LIM_n to reset
 * counter/timer n.
 */

#define CTR_CNT_10 0x00
#define CTR_LIM_10 0x04
#define CTR_CNT_14 0x08
#define CTR_LIM_14 0x0C
#ifndef lint
static char sccsid[] = "@(#)counter.c  1.3 89/09/07 Copyr 1989 Sun Micro";
#endif lint
```

```
/*
 * Machine-dependent counter/timer routines.
 *
 * Startrtclock restarts the counter/timers, which provide
 * hardclock interrupts to kern_clock.c.
 */


#include <sys/param.h>
#include <sys/time.h>
#include <sys/kernel.h>

#include <machine/clock.h>
#include <machine/intreg.h>
#include <machine/counter.h>

/*
 * Start the real-time clock.
 */
startrtclock()
{

        /*
         * We will set things up to interrupt every 1/100 of a second.
         */
        if (hz != 100)
                panic("startrtclock");

        COUNTER->limit10 = ((1000000 / hz) << CTR_USEC_SHIFT) & CTR_USEC_MASK;
/*
 * Turn on level 10 counter/timer interrupt.
 */
        set_clk_mode(IR_ENA_CLK10, 0);
}       /* end of startrtclock */


/*
 * Set and/or clear the desired clock bits in the interrupt
 * register. We have to be extremely careful that we do it
 * in such a manner that we don't get ourselves lost.
 */
set_clk_mode(on, off)
        u_char on;
        u_char off;
{
        register u_char intreg;
        register u_int  dummy;
        register int    s;

        /*
         * make sure that we are only playing w/
         * clock interrupt register bits
         */
        on &= (IR_ENA_CLK14 | IR_ENA_CLK10);
        off &= (IR_ENA_CLK14 | IR_ENA_CLK10);

        /*
         * Get a copy of current interrupt register,
         * turning off any undesired bits (aka 'off')
         */
        s = spl7();

        intreg = *INTREG & ~(off | IR_ENA_INT);
```

```
        /*
         * Next we turn off the CLK10 and CLK14 bits to avoid any triggers.
         * Then we clear any outstanding clock interrupts.
         */
        *INTREG &= ~(IR_ENA_CLK14 | IR_ENA_CLK10);
        dummy = COUNTER->limit10;              /* clear counter/timer */
        dummy = COUNTER->limit14;              /* clear counter/timer */

        /*
         * Now we set all the desired bits
         * in the interrupt register.
         * finally we can enable all interrupts.
         */
        *INTREG |= (intreg | on);              /* enable flip-flops */
        (void) splx(s);
}       /* end of set_clk_mode */
```

Memory Error Register

The SPARCengine 1E CPU card uses a single parity control register located at address 0xE8000000. Bits D[31:8] read back as 0's, and bits D[7:0] are used as follows:

D7 — Parity Error
This bit is set by the hardware to indicate that a parity error has occurred.

D6 This bit is set by the hardware to indicate that multiple parity errors have occurred (a parity error occurred while D7 = 1).

D5 — Parity Test
Setting this bit generates inverse parity.

D4 — Parity Check
Setting this bit enables parity checking.

D3 — Parity Error 00
The hardware sets this bit to indicate that a parity error occurred in data bits D[31:24].

D2 — Parity Error 08
The hardware sets this bit to indicate that a parity error occurred in data bits D[23:17].

D1 — Parity Error 16
The hardware sets this bit to indicate that a parity error occurred in data bits D[16:08].

D0 — Parity Error 24
The hardware sets this bit to indicate that a parity error occurred in data bits D[07:00].

Reading the memory error register clears it.

Interrupt Register

The interrupt register is an 8-bit register in device space at 0xEA000000. It enables all interrupts, causes software interrupts, and enables certain interrupts. For more on interrupts, see the chapter *Interrupts*.

The bits are:

Table 6-6 *Interrupt Register Bits*

Bit	Name	Function
D0	EN.INT	Enables all interrupts
D1	EN.INT1	Generate software interrupt level 1
D2	EN.INT2	Generate software interrupt level 4
D3	EN.INT6	Generate software interrupt level 6
D4	EN.INT8	Enable video interrupt level 8
D5	EN.INT10	Enable clock interrupt level 10
D6	EN.INT13	reserved
D7	EN.INT14	Enable clock interrupt level 14

EN.INT

This bit enables all interrupts. When it is off, no interrupts occur.

EN.INT2, EN.INT4, and EN.INT6

These bits cause software interrupts on their respective levels. The interrupts caused by these bits stay active until software clears the corresponding bit.

EN.INT8

This bit enables video interrupt requests on level 8. When it is enabled, a level 8 interrupt request is sent on the rising edge of vertical retrace. To clear this interrupt, momentarily turn off EN.INT8.

EN.INT10

This bit enables clock interrupts clock interrupt requests on level 10. When these interrupts are enabled, a level 10 request is sent on the rising edge of the clock interrupt output. To clear the interrupt, momentarily turn off EN.INT10.

EN.INT14

This bit enables clock interrupts clock interrupt requests on level 14. When these interrupts are enabled, a level 14 request is sent on the rising edge of the clock interrupt output. To clear the interrupt, momentarily turn off EN.INT14.

EPROM

The SPARCengine 1E CPU card has 256KB of EPROM containing the boot monitor beginning at location 0xEC000000 in Type-1 physical space. The EPROM is also referenced by all Supervisor Virtual addresses when the ENA_NOTBOOT bit in the System Enable Register is zero; this occurs at boot time. The boot code must initialize the MMU to at least map itself before setting the ENA_NOTBOOT bit to one.

Note that the EPROM does not obey the normal mapping rules. PA[16:0] in the EPROM always come from VA[16:0]. Although VA[29:13] are processed by the MMU to select a physical address, when PA[27:24] of that physical address select the EPROM then bits PA[23:13] from the MMU are ignored. This means that, for proper operation of the EPROM, it must be mapped one-for-one to contiguous virtual pages beginning on a 256K boundary.

Diagnostic Output Register

The diagnostic output register is a one-byte write only register located at address 0xEE800000 in type 1 device space. The 8 bits are encoded to identify device or board failures. Each data bit drives a corresponding LED display bit. During selftest, the 8 LEDs provide status information; see the **Diagnostics** chapter later in this manual.

6.6. SBus Slots

SBus Slot 0 is assigned to the onboard SCSI and Ethernet interfaces. SBus slot 1 is used for frame buffer and I/O (SBus card) expansion. SBus slot 2 is reserved for future use.

For detailed information on the SBus, see the chapter *SBus Interface* in this manual.

Programming Example for the Diagnostic Output Register

This example is merely an infinite loop through the various LED pattern values. It demonstrates how to access the Diagnostic register so as to produce a changing LED display.

The pattern that we use is the familiar SunOS "IDLE" pattern. This causes the LEDs on the CPU board to be strobed in the familiar "cylon" display used by SunOS.

```
/*
 * LED data for idle pattern of diagnostic register.
 */

LEDPTR          =       18
LEDPATCNT =     18

        .seg    "data"
led:
        ! LEDPAT
        .byte   0x7e, 0x7e
        .byte   0x7e, 0xbd
        .byte   0xbd, 0xbd
        .byte   0xdb, 0xdb
        .byte   0xdb, 0xe7
        .byte   0xe7, 0xe7
        .byte   0xdb, 0xdb
        .byte   0xdb, 0xbd
        .byte   0xbd, 0xbd
        ! end of LEDPAT
        .byte   0                       ! LEDPTR offset in pattern

        .seg    "text"

        set     led, %l5                ! %l5 has address of LED pattern
1:
        mov     LEDPATCNT-1, %g1 ! # of LED patterns to loop through
        stb     %g1, [%l5 + LEDPTR]     ! initialize LED pattern index
2:
        ldub    [%l5 + LEDPTR], %g1     ! %g1 gets index into LED pattern
        ldub    [%l5 + %g1], %g2        ! %g2 gets LED pattern
        subcc   %g1, 1, %g1             ! point to next one
        stb     %g1, [%l5 + LEDPTR]     ! update pattern pointer
        set     DIAG_ADDR, %g3          ! %g3 gets virtual address of the Diag Register
        stb     %g2, [%g3]              ! write LED pattern to LEDs
        bneg    1b
        nop
        b       2b
        nop
```

6.7. P2 Bus Slot

A P2 Bus slot is reserved in addition to the SBus slots. This slot resides in type 1 space just as the SBus slot. This device should not be confused with the P2 Bus type 0 memory space. For implementations using the 4E Memory board the ECC registers will reside at address 0xFC000000 to 0xFC0000FF. Refer to the 4E Memory board specification for details.

7

Memory Management Unit

The MMU translates virtual addresses to physical addresses. In the process, it keeps track of the context, the segment and the page, and it provides memory protection and updating information.

Virtual memory occupies the first and last half GB of a 4 GB space. Accesses to the middle 3 GBs produce errors. Acceptable virtual memory addresses are:

0x00000000 to 0x1FFFFFFF, and

0xE0000000 to 0xFFFFFFFF

Note that to fall into the acceptable range, virtual address bits VA[31:29] must be either all 0's or all 1's.

The following list shows the map's capabilities:

Table 7-1 *MMU Attributes*

Attribute	Value
Page size	8 KBs
Segment size	256 KBs
Process size (context)	1 GB
Number of contexts	8
Pages per segment	32
Number of pmegs	256
Number of page map entries	8K
Number of segment map entries	32K

Accesses through the map take the following steps:

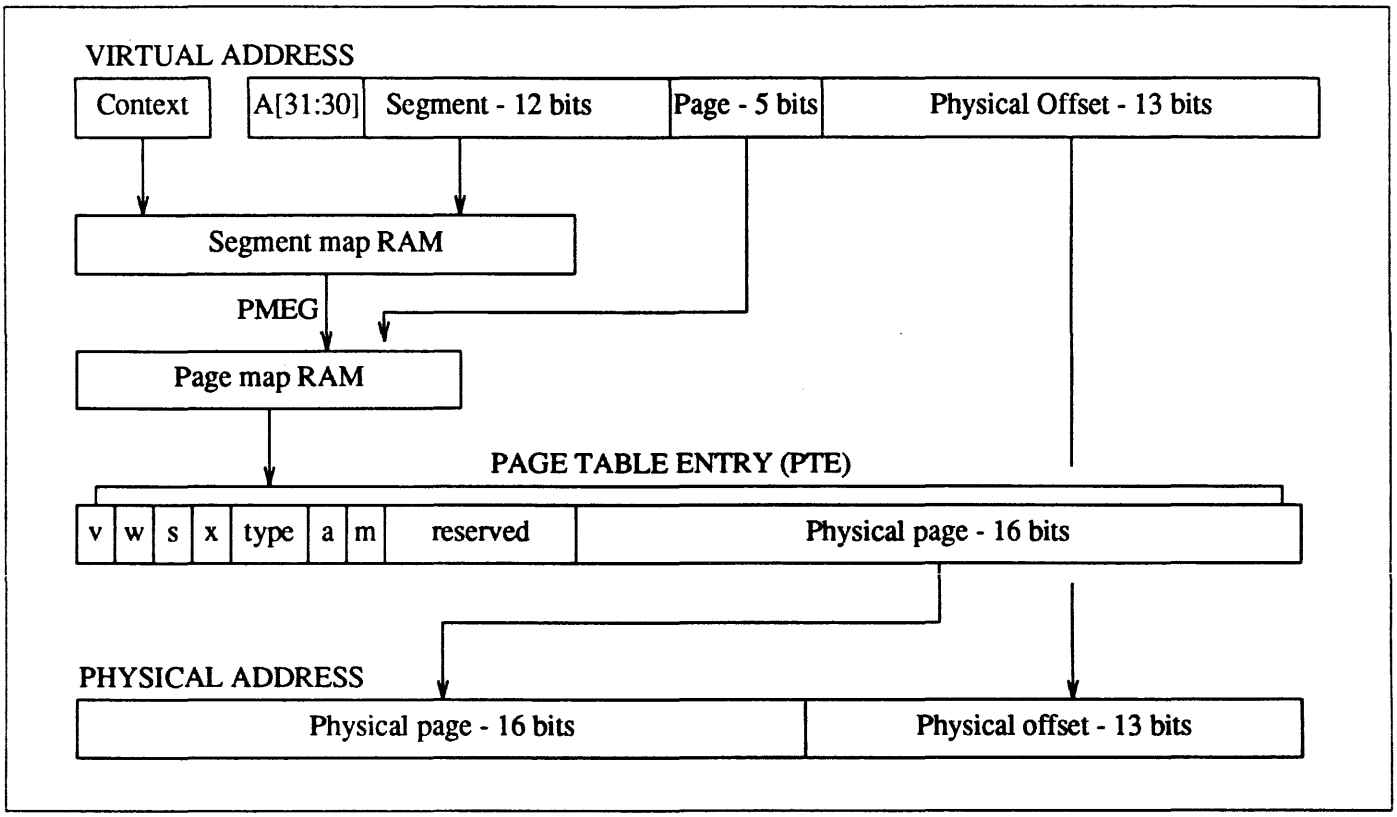
The context bits are concatenated to the segment number to select an address in the segment map, which puts out an 8-bit Page Map Entry Group (PMEG).

The PMEG is concatenated to the 5-bit page field of the virtual address, and these 13 bits select an address in the page map. The page map puts out 32-bits, containing 8-bits of information about the page (type, privilege, etc), and 16 bits of high-order address information which actually points to a physical page.

The physical page number is concatenated with the 13-bit physical offset from the virtual address, and this is a complete physical address.

The mapping appears in the following Figure:

Figure 7-1 *Memory Management Unit*



NOTE For compatibility with other Sun-4 architectures, bits PA[31:29] are assumed to be all zero's (000) for type 0 space, and all ones (111) for type 1 space. The map actually outputs bits PA[28:00].

7.1. Page ID bits

Bits 31 through 16 of the page map entry provide the following information about the page:

31 Valid bit. When HIGH, it means the page entry is valid and it allows read and execute access to the page.

30 Write access bit. When HIGH, page is enabled for writing.

29 Supervisor access. When HIGH, only supervisor access is allowed.

28 Don't cache. When HIGH, forces accesses to main memory instead of cache.

27 and 26
Type bits, which select device space access as follows:

Table 7-2 *Page ID Bit Definitions*

Bit 27	Bit 26	Type
0	0	Main memory (type 0)
0	1	I/O space (type 1)
1	0	VME D16 Port (type 2)
1	1	VME D32 Port (type 3)

25 Accessed bit. When HIGH, this bit indicates that the page has been accessed by the CPU or DVMA. This bit does not specify whether the access was a write or a read.

24 Modified bit. This bit is set when the page is modified.

23 through 16
Reserved for future use.

15 through 0
Physical page number. These bits identify an actual physical page. They are concatenated with bits 0 through 11 of the virtual address to form an actual physical address.

The MMU is loaded dynamically by the kernel, which keeps track of where it puts things.

Page ID Bits Programming Example

The following code fragment demonstrates the use of the page ID bits. In the example, we construct a mapping between a selected virtual address and the Diagnostic Register. Since the Diagnostic Register is in type 1 space (OBIO), we must make use of the type space bits. Furthermore, we wish to provide write access for the kernel, but not to users; hence, we must set the Valid (PG_V), Write Enable (PG_W), and Supervisor Access Only (PG_S) bits in the protection fields of the page ID bits.

Since type 1 (OBIO) space is not cacheable, we must also set the No Cache (PG_NC) bit of the page ID bits.

```
#define PG_V        0x80000000   /* page is valid */
#define PG_W        0x40000000   /* write enable bit */
#define PG_S        0x20000000   /* system page */
#define PG_NC             0x10000000   /* no cache bit */

#define OBIO        0x1


DIAG_ADDR =      0xDFFEE000  ! virtual address in type 1 space
OBIO_DIAG_ADDR =      0xEE800000   ! physical address of Diag reg

PROT        =    (PG_V | PG_W | PG_S | PG_NC)
TYPESPACE =      OBIO << 26
PGFRAME          =      (OBIO_DIAG_ADDR >> 13)
DIAG_PTE    =    PROT | TYPESPACE | PGFRAME

#define ASI_PM            0x4          ! page map address space indicator (ASI)
```

```
        set     DIAG_ADDR, %g1          ! map in Diagnostic Register
        set     DIAG_PTE, %g2           ! phys. page # for Diagnostic Register
        sta     %g2, [%g1]ASI_PM  ! put it in the page map.
```

7.2. Context Register

Bits 0 through 2 of the context register identify the current context; these bits are appended to virtual addresses before they enter the map.

7.3. Page Map

The page map is a 8K-by-32 bit RAM bank. Its 13 address inputs are formed by concatenating the PMEG output by the segment map with the page field of the virtual address. Effectively, the PMEG index identifies one of 256 sections, and virtual address bits A[17:13] select one of 32 pages within that section.

Programming Example for Using the Page Map Register

The following programming example is provided simply as a guide for your programming, and is not intended to be used as is.

```
/*
 * pagmap.s
 * Copyright (c) 1989 by Sun Microsystems, Inc.
 */

#define CONTEXTBASE    0x30000000    /* context reg */
#define ASI_CTL        0x2           /* control space*/
#define PAGTSTMAX      0x0ffc0000    /* full pmeg in each of 256 segm entries */
#define PAGINCR        0x2000        /* offset between adjacent pages */
#define SEGINCR        0x40000       /* offset between adjacent segments */
#define SGSHIFT        18          /* LOG2(NBSG) */
#define ASI_SM         0x3           /* segment map */
#define PG_MAP_VALID   0xff00ffff    /* valid r/w bits of page map */
#define TEST_PATT      0x5a972c5a    /* 3-pattern base pattern */
#define PGA_NUM        0x06          /* Page Map Address test */
#define ASI_PM         0x4           /* page map */
#define PG3_NUM        0x06          /* Page Map 3 pattern RAM test */o
#define PGWR_NUM       0x04          /* Page Map write-write-read test */
#define PATT_END       0x972c5a97    /* 3-pattern test finish pattern */
#define BUSERRBASE     0x60000000    /* bus error reg */
#define quit_key       0x11        /* Cntl-q: quit this test,go to next */


/*#define FORCE_PAGMAP_ERRORS*/
/*#define DEBUG_PAGMAP*/

        .seg    "text"
!-------------------------------------------------------------------------
!       <PAGE MAP TESTING SETUP>
!
! Initialize Context Reg and Segment Map for the following Page Map Tests.
!
! Polaris:
!       Set Context to 0, linearly map segment table.
!


        .global PM_setup
PM_setup:
    save   %sp, -MINFRAME, %sp ! Preserve calling window.

        set     CONTEXTBASE, %l4        ! Pointer to context register.
        stba    %g0, [%l4]ASI_CTL  ! Start with context 0.

        /* Linearly map the segment table from bottom to top. */
        set     0x00000000, %o4         ! Bottom of segment map
        set     PAGTSTMAX, %i4          ! Last address to write.
```

```
        set     SEGINCR, %l3            ! Address increment for adjacent
                                        ! segment map entries.
3:
        srl     %o4, SGSHIFT, %o2 ! Make entry from the seg map index.
        stha    %o2, [%o4] ASI_SM ! Store into segment map.
        cmp     %o4, %i4          ! Just stored last address?
        bne     3b                ! If not then loop back and
        add     %o4, %l3, %o4           ! point to next segment map entry.

!-----------------------------------------------------------------------
!       <PAGE MAP WRITE-WRITE-READ-READ TEST>
!
! For each page table entry from bottom to top:
!       Write data to entry x.
!       Write inverse data to entry x+1.
!       Verify entry x+1.
!       Verify entry x.
!
! Register usage:
!       %l0 - Test pattern.
!       %l3 - Address increment for adjacent page map entries.
!       %l4 - Address of Context register in control space.
!       %l6 - Page map entry valid bit mask.
!       %o4 - Address of page map entry being tested.
!       %i4 - Address of last page map entry to test.
!

Test_0b:
        set     pagemwr_tst_txt, %o0! Test descriptor text.
        call    test$             ! Display text and test number.
        set     ~PGWR_NUM, %o1          ! Set inverse test number.
        set     0x00000000, %o4         ! Addr of first Page map entry.
        set     PAGTSTMAX-PAGINCR, %i4  ! Addr of last Page map entry.
        set     PAGINCR, %l3            ! Address increment for adjacent
                                        ! page map entries.
        set     PG_MAP_VALID, %l6       ! Page map entry valid bit mask.
        set     TEST_PATT, %l0          ! Test pattern source.
        and     %l0, %l6, %o2     ! Strip unused bits from pattern.

        set     2f, %g4                 ! <<<TOP OF TEST LOOP>>>
2:
        sta     %o2, [%o4] ASI_PM ! Store pattern into pagemap RAM.

        xor     %o2, %l6, %o2           ! Invert pattern.
        add     %o4, %l3, %o4           ! Increment to adjacent pagmap entry.
        sta     %o2, [%o4] ASI_PM ! Store invert pattern in pagmap RAM.

        lda     [%o4] ASI_PM, %o1 ! Get pattern from high location.
        and     %o1, %l6, %o1           ! Strip unused bits from pattern.
        cmp     %o1, %o2          ! Data okay?
        be      3f
        nop
        set     pagemwr_err_txt, %o0    ! Error message text.
        call    error$
        nop
3:
        sub     %o4, %l3, %o4           ! Decr down to original pagmap entry.
        lda     [%o4] ASI_PM, %o1 ! Get pattern from low location.
        and     %o1, %l6, %o1           ! Strip unused bits from pattern.
        xor     %o2, %l6, %o2           ! Re-invert source pattern.
        cmp     %o1, %o2          ! Data okay?
        be      4f
        nop
        set     pagemwr_err_txt, %o0    ! Error message text.
```

```
        call    error$
        nop
4:
        call    loop$end            ! <<<BOTTOM OF TEST LOOP>>>
        nop
        cmp     %o0, quit_key       ! Skip to next test?
        bz      9f
        nop

        cmp     %o4, %i4            ! Just tested last location?
        bne     2b                  ! If not then loop back and
        add     %o4, %l3, %o4               ! increment pagmap address in slot.
9:

!-------------------------------------------------------------------------
!       <PAGE MAP ADDRESS TEST>
!
! For each page table entry from bottom to top:
!       Write address of entry x as data into entry x.
! For each page table entry from bottom to top:
!       Verify data in entry x is address of entry x.
!
! Register usage:
!       %l3 - Address increment for adjacent page map entries.
!       %l4 - Address of Context register in control space.
!       %l6 - Page map entry valid bit mask.
!       %o4 - Address of page map entry being tested.
!       %i4 - Address of last page map entry to test in each context.
!
!

Test_0c:
        set     pagema_tst_txt, %o0 ! Test descriptor text.
        call    test$               ! Display text and test number.
        set     ~PGA_NUM, %o1               ! Set inverse test number.
        set     0x00000000, %o4             ! Addr of first Page map entry.
        set     PAGINCR, %l3                ! Address increment for adjacent
                                    ! page map entries.
        set     PG_MAP_VALID, %l6           ! Page map entry valid bit mask.

        set     1f, %g4                     ! <<<TOP OF TEST LOOP>>>
1:
        set     PAGTSTMAX, %i4              ! Addr of last Page map entry.

        /* Write page map address as data into each page map entry. */
2:
        and     %o4, %l6, %o2               ! Strip unused bits.
        sta     %o2, [%o4] ASI_PM   ! Store pattern into page map RAM.
        cmp     %o4, %i4            ! Just wrote last entry?
        bne     2b                  ! If not then loop back and
        add     %o4, %l3, %o4               ! increment page map addr in slot.

        /* Verify that all page map entries have correct data (page table addr). */
        set     PAGTSTMAX, %o4              ! Address of last page map entry.
3:
        and     %o4, %l6, %o2               ! Strip unused bits.
        lda     [%o4] ASI_PM, %o1   ! Read pattern from high location.
        and     %o1, %l6, %o1               ! Strip unused bits.
        cmp     %o1, %o2            ! Data okay?
        be      4f
        nop
        set     pagema_err_txt, %o0 ! Error message text.
        call    error$
        nop
```

```
4:
        call    loop$end            ! <<<BOTTOM OF TEST LOOP>>>
        nop
        cmp     %o0, quit_key       ! Skip to next test?
        bz      6f
        nop

        cmp     %o4, %i4            ! Just tested last entry?
        bne     3b                  ! If not then loop back and
        add     %o4, %l3, %o4               ! increment pagmap addr in slot.
6:

!-------------------------------------------------------------------------
!       <PAGE MAP 3-PATTERN TEST>
!
! Pass 1: Writes 5A,2C,97 to consecutive seg map entries from bottom to top.
!         Verifies entire page map.
! Pass 2: Writes 2C,97,5A to consecutive seg map entries from bottom to top.
!         Verifies entire page map.
! Pass 3: Writes 97,5A,2C to consecutive seg map entries from bottom to top.
!         Verifies entire page map.
!
! Register usage:
!       %l0 - Test pattern.
!       %l3 - Address increment for adjacent page map entries.
!       %l4 - Address of Context register in control space.
!       %l7 - Ending test pattern.
!       %o4 - Address of page map entry being tested.
!       %i4 - Address of last page map entry to test in each context.
!

Test_0d:
        set     pagem3_tst_txt, %o0 ! Test descriptor text.
        call    test$               ! Display text and test number.
        set     ~PG3_NUM, %o1               ! Set inverse test number.

        set     BUSERRBASE, %l5             ! Ponit to the bus err reg
        lda     [%l5]ASI_CTL, %l2           ! clear bus error reg
        nop
        set     PAGTSTMAX, %i4              ! Addr of last Page map entry.
        set     PAGINCR, %l3                ! Address increment for adjacent
                                    ! page map entries.
        set     PG_MAP_VALID, %l6           ! Page map entry valid bits mask.
        set     PATT_END, %l7               ! Ending pattern.
        set     TEST_PATT, %l0              ! Test pattern source.

        set     1f, %g4                     ! <<<TOP OF TEST LOOP>>>
1:
        set     0x00000000, %o4             ! Address of first Page Map entry.
2:
        /* Fill the page map with 3 repeating patterns. */
        mov     %l0, %l1            ! Set up pattern for this pass.
3:
        and     %l1, %l6, %o2       ! Generate test patt in o2.
        sta     %o2, [%o4] ASI_PM   ! Store pattern into pagemap RAM.
        cmp     %o4, %i4            ! Just wrote last location?
        be      4f
        nop
        add     %o4, %l3, %o4               ! Increment pagmap addr to next entry.
        srl     %l1, 0x8, %l1       ! Shift pattern right a byte.
        sll     %l1, 0x18, %l5              ! Duplicate low byte in high byte.
        or      %l1, %l5, %l1
        ba      3b                  ! Do the next mod3 pattern.
        nop
```

```
4:
/* Verify the page map with 3 repeating patterns. */
#ifdef DEBUG_PAGMAP
        set     dbgmsg1, %o0
        call    print$
        nop
#endif DEBUG_PAGMAP
        set     BUSERRBASE, %l5         ! Ponit to the bus err reg
        lda     [%l5]ASI_CTL, %l2  ! clear bus error reg
        nop
        set     PAGTSTMAX, %o4          ! Address of last page map entry.
2:
        mov     %l0, %l1                ! Set up pattern for this pass.
3:
        and     %l1, %l6, %o2           ! Generate test patt in o2.
        lda     [%o4] ASI_PM, %o1  ! Read pattern from page map RAM.
        and     %o1, %l6, %o1           ! Strip unused bits.
        cmp     %o2, %o1                ! Data okay?
        be      5f
        nop
        set     pagem3_err_txt, %o0 ! Pass addr of error message.
        call    error$                  ! Show error text and loop for scoping.
        nop

        set     BUSERRBASE, %l5         ! Ponit to the bus err reg
        lda     [%l5]ASI_CTL, %l2  ! clear bus error reg
        nop
5:
        cmp     %o4, %i4                ! Just read last location?
        be      4f
        nop
        add     %o4, %l3, %o4           ! Increment pagmap addr to next entry.
        srl     %l1, 0x8, %l1           ! Shift pattern right a byte.
        sll     %l1, 0x18, %l5          ! Duplicate low byte in high byte.
        or      %l1, %l5, %l1
        ba      3b                      ! Do the next mod3 pattern.
        nop
4:
        call    loop$end                ! <<<BOTTOM OF TEST LOOP>>>
        nop
        cmp     %o0, quit_key           ! Skip to next test?
        bz      5f
        nop

/* Just finished writing and verifying page map for all segments for one
  of 3 different passes. Now see if just did last of 3 passes. If not then
  shift starting pattern and go back to first segment.
*/
#ifdef DEBUG_PAGMAP
        set     dbgmsg2, %o0
        call    print$
        nop
#endif DEBUG_PAGMAP
        cmp     %l0, %l7                ! Arrived at terminating pattern?
        bz      5f                      ! If so go to next test.
        srl     %l0, 0x8, %l0           ! Shift pattern a byte.
        sll     %l0, 0x18, %l1          ! Duplicate low byte in high byte.
        or      %l0, %l1, %l0
        ba      1b
        nop
5:
#ifdef DEBUG_PAGMAP
        set     dbgmsg3, %o0
        call    print$
```

```
        nop
#endif DEBUG_PAGMAP

        ret
        restore

#ifdef DEBUG_PAGMAP
dbgmsg1:
        .asciz  " 15  12 DEBUG: Done writing, time to read."
dbgmsg2:
        .asciz  " 15  12 DEBUG: Done with this pattern, do next."
dbgmsg3:
        .asciz  " 15  12 DEBUG: All done with 3 patterns."
#endif DEBUG_PAGMAP
```

7.4. Segment Map

The segment map is a 32-by-9 bit RAM bank. Its 15 address inputs are formed by concatenating the contents of the context register (3 bits) to bits A[29:18] of the virtual address. Each of these addresses contains a 8-bit PMEG index number.

Programming Example for Setting the Segment Map Register

The following programming example is provided simply as a guide for your programming, and is not intended to be used as is.

```
/*
 * segmap.s
 * Copyright (c) 1989 by Sun Microsystems, Inc.
 */

#define CONTEXTBASE    0x30000000     /* context reg */
#define ASI_CTL        0x2            /* control space*/
#define SEGINCR        0x40000        /* offset between adjacent segments */
#define SGSHIFT        18             /* LOG2(NBSG) */
#define ASI_SM         0x3            /* segment map */
#define TEST_PATT      0x5a972c5a     /* 3-pattern base pattern */
#define ASI_PM         0x4            /* page map */
#define PATT_END       0x972c5a97     /* 3-pattern test finish pattern */
#define quit_key       0x11           /* Cntl-q: quit this test,go to next */
#define SGWR_NUM       0x05           /* Segment Map write-write-read test */
#define SEGMAPMAX      0x0ffc0000     /* sunrise, all full seg map bits */
#define SG_MAP_VALID   0xff           /* valid r/w bits of segment map */
#define NCONTEXT       8              /* 8 context */
#define SGA_NUM        0x05           /* Segment Map Address test */
#define SG3_NUM        0x05           /* Segment Map 3 pattern RAM test */

/*#define FORCE_SEGMAP_ERRORS*/
/*#define DEBUG_SEGMAP*/

        .seg    "text"

!-------------------------------------------------------------------------
!       <SEGMENT MAP WRITE-WRITE-READ-READ TEST>
!
! Polaris: (16 context)(4096 entries/context) = 65536 segment map entries.
!
! For each segment table entry from bottom to top:
!       Write data to entry x.
!       Write inverse data to entry x+1.
!       Verify entry x+1.
!       Verify entry x.
!
! Register usage:
!       %l0 - Test pattern.
```

```
!       %l2 - Context #
!       %l3 - Address increment for adjacent segment map entries.
!       %l4 - Address of Context register in control space.
!       %o4 - Address of segment map entry being tested.
!       %i4 - Address of last segment map entry to test in each context.
!

        .global segm_tst
segm_tst:
    save  %sp, -MINFRAME,%sp    ! preserve calling window

        set     CONTEXTBASE, %l4        ! Set context to 0.
        stba    %g0, [%l4] ASI_CTL

        set     segmwr_tst_txt, %o0  ! test descriptor text
        call    test$                ! display text and test number
        set     ~SGWR_NUM, %o1              ! set inverse test number
        mov     %g0, %l2             ! Holds context number.
        set     SEGINCR, %l3                ! Address increment for adjacent
                                     ! segment map entries.
2:
        stba    %l2, [%l4] ASI_CTL  ! Set context.
        set     0x00000000, %o4             ! Address of first segment map entry.
        set     SEGMAPMAX-SEGINCR, %i4      ! Address of last segment map entry.
        set     TEST_PATT, %l0              ! Test pattern.

        set     3f, %g4                     ! <<<TOP OF TEST LOOP>>>
3:
        and     %l0, SG_MAP_VALID, %o2 ! Strip unused bits from pattern.
4:
        stha    %o2, [%o4] ASI_SM  ! Store pattern in segmap RAM.
        xor     %o2, SG_MAP_VALID, %o2      ! Invert pattern.
        add     %o4, %l3, %o4               ! Increment to adjacent segmap entry.
        stha    %o2, [%o4] ASI_SM  ! Store invert pattern in segmap RAM.

        lduha   [%o4] ASI_SM, %o1  ! Get pattern from high location.
        and     %o1, SG_MAP_VALID, %o1      ! Strip unused bits from pattern.
        cmp     %o1, %o2           ! Data okay?
#ifndef FORCE_SEGMAP_ERRORS
        be      5f
#endif FORCE_SEGMAP_ERRORS
        nop
        set     segmwr_err_txt, %o0  ! Error message text.
        call    error$
        nop
5:
        sub     %o4, %l3, %o4               ! Decr down to original segmap entry.
        lduha   [%o4] ASI_SM, %o1  ! Get pattern from low location.
        and     %o1, SG_MAP_VALID, %o1      ! Strip unused bits from pattern.
        xor     %o2, SG_MAP_VALID, %o2      ! Re-invert source pattern.
        cmp     %o1, %o2           ! Data okay?
#ifndef FORCE_SEGMAP_ERRORS
        be      6f
#endif FORCE_SEGMAP_ERRORS
        nop
        set     segmwr_err_txt, %o0  ! Error message text.
        call    error$
        nop
6:
        call    loop$end           ! <<<BOTTOM OF TEST LOOP>>>
        nop
        cmp     %o0, quit_key      ! Skip to next test?
        bz      9f
        nop
```

```
        cmp     %o4, %i4            ! Just tested last location?
        bne     4b                  ! If not then loop back.
        add     %o4, %l3, %o4               ! Increment segmap addr in slot.

        cmp     %l2, NCONTEXT-1             ! Just tested last context?
        bne     2b
        add     %l2, 1, %l2         ! Increment context number.
9:


!-----------------------------------------------------------------------
!       <SEGMENT MAP ADDRESS TEST>
!
! For each segment table entry from bottom to top:
!       Write address of entry x as data into entry x.
! For each segment table entry from bottom to top:
!       Verify data in entry x is address of entry x.
!
! Register usage:
!       %l2 - Context # (goes from 0 to NCONTEXT for Polaris)
!       %l3 - Address increment for adjacent segment map entries.
!       %l4 - Address of Context register in control space.
!       %o4 - Address of segment map entry being tested.
!       %i4 - Address of last segment map entry to test in each context.
!

Test_09:
        set     segma_tst_txt, %o0  ! Test descriptor text.
        call    test$               ! Display text and test number.
        set     ~SGA_NUM, %o1               ! Set inverse test number.
        mov     %g0, %l2            ! Holds context number.
        set     SEGINCR, %l3                ! Address increment for adjacent
                                    ! segment map entries.
        set     CONTEXTBASE, %l4            ! Pointer to context register.

        set     1f, %g4                     ! <<<TOP OF TEST LOOP>>>
1:
        stba    %l2, [%l4] ASI_CTL  ! Set context.
        set     0x00000000, %o4             ! Address of first segment map entry.
        set     SEGMAPMAX, %i4              ! Address of last segment map entry.

/* Write segment map address as data into each segment map entry. */
2:
        srl     %o4, SGSHIFT, %o2   ! Use segment map index for data.
        stha    %o2, [%o4] ASI_SM   ! Write data into segment map RAM.
        cmp     %o4, %i4            ! Just wrote last location?
        bne     2b                  ! If not then loop writing and
        add     %o4, %l3, %o4               ! increment segment map addr in slot.

        set     0x00000000, %o4             ! Address of first segment map entry.
        set     SEGMAPMAX, %i4              ! Address of last segment map entry.

/* Verify that all segment map entries have correct data (seg table addr). */
3:
        srl     %o4, SGSHIFT, %o2   ! Use segment map index for data.
        and     %o2, SG_MAP_VALID, %o2      ! Strip unused bits from expected data.
        lduha   [%o4] ASI_SM, %o1   ! Read data from segment map RAM.
        and     %o1, SG_MAP_VALID, %o1      ! Strip unused bits from actual data.
        cmp     %o1, %o2            ! Data okay?
#ifndef FORCE_SEGMAP_ERRORS
        be      4f
#endif FORCE_SEGMAP_ERRORS
        nop
        set     segma_err_txt, %o0  ! Error message text.
```

```
        call    error$
        nop
4:
        call    loop$end            ! <<<BOTTOM OF TEST LOOP>>>
        nop
        cmp     %o0, quit_key       ! Skip to next test?
        bz      9f
        nop

        cmp     %o4, %i4            ! Just tested last location?
        bne     3b                  ! If not then loop back.
        add     %o4, %l3, %o4               ! Increment segmap addr in slot.

        cmp     %l2, NCONTEXT-1             ! Just tested last context?
        bne     2b                  ! If not then loop back, and
        add     %l2, 1, %l2         ! increment context number in slot.
9:

!-------------------------------------------------------------------------
!       <SEGMENT MAP 3-PATTERN TEST>
!
! Pass 1: Writes 5A,2C,97 to consecutive seg map entries from bottom to top.
!         Verifies entire segment map.
! Pass 2: Writes 2C,97,5A to consecutive seg map entries from bottom to top.
!         Verifies entire segment map.
! Pass 3: Writes 97,5A,2C to consecutive seg map entries from bottom to top.
!         Verifies entire segment map.
!
! Register usage:
!       %l0 - Test pattern.
!       %l2 - Context # (goes from 0 to NCONTEXT for Sunrise,Cobra,Stingray)
!       %l3 - Address increment for adjacent segment map entries.
!       %l4 - Address of Context register in control space.
!       %l7 - Ending test pattern.
!       %o4 - Address of segment map entry being tested.
!       %i4 - Address of last segment map entry to test in each context.
!

Test_0a:
        set     segm3_tst_txt, %o0  ! Test descriptor text.
        call    test$               ! Display text and test number.
        set     ~SG3_NUM, %o1               ! Set inverse test number.
        set     SEGINCR, %l3                ! Address increment for adjacent
                                    ! segment map entries.
        set     CONTEXTBASE, %l4            ! Pointer to context register.
        set     0xffffff, %l6       ! Segment map entry mask.
        set     PATT_END, %l7               ! End of mod3 shift pattern.
        set     TEST_PATT, %l0              ! Test pattern source.
1:
        mov     %g0, %l2            ! Holds context number.

        set     2f, %g4                     ! <<<TOP OF TEST LOOP>>>
2:
        stba    %l2, [%l4] ASI_CTL  ! Set context.

/* Fill the segment map for one context/region with 3 repeating patterns. */
        set     0x00000000, %o4             ! Address of first segment map entry.
        set     SEGMAPMAX, %i4              ! Address of last segment map entry.
3:
        and     %l0, %l6, %l1       ! Set up pattern for this pass.
4:
        and     %l1, SG_MAP_VALID, %o2 ! Generate write data.
        stha    %o2, [%o4] ASI_SM   ! Write data into segment map RAM.
        cmp     %o4, %i4            ! Just wrote last location?
```

```
        be      4f                      ! If so go check data.
        nop
        add     %o4, %l3, %o4                   ! Increment segmap addr to next entry.
        srl     %l1, 0x8, %l1           ! Shift pattern right a byte.
        tst     %l1                     ! Shifted all the way out?
        bz      3b                      ! If so start with first of 3 patterns.
        nop
        ba      4b                      ! If not then next pattern.
        nop
4:
/* Verify the segment map for one context/region with 3 repeating patterns. */
#ifdef DEBUG_SEGMAP
        set     dbgmsg1, %o0
        call    print$
        nop
#endif DEBUG_SEGMAP

        set     0x00000000, %o4                 ! Address of first segment map entry.
        set     SEGMAPMAX, %i4          ! Address of last segment map entry.
5:
        and     %l0, %l6, %l1           ! Set up pattern for this pass.
6:
        and     %l1, SG_MAP_VALID, %o2 ! Generate test data.
        lduha   [%o4] ASI_SM, %o1  ! Read data from segment map RAM.
        and     %o1, SG_MAP_VALID, %o1          ! Strip out unused bits.
        cmp     %o2, %o1                ! Data okay?
#ifndef FORCE_SEGMAP_ERRORS
        be      7f
#endif FORCE_SEGMAP_ERRORS
        nop
        set     segm3_err_txt, %o0  ! Pass addr of error message.
        call    error$                  ! Show error text and loop for scoping.
        nop
7:
        cmp     %o4, %i4                ! Just verified last location?
        be      8f
        nop
        add     %o4, %l3, %o4                   ! Increment segment map address.
        srl     %l1, 0x8, %l1           ! Shift pattern right a byte.
        tst     %l1                     ! Shifted all the way out?
        bz      5b                      ! If so start another mod3.
        nop
        ba      6b                      ! If not do the next mod3 pattern.
        nop
8:
        call    loop$end                ! <<<BOTTOM OF TEST LOOP>>>
        nop
        cmp     %o0, quit_key           ! Skip to next test?
        bz      9f
        nop
        cmp     %l2, NCONTEXT-1                 ! Just tested last context?
        bne     2b                      ! If not then loop back and
        add     %l2, 1, %l2             ! increment context number in slot.

/* Just finished writing and verifying segment map for all contexts for one
  of 3 different passes. Now see if just did last of 3 passes. If not then
  shift starting pattern and go back to first context.
*/
#ifdef DEBUG_SEGMAP
        set     dbgmsg2, %o0
        call    print$
        nop
#endif DEBUG_SEGMAP
```

```
        cmp     %l0, %l7                ! Arrived at terminating pattern?
        beq     9f                      ! If so go to next test.
        srl     %l0, 0x8, %l0           ! Shift pattern a byte.
        sll     %l0, 0x18, %l1          ! Duplicate low byte in high byte.
        or      %l0, %l1, %l0
        ba      1b
        nop
9:
#ifdef DEBUG_SEGMAP
        set     dbgmsg3, %o0
        call    print$
        nop
#endif DEBUG_SEGMAP

        ret
        restore

#ifdef DEBUG_SEGMAP
dbgmsg1:
        .asciz  " 15  12 DEBUG: Done writing, time to read."
dbgmsg2:
        .asciz  " 15  12 DEBUG: Done with this pattern, do next."
dbgmsg3:
        .asciz  " 15  12 DEBUG: All done with 3 patterns."
#endif DEBUG_SEGMAP
```

7.5. Direct Access to Map Data

Direct accesses to the segment and page maps are performed using *asi* = 0x3 for the segment map and *asi* = 0x4 for the page map. For the page map, use 32-bit accesses, and for the segment map, use 16-bit accesses.

These accesses are typically used to set up and modify the map.

7.6. Initialization of the UART

Function "UARTinit" initializes serial ports A and B by way of the MMU bypass. Serial port A will be set up at baud rate 9600 as the i/o device while serial port B will be set up at baud rate 1200.

```
/*
 * uartinit.s
 *
 * @(#)uartinit.s 1.0 90/03/26
 * Copyright (c) 1987 by SUN Microsystems, Inc.
 */

#define  UARTACNTL      ZSBASE + 4     /* Port A Control Address */
#define  ZSBASE         0xF0000000     /* ZS serial port base */
#define  ASI_CTL 0x2  /* control space*/
#define  ASI_SP  0x9  /* supervisor program */
#define  UARTrecov      4*MSEC         /* must make delay > 1.6 uSec */
#define  MSEC (CLOCK_RATE/3)/1000000  /* one microsecond worth of ticks */
#define  CLOCK_RATE     25000000       /* 25.0 MHz */
#define  UARTBCNTL      ZSBASE         /* base address for the UART */



/*
 * Function "UARTinit" initializes serial ports A and B by way of the mmu
 * bypass. Serial port A will be set up at baud rate 9600 as the i/o device
 * while serial port B will be set up at baud rate 1200.
 */
_UARTinit:
UARTinit:
        save %sp, -MINFRAME, %sp
```

```
        /*
         * With the mmu bypass, it is possible to access the uart without
        * going through the mmu. This enables us to print messages on a
        * dumb terminal before the mmu is tested. Helpful if mmu is broken.
        *
        * Initialize channel A at 9600 baud.
         */
        set     UARTACNTL, %l0          ! Address of port A in control space.
        stba    %g0, [%l0]ASI_CTL  ! Clear A control.
        set     uartinita, %l4          ! Address of port A uart initialization
1:      lduba   [%l4]ASI_SP, %l1    ! table in ROM. Get a byte from table.
        cmp     %l1, 0xff               ! Check for end of table.
        be      3f              ! Channel A initialized.
        nop                     ! Now initialize channel B.
        stba    %l1, [%l0]ASI_CTL  ! Write byte to SCC chip.
        orcc    %g0, UARTrecov, %l2     ! Set up real delay count.
2:      deccc   %l2                     ! Count down for delay.
        bg      2b              ! Delay required when talking to SCC.
        nop
        ba      1b                      ! Read and write another byte.
        inc     %l4                     ! Bump table pointer to next byte.

3:      /*
         * Initialize channel B at 1200 baud.
         */
        set     uartinitb, %l4          ! Address of port B uart initialization
      orcc      %g0, UARTrecov, %l2     ! Set up real delay count.
2:      deccc   %l2             ! Count down for delay.
        bg    2b                ! Delay required when talking to SCC.
        nop
        set     UARTBCNTL, %l0          ! Address of port B in control space.
        stba    %g0, [%l0]ASI_CTL  ! Clear B control.
1:      lduba   [%l4]ASI_SP, %l1    ! Get a byte from the table in ROM.
        cmp     %l1, 0xff               ! Check for end of table.
        be      3f              ! Channel B initialized.
        nop
        stba    %l1, [%l0]ASI_CTL  ! Write byte to SCC chip.
        orcc    %g0, UARTrecov, %l2     ! Set up real delay count.
2:      deccc   %l2                     ! Count down for delay.
        bg      2b              ! Delay required when talking to SCC.
        nop
        ba      1b                      ! Read and write another byte.
        inc     %l4                     ! Bump table pointer to next byte.
3:      ret
        restore

        .globl  _UARTBinit
_UARTBinit:
        /*
         * Initialize channel B at baud rate specified by %i0.
         */
        save %sp, -MINFRAME, %sp
        mov     %i0, %l4                ! Address of port B uart initialization
      orcc      %g0, UARTrecov, %l2     ! Set up real delay count.
2:      deccc   %l2             ! Count down for delay.
        bg    2b                ! Delay required when talking to SCC.
        nop
        set     UARTBCNTL, %l0          ! Address of port B in control space.
        stba    %g0, [%l0]ASI_CTL  ! Clear B control.
1:      lduba   [%l4]ASI_SP, %l1    ! Get a byte from the table in ROM.
        cmp     %l1, 0xff               ! Check for end of table.
        be      3f              ! Channel B initialized.
        nop
        stba    %l1, [%l0]ASI_CTL  ! Write byte to SCC chip.
```

```
        orcc    %g0, UARTrecov, %l2         ! Set up real delay count.
2:      deccc   %l2                         ! Count down for delay.
        bg      2b              ! Delay required when talking to SCC.
        nop
        ba      1b                          ! Read and write another byte.
        inc     %l4                         ! Bump table pointer to next byte.
3:      ret
        restore

/*
 * Initialization table for Channel A.
 */
uartinita:
/*
 * Time constant defined by this formula: ((4915200/32)/(baud) -2)
 *
 * low byte of time constant
 */
#ifdef FORTH
#else FORTH
#endif FORTH
/*
 * high byte of time constant
 */

/*
 * Initialization table for Channel B.
 */
uartinitb:
/*
 * Do not do the hardware reset again or channel A will be cleared.
 */
/*
 * Time constant defined by this formula: (using X16 mode above)
 * ((4915200/32)/(baud) -2)
 */
#ifdef FORTH
#else FORTH
#endif FORTH



#if defined(FORTH) || defined(DEMON)
_baud1200:
/*
 * don't do the hardware reset again or channel A will be cleared...
 */
/*
 * Time constant defined by this formula: (using X16 mode above)
 * ((4915200/32)/(baud) -2)
 */

_baud9600:
/*
 * don't do the hardware reset again or channel A will be cleared...
 */
/*
 * Time constant defined by this formula: (using X16 mode above)
 * ((4915200/32)/(baud) -2)
 */

_baud19200:
/*
 * don't do the hardware reset again or channel A will be cleared...
```

```
 */
/*
 * Time constant defined by this formula: (using X16 mode above)
 * ((4915200/32)/(baud) -2)
 */

_baud38400:
/*
 * don't do the hardware reset again or channel A will be cleared...
 */
/*
 * Time constant defined by this formula: (using X16 mode above)
 * ((4915200/32)/(baud) -2)
 */
#endif defined(FORTH) || defined(DEMON)
```

8

Boot PROM

8.1. Required Reference Material for the Boot PROM

Reference material required for a complete definition of the SBus:

SBus Developer's Kit, Sun Part No. 825-1219-xx (Sun SBus Information).

Within the ***SBus Developer's Kit***, there is a book entitled ***Open PROM Toolkit User's Guide***. While this book was written for use with the SPARCstation 1, it can be easily amended to use with the SPARCengine 1E CPU card. Below, a collection of notes are presented that amend and add to the contents of the Open PROM User's Guide.

SPARCengine 1E Notes for the Open PROM Toolkit User's Guide

The following notes add or delete information in the Open PROM User's Guide. Use these notes to convert the User's Guide from SPARCstation 1 to SPARCengine 1E. In general, replace all mention of SPARCstation 1 with SPARCengine 1E. Because the SPARCengine 1E is not delivered with tape or hard disk (i.e., delivered as a card, not a system) the text of the manual should be interpreted for the card level, without hard disk or floppy disk as standard equipment.

Chapter 1 Notes

Chapter 1, Page 3: At the end of the paragraph on the Forth Toolkit, add the following text. "A password may be required to enter the Forth toolkit."

Chapter 2 Notes

Chapter 2, Page 7: In the second paragraph, add the following text: "If in the diagnostic mode, the extended selftests begin immediately, otherwise, the normal power-on self-test sequence begins."

Chapter 2, Page 8: In the first paragraph, delete the last phrase "systems internal hard disk (SCSI) drive." and replace it with the following text: "...a hard disk drive at target 0."

Chapter 3 Notes

Chapter 3, Page 14: In step 2 of Aborting a Hung System, add the text "enter password, if required."

Chapter 3, Page 18: In Figure 3-2, the third device is "st (c,u,p) or tape SCSI Tape " Also, in the Note immediately below, change "When using le, sd and fd..." to "When using le, sd and st..."

Chapter 5 Notes

Chapter 5, Page 33: Delete the entry "Ejecting a Floppy Diskette."

Chapter 5, Page 35: Delete the "test floppy" line from Figure 5-2.

Chapter 5, Page 36: Delete the section "Testing the Diskette Drive System."

Chapter 5, Page 48: Delete the section "Ejecting a Floppy Diskette."

Chapter 5, Page 49: Delete the line beginning "Eject floppy...."

Chapter 6 Notes

Chapter 6, Page 52: In Figure 6-2, add the following items:

Table 8-1 *Open PROM Toolkit User's Guide Figure 6-2: Additional Items for the SPARCengine 1E*

Parameter Name	Value	Default Value
sbus-probe-list	01(delete 23)	01(delete 23)
ttyb-its-dtr-off	(Def: fake)	
ttya-its-dtr-off	(Def: fake)	
sd-targets	0123456	0123456
st-targets	0123456	0123456

Chapter 7 Notes

Chapter 7, Page 79: Delete all "Slot offsets" at the bottom of the page, and replace with: "Slot #1 - 400.000"

9

Cache

9.1. Overview of the Cache

To improve performance, the SPARCengine 1E contains a 64KB virtual address cache. The cache is implemented as a write-through, mixed instruction and data cache with a 16-byte line size. Caching is provided for type 0 (OBMEM) memory only--OBIO space and VME space memory accesses are not cached.

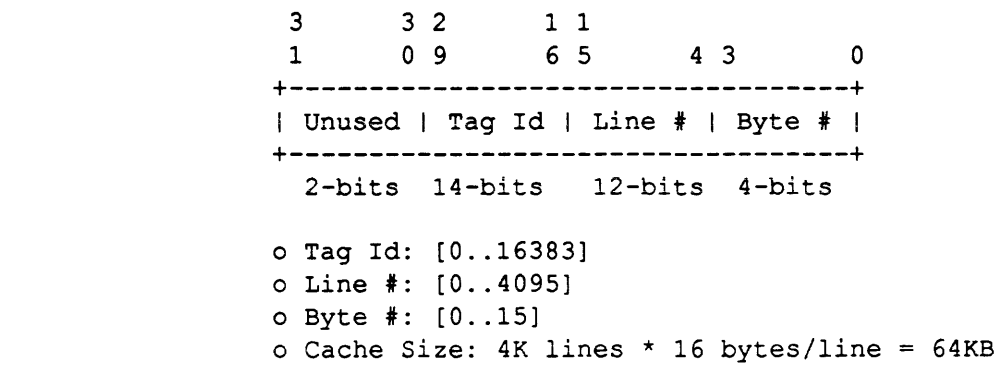
The cache is organized as 4096 lines of 16-bytes each and is one-way set associative, with each virtual address mapping to one and only one possible cache line. Each cache line has a 4-byte cache tag associated with it. This cache tag is used to lookup and match virtual addresses asserted by the IU.

9.2. Organization of the Cache

As mentioned earlier, the SPARCengine 1E cache is a virtual address cache. This means that cache addresses are derived directly from the virtual addresses asserted by the SPARC IU. Indices for the cache tags, cache lines, and the individual bytes within a cache line are extracted as fixed fields within a 30-bit virtual address.

The cache address decoding of virtual addresses is given in the figure below.

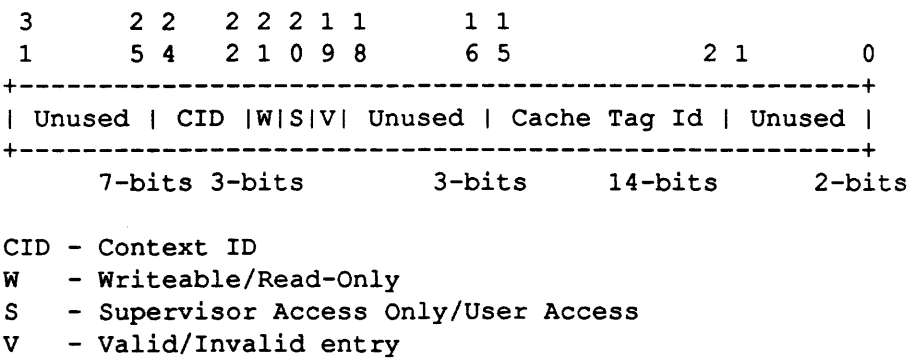
Figure 9-1 *Cache Address Decoding of Virtual Addresses*



- Tag Id: [0..16383]
- Line #: [0..4095]
- Byte #: [0..15]
- Cache Size: 4K lines * 16 bytes/line = 64KB

There is a 4-byte cache tag associated with each data line. The format of a cache tag is given in the figure below.

Figure 9-2 *Cache Tag Format*



If an accessed page is not a supervisor-access-only page (if the "S" bit is not set), then, if there is a cache entry such that the cache tag id and the context id match those of the accessed page, a cache hit occurs.

If the accessed page is supervisor-access-only, the context id is not checked. Because of this, the kernel mode segments should be identical in each context.

9.3. General Operation Considerations

It is possible to map a physical page to two or more distinct virtual addresses. This condition is known as aliasing. A physical address that is thus aliased could result in data from the same physical address appearing in two or more cache lines. This situation is not detected by the hardware.

There are two methods for avoiding this problem in software. The first, and easiest method is to set the "Don't Cache" bit in each Page Map Entry that aliases the physical page. The second method requires that all virtual addresses must be identical in bits A[15:12]. This is equivalent to saying that the virtual addresses of the aliased physical page must be equal, modulo 64K. If this guideline is followed, each virtual address that aliases a physical page will map to the same cache line. Alternate accesses to the aliasing virtual addresses will result in invalidation and refilling of the cache line from the same physical address. The hardware automatically invalidates a cache line when a cache miss occurs on a write operation. This insures that the cache and memory remain consistent.

Direct Virtual Memory Access (DVMA)

The SPARCengine 1E does not provide a hardware mechanism for maintaining consistency between memory and the cache when DVMA writes to memory take place. This means that, after DVMA is performed to a buffer in memory, it is possible that stale data may reside in the cache. An attempt by the SPARC IU to read from this memory will produce erroneous results.

There are two ways to avoid this problem. The first method is to set the "Don't Cache" bit in the Page Map Entry that maps the DVMA memory buffer. The alternative method requires one to flush all references to the DVMA memory buffer from the cache.

Multi-processor Cache Coherency

No hardware support exists on the SPARCengine 1E to provide cache coherency in a shared memory multi-processor environment. The mechanism is provided to un-cache pages shared among several processors in a multiprocessing configuration, however. Please refer to the chapter on Memory Management Unit for bit assignments of the Page ID bits.

9.4. Cache Programming Examples

The cache can be enabled or disabled by setting or clearing (respectively) the appropriate bit in the System Enable Register.

Enabling the Cache

```
        .seg    "data"
_enablereg:  .byte   0                  ! a software copy of the system enable register

#define ENA_CACHE 0x10                  ! enable cache bit field
#define ENABLEREG 0x40000000            ! address of system enable register in ASI_CTL space
#define ASI_CTL           0x2           ! control space

        .seg    "text"
        mov     ENA_CACHE, %o0
        mov     %psr, %o3
        or      %o3, PSR_PIL, %g1       ! spl() high to lock enable register update
        mov     %g1, %psr
        nop; nop                        ! psr delay
        set     _enablereg, %o1         ! address of software copy of system enable register
        ldub    [%o1], %g1              ! %g1 gets software copy
        set     ENABLEREG, %o2          ! hardware address of system enable register
        bset    %o0, %g1                ! turn on "enable cache" bit
        stb     %g1, [%o1]              ! update software copy
        stba    %g1, [%o2]ASI_CTL       ! enable cache
        mov     %o3, %psr               ! restore psr
        nop; nop                        ! psr delay
```

Disabling the Cache

```
        .seg    "data"
_enablereg:  .byte   0                  ! a software copy of the system enable register

#define ENA_CACHE 0x10                  ! enable cache bit field
#define ENABLEREG 0x40000000            ! address of system enable register in ASI_CTL space
#define ASI_CTL           0x2           ! control space

        .seg    "text"
        mov     ENA_CACHE, %o0
        mov     %psr, %o3
        or      %o3, PSR_PIL, %g1       ! spl() high to lock enable register update
        mov     %g1, %psr
        nop; nop                        ! psr delay
        set     _enablereg, %o1         ! address of software copy of system enable register
        ldub    [%o1], %g1              ! %g1 gets software copy
        set     ENABLEREG, %o2          ! hardware address of system enable register
        bclr    %o0, %g1                ! turn off "enable cache" bit
        stb     %g1, [%o1]              ! update software copy
        stba    %g1, [%o2]ASI_CTL       ! enable cache
        mov     %o3, %psr               ! restore psr
        nop; nop                        ! psr delay
```

Flushing

In this section, we will show how to flush a page, a segment, and a whole context from the virtual address cache. Note that, before the cache can be flushed, the cache must be first disabled. See the section on enabling and disabling the cache for more how this is done.

Page Flush

Below we demonstrate how to flush a page from the virtual address cache.

```
!
! Flush a page from the cache.
! To flush the page containing the argument virtual address from
! the cache we hold the bits that specify the page constant and
! issue a store into alternate space command for each line of
! the cache. Since the match part of the address is larger we
! have less lines to cycle through. We increment the lower bits
! of the address by VAC_LINESIZE to cycle through lines of the
! cache.
!
! vac_pageflush(v)
!       addr_t v;
!

#define VAC_LINESIZE            16      ! 16 bytes per cache line
#define VAC_PGLINES             512     ! mmu-pagesize(8192) / VAC_LINESIZE
#define PGSHIFT                 13      ! log2(mmu-pagesize)
#define NBPG                    8192    ! # bytes per mmu-page
#define ASI_FCP                 0xD     ! flush cache page

_vac_pageflush:
        set     VAC_LINESIZE, %i5
        set     VAC_PGLINES, %i1
        srl     %i0, PGSHIFT, %i0       ! mask off lo bits
        sll     %i0, PGSHIFT, %i0

        mov     NBPG/16, %l0            ! 512
        add     %l0, NBPG/16, %l1       ! 1024
        add     %l1, NBPG/16, %l2       ! ...
        add     %l2, NBPG/16, %l3
        add     %l3, NBPG/16, %l4
        add     %l4, NBPG/16, %l5
        add     %l5, NBPG/16, %l6
        add     %l6, NBPG/16, %l7
        add     %l7, NBPG/16, %o0
        add     %o0, NBPG/16, %o1
        add     %o1, NBPG/16, %o2
        add     %o2, NBPG/16, %o3
        add     %o3, NBPG/16, %o4
        add     %o4, NBPG/16, %o5
        add     %o5, NBPG/16, %i4
        sub     %l0, %i5, %i3           ! NBPG/16 - linesize
        add     %i0, %i3, %i0           ! page addr + 512 - 16
1:
        sta     %g0, [%i0]ASI_FCP
        sta     %g0, [%i0 + %l0]ASI_FCP
        sta     %g0, [%i0 + %l1]ASI_FCP
        sta     %g0, [%i0 + %l2]ASI_FCP
        sta     %g0, [%i0 + %l3]ASI_FCP
        sta     %g0, [%i0 + %l4]ASI_FCP
        sta     %g0, [%i0 + %l5]ASI_FCP
        sta     %g0, [%i0 + %l6]ASI_FCP
        sta     %g0, [%i0 + %l7]ASI_FCP
        sta     %g0, [%i0 + %o0]ASI_FCP
        sta     %g0, [%i0 + %o1]ASI_FCP
```

```
        sta     %g0, [%i0 + %o2]ASI_FCP
        sta     %g0, [%i0 + %o3]ASI_FCP
        sta     %g0, [%i0 + %o4]ASI_FCP
        sta     %g0, [%i0 + %o5]ASI_FCP
        sta     %g0, [%i0 + %i4]ASI_FCP
        subcc   %i1, 16, %i1            ! decrement loop count
        bg      1b                      ! are we done yet?
        sub     %i0, %i5, %i0           ! generate next match address
        ret
        restore
```

Segment Flush

Below we demonstrate how to flush a segment from the cache.

```
!
! Flush a segment from the cache.
! To flush the argument segment from the cache we hold the bits that
! specify the segment in the address constant and issue a store into
! alternate space command for each line of the cache by incrementing
! the lower bits of the address by VAC_LINESIZE (cache line size - 16).
!
! vac_segflush(v)
!       addr_t v;
!

#define VAC_SIZE            0x10000     ! size of the cache (64KB)
#define VAC_LINESIZE           16       ! 16 bytes per cache line
#define VAC_NLINES          0x1000      ! # of lines in the cache (4096)
#define PMGRPSHIFT             18       ! pageshift + segment shift
#define ASI_FCS                0xC      ! flush cache segment

_vac_segflush:
        set     (VAC_SIZE/16), %l0      ! cachesize / # of steps in loop
        set     VAC_LINESIZE, %i5
        set     VAC_NLINES, %i1         ! nlines to flush / # steps in loop

        srl     %i0, PMGRPSHIFT, %i0    ! mask off lo bits
        sll     %i0, PMGRPSHIFT, %i0

        !
        ! preload a bunch of offsets
        ! Avoid going through sequentially by flushing
        ! 16 lines spread evenly through the cache.
        !
        add     %i0, %l0, %i0
        sub     %i0, %i5, %i0           ! base address + cachesize - linesize
        add     %l0, %l0, %l1           ! cachesize/16*2
        add     %l1, %l0, %l2           ! cachesize/16*3
        add     %l2, %l0, %l3           ! ...
        add     %l3, %l0, %l4
        add     %l4, %l0, %l5
        add     %l5, %l0, %l6
        add     %l6, %l0, %l7
        add     %l7, %l0, %o0
        add     %o0, %l0, %o1
        add     %o1, %l0, %o2
        add     %o2, %l0, %o3
        add     %o3, %l0, %o4
        add     %o4, %l0, %o5
        add     %o5, %l0, %i4
1:
        sta     %g0, [%i0]ASI_FCS
        sta     %g0, [%i0 + %l0]ASI_FCS
        sta     %g0, [%i0 + %l1]ASI_FCS
```

```
        sta     %g0, [%i0 + %l2]ASI_FCS
        sta     %g0, [%i0 + %l3]ASI_FCS
        sta     %g0, [%i0 + %l4]ASI_FCS
        sta     %g0, [%i0 + %l5]ASI_FCS
        sta     %g0, [%i0 + %l6]ASI_FCS
        sta     %g0, [%i0 + %l7]ASI_FCS
        sta     %g0, [%i0 + %o0]ASI_FCS
        sta     %g0, [%i0 + %o1]ASI_FCS
        sta     %g0, [%i0 + %o2]ASI_FCS
        sta     %g0, [%i0 + %o3]ASI_FCS
        sta     %g0, [%i0 + %o4]ASI_FCS
        sta     %g0, [%i0 + %o5]ASI_FCS
        sta     %g0, [%i0 + %i4]ASI_FCS
        subcc   %i1, 16, %i1                    ! decrement loop count
        bg      1b                              ! are we done yet?
        sub     %i0, %i5, %i0                   ! generate next address
        ret
        restore
```

Context Flush

Below we demonstrate how to flush a context from the cache.

```
!
! Flush a context from the cache.
! To flush a context we must cycle through all lines of the
! cache issuing a store into alternate space command for each
! line whilst the context register remains constant.
! We'll start at the end and work backwards to use only
! one variable for the loop and test.
!
! void
! vac_ctxflush()
!
_vac_ctxflush:

#define VAC_SIZE                0x10000     ! size of the cache (64KB)
#define VAC_LINESIZE            16          ! 16 bytes per cache line
#define ASI_FCC                 0xE         ! flush cache segment

        set     (VAC_SIZE/16), %l0          ! cachesize / number of steps in loop
        set     VAC_LINESIZE, %i5
        !
        ! preload a bunch of offsets
        ! Avoid going through the cache sequentially by flushing
        ! 16 lines spread evenly through the cache.
        !
        sub     %l0, %i5, %i0               ! cachesize/16 - linesize
        add     %l0, %l0, %l1               ! cachesize/16*2
        add     %l1, %l0, %l2               ! cachesize/16*3
        add     %l2, %l0, %l3               ! ...
        add     %l3, %l0, %l4
        add     %l4, %l0, %l5
        add     %l5, %l0, %l6
        add     %l6, %l0, %l7
        add     %l7, %l0, %o0
        add     %o0, %l0, %o1
        add     %o1, %l0, %o2
        add     %o2, %l0, %o3
        add     %o3, %l0, %o4
        add     %o4, %l0, %o5
        add     %o5, %l0, %i4
        sta     %g0, [%i0]ASI_FCC
1:
        sta     %g0, [%i0 + %l0]ASI_FCC
```

```
sta     %g0, [%i0 + %l1]ASI_FCC
sta     %g0, [%i0 + %l2]ASI_FCC
sta     %g0, [%i0 + %l3]ASI_FCC
sta     %g0, [%i0 + %l4]ASI_FCC
sta     %g0, [%i0 + %l5]ASI_FCC
sta     %g0, [%i0 + %l6]ASI_FCC
sta     %g0, [%i0 + %l7]ASI_FCC
sta     %g0, [%i0 + %o0]ASI_FCC
sta     %g0, [%i0 + %o1]ASI_FCC
sta     %g0, [%i0 + %o2]ASI_FCC
sta     %g0, [%i0 + %o3]ASI_FCC
sta     %g0, [%i0 + %o4]ASI_FCC
sta     %g0, [%i0 + %o5]ASI_FCC
sta     %g0, [%i0 + %i4]ASI_FCC
subcc   %i0, %i5, %i0           ! generate next address
bge,a   1b                      ! are we done yet?
sta     %g0, [%i0]ASI_FCC
ret
restore
```

9.5. Cache Flush Operations

The cache flush operations provide a means of flushing lines from the cache, based on matching criteria. When a line is flushed, if the line is valid the valid bit is reset.

Cache flush operations require the following conditions:

D[31:00] must be all zeroes.

No MMU protection check is done, and no update is performed on MMU accessed of modified bits.

The SPARCengine 1E CPU card provides cache flush match criteria for contexts, pages, and segments. The following sections describe each:

Flush Cache Line based on Context Match

To flush all references to a context from the cache:

1) Store the context that you wish to flush in the context register. To do this, perform a byte store into the context register (*asi* = 0x2, A[31:28] = 0x3).

2) Issue a fullword "store into alternate space" instruction with *asi* = 0xE, and D = 0x0, to every combination of addresses between A[15:4] = 0x0 to A[15:4] = 0xFFF. This requires starting with A[15:4] = 0x0, then incrementing A[15:4], and issuing a new "store into alternate space" instruction. Repeat until A[15:4] = 0xFFF.

Flush Cache Line based on Page Match

To flush all references to a page from the cache:

1) Store the context that you wish to flush in the context register. To do this, perform a byte store into the context register (*asi* = 0x2, A[31:28] = 0x3).

2) Issue fullword "store into alternate space" instructions with *asi* = 0xD, A[31:12] = the page you want to flush, and D = 0x0, to every combination of addresses between A[11:4] = 0 to A[11:4] = 0xFF. This requires starting with A[11:4] = 0x0, incrementing A[11:4] by one, and reissuing the instruction. Repeat until A[11:4] = FF.

Flush Cache Line based on Segment Match

To flush all references to a segment from the cache:

1) Store the context that you wish to flush in the context register. To do this, perform a byte store into the context register (*asi* = 0x2, A[31:28] = 0x3).

2) Issue fullword "store into alternate space" instructions with *asi* = 0xC, D = 0x0, and A[29:18] = the segment you want to flush, to every combination of addresses between A[15:4] = 0x0 to A[15:4] = 0xFFF. This requires starting with A[15:4] = 0x0, incrementing A[15:4], and reissuing the instruction. Repeat until A[15:4] = 0xFFF.

   A[17:16] don't matter.

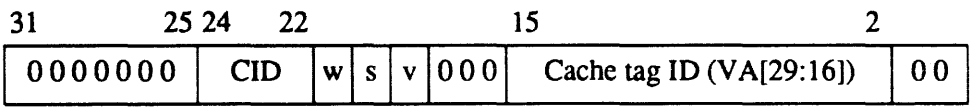
9.6. Additional Thoughts

The cache provides 4K addresses, and each of these addresses accesses a cache "line". Each line contains 4 bytes of control information, called a "cache tag", and 16 bytes of data. Each cache tag contains a field that corresponds to VA[29:16]. This is called a "cache tag ID".

When the processor issues a memory address, the cache uses bits VA[15:4] to address one of the 4K locations in its memory array. It compares bits VA[29:16] to the cache tag ID. The cache tag ID field contains VA[29:16] from a prior memory access. If they match, it is a **hit**. The cache provides the *w*, *s*, and *v* bits, and the context, plus the data byte(s) requested by the processor (1, 2, or 4 bytes, depending on whether the access was a byte, halfword, or fullword access).

If bits VA[29:16] are not identical to the corresponding bits from the cache tag ID field, it is a **miss**. The system goes to memory for the required data, and updates the entire cache line with the contents of (the corresponding locations) in memory. Note that the cache fills the line from memory and provides the byte(s) requested by the CPU simultaneously. By the time the CPU issues a subsequent request for adjacent memory byte(s), they should be already in the cache.

A cache tag has the following format:



w = 1 means write access is allowed. The *w* bit is copied from the MMU when the cache line is filled.

s = 1 means only supervisor access is allowed. The *s* bit is copied from the MMU when the cache line is filled.

v = 1 means the cache line is valid.

CID is the context.

The cache tags must be initialized by software before the cache is enabled. To do this, clear the valid bit of every cache line. Do fullword stores of 0 into (ASI = 2, A[31:28] = 0x8, A[15:4] = 0x0 to 0xFFF). Note that to initialize the entire cache, you must do the full range of accesses where A[15:4] = 0x0 to 0xFFF.

10

Diagnostics

10.1. Required Reference Material for the Diagnostics

Reference material required for a complete definition of the SPARCengine 1E on-board diagnostics:

PROM User's Guide, Sun Microsystems, Inc., 1986.
Sun Part No. 800-1736-xx

10.2. Diagnostic LED Interpretation

LEDs 0 through 4 display the number of the test that is running. LEDs 0 through 3 display the first hex digit, and LED 4 displays the least significant bit of the second digit. If a test encounters an error and enters a scopeloop, the test number remains displayed.

LED 5 is the heartbeat; after selftest, it blinks on and off to indicate that the CPU is processing instructions. If it remains either ON or OFF, the CPU is hung.

LED 6 ON indicates an exception class error.

LED7 ON indicates an error. During a typical error, LEDs 0 through 4 should also display the test number.

After selftest (but before unix boot), all LEDs should be OFF except LED 5, the heartbeat. After unix boot, the LEDs will display a "converging/diverging" pattern. Refer to the Sun ***PROM User's Manual*** for further information.

Figure 10-1 *LED Light Location (CPU Backside)*

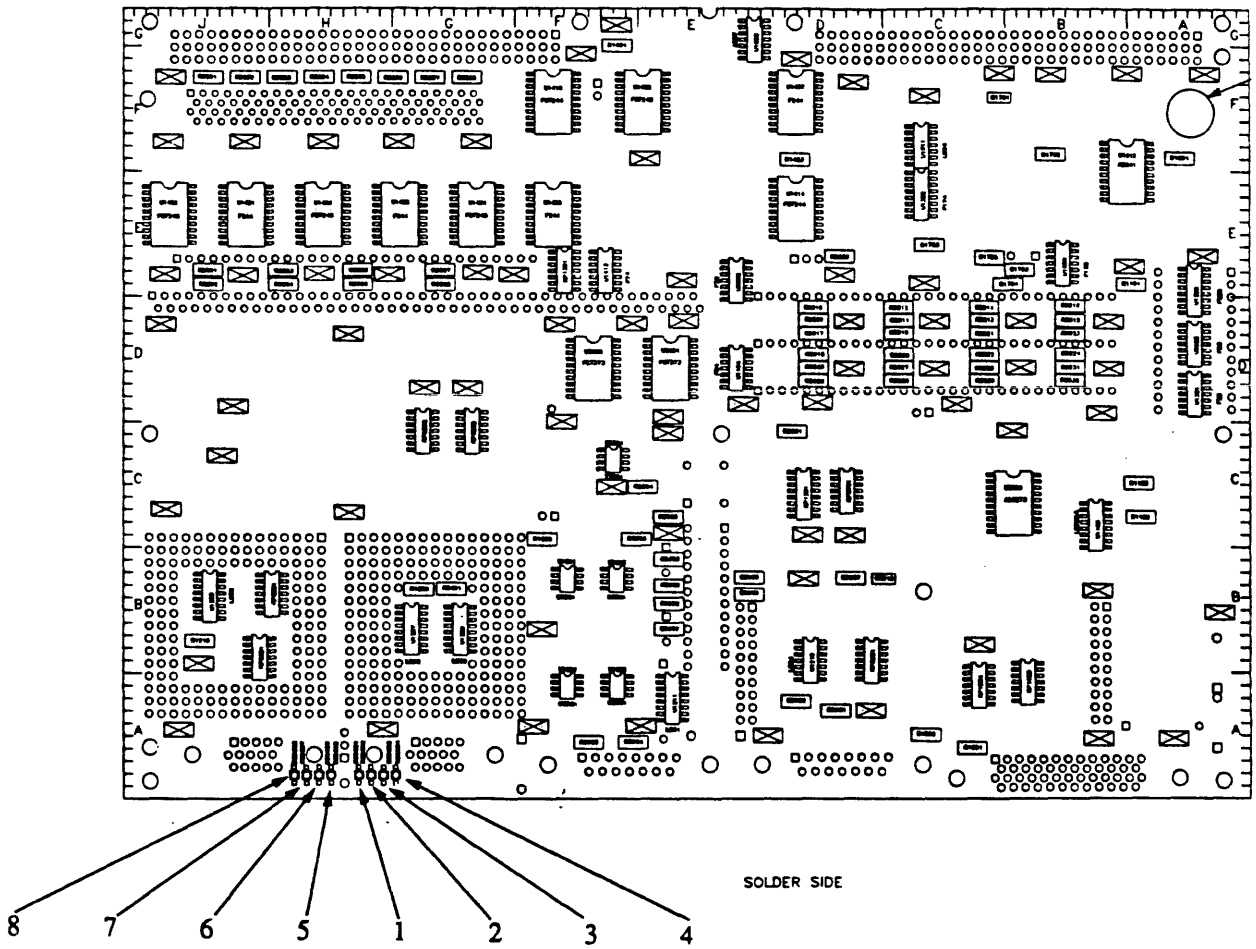


Figure 10-2 *LED Light Diagnostic Table*

LEDs ●= ON, o= OFF 0 7	***What the System is Doing When These LEDs Are Cycling***	***What Might Be Bad If This Indication Stays On And LED 6/7 Lights***
● ● ● ● ● ● ● ●	A reset sets LEDs to this state	CPU or PROMs bad or +5VDC is low
● o o o o o o o	Test 0x01 checks PROM checksum	CPU Board (Boot PROM)
o ● o o o o o o	Test 0x02 checks UDVMA enable register	CPU Board
● ● o o o o o o	Test 0x03 checks UDVMA map	CPU Board
o o ● o o o o o	Test 0x04 checks the context register	CPU Board (MMU)
● o ● o o o o o	Test 0x05, performs segment map tests	CPU Board (MMU)
o ● ● o o o o o	Test 0x06 checks page map RAM	CPU Board (MMU)
● ● ● o o o o o	Test 0x07 performs software traps test	CPU Board (IU)
o o o ● o o o o	Test 0x08 performs interrupt register test	CPU Board
● o o ● o o o o	Test 0x09 performs software interrupts test	CPU Board
o ● o ● o o o o	Test 0x0A performs TOD clock interrupt test	CPU Board
● ● o ● o o o o	Test 0x0B checks video memory	CPU Board
o o ● ● o o o o	Test 0x0C performs limited main memory tests	CPU or Memory Board
● o ● ● o o o o	Test 0x0D performs MMU read/write tests	CPU Board (MMU)
o ● ● ● o o o o	Test 0x0E performs MMU write to protected page test	CPU Board (MMU)
● ● ● ● o o o o	Test 0x0F performs MMU read invalid page tests	CPU Board (MMU)
o o o o ● o o o	Test 0x10 performs MMU write invalid page test	CPU Board (MMU)
● o o o ● o o o	Test 0x11 performs main memory timeout test	Memory Board
o ● o o ● o o o	Test 0x12 performs control space timeout test	CPU Board
● ● o o ● o o o	Test 0x13 performs range error test	CPU Board
o o ● o ● o o o	Test 0x14 performs size error test	CPU Board
● o ● o ● o o o	Test 0x15 tests ECC circuits	Memory Board
o ● ● o ● o o o	Test 0x16 tests cache tag memory	CPU Board
● ● ● o ● o o o	Test 0x17 tests cache data memory	CPU Board
o o o ● ● o o o	Test 0x18 is cache write/read hit/miss verify test	CPU Board
● o o ● ● o o o	Test 0x19 is cache write/read/flush verify test	CPU Board
o ● o ● ● o o o	Test 0x1A runs main memory tests	Memory Board
o o o o o o o ●	Self-tests have found an error	CPU or Memory Board
o o o o o o ● o	An exception class error is found	CPU or Memory Board
o o o o o ● o o	Self-tests done, UNIX in boot state or monitor quiescent (LED is blinking)	CPU or Memory Board
●⇒ o⇒ o⇒ o o⇐ o⇐ o⇐ ●	"converging/diverging" pattern	UNIX running okay

10.3. Power Up Diagnostics

Looking along the top edge of the SPARCengine 1E from the left to the right one can see the SCSI, KEYBOARD, ETHERNET, SERIAL B, and SERIAL A connectors which are all labeled accordingly on the Front Panel. These are all keyed connectors designed to prevent misinsertion.

Each connector can be hooked up as desired; to power up the board with the minimum number of cable connections, there are two options:

1. Connect SERIAL A via a cable to a terminal.

2. With a monitor and a Frame Buffer board present, connect KEYBOARD via a cable to a keyboard.

Note: When first bringing up the board, option 1 MUST be used to receive the output of the extended selftests (through serial port A). If option 2 is used then the results of the extended selftests will not be reported, and if there is a failure a scope loop will be entered and control will not be passed to the monitor and keyboard. Use of option 1 in conjunction with option 2 is recommended.

Note: When in autoboot mode a SCSI disk or ethernet should be attached. Once the SPARCengine 1E is inserted into a backplane and the cables connected, power can be applied to the SPARCengine 1E by turning on the backplane's power source.

Since the diagnostic mode is enabled at the factory, the extended selftests will immediately begin execution. See the ***Open Boot PROM Toolkit User's Manual*** for a description of how to use the extended selftests.

If everything is working correctly, then the eight LED lights located under SERIAL A will begin flashing as board begins running its diagnostics. IF no LED comes on, or if all the LEDs come on and stay on, then something is wrong.

NOTE: I/O for the extended selftests is through serial port A (ttya). Extended selftests may be disabled from the monitor by:

```
setenv diag-switch? false
```

When the extended selftests have completed the Open PROM power-up sequence commences.

11

System Reset and the Reset Switch

11.1. Five Sources for a System Reset

- Power-on Reset.
- User Reset (initiating the Reset Switch).
- Watchdog Reset.
- Software Reset.
- VMEbus Reset.

The Power-On Reset

On power-up, the SPARCengine 1E CPU Board generates a system reset on the VMEbus for a minimum of 200 ms if it is configured as the slot 1 (VMEbus system controller) CPU. All SPARCengine 1E CPUs on the VMEbus are reset in a system reset.

The User Reset

The user reset is activated by the reset toggle switch. If there are multiple SPARCengine 1E CPU cards in the backplane, toggling the slot 1 CPU will generate a VMEbus reset to reset all other non-slot 1 SPARCengine 1E CPUs.

The Reset Switch

The Reset switch is the only switch on the SPARCengine 1E CPU card. The switch is located on the front panel of the card, labeled Reset. The switch rests in a null position. When it is activated, a reset procedure is activated that is performed by the VME Gate Array. For additional information, see the chapter **VMEbus Interface** in this manual.

The Watchdog Reset

The watchdog error is the result of a doule bus error as detected by the SBus Controller on a SPARCengine 1E CPU. If this occurs, an on-board reset is generated and if it is configured as the slot 1 CPU, a VMEbus reset will be generated to reset all other non-slot 1 SPARCengine 1E CPUs.

The Software Reset

As the name implies, this reset is generated by a SPARCengine 1E CPU through a software trap which generates an on-board reset. If the board is configured as the slot 1 CPU, a VMEbus reset will be generated to reset all other non-slot 1 SPARCengine 1E CPUs.

The VMEbus Reset

The incoming VMEbus reset will generate an on-board reset to all SPARCengine 1E CPUs.

11.2. Local Reset of Non-Slot 1 CPUs

If there are multiple SPARCengine 1E CPU cards in the VMEbus, resetting a non-slot 1 CPU card results in a local reset of that CPU only, not a system reset.

12

Multiprocessing Capabilities of the SPARCengine 1E

The SPARCengine 1E is designed to run in a multiprocessor environment using the Mail Box Interrupt, local Read-Modify-Write (RMW) with a Bus Locker and VME RMW facilities. Local memory is dual-ported, accessible from the VMEbus in a 1MB window. This window should not be cached.

The following text is an introduction to the multiprocessing capabilities of the SPARCengine 1E. The details of the VMEbus are defined in the chapter entitled "VMEbus Interface".

NOTE *At this time, SunOS 4.0.3e does not support multiprocessing capabilities. The hardware capability described below is not supported by the SunOS.*

12.1. Mail Box Interrupt

The Mail Box Interrupt is used to allow other VMEbus drivers to interrupt the SPARCengine 1E by accessing in byte or word to a pre-programmed location in the A16 address space (32-bit accesses are not allowed). This location is programmed via the Mail Box Register. The SPARCengine 1E responds to this access by returning a DTACK. If the Enable bit is set, an on-board interrupt level 13 is generated to the CPU. The CPU then jumps to the Mail Box Interrupt service routine and executes the program there.

Programming the location monitor consists of setting up the comparison bits A15,A14 and A3,A2,A1 and the Mail Box enable bit in the Mail Box register. This selects one out of eight 16-bit words (or one out of sixteen 8-bit words) in one of the four 16K blocks of the address range. Only 8-bit accesses to the Mail Box Register are allowed.

Since the address mapping of the location monitor compares only address bits 15,14 and 3,2,and 1, (address bits 13 to 4 are not used), the location monitor is mirrored 1023 times in the selected 16K range. The user is advised to use addresses within address bits a13 to a4 set to zero to prevent confusion.

Due to the existing architecture of the Mail Box Interrupting scheme, only one pending interrupt is stored in the Mail Box Register at a time. Further interrupts are not recorded and will not be acknowledged until the CPU clears the interrupt pending flag by reading the Mail Box Register.

12.2. Bus Locker

A VME Bus Locker is provided to allow the SPARCengine 1E CPU to perform a non-disturbed RMW cycle to its local memory because the SBus RMW cycle is non-atomic. Before doing so, the CPU will set the Bus Locker flag in the Bus Locker Register to instruct the VME gate array to acquire the VMEbus and keep the Bus Busy signal asserted. This action will lock the VMEbus and no other VME master can gain access to the VMEbus and indirectly access the local memory of the SPARCengine 1E.

When the VMEbus has been acquired, the OWN flag in the Bus Locker Register is set. By reading this flag, the CPU can make a decision whether or not to start the RMW cycle to it's local memory. When the RMW transaction is completed, the CPU should unlock the bus by clearing the Bus Lock Flag in the Bus Locker Register.

NOTE *This process has to be performed very fast to reduce the latency of the VMEbus. Refer to section 13.12 for implementation details.*

12.3. Possible RMW Deadlock Condition Across the VMEbus

The SPARCengine 1E is capable of performing RMW cycles across the VMEbus. A deadlock might occur when there is another bus master who is already on the bus, trying to do a RMW access to the local SPARCengine 1E CPU memory. Since the SPARCengine 1E CPU cannot give up the local bus during the RMW cycle, the VMEbus master cannot finish his RMW transaction. At the same time, the SPARCengine 1E CPU cannot access the VMEbus. This deadlock situation results in a time-out bus error on the SPARCengine 1E CPU. In order to recover from the bus error, a re-run of the RMW must be done in software.

12.4. Procedure for Enabling Multiprocessor Operation

The SPARCengine 1E is capable of running as a VMEbus slot one board or a non-slot one board. To setup multiple SPARCengine 1E's in the same VME card cage it is necessary to properly set the board jumpers and setup the Open PROM so that VME addressing errors and "probes" to the VME bus from different SPARCengine 1E's don't overlap. Refer to the Boot PROM section of the manual for more details on the forth monitor in the prom.

The following steps are required to run multiple SPARCengine 1E CPUs in the same VME backplane:

1. The SPARCengine 1E placed in slot 1 must have it's VME slot 1 jumper installed.

2. The SPARCengine 1E's placed in other slots except 1 must have their VME slot 1 jumper removed.

3. If the VMEbus backplane has a special Sun P2 Bus for ECC memory boards, then make sure the extra SPARCengine 1E CPU cards are not placed in these slots. The P2 Bus slots can only be used by the slot 1 SPARCengine 1E CPU.

4. Each SPARCengine 1E CPU should be set to "not autoboot". This is accomplished by setting the "autoboot" parameter in the boot prom to false. The procedure for setting this parameter is as follows: If in unix, fasthalt the system to get to the boot prom prompt ">" then press "n" to get to the forth system. Enter the command "SETENV AUTO-BOOT? FALSE". This disables the autoboot

feature.

This procedure must be done for each SPARCengine 1E CPU in the system.

5. Each SPARCengine 1E CPU should be set to a different slavemap. In any event no cpu can have it's VME-slavemap set to 0. There are 16 VME-slavemaps 0 thru 15. You can set the first cpu to 1, the second to 2, etc. While in FORTH (as in step 4 above) enter the following command "1 VME-slavemap!" for the first SPARCengine 1E CPU, then go to the second SPARCengine 1E CPU and enter "2 VME-slavemap!", etc. When all cpu's have been setup, enter "boot" to boot the default boot device on each cpu.

13

VMEbus Interface

13.1. Required Reference Material for VME

Reference material required for a complete definition of the VMEbus Interface:

VMEbus Specification Revision C.1, October, 1985. Also known as: IEC 821 BUS and IEEE P1014/D1.2.

SBus Developer's Kit, Sun Part No. 825-1219-xx

NOTE: Within the ***SBus Developer's Kit***, there is a book entitled ***Open PROM Toolkit User's Guide***. The additional notes in the chapter of this manual entitled **Boot PROM** are required to make it applicable to the SPARCengine 1E.

13.2. Features of the SPARCengine 1E VMEbus Interface

- A32/A24/A16 Master (Sun-4 VME Device) and A24/A32 Slave DVMA device as SBus DMA device.
- Single-level and round-robin arbitration with bus arbiter timer.
- VME Interrupt handler with IACK Daisy Chain Driver.
- Clock driver.
- System resetter.
- Fast bus arbitration by supporting early release enabling bus arbitration concurrent with Data accesses.
- Multiprocessing support with Mail Box Interrupts, programmable slave base address ranges, true Read Modify Write to local memory, and to shared VME resource, with the use of the VME Bus Locker.
- Full 4GByte VME addressing with mapping register.
- Simple user-setup provided, only one jumper is required (slot one). All other options are selected through register settings, stored in EEPROM.
- A special loopback cycle is provided to enable stand-alone testing of the interface.

The heart of the VME Interface is the S4-VME Chip, a 7000 gate CMOS LMA9K 1.5 u Gate Array. It includes all VME logic. Address and Data paths are external, using 29FCT521, 29FCT52, 29821 and 29FCT821 registers. Decoding of VME address A(31:24) is done externally using AS27 gates. Decoding of Physical address PA(19:16) is done externally using an F20. All VMEbus outputs are driven through bipolar drivers using AS641, LS04 and F125.

13.3. VMEbus Basics -- An Introduction

The VMEbus is an asynchronous 32 bit bus, with multi master, multi level arbitration specifying Single Cycles, Block Mode Cycles and Read-Modify-Write Cycles. See next section regarding current revision and publisher.

Certain conceptual logical units are defined in the table below.

Table 13-1 *VME Concept Definitions*

Concept	Definition
MASTER	accesses SLAVEs.
SLAVE	responds with [DTACK/BERR] to MASTERs.
BUS REQUESTER	Requests and keeps the bus on order from a MASTER.
BUS ARBITERS	Grants bus to REQUESTERs through [BG] daisy chain. Must be in slot 1.
INTERRUPTER	Interrupts bus, responds to Interrupt handlers with [DTACK/BERR] and supplies an Interrupt Vector.
IACK DAISY CHAIN Drive	Drives [IACKIN/IACKOUT] chain when [IACK] and [DS(1:0)] are asserted. Must be in slot 1.
SYS CLOCK DRIVER	Drives [SYSCLK]. Must be in slot 1.
POWER MONITOR	Asserts [SYSRESET].

13.4. VME Performance

The timing for the VMEbus interface is as follows:

8 Bit Transfers 1.6 MB per second.
16 Bit Transfers 3.2 MB per second.
32 Bit Transfers 6.4 MB per second.

The VME Gate Array logic meets the Sbus and VMEbus specifications at a 20 MHz clock, correlating to a 50 ns clock cycle.

Bus arbitration for other VMEbus masters takes place faster by using the Early Release feature.

NOTE *These numbers are given only as a way to describe the SPARCengine 1E CPU performance. The overall bandwidth depends on the other card's response time, the application code, amount of other tasks running concurrently, other concurrent DMA activity, cache hit/miss ratio, etc. These figures should NOT be expected in a real world system.*

13.5. VME Addresses

Three address spaces are defined. Address Modifier bits [AM(5:4)] indicates which:

Table 13-2 *VME Addresses*

Concept	Definition	Address Modifier
A16	16 address bits are needed. Can address 64 KB.	[AM(5:4)] = 10
A24	24 address bits are needed. Can address 16 MB.	[AM(5:4)] = 11
A32	32 address bits are needed. Can address 4 GB.	[AM(5:4)] = 00

Address modifier bits [AM(2:0)] indicates what privilege and type of access: Supervisor, Non-Privileged, Program Data or Block Transfer access. Data transfers can be D32/D16/D8 (i.e., quad-byte, double-byte or single-byte size. Unaligned transfer means that a quad-byte or a double-byte transfer takes place on a non-even boundary. Unaligned transfers are not supported by the SPARC-engine 1E CPU card.

13.6. VME Implementation

The VME Interface adheres to the VMEbus Specification Rev C.1. A later version of the specification, released by ANSI, the ANSI/IEEE 1014-1987, has changed some optional features to mandatory. The new rules are: 2.61-68 and 4.49. The SPARCengine 1E VME Interface follows all these new rules except for 2.67 requiring ALL D32 slaves to include Unaligned Transfer capability.

- Address spaces:

Table 13-3 *VME Address Spaces*

Addresses	Comments
A16	Master accesses are possible.
A24	Master accesses are possible.
A32	Master accesses are possible.
D16	Master accesses are possible.
D32	Master accesses are possible.
A16	Mailbox cycles are monitored, and can generate the interrupt.
D8 (only)	Mailbox cycles are monitored, and can generate the interrupt.
A24	Slave cycles are supported.
A32	Slave cycles are supported.
D8	Slave cycles are supported.
D16	Slave cycles are supported.
D32	Slave cycles are supported.

- No VME Bus Timer is provided as specified in VME. Own Master cycles times out after 300 us generating a Bus Error on VMEbus.
- Single-level VMEbus arbitration (SGL) and round-robin arbitration (RRS) are supported.
- Read-Modify-Write (RMW) accesses to local and VME memory are supported.
- Unaligned transfers are not implemented nor supported. Unaligned accesses get a Bus error returned. Unaligned transfers are: 32bit accesses not to address(1:0)=00, 16bit accesses not to a(0)=0. See ANSI/IEEE change described above.
- Address only cycles are not decoded, an active data strobe is required for the start of any DVMA access.
- The VME Interface allow Address pipe-lining during Slave accesses, but do not utilize the function on VME Master cycles.
- Early release of bus is possible, i.e., other masters can arbitrate getting bus mastership while VME Master access is finishing (i.e. Address Strobe still asserted). The VME Interface is also prepared for other cards early release of the bus, by awaiting the negation of the Address Strobe.
- The VME Interface requests VME mastership on bus level 3 as an ROR (Release On Request).
- Bus Clear is not monitored.
- An Interrupt handler that can serve any of the seven interrupts is available. A selection of which levels to serve is done by programming a SW register. The Interrupt handler is a D08(O), handling only 8 bit interrupt vectors.
- No Interrupter function is provided. Signaling between CPU cards in multiprocessor configurations is done through the Mail Box Interrupt.
- An IACK Daisy Chain function is provided when the card is in slot one.
- A System Clock Driver is active, if the card is in slot one.
- No Serial Clock Driver is implemented.
- An on card Voltage Supervisor asserts SYSRESET for a minimum of 200 ms after +5V DC has reached 4.5 Volts, if the slot 1 jumper is inserted. The ACFAIL and SYSFAIL is not driven or monitored. Incoming SYSRESET generates an on-card reset.

13.7. VME Registers -- Major Groups

The registers can be divided into the following groups:

1. Registers set on system initialization, replacing card jumpers:

   Interrupt Enable Register.
   Interrupts handled.
   Arbitration mode.

2. Registers used for multi-processing or co-processing:

   Mail Box Interrupt Register.

   Slave Map Register (also replacing jumper).

3. Others:

   A32 VME Address mapping.
   Move 512 MB window.
   Enable diagnostic loop back.

   Slave Map register bit0.
   Enable Block Mode transfers.

Table 13-4 *VME Registers*

Type	Name	Physical Base	Size
1	VME Bus Locker	0xEFE00000	byte
1	VME IACK cycle	0xEFE00001	byte A[3:1], A0=1
1	Mail Box Register	0xEFE00010	byte
1	VME Interrupt Enable Register	0xEFE00014	byte
1	A32MAP Register	0xEFE00018	byte
1	Slave Map Register	0xEFE0001C	byte

13.8. Master Interface

A full 32 bit Master Interface is provided. Complete mapping of 512 MB is possible all the time, but this 512 MB window can be moved through all 4 GB through a VME Mapping register.

A32, A24, A16 Address Modes, D32, D16, D8(EO) Data Sizes are supported.

On Master cycles, RMW, i.e. atomic load-store cycles are implemented according to the VME spec. The VME data strobe asserts two times as the VME address strobe is hold asserted. This differs from other Sun cards, where instead the ownership of VME is kept while a read and a write takes place. The problem with this is that if the access is to a card with several devices competing for an on-card bus, there is no way this card can recognize the access as a RMW. For example.: a disk card with multiple on card devices would not be able to prevent a shared resource to be stolen in between what looks like a standard Read and Write even though the VMEbus is locked.

Block mode transfers are not supported. (They are only supported by the Slave interface.)

Unaligned transfers (UAT) are not supported. Accesses of this kind, i.e. 16-bit accesses to base addr. + 1 or addr. + 3, or 32 bit accesses to base addr. + 1, base addr. + 2 or base addr. + 3 is not recognized, resulting in a Bus error time out. 32 bit accesses to type 2 space, the 16 bit VME port is not recognized, resulting in a Bus error time out.

```
 * VME interrupt. Do vectoring.
 * The VME vector is read off the VME bus. If this fails (data fault),
 * _trap will check the PC of the fault and jump to _spurious.
 *
 * The vmelevel# code sets up %g1 to contain the address of the
 * appropriate VME IACK register before it branches here.
 */
        .global _vme_read_vector

#define VEC_MAX         64
#define VEC_MIN         255


_vme_read_vector:
        ldub    [%g1], %l6              ! read vector #, acknowledge int.
        cmp     %l6, VEC_MAX            ! check vector limits
        bg      spurious_vme
        subcc   %l6, VEC_MIN, %g3 ! normalize vector
        bl      spurious_vme
        sll     %g3, 2, %g5             ! scale for interrupt counting
        sll     %g3, 3, %g3             ! scale vector
        set     _intrcnt, %g4           ! per-device interrupt counts table
        ld      [%g5 + %g4], %g1        ! interrupt count
        set     _vme_vector, %g2        ! table of interrupt vectors
        inc     %g1                     ! count interrupt
        st      %g1, [%g5 + %g4]        ! and store result
        ld      [%g2 + %g3], %g1        ! get handler address
        add     %g2, 4, %g2             ! generate address of arg ptr
        ld      [%g2 + %g3], %o0        ! delay slot, get arg ptr
        call    %g1                     ! call handler
        ld      [%o0], %o0              ! read arg ptr

        b,a     int_rtt                 ! restore previous stack pointer
/* end _vme_read_vector */


/*
 * Vme vectors are compatible with the sun3 family in which
 * there were possible valid vectors from 64 to 255 inclusive.
 * This requires 192 vectors, each vector is two words long
 * the first word being the interrupt routine address and the
 * second word is the arg.
 *
 * Vectors 0xC8-0xFF (200-255) are reserved for customer use.
 */

/*
 * The vme vectoring uses the following table of routines
 * and arguments. The values for the vectors in the following
 * table are loaded by autoconf at boot time.
 */
#define ERRV .word _spurious, 0

        .seg    "data"
        .align  4

        .global _vme_vector
_vme_vector:                    ! vector numbers
        ERRV; ERRV; ERRV; ERRV  ! 0x40 - 0x43  sc0 | sc?
        ERRV; ERRV; ERRV; ERRV  ! 0x44 - 0x47  xdc0 | xdc1 | xdc2 | xdc3
        ERRV; ERRV; ERRV; ERRV  ! 0x48 - 0x4B  xyc0 | xyc1 | xyc?
        ERRV; ERRV; ERRV; ERRV  ! 0x4C - 0x4F  future disk controllers
        ERRV; ERRV; ERRV; ERRV  ! 0x50 - 0x53  future disk controllers
        ERRV; ERRV; ERRV; ERRV  ! 0x54 - 0x57  future disk controllers
```

```
ERRV; ERRV; ERRV; ERRV  ! 0x58 - 0x5B  future disk controllers
ERRV; ERRV; ERRV; ERRV  ! 0x5C - 0x5F  future disk controllers
ERRV; ERRV; ERRV; ERRV  ! 0x60 - 0x63  tm0 | tm1 | tm?
ERRV; ERRV; ERRV; ERRV  ! 0x64 - 0x67  xtc0 | xtc1 | xtc?
ERRV; ERRV; ERRV; ERRV  ! 0x68 - 0x6B  future tape controllers
ERRV; ERRV; ERRV; ERRV  ! 0x6C - 0x6F  future tape controllers
ERRV; ERRV; ERRV; ERRV  ! 0x70 - 0x73  ec?
ERRV; ERRV; ERRV; ERRV  ! 0x74 - 0x77  ie0 | ie1 | ie?
ERRV; ERRV; ERRV; ERRV  ! 0x78 - 0x7B  future ethernet devices
ERRV; ERRV; ERRV; ERRV  ! 0x7C - 0x7F  future ethernet devices
ERRV; ERRV; ERRV; ERRV  ! 0x80 - 0x83  vpc0 | vpc1 | vpc?
ERRV; ERRV; ERRV; ERRV  ! 0x84 - 0x87  vp?
ERRV; ERRV; ERRV; ERRV  ! 0x88 - 0x8B  mti0 | mti1 | mti2 | mti3
ERRV; ERRV; ERRV; ERRV  ! 0x8C - 0x8F  SunLink SCP (Systech DCP-8804)
ERRV; ERRV; ERRV; ERRV  ! 0x90 - 0x93  Sun-3 zs0 (8 even vectors)
ERRV; ERRV; ERRV; ERRV  ! 0x94 - 0x97  Sun-3 zs1 (8 odd vectors)
ERRV; ERRV; ERRV; ERRV  ! 0x98 - 0x9B  Sun-3 zs0 (8 even vectors)
ERRV; ERRV; ERRV; ERRV  ! 0x9C - 0x9F  Sun-3 zs1 (8 odd vectors)
ERRV; ERRV; ERRV; ERRV  ! 0xA0 - 0xA3  future serial
ERRV; ERRV; ERRV; ERRV  ! 0xA4 - 0xA7  pc0 | pc1 | pc2 | pc3
ERRV; ERRV; ERRV; ERRV  ! 0xA8 - 0xAB  cg2 | future frame buffers
ERRV; ERRV; ERRV; ERRV  ! 0xAC - 0xAF  gp1 | future graphics processors
ERRV; ERRV; ERRV; ERRV  ! 0xB0 - 0xB3  sky0 | ?
ERRV; ERRV; ERRV; ERRV  ! 0xB4 - 0xB7  SunLink / channel attach
ERRV; ERRV; ERRV; ERRV  ! 0xB8 - 0xBB  (token bus) tbi0 | tbi1 | ?
ERRV; ERRV; ERRV; ERRV  ! 0xBC - 0xBF  Reserved for Sun
ERRV; ERRV; ERRV; ERRV  ! 0xC0 - 0xC3  Reserved for Sun
ERRV; ERRV; ERRV; ERRV  ! 0xC4 - 0xC7  Reserved for Sun
ERRV; ERRV; ERRV; ERRV  ! 0xC8 - 0xCB  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xCC - 0xCF  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xD0 - 0xD3  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xD4 - 0xD7  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xD8 - 0xDB  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xDC - 0xDF  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xE0 - 0xE3  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xE4 - 0xE7  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xE8 - 0xEB  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xEC - 0xEF  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xF0 - 0xF3  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xF4 - 0xF7  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xF8 - 0xFB  Reserved for User
ERRV; ERRV; ERRV; ERRV  ! 0xFC - 0xFF  Reserved for User
```

13.10. Slave Interface

A32, A24 address mode.

1 MB of memory is accessible, normally the lowest MB on VME, but this can be changed to be in the 1-16 MB range to support multi-CPU card implementations. The address range is incremented in 1 MB steps.

Unaligned accesses to the Slave Interface range returns a Bus Error back to the external VME Master.

The address modifiers on VME are used to define Address mode (A32/A24/A16), privilege, and data/program/block-transfer access. The Slave decoder and Mail Box Interrupter monitors all address modifier bits with the exception of AM2, the access privilege bit. All slave accesses as a DVMA device is set to supervisor mode, which is in accordance with the Sun-4 Architecture.

The 1 MB VME address space selected is always mapped to the highest megabyte in the Virtual Address Space, again in accordance with the Sun-4 Architecture.

Single Transfers

A bit in the S4-CACHE SYSTEM Enable Register can disable SDVMA and therefore all Slave cycles. If this bit is set, single slave cycle can take place.

Slave Map Register

The address space for System DVMA accesses is 1 MB, normally the lowest Megabyte in accordance with the Sun-4 Architecture, but bits 27:24 can be used to re-map the SDVMA address space. Slave Block Mode transfers can also be enabled by setting bit 31. This lowest 1 MB on VME is mapped to the highest 1 MB within the Virtual Address space. During register accesses, only 8-bit accesses should be used. 16- or 32-bit accesses are not acknowledged, resulting in a bus error time out.

Slave Map Register Initialization

The register is reset to all 0's on resets. Thus, the mapping defaults to 0-1 MB and block mode transfers is disabled.

Table 13-8 *Slave Map Register Address*

Type	Device Address	Device	Physical Space
1	0xEFE0001C	Slave Map Register	1 byte

Table 13-9 *Slave Map Register Address Bits*

31	30	29	28	27	26	25	24
0	0	0	0	VME Slave Address base			

Table 13-10 *Slave Master Register Bit Definitions*

Bit	Range	Value	Range Start	Range End
31	This bit is not used, and must be 0.			
30:28	This bit is not used, and is read back as 0.			
27:24	0x0 Maps System DVMA Space to:	0x0000000-0x000FFFFF	0 MB	1 MB
27:24	0x1 Maps System DVMA Space to:	0x0010000-0x001FFFFF	1 MB	2 MB
27:24	0x2 Maps System DVMA Space to:	0x0020000-0x002FFFFF	2 GB	3 GB
27:24	0x3 Maps System DVMA Space to:	0x0030000-0x003FFFFF	3 GB	4 GB
27:24	0x4 Maps System DVMA Space to:	0x0040000-0x004FFFFF	4 GB	5 GB
27:24	0x5 Maps System DVMA Space to:	0x0050000-0x005FFFFF	5 GB	6 GB
27:24	0x6 Maps System DVMA Space to:	0x0060000-0x006FFFFF	6 GB	7 GB
27:24	0x7 Maps System DVMA Space to:	0x0070000-0x007FFFFF	7 GB	8 GB
27:24	0x8 Maps System DVMA Space to:	0x0080000-0x008FFFFF	8 GB	9 GB
27:24	0x9 Maps System DVMA Space to:	0x0090000-0x009FFFFF	9 GB	10 GB
27:24	0xA Maps System DVMA Space to:	0x00A0000-0x00AFFFFF	10 GB	11 GB
27:24	0xB Maps System DVMA Space to:	0x00B0000-0x00BFFFFF	11 GB	12 GB
27:24	0xC Maps System DVMA Space to:	0x00C0000-0x00CFFFFF	12 GB	13 GB
27:24	0xD Maps System DVMA Space to:	0x00D0000-0x00DFFFFF	13 GB	14 GB
27:24	0xE Maps System DVMA Space to:	0x00E0000-0x00EFFFFF	14 GB	15 GB
27:24	0xF Maps System DVMA Space to:	0x00F0000-0x00FFFFFF	15 GB	16 GB

13.11. Mail Box

A Mail Box interrupt function is provided to allow other devices to interrupt the SPARCengine 1E. This mail box detects accesses to the A16 address space to a location programmed in the mail box register. No real memory is provided at this location, but the mail box responds with a VME DTACK, acknowledging the access and generate an on card interrupt Level 13 if the Enable bit is set. VME Address bits A(15:14, 3:1) are in the Mail Box Register. This corresponds to programming the location to one of the eight 16-bit words in one of the four 16K blocks of the address range. Any VME A16 address space, 8-bit or 16-bit read or write to the location programmed into the Mail Box Register generates an on-card interrupt on level 13 if enabled. 32-bit wide access is not allowed and is not acknowledged, resulting in a Time out Bus Error for the other accessing VME Master.

Since the address mapping of the location monitor compares only the address bits A15, A14 and A3, A2, A1 (Bits A13 up to A4 are NOT compared at all), the location monitor address is mirrored 1023 times in the selected 16K range. The user is advised to use addresses with address bits A13 to A4 set to zero.

Mail Box Register Base Location

This register is used to program the VME address to be monitored and, if enabled, mailbox interrupts the IU via an on-card interrupt. Only 8-bit accesses to the register should be used, 16- or 32-bit accesses are not acknowledged, resulting in a bus error time out. For mail box cycles, the VME bus is only monitored for A16, D8 or D16 accesses.

Mail Box Register Initialization

All bits are initialized to 0's.

Table 13-11 *Mail Box Register Address*

Type	Device Address	Device	Physical Space
1	0xEFE00010	Mail Box Register	1 byte

Table 13-12 *Mail Box Register Address Bits*

31	30	29	28	27	26	25	24
I-fig	En I	0	Comp Address (15:14)		Comp Address (3:1)		

Mail Box Register Interrupt Level

Level 13.

Table 13-13 *Mail Box Register Bit Definitions*

Bit	Range	Value	Range Start	Range End
31	This bit set indicates that a mailbox interrupt is pending. A read of this register resets the interrupt on the trailing edge of the read pulse. A write to this bit is ignored.			
30	This bit set enables the mail box interrupt.			
29	This bit is ignored, and is read back as 0.			
28	Compared with VME Address Bit 15.			
27	Compared with VME Address Bit 14.			
26	Compared with VME Address Bit 3.			
25	Compared with VME Address Bit 2.			
24	Compared with VME Address Bit 1.			

13.12. Bus Locker

A Bus locker function is provided to enable the CPU to do an atomic Read-Modify-Write (RMW) to its on-card memory without being interrupted by an incoming RMW from the another VME Master. This function is only to be used when the card is running together with one or more external CPU cards (multiprocessing). In these cases, message passing is done through shared memory locations that need to be accessed in a defined way.

VME Bus Locker Register

In a non-multiprocessing application, this register should be left alone. When used, bit 24 of the register should ALWAYS be SET for each update, to guarantee a consistent bus arbitration behavior.

Table 13-14 *Bus Locker Register Address Bits*

31	30	29	28	27	26	25	24
Owned	0	0	0	0	0	Req Bus	Use locker

Initialization

All bits are initialized to 0's. Thus, the VME bus locker is inactive and does not affect the card's bus arbitration.

Table 13-15 *Bus Locker Register Bit Definitions*

Bit	Range	Value	Range Start	Range End
31	This bit indicates the the VMEbus is owned. No external VME card can access memory. Bit is read only, a write to this bit is ignored.			
30	This bit is ignored, and is read back as 0.			
29	This bit is ignored, and is read back as 0.			
28	This bit is ignored, and is read back as 0.			
27	This bit is ignored, and is read back as 0.			
26	This bit is ignored, and is read back as 0.			
25	This bit initiates a VMEbus request. Once owned, it is held.			
24	This bit enables the Bus request function. It should only be set for multiprocessing applications. If set, it should never be changed (unless system is restarted).			

In multiprocessing applications, update the register at the system initialization time by writing 0x01 to the register. The procedure for accessing an on-card memory location is:

1. Disable all interrupts. (VME IACK cycles need to acquire control of VME.)
2. Request VMEbus locking by writing 0x3 to the register.
3. Read register and check bit 31 to until VMEbus is owned and locked.
4. Do Read-Modify-Write (RMW) (atomic Load-Store) to on-card memory.
5. Unlock VME by writing 0x1 to register.
6. Enable all interrupts.

NOTE *If the VME is not unlocked in time, other accesses might time out causing a system crash. Always unlock VME ASAP after the RMW is completed!*

NOTE *VME interrupts must be disabled during the bus locking procedure to prevent potential deadlock, since VME IACK cycles requires access to VME. Interrupts could otherwise cause a deadlock situation if it happens while the bus is locked.*

13.13. Interrupt Handler

The interrupt handler can selectively support all interrupt levels. These are enabled through the VME Interrupt/Bus Arbiter register. Note that this register is a replacement for a jumper and should normally be fixed depending on the VME Interrupt handler configuration. Run-time enabling of the interrupts are normally done with the Interrupt Enable register, see the System Specification.

Interrupt Enable/Bus Arbiter Mode Register

The Interrupt Enable Register can selectively enable all VME interrupt levels.

This Register enables VME interrupts to be fed to the Interrupt logic in the S-4 MMU chip. Only 8-bit accesses should be used, 16- or 32-bit accesses are not acknowledged, resulting in a bus error time out.

Interrupt Enable Register Initialization

Bit set 31:25 is set and bit 24 reset. Thus, VME interrupts are enabled and the arbiter mode is SGL, single level arbitration.

NOTE *This register replaces on-card jumpers. Changes to this register should only be done at system initialization time, and should not be changed at a later time unless the system is restarted.*

Table 13-16 *Interrupt Enable Register Address*

Type	Device Address	Device	Physical Space
1	0xEFE00014	VME Interrupt Enable Register	1 byte

Table 13-17 *Interrupt Enable Register Address Bits*

31	30	29	28	27	26	25	24
En 7	En 6	En 5	En 4	En 3	En 2	En 1	Round-Robin Arbitration

Table 13-18 *Interrupt Enable Register Bit Definitions*

Bit	Range	Value	Range Start	Range End
31	This bit enables VME Interrupt 7/Sun-4 level 13 to Internal Logic.			
30	This bit enables VME Interrupt 6/Sun-4 level 11 to Internal Logic.			
29	This bit enables VME Interrupt 5/Sun-4 level 9 to Internal Logic.			
28	This bit enables VME Interrupt 4/Sun-4 level 8 to Internal Logic.			
27	This bit enables VME Interrupt 3/Sun-4 level 5 to Internal Logic.			
26	This bit enables VME Interrupt 2/Sun-4 level 3 to Internal Logic.			
25	This bit enables VME Interrupt 1/Sun-4 level 2 to Internal Logic.			
24	This bit enables Round-Robin Arbitration.			

NOTE Enabling the bus arbiter function is done via a jumper (Slot 1 jumper).

13.14. Bus Requester

The Bus requester is an ROR requester. It releases the bus when other VME Masters/Requesters requests the bus. The SPARCengine 1E can only request on Bus Request level 3.

Bus Arbiter

A Single Level Arbiter (level 3) and a Round-Robin Arbiter are provided. Note that the arbiter is enabled only if the card resides in slot 1. (Slot1 jumper.)

In arbitration mode, locked arbitration is detected and released by the arbiter after 3.3 ms. The Bus arbiter then drives BBSY active for the minimum period required by the VMEbus spec. This is to prevent requesters that in the mean time have decided to withdraw their requests to be able to do so. Note that a withdrawal of a VME Request normally is in violation of the VME Specification. (Rule 3.11).

13.15. Bus Time Out Period

There are two time out parameters in a VME cycle in addition to the On Board Bus arbitration timer: Rerun Time Out and Abort.

Rerun Time Out

If the time from the Master cycle starts until the VME slave responds with a DTACK exceeds 1.6 us, the cycle is rerun by the IU so high-priority devices can get on the Sbus. In the mean time, the Master cycle and all output signals is frozen, ignoring any DTACK coming in. When the cycle is rerun, it is reconnected again.

Abort

Two stages of the access exists. If the bus is not owned, it has to be acquired. Then the actual VMEbus cycle can start. A Master cycle times out after 300 us. That means, the response time from another VME slave including the time for acquiring the bus must be less than 300 us.

If the VMEbus is not owned when an abort occurs, the bus requester keeps the VMEbus Request asserted even though the CPU cycles has been aborted. It then asserts a dummy Bus Busy (BBSY) when the Bus Grant is received. This is to comply with VME rule 3.11.

If the bus is owned when the abort takes place, a Bus Error is generated. This Bus Error would normally be generated by an accessed VME slave. This is to comply with VME rule 2.48.

13.16. System Reset

On power up, the CPU card generates SYSRESET for a minimum of 200 ms, if the Slot1 jumper indicates the card resides in slot 1. Incoming VMEbus resets always generate an on-card reset.

13.17. Jumper

Only one jumper is used for the VME Interface. If jumpered, the SPARCengine 1E operates as the VME system arbiter and slot 1 card.

Table 13-19 *Slot 1 Jumper & Functions*

Function	"Slot 1" Position	"Not in Slot 1"
VME System Clock	Enabled.	Tri-state.
VME IACK Daisy-chain Driver	Enabled.	IACKIN drives IACKOUT.
SYSRESET	Generates VME SYSRESET.	Does not generate VME SYSRESET.

13.18. Programmable Register Settings

The following registers include settings otherwise often provided by jumpers.

Table 13-20 *Programmable Register Settings*

Register	Setting Definition
Slave Map Register	Defines memory address space of the 1 MB on-card VME Slave (System DVMA).
VME Interrupt Handler Register	Defines which interrupt levels to handle. Defines what type of VME Bus arbiter (only if the card resides in slot 1).
A32 Map Register	Defines where to map the CPU Master Address Space of 512 MB.

13.19. VME IACK Cycles

When the SPARCengine 1E is set up to respond to Vectored VME interrupt requests, it drives the IACK line to acquire the interrupt vector from interrupting devices though software routines. A device space operation is used by the Interrupt handler to generate an Interrupt Acknowledge cycle. The operation is done only by doing a Device Space byte read at location 0xEFE0000X with Address bits 3 to1 set to thc VMEbus interrupt level being acknowledged. A 16- or 32-bit access is not acknowledged and results in a bus error time-out. By doing this, an acknowledge process is started by first acquiring the bus, then by driving the IACK signal at the same time as with other control lines such as AS*, DS*, A1-3, etc.

NOTE *An access to these seven addresses are not to an on chip register, but instead to execute a VME Interrupt Acknowledge cycle, acquiring the Interrupt Vector from the interrupting device.*

Table 13-21 *Mail Box Register Address*

Interrupt Type	Device Address	Device	Physical Space
1	0xEFE00000	VME Int. Vector	1 A[3:1]=VME Int. level. A0=1, Read, only 8-bit access should be used.

Table 13-22 *VME IACK Cycles Register Address Bits*

31	30	29	28	27	26	25	24
8-bit Interrupt Vector from interrupting VME Device Level A(3:1)							

This translates into the following interrupt responses:

Table 13-23 *VME IACK Cycles Interrupt Responses*

Acknowledging VME Interrupt	Device Address
1	0xEFE00003
2	0xEFE00005
3	0xEFE00007
4	0xEFE00009
5	0xEFE0000B
6	0xEFE0000D
7	0xEFE0000F

Daisy Chain IACK Driver

Since the SPARCenginc 1E does not have the capability to interrupt the VMEbus through the VME interrupt request lines, it does not provide interrupt vectors at all. Thus, it drives the daisy chain IACKOUT when an IACKIN is active. When set up as a slot 1 card, it drives the IACKOUT as soon as the non-daisy chain IACK signal is active.

Master Cycles

Ideal VME slave accessed (30 ns DS to DTACK response). Bus already owned. No lingering DTACK or AS.

A Master cycle is 9 cycles, corresponding to a theoretical 9 MB/s bandwidth for back-to-back cycles.

Slave Cycles

Ideal VME master (0 ns DTACK to DS response). Fast Sbus grant (two clock periods).

Table 13-24 *Slave Cycles Duration and Theoretical Bandwidth*

Cycle	Duration (ns)	Theoretical Bandwidth
Single Read	560-630 ns	6.9 to 7.9 MB
Single Write	510-580 ns	6.3 to 7.2 MB

13.20. Bus Arbitration

Table 13-25 *Bus Arbitration*

Arbiter	Conditions
Single Level Arbiter	No MVE master owns the bus, Bus Locker disabled. VME Bus Request to VME Bus Grant: 2 Cycles. No MVE master owns the bus, Bus Locker enabled. VME Bus Request to VME Bus Grant: 3 Cycles.
Round-Robin Arbiter	No MVE master owns the bus, Bus Locker disabled. VME Bus Request to VME Bus Grant: 2-5 Cycles. No MVE master owns the bus, Bus Locker enabled. VME Bus Request to VME Bus Grant: 3-6 Cycles.

13.21. Interrupts

The following table defines the interrupt sources and their levels:

Table 13-26 *VME Interrupt Levels and Sources*

Level	Source	Additional Source
15	Memory: Parity Error	ECC Uncorrectable Error
14	Counter/Timer 1	
13	VME Level 7	VME Mailbox interrupt
12	Serial Ports	
11	VME Level 6	
10	Counter/Timer 0	
9	VME Level 5	SBUS 7
8	VME Level 4	SBUS 6
7	P2 Bus interrupt (ECC Correctable Error)	SBUS 5 (Video)
6	Ethernet	SW IRQ6
5	VME Level 3	SBUS 4
4	SCSI	SW IRQ4
3	VME Level 2	SBUS 3
2	VME Level 1	SBUS 2
1	SW IRQ1	SBUS 1

All type 1 devices defined in the architecture, including the serial ports, use auto-vectored interrupts.

The interrupt register in device space enables all interrupts, causes software interrupts, and enables specified interrupts. It is described in the chapter *Device Space*. The memory error registers in system space, provide information about any memory errors that occur. They are described in the chapter *System Space*.

13.22. VME Programming Examples

Two code examples are presented below, one written in FORTH, and the other written in C. These examples provide an indication to you on how to program for the VMEbus.

Additional reference material required for a complete definition of the FORTH and C words used in the following code examples: ***SBUS Developer's Kit***, Sun Part No. 825-1219-xx, (Sun SBUS Information).

FORTH Programming Example

The FORTH example below shows how to map a page to obtain a virtual address that allows access to the VME-bus. The example shows how to map the slave port (what other masters read/write to obtain access to your local memory) and how to map the master port (what you use to read/write the VME-bus).

The routines map-master and map-slave translate a slave window (1 of 16 one megabyte windows that can be mapped to the VME-bus) into a virtual address that accesses the VME-bus. The routines use subroutines map-segments and

map-pages. Map-segments has 3 arguments, segment map entry, virtual address, and length. It stores a new segment map entry for the virtual address supplied. Map-pages has 4 arguments, physical address, space type, virtual address, and length. It maps conse- cutive pages to map a region of size length.

```
constants
800000 constant vmed32-mem                    (VMED32 memory virtual address)
100000 constant 1mb                           (1 megabyte constant)
0fe000 constant window-size                   (size of window monitored by 4e-slave)
fff00000 constant sdvma                       (SDVMA virtual address)
0 constant local-mem                          (local memory address)

: map-master ( vme-slave-window -- )
  20 vmed32-mem window-size map-segments      (map in segments for vmed32-mem)
  ( vme-slave-window ) 1mb *
  vmed32a32 ( space type) vmed32-mem window-size map-pages
;

: map-slave ( vme-slave-window -- )
  vme-slavemap!                               (window for SDVMA)
  f4 sdvma window-size map-segments           (map in segments for SDVMA)
  local-mem obmem ( space type) sdvma window-size map-pages
;

: map-m8s2 ( -- )                             (map master to global slave8 and select slave2)
  8 map-master
  2 map-slave
;

: map-m2s8 ( -- )                             (map master to global slave2 and select slave8)
  2 map-master
  8 map-slave
;

800000 l@ .                                   (read VME-bus location 0 and print results)
```

C Programming Example

The C example below shows how to open a VME space and obtain a virtual pointer so that the VME space can be read/written to.

In the Unix environment several options are available to the programmer. The VME-bus supports 16, 24, and 32 bit addressing, along with 8, 16, and 32 bit data transfers. In /dev several special files are provided that allow for the different configurations. For example vme24d32 provides a 24 bit address with 32 bit data transfers. These special files can be opened and seeks performed. (See MEM in section 4S of the manual pages) However it is usually more efficient to use pointers. A pointer to an opened special VME file can be obtained using mmap().(See MMAP in section 2 of the manual pages)

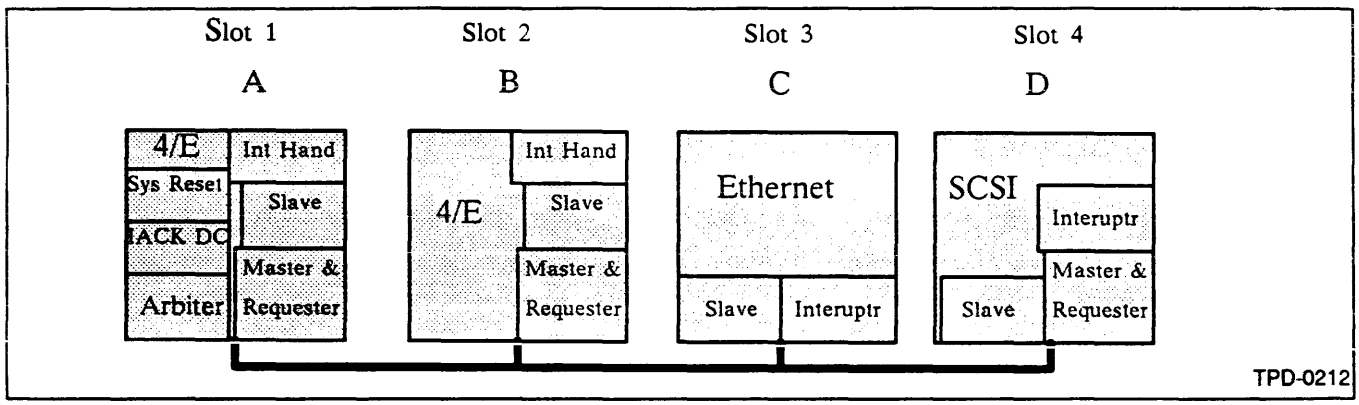
```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/mman.h>

main(argn, argv)
int argn;
char **argv;
{
int     fd, i, len, off;
off_t   addr;
caddr_t ptr;
unsigned short *p;
unsigned short pat;

        if(argn < 3) {
                printf("usage: %s addr len [pat]0, argv[0]);
                return;
        }
        pat = 0;
        addr = (int)strtol(argv[1], (char **)NULL, 0);
        off = (addr & 0x7fff) >> 1;             /* convert offset to a short */
        addr &= 0xffff8000;                     /* use a 32k window */
        len = (int)strtol(argv[2], (char **)NULL, 0);
        if(argn == 4) pat = (short)strtol(argv[3], (char **)NULL, 0);
        if(pat == 0) pat = 0xaaaa;
        printf("addr = 0x%x0, addr);
        printf("off  = 0x%x0, off*sizeof(pat));
        printf("len  = %d0, len);
        printf("pat  = 0x%x0, pat);

                                                /* Now open VME space as address 24 data 16 */
        if((fd = open("/dev/vme24d16",O_RDWR)) == -1) {
                printf("error opening vme bus0);
                return;
        }

                                                /* Use mmap to obtain a virtual pointer to VME space */
        if((ptr = mmap(0, len, (PROT_READ|PROT_WRITE),
                                MAP_SHARED, fd, addr)) == (caddr_t)-1) {
                printf("mmap failed0);
                return;
        }

                                                /* convert pointer to unsigned short for 16 bit read/writes */
        p = (unsigned short *)ptr;

        for(i=0; i<len/2; i++) *(p+i+off) = pat; /* initialize memory to pat */
```

```
        while(p) {
          for(i=0; i<len/2; i++) {
              if(*(p+i+off) != pat) {          /* read and compare pat */
                printf("fail location %x read %x exp %x0, p+i+off, *(p+i+off), pat);
                p = 0;                         /* break out of loop */
                break;
              }
          }
        }
}
```

13.23. Example of a VME System

Below is a diagram of a sample VME system with two SPARCengine 1E cards, a third-party Ethernet card, and a third-party SCSI card. The text below describes how this system configuration should be defined and implemented. This example is generic, and is provided only as a guide for understanding the SPARCengine 1E VMEbus.

Figure 13-1 *Sample VME System Diagram*



Board A in slot 1 is a 4/E with the slot 1 functions enabled (ARBITER,IACK DAISY chain driver, SYS CLOCK Driver). It is an INTERRUPT HANDLER for level 1-3. Board B is also a 4/E with the slot 1 functions disabled. It is an INTERRUPT HANDLER for level 4-7. Board C is an Ethernet card with a SLAVE and an INTERRUPTER. Board D is a SCSI card with DMA capabilities. It has a MASTER, a SLAVE and an INTERRUPTER.

Board A needs to start a disk transfer by writing to command registers in card D. But first he has to acquire the bus. His MASTER orders the on card REQUESTER to get the bus. The REQUESTER asserts [BR]. If no one owns the bus, indicated by an inactive [BBSY], the arbiter grants the bus with [BG]. Board A's REQUESTER receives this, assert [BBSY] and allow his MASTER to start a bus cycle.

If another card 's REQUESTER was requesting the bus at the same time with a [BR], card A would still get it since the bus grant from the arbiter in slot 1 is passed from card to card via a Daisy Chain [BGIN,BGOUT] and card A is the first card in the Chain.

Board A's MASTER asserts addresses, data and then strobe [AS] and [DS(1:0)].

Next, card D's SLAVE detects that the cycle is aimed at him and respond with a [DTACK] once the data has been captured (the write to the command register).

The SCSI card starts the disk access and when he has filled his buffer with data, activate his MASTER. The MASTER requests the bus via it's on-card REQUESTER. The SPARCengine 1E requester is of Release On Request type, so he keeps the bus with [BBSY] until someone else asks for it. When Board D's REQUESTER requests the bus, BOARD A's REQUESTER drops [BBSY]. Now the ARBITER can grant the bus to card D with a [BG].

BOARD D acquires the bus asserting [BBSY] and the MASTER can start a DMA transfer to card A's on card memory via A's SLAVE. the SLAVE responds with a DTACK for every cycle. Once the whole transfer is complete, card D asserts interrupt 3 via his INTERRUPTER signaling he's finished.

Board A's INTERRUPT HANDLER is activated and does an Interrupt Acknowledge cycle. This cycle is almost identical to a MASTER read, but instead of reading data, the interrupt vector is read. [IACK] is asserted to signal an IACK cycles, and address bits 1-3 indicates what level the IACK cycle is responding to.

[IACK] from any interrupt handler is passed to slot1, where the back plane directly connects it to the start of the [IACKIN/IACKOUT] daisy-chain. This is done to enable INTERRUPT HANDLERS to be spread out on different cards if needed. The IACK Daisy Chain Driver in slot 1 drives the chain. Board B is not an INTERRUPTER and therefore always passes [IACKIN] to [IACKOUT].

Board C passes the incoming [IACKIN] to its [IACKOUT], unless he is interrupting on the same level at the same time. If so, he does not pass the [IACKIN] signal, but instead respond to the IACK cycle with a DTACK and his vector, different from card D's. In the mean time, card D would not see any [IACKIN] and therefore just keep his interrupt asserted until card C passes it on.

Board D's INTERRUPTER responds to an IACK cycle with the corresponding bits set to the level he is interrupting on. Board A jumps to the disk card interrupt routine, with the understanding that the disk transfer has finished and now is ready and stored in the DVMA buffer. The transfer is complete.

13.24. Sample VMEbus Interface Driver

The following code is a listing of the current VMEbus driver in SunOS. It is being provided as a developmental aide, not as a substitute for your own development.

```
/*
 * Copyright (c) 1989 by Sun Microsystems, Inc.
 */

/*
 * VME special file
 */

#include <sys/param.h>
#include <sys/user.h>
#include <sys/buf.h>
#include <sys/systm.h>
#include <sys/vm.h>
#include <sys/uio.h>
#include <sys/mman.h>
#include <vm/seg.h>

#include <machine/pte.h>
#include <machine/mmu.h>
```

```
#include <machine/cpu.h>
#include <machine/eeprom.h>
#include <machine/seg_kmem.h>

#define M_VME16D16      5       /* /dev/vme16d16 - VME 16bit addr/16bit data */
#define M_VME24D16      6       /* /dev/vme24d16 - VME 24bit addr/16bit data */
#define M_VME32D16      7       /* /dev/vme32d16 - VME 32bit addr/16bit data */
#define M_VME16D32      8       /* /dev/vme16d32 - VME 16bit addr/32bit data */
#define M_VME24D32      9       /* /dev/vme24d32 - VME 24bit addr/32bit data */
#define M_VME32D32      10      /* /dev/vme32d32 - VME 32bit addr/32bit data */

/* transfer sizes */
#define SHORT       1
#define LONG 2

/*
 * Check bus type memory spaces for accessibility on this machine
 */
mmopen(dev)
        dev_t dev;
{
        switch (minor(dev)) {
        case M_VME16D16:
        case M_VME24D16:
        case M_VME32D16:
        case M_VME16D32:
        case M_VME24D32:
        case M_VME32D32:
                break;
        default:
                return EINVAL;          /* Unsupported or unknown type */
        }
        return 0;
}       /* end of mmopen */

mmread(dev, uio)
        dev_t dev;
        struct uio *uio;
{

        return (mmrw(dev, uio, UIO_READ));
}

mmwrite(dev, uio)
        dev_t dev;
        struct uio *uio;
{

        return (mmrw(dev, uio, UIO_WRITE));
}

mmrw(dev, uio, rw)
        dev_t dev;
        struct uio *uio;
        enum uio_rw rw;
{
        register int            o,
                                xfersize;
        register u_int          c,
                                v;
        register struct iovec   *iov;
        int                     error = 0;
        int                     pgsp;
        struct memlist          *pmem;
```

```
while (uio->uio_resid > 0 && error == 0) {
        iov = uio->uio_iov;
        if (iov->iov_len == 0) {
                uio->uio_iov++;
                uio->uio_iovcnt--;
                if (uio->uio_iovcnt < 0)
                        panic("mmrw");
                continue;
        }
        switch (minor(dev)) {

        case M_VME16D16:
                if (uio->uio_offset >= VME16_SIZE)
                        return EFAULT;
                v = uio->uio_offset + VME16_BASE;
                pgsp = PGT_VME_D16;
                xfersize = SHORT;
                goto vme;

        case M_VME16D32:
                if (uio->uio_offset >= VME16_SIZE)
                        return EFAULT;
                v = uio->uio_offset + VME16_BASE;
                pgsp = PGT_VME_D32;
                xfersize = LONG;
                goto vme;

        case M_VME24D16:
                if (uio->uio_offset >= VME24_SIZE)
                        return EFAULT;
                v = uio->uio_offset + VME24_BASE;
                pgsp = PGT_VME_D16;
                xfersize = SHORT;
                goto vme;

        case M_VME24D32:
                if (uio->uio_offset >= VME24_SIZE)
                        return EFAULT;
                v = uio->uio_offset + VME24_BASE;
                pgsp = PGT_VME_D32;
                xfersize = LONG;
                goto vme;

        case M_VME32D16:
                pgsp = PGT_VME_D16;
                v = uio->uio_offset;
                xfersize = SHORT;
                goto vme;

        case M_VME32D32:
                pgsp = PGT_VME_D32;
                v = uio->uio_offset;
                xfersize = LONG;
                /* FALL THROUGH */

        vme:
                v = btop(v);

                /* Mapin for VME, no cache operation is involved. */
                segkmem_mapin(&kseg, vmmap, MMU_PAGESIZE,
                  (u_int)(rw == UIO_READ ? PROT_READ :
                  PROT_READ | PROT_WRITE), pgsp | v, 0);
                o = (int)uio->uio_offset & PGOFSET;
                c = min((u_int)(NBPG - o), (u_int)iov->iov_len);
```

```
                        error = mmpeekio(uio, rw, &vmmap[o], (int)c, xfersize);
                        segkmem_mapout(&kseg, vmmap, MMU_PAGESIZE);
                        break;

                default:
                        panic("mmrw: invalid minor(dev)0);
                }
        }
        return error;
}

static
mmpeekio(uio, rw, addr, len, xfersize)
        struct uio *uio;
        enum uio_rw rw;
        caddr_t addr;
        int len;
        int xfersize;
{
        register int c;
        int o;
        short sh;
        long lsh;

        while (len > 0) {
                if ((len|(int)addr) & 1) {
                        c = sizeof (char);
                        if (rw == UIO_WRITE) {
                                if ((o = uwritec(uio)) == -1)
                                        return (EFAULT);
                                if (pokec(addr, (char)o))
                                        return (EFAULT);
                        } else {
                                if ((o = peekc(addr)) == -1)
                                        return (EFAULT);
                                if (ureadc((char)o, uio))
                                        return (EFAULT);
                        }
                } else {
                        if ((xfersize == LONG) &&
                          (((int)addr % sizeof (long)) == 0) &&
                          (len % sizeof (long)) == 0) {
                                c = sizeof (long);
                                if (rw == UIO_READ) {
                                        if (peekl((long *)addr, (long *)&o) == -1)
                                                return (EFAULT);
                                        lsh = o;
                                }
                                if (uiomove((caddr_t)&lsh, c, rw, uio))
                                        return (EFAULT);
                                if (rw == UIO_WRITE) {
                                        if (pokel((long *)addr, lsh))
                                                return (EFAULT);
                                }
                                break;
                        } else {
                                c = sizeof (short);
                                if (rw == UIO_READ) {
                                        if ((o = peek((short *)addr)) == -1)
                                                return (EFAULT);
                                        sh = o;
                                }
                                if (uiomove((caddr_t)&sh, c, rw, uio))
                                        return (EFAULT);
```

```
                        if (rw == UIO_WRITE) {
                                if (poke((short *)addr, sh))
                                        return (EFAULT);
                        }
                        break;
                }
        }
        addr += c;
        len -= c;
    }
    return 0;
}

/*ARGSUSED*/
mmmmap(dev, off, prot)
    dev_t dev;
    off_t off;
{
    int                 pf;
    struct pte          pte;
    register struct memlist *pmem;

    switch (minor(dev)) {

    case M_VME16D16:
        if (off >= VME16_SIZE)
            break;
        return (PGT_VME_D16 | btop(off + VME16_BASE));

    case M_VME16D32:
        if (off >= VME16_SIZE)
            break;
        return (PGT_VME_D32 | btop(off + VME16_BASE));

    case M_VME24D16:
        if (off >= VME24_SIZE)
            break;
        return (PGT_VME_D16 | btop(off + VME24_BASE));

    case M_VME24D32:
        if (off >= VME24_SIZE)
            break;
        return (PGT_VME_D32 | btop(off + VME24_BASE));

    case M_VME32D16:
        return (PGT_VME_D16 | btop(off));

    case M_VME32D32:
        return (PGT_VME_D32 | btop(off));
    }
    return -1;
}
```

14

SCSI Interface

14.1. Required Reference Material for the SCSI Interface

Reference material required for a complete definition of the SCSI interface:

SCSA (SUN Common SCSI Architecture)
Sun Part Number: 800-4701-10 (Rev A of 15 Nov 1989)

SCSI Implementation Guide SUN Common SCSI Architecture
Sun Part Number: 800-4700-10 (Rev A of 15 Nov 1989)

NCR 53C90 Enhanced SCSI Processor Data Sheet, (SCSI Registers Definition), NCR Corporation, 11/87.

SCSI, Understanding the Small Computer System Interface
NCR Corporation, Prentice-Hall, 1990

ANSI Specification X3.133, X3.131_1986

Federal Information Processing Standard (FIPS), X3.131_1986

14.2. SCSI Performance

The NCR 54C90 Extended SCSI Processor timing is as follows:

Synchronous Mode Speed:

Absolute Maximum:	5.0 MB per second
Typical:	2.5 MB per second

Asynchronous Mode Speed:

Absolute Maximum:	3.0 MB per second
Typical:	1.75 MB per second

14.3. SCSI Addresses

The SCSI registers are accessed via byte loads and stores to the following physical addresses:

Table 14-1 *SCSI Register Addresses*

Address	Description
0xF0800000	Transfer Count Low
0xF0800004	Transfer Count High
0xF0800008	FIFO Data
0xF080000C	Command
0xF0800010	Status/Bus ID
0xF0800014	Interrupt/Status Timeout
0xF0800018	Sequential Step/Synchronization Transfer Period
0xF080001C	FIFO Flags/Synchronization Offset
0xF0800020	Configuration
0xF0800024	Clock Conversion Factor (write only)
0xF0800028	Chip Test (Extended SCSI Processor test use only)
0xF080002C	Chip (Extended SCSI Processor) Configuration-2

NOTE ***Byte accesses must be performed even though the addresses are all fullword-aligned.***

14.4. Interface Programming

Since the SCSI controller uses the DMA controller to perform the actual transfer of data to and from memory, the two devices must be programmed together. One possible algorithm is as follows:

```
scsi_start()
{
        /*start an operation on the SCSI*/
        lock data pages into contiguous virtual memory;
        DMA_address_register = starting virtual address;
        setup SCSI registers (except for "go");
        DMA_control_status_register = (EN__DMA|INT_EN|(other bits));
        start SCSI;
        /*The SCSI will interrupt us when it is done.*/
}

scsi_interrupt()
{
        /*must drain DMA on a read from disk/write to memory*/
        if (last operation==READ){
                        DMA_control_status_register=(DRAIN);
        }
}
```

For a detailed description of the SCSI registers, see th ***NCR 53C90 Data Sheet.***

14.5. SCSI Connector Pinout List

Table 14-2 *SCSI Pinout List -- Connector J1001*

Signal	Pin #	Comments
GND	1	Ground
GND	2	Ground
GND	3	Ground
GND	4	Ground
GND	5	Ground
GND	6	Ground
GND	7	Ground
GND	8	Ground
GND	9	Ground
GND	10	Ground
GND	11	Ground
NC	12	No Connect
NC	13	No Connect
NC	14	No Connect
GND	15	Ground
GND	16	Ground
GND	17	Ground
GND	18	Ground
GND	19	Ground
GND	20	Ground
GND	21	Ground
GND	22	Ground
GND	23	Ground
GND	24	Ground
GND	25	Ground
SD0-	26	SCSI Data0-
SD1-	27	SCSI Data1-
SD2-	28	SCSI Data2-
SD3-	29	SCSI Data3-
SD4-	30	SCSI Data4-
SD5-	31	SCSI Data5-
SD6-	32	SCSI Data6-
SD7-	33	SCSI Data7-
SDP-	34	SCSI Parity-
GND	35	Ground
GND	36	Ground
NC	37	No Connect
TRMPWR	38	Terminator Power (+5 Volts DC, Fused, 3 Amps)
NC	39	No Connect
GND	40	Ground
ATN-	41	Attention-
NC	42	No Connect
BSY-	43	Busy-
ACK-	44	Acknowledge-
RST-	45	Reset-
MSG-	46	Message-
SEL-	47	Select-
CD-	48	Command/Data-
REQ-	49	Request-
IO-	50	Input/Output-

15

Ethernet Interface

15.1. Required Reference Material for the Ethernet Interface

Reference material required for a complete definition of the Ethernet interface:

Am7990 Local Area Network Controller (LANCE) Technical Manual, Advanced Micro Devices, Inc., 1986.

Am7992B Serial Interface Adapter (SIA) Technical Manual, Advanced Micro Devices, October, Inc., 1985.

ANSI Specification 802.3, 1986, also known as:
ISO/Draft International Standard 8802/3, February 1989.
Federal Information Processing Standard (FIPS) 107, February 1989.

15.2. Introduction

The SPARCengine 1E card uses the AMD AM7990 Ethernet Controller chip to perform Ethernet processing.

15.3. Ethernet Interface Definition

The Ethernet controller is the AMD AM7990 LANCE chip. The data is translated for the Ethernet by the AM7992 Serial Interface Adapter (SIA). The LANCE connects directly to the S4-DMA, over unique Ethernet interface signals. The S4-DMA will provide all necessary buffering and arbitration functions to allow the Ethernet chip to access main memory. The S4-DMA Ethernet interface contains a 1 word (32bit) pack/unpack register with consistency control logic. Consistency control ensures that all data written by the Ethernet chip gets to main memory in a deterministic manner.

The LANCE uses multiplexed address and data, so the S4-DMA demultiplexes them internally. The address supported by the LANCE is 24 bits; as per the SPARCengine 1E specification the upper 8 bits of the 32 bit virtual address are driven to 0xFF. The LANCE has a 16 bit data path which can accommodate 8 or 16 bit accesses, by the use of byte masking capabilities.

15.4. Ethernet Performance

Maximum Transfer Rate:	10 Mbits per second
Maximum Observed Rate:	7.5 Mbits per second

Ethernet Addresses

The Ethernet Controller chip is in slot 0 address space at 0xF0C000000. The following table shows the address and location of its internal registers:

Table 15-1 *Ethernet Registers*

Address	Description
0xF0C00000	Register Data Port (RDP)
0xF0C00002	Register Address Port (RAP)

For additional information, see the *AMD Am7990 Data Sheet.*

15.5. Ethernet Interrupt

Level 6

15.6. Ethernet Bandwidth and Capability

SPARCengine 1E can transfer data from the Ethernet into the on-card buffer memory at a maximum(estimated) rate of 10 megabits/sec.

15.7. Ethernet Transmits and Receives

When it is in transmit mode, the AM7990 LANCE transfers data from the current buffer in LANCE to a register called a Silo. The output of the Silo is serialized and then goes to the Serial Interface Adapter(AM7992). The output of the Serial Interface Adapter is connected to the Ethernet lines through a transformer.

In the receive mode, the Serial Interface Adapter(AM7992) receives data from the Ethernet cable. It transfers the data to the Silo in the LANCE. The LANCE transfers the data from the silo to the correct buffer in local memory.

15.8. Ethernet Buffer

The LANCE Controller utilizes up to 128 Kilobytes of memory for buffers. These buffers are usually allocated from the SPARCengine 1E's main memory.

15.9. Ethernet Connector Pinout List

Table 15-2 *Ethernet Pinout List -- Connector J0901*

Signal	Pin #	Comments
NC	1	No Connect
COL+	2	Collision+
TxD+	3	Transmit Data+
NC	4	No Connect
RxD+	5	Receive Data+
GND	6	Ground
NC	7	No Connect
NC	8	No Connect
COL-	9	Collision-
TxD-	10	Transmit Data-
NC	11	No Connect
RxD-	12	Receive Data-
+12V	13	+12 Volts DC, Fused, 3 Amps
NC	14	No Connect
NC	15	No Connect

15.10. Ethernet Interface Programming

```
/* Initializing on-card ethernet chip                                    */
 *Abstract:        Performs Lance hardware dependent variable *
 *                 initialization for Sun Ethernet diagnostic.  *
 *Algorithm:       Initialize CSR 3 (describe bus acquisition   *
 *                 procedures to the emulator).                 *
 *                 Initialize global variables and i/o data     *
 *                 structures.                                  *
 *Routines called: ini_print, ini_read                          *
 *                                                              *
 ****************************************************************************
 */

long le_sys_init(argc,argv)
        int             argc ;
        char            *argv[] ;
{

        extern int      byteswap() ;

        extern short    DSPL_ON;
        extern short    err_dsp;
        extern char     home_adr[6];
        extern short    INTR_ON ;
        extern etraddr  myaddr ;
        extern char     net_data [];
        extern short int ringsize,bufsize;
        extern int      totbuf;
        u_short *rdpp  ;                /* ptr to reg data port */
        u_short *rapp  ;                /* ptr to reg adr port */

        long    mrc ;
        long    rc ;
        extern int slot;
        int     i;
```

```
rc = PASS ;

                                        /* init etherbase_va array */
for (i=0; i<= NUM_SLOTS; i++)
etherbase_va[i] = 0;

ether.etherbase_size = (int)ROUNDUP(sizeof(LE_BLOCK),PAGESZ);
etherbase_va[0] = ether.dvma_start ;

                                        /* Initializing on-card ethernet chip */
                                        /* set remaining chip/diagnostic ether mem variables */
ether.txbuf_n = ONE << TLENMAX ;
ether.rxbuf_n = ONE << RLENMAX ;
ether.txbuf_size = 1024 ;               /* may need to tweek this */
ether.rxbuf_size = 1024 ;               /* may need to tweek this */

                                        /* map on-card ethernet control block & buffers here */
                                        /* find the first available block for control block */
for(ether.etherbase_va=etherbase_va[0],mrc=FAIL;
PASSED(rc) && FAILED(mrc);)
{
        if(FAILED(mrc = execmap(&ether.etherbase_va,
                        &ether.etherdum_pa,
                        ether.etherbase_size,
                        ether.ether_flags)))
{
                                        /* increment va by a page */
ether.etherbase_va += PAGESZ;
                                        /* will we go beyond dvma space ? */
if(((u_long)(ether.etherbase_va+ether.etherbase_size))
        >= ((u_long)(ether.dvma_start+ether.dvma_size)) )
{
rc = mrc ;
exec_log(ERROR,ALL_DST,
        "le_sys_init(le%d): can not map on-card etherbase{%d}@0x%x0,
                testing_LE,rc,ether.etherbase_va) ;
}
}
}                                       /* for */
etherbase_va[0] = ether.etherbase_va;
ether.ethermin_va = ether.etherbase_va + ether.etherbase_size;

rdpp = (u_short *)(ethernet_va[0]) ;    /* ptr to reg data port */
rapp = (u_short *)(ethernet_va[0] + 2) ;        /* ptr to reg adr port */
*rapp = 0;                              /* Select CSR 0 */
                                        /* set the stop bit in csr0 to inactivate the lance chip */
*rdpp = (*rdpp) | 0x0004;

for (i=1; i<=slots[0]; i++)
{                                       /* Initializing 2nd ethernet chip. */
                                        /* map 2nd ethernet control block & buffers here */
                                        /* find the first available block for control block */
for(ether.etherbase_va=etherbase_va[i-1]+ether.etherbase_size,mrc=FAIL;
        PASSED(rc) && FAILED(mrc);)
{
        if(FAILED(mrc = execmap(&ether.etherbase_va,
                        &ether.etherdum_pa,
                        ether.etherbase_size,
                        ether.ether_flags)))
{
                                        /* increment va by a page */
ether.etherbase_va += PAGESZ;
                                        /* will we go beyond dvma space ? */
```

```
if(((u_long)(ether.etherbase_va+ether.etherbase_size))
        >= ((u_long)(ether.dvma_start+ether.dvma_size)) )
{
        rc = mrc ;
        exec_log(ERROR,ALL_DST,
                "le_sys_init(le%d): can not map 2nd etherbase{%d}@0x%x0,
        testing_LE,rc,ether.etherbase_va) ;
}
}
}                                                   /* for */
etherbase_va[i] = ether.etherbase_va;
ether.ethermin_va += ether.etherbase_size;
rdpp = (u_short *)(ethernet_va[i]) ;            /* ptr to reg data port */
rapp = (u_short *)(ethernet_va[i] + 2) ;        /* ptr to reg adr port */
*rapp = 0;                                  /*Select CSR0 */
                                            /* set the stop bit in csr0 to inactivate the lance chip */
*rdpp = (*rdpp) | 0x0004;                   /* set stop bit in csr0 */

} /* for */

totbuf = BUFSPC ;
bufsize = 1024;
ringsize = TLENMAX;

                                            /* copy ethernet address from myaddr to home_adr */
bcopy(myaddr,home_adr,sizeof(etraddr)) ;    /* 1 to 1 copy */

net_data[0] = ' '; net_data[1] = ' '; net_data[2] = ' ';
net_data[3] = ' '; net_data[4] = ' '; net_data[5] = ' ';
net_data[6] = ' '; net_data[7] = ' '; net_data[8] = ' ';
net_data[9] = ' '; net_data[10] = 'T'; net_data[11] = 'h';
net_data[12] = 'i'; net_data[13] = 's'; net_data[14] = ' ';
net_data[15] = 'i'; net_data[16] = 's'; net_data[17] = ' ';
net_data[18] = 'a'; net_data[19] = ' '; net_data[20] = 't';
net_data[21] = 'e'; net_data[22] = 's'; net_data[23] = 't';
net_data[24] = ' '; net_data[25] = 'o'; net_data[26] = 'f';
net_data[27] = ' '; net_data[28] = 't'; net_data[29] = 'h';
net_data[30] = 'e'; net_data[31] = ' '; net_data[32] = 'L';
net_data[33] = 'A'; net_data[34] = 'N'; net_data[35] = 'C';
net_data[36] = 'E'; net_data[37] = ' '; net_data[38] = 'E';
net_data[39] = 'm'; net_data[40] = 'u'; net_data[41] = 'l';
net_data[42] = 'a'; net_data[43] = 't'; net_data[44] = 'o';
net_data[45] = 'r';

err_dsp = SET;                              /* error display flag */
DSPL_ON = 0;                                /* By default, no debug msgs */
INTR_ON = 0 ;

go_reset();

return(rc) ;
}                                       /* le_sys_init */
```

16

SBus Interface

16.1. Required Reference Material for SBus

Reference material required for a complete definition of the SBus:

SBus Developer's Kit, Sun Part No. 825-1219-xx.

The SPARCengine 1E Color & Monochrome Video Cards User's Manual.

16.2. Introduction

The SBus is a non-proprietary I/O expansion bus developed by Sun Microsystems for some new Sun systems and board products. The bus lives in type 1 device space.

16.3. SBus Slot Addresses

Address bits PA[27:24] select one of two SBus slots, and bits PA[23:0] are available for addressing devices on the SBus card.

Bits PA[27:24] select SBus slots as follows:

Table 16-1 *SBus Slot Addresses*

PA[27:24]	Slot	Address Range
00	SBus slot 0	0xF0000000 to 0xF3FFFFFF
01	SBus slot 1	0xF4000000 to 0xF7FFFFFF

SBus slot 0 is not a physical slot; it is occupied by devices residing on the SPARCengine 1E CPU card.

SBus slot 1 on the SPARCengine 1E is a physical slot, where cards such as the SBus Video Cards can be utilized.

The SBus device addresses are described in the following sections.

16.4. SBus Slot 0 Devices

The SBus Slot 0 devices are physically on the CPU card. SBus slot 0 is in type 1 space at a base address of 0xF0000000. The base address contains the card ID number, DMA registers, SCSI registers, and Ethernet registers. These are located at the following addresses:

Table 16-2 *SBus Slot 0 Addresses*

Offset	Description
0xF0000000	Card ID (0xFE810101)
0xF0400000	**DMA Registers:** Control/Status Address Byte Count Diagnostic
0xF0800000	**SCSI Registers:** Transfer Count Register FIFO Register Command Register Status Register Select/Reselect Bus ID Register Sequence Step Synchronous Transfer Period Register FIFO Flags Register Synchronous Offset Register Configuration Register Clock Conversion Register Test Register
0xF0C00000	**Ethernet Registers:** Data Port Address Port

NOTE *The slot 0 space from 0xF2000000 to 0xF3FFFFFF is not used.*

Card ID

The card ID (data 0xFE810101 at location 0xF0000000) is a line of Forth code that identifies the card in SBus slot 0 to the operating system (in this case, the CPU card identifies itself).

DMA Registers

The DMA register space contains four registers, addressed by fullword loads and stores to the following addresses:

Table 16-3 *DMA Register Addresses*

Address	Description
0xF0400000	DMA Control/Status Register
0xF0400004	DMA Address Register
0xF0400008	DMA Byte Count Register
0xF040000C	DMA Diagnostic Register

DMA Control/Status Register

The DMA control/status register has the following format:

D[31:27] — DEV_ID
These read-only bits contain the data 0x0B1000. They identify the Ethernet device type.

D[27:15]
These bits are unused; they read back as 0's.

D14 — TC
This read-only bit indicates the that the byte counter has expired (decremented to 0).

D13 — EN_CNT
This read/write bit enables the DMA byte count register.

D[12:11] — BYTE_ADDR
These two read-only bits indicate the next byte number to be accessed.

D10 — REQ_PEND
This read-only bit is set when the DMA interface is active. Note that RESET and FLUSH must not be asserted when this bit is active.

D9 — EN_DMA
This read/write bit is set to enable DMA activity and reset to disable it.

D8 — WRITE
This read/write bit is set for DMA from memory to device (write) and reset for DMA from device to memory (read).

D7 — RESET
Setting this read/write bit causes a hardware reset. ERR_PEND, PACK_CNT, INT_EN, FLUSH, DRAIN, WRITE, EN_DMA, REQ_PEND, EN_CNT, and TC are all set to zero. RESET remains set, and must be reset to resume operation.

D6 — DRAIN
Setting this read/write bit forces remaining pack register bytes to be drained to memory. It resets itself.

D5 — FLUSH
This write-only bit forces PACK_CNT and ERR_PEND to zero. It always reads as zero.

D4 — INT_EN
This read/write bit enables interrupts when it is set.

D[3:2] — PACK CNT
This read only field provides the number of bytes in the pack register.

D1 — ERR_PEND
This read-only bit is set when a memory exception occurs, and is reset by setting FLUSH. DMA activity stops until it is reset.

D0 — INT_PEND
This read-only bit is set when TC=1, or when an external device raises an interrupt.

DMA Address Register

The DMA address register is located at address 0xF04000004. It holds the virtual address of the DMA transfer. Initially, it should be loaded with the virtual address where the DMA will start. As the DMA proceeds, both the DMA byte count register and bits [23:0] of the DMA address register are decremented.

Bits [31:24] of the DMA address register indicate which 16-megabyte portion of virtual memory is accessed. These bits are latched; they do not change as the DMA proceeds.

DMA Byte Count Register

When the EN_CNT bit in the DMA control/status register is set, bits [23:0] of this register keep a count of the number of bytes transferred by the DMA circuits. It should be loaded with the total number of bytes to be transferred; it decrements towards zero as the bytes are transferred. When it reaches zero, it sets TC and INT_PEND in the DMA status/control register.

Note that bits [31:24] are hardwired to zero.

DMA Diagnostic Register

This register is not implemented at this time.

16.5. SBus Slot 1 Devices

Any device plugged into the actual SBus connector on the CPU card is a "slot 1" device.

As of the print date of this manual, the SPARCengine 1E supports two video cards, color and monochrome, that can be plugged into the SBus connector on the CPU card. These cards are interchangeable at the card level, but require

different cables and monitors. The two cards have different registers. For specific information on these cards, please refer to **The SPARCengine 1E Color & Monochrome Video Cards User's Manual.**

For all other SBus cards, refer to the technical documentation & manuals provided with the card.

SPARC Assembly Language Example

```
/*
 * sbus.s
 */
#define sbud_id                 0x00010000   /* sbus ID virtual address */
#define sbus_phy                0xf0000000   /* sbus ID physical address */
#define TYPE1_ATTRIBUTES        0xd4000000   /* valid, writable, don't cache */

    .seg   "text"
    .global Sbus_id
!-------------------------------------------------------------------------
! Read SBus slot 0 card ID register
Sbus_id:
        save    %sp, -MINFRAME,%sp      ! preserve calling indow
        set     sbus_id, %o0            ! Virtual address to be mapped.
        set     TYPE1_ATTRIBUTES, %o2 ! Page table entry attributes.
        set     sbus_phy, %o3           ! Physical address to be mapped.

        call    _pmap_memory            ! Map in specified block of memory.
        mov     %o0, %o1                ! Upper virtual address to be mapped.
        set     sbus_virt, %l5          ! Address of SBus register.
        ld      [%o0],%l4               ! Read Card ID status
        nop
        ret
        restore

/*
 *      PMAP.S
 */
#define SEGINCR         0x40000     /* offset between adjacent segments */
#define SGSHIFT         18          /* LOG2(NBSG) */
#define ASI_SM          0x3         /*segment map */
#define PAGINCR         0x2000      /* offset between adjacent pages */
#define PGSHIFT         13          /* LOG2(NBPG) */
#define ASI_PM          0x4         /* page map */

        .seg   "text"
        .global _pmap_memory
/*
 * Synopsis: status=pmap_memory(low_va, hi_va, attributes, low_pa);
 *      status              : (int) don't care
 *      low_va      (%i0)   : (unsigned long) initial virtual address
 *      hi_va       (%i1)   : (unsigned long) final virtual address
 *      attributes  (%i2)   : (unsigned long) page table entry attributes
 *      low_pa      (%i3)   : (unsigned long) initial physical address
 *
 * Function "pmap_memory" (physical map memory) linearly maps the range of
 * virtual addresses specified by 'low_va' and 'hi_va', beginning at physical
 * address 'low_pa', with page table entry attributes 'attributes'.
 */
_pmap_memory:
        save   %sp, -MINFRAME,%sp
        /*
         * Fill in segment table entries for a linear mapping of main memory.
         */
        set     SEGINCR, %l3        ! Adjacent segment table entry offset.
```

```
        sub     %l3, 0x1, %l4           ! Mask for segment table addressing.
        andn    %i0, %l4, %l2           ! Adjust initial virtual address.
3:      srl     %l2, SGSHIFT, %l0       ! Segment table entry (pmeg number).
        dd      %l0, [%l2]ASI_SM        ! Set segment table entry.
        add     %l2, %l3, %l2           ! Increment address by one segment.
        cmp     %l2, %i1                ! If current virtual address is not
        bgt     4f
        nop
        bleu    3b                      ! greater than the final virtual
        nop                             ! address, continue.
4:
        /*
         * Fill in page table entries for a linear mapping of main memory.
         */
        set     PAGINCR, %l3            ! Adjacent page table entry offset.
        sub     %l3, 0x1, %l4           ! Mask for page table addressing.
        andn    %i0, %l4, %l2           ! Adjust initial virtual address.
        /*
         * Generate the initial page table entry.
         */
        andn    %i3, %l4, %l1           ! Adjust initial physical address.
5:      srl     %l1, PGSHIFT, %l0       ! Page number of page table entry.
        or      %l0, %i2, %l0           ! Combine attributes and page number.
        sta     %l0, [%l2]ASI_PM        ! Set page table entry.
        add     %l1, %l3, %l1           ! Increment physical address by a page.
        add     %l2, %l3, %l2           ! Increment virtual address by a page.
        cmp     %l2, %i1                ! If current virtual address is not
        bgt     6f
        nop
        bleu    5b                      ! greater than the final virtual
        nop                             ! address, continue.
6:
        /*
         * Mapping completed.
         */
        ret
        restore
```

C Example

```
/*
 * caddr_t map_SBus_address(u_int p_addr, u_int num_pages)
 *
 * Inputs: Physical address and the # of pages to map.
 * Returns: Virtual address of the mapped page(s).
 */

caddr_t map_SBus_address(p_addr, num_pages)
u_int   p_addr;             /* physical address on the SBus to map */
u_int   num_pages;          /* # of pages to allocate/map */
{
        long    v_pgnum;
        u_int   v_addr;
        u_int   offset;
        u_int   pg_tbl_ent;

/*
 * Save the offset from the page boundary so that
 * we can apply it to the virtual address once the
 * page is mapped in.
 */
        offset = p_addr & MMU_PAGEOFFSET;

/*
 * Create a page table entry that maps the physical address
 * "p_addr" into SBus space.
 */

        pg_tbl_ent = PGT_OBIO | PG_NC | btop(p_addr);

/*
 * Allocate virtual pages that we can use from the
 * kernel map of available virtual addresses.
 */
        v_pgnum = rmalloc(kernelmap, (long)btoc(num_pages));
        if (v_pgnum == 0)     /* no addresses available from kernelmap */
                return (caddr_t)0;
/*
 * Convert the virtual page # to a virtual address
 */
        v_addr = (u_int)kmxtob(v_pgnum);

/*
 * Map the physical address into our allocated virtual address.
 */
        segkmem_mapin(&kseg, v_addr, num_pages, PROT_READ | PROT_WRITE,
                        pg_tbl_ent, 0);

/*
 * Return to the caller of this function the virtual address of
 * the mapped SBus physical address.
 */
        return (caddr_t)(v_addr | offset);
}       /* end of of map_SBus_address */
```

Forth Example

```
f000.0000 constant SBUS-SLOT0          physical address
0001.0000 constant SBUS-ID             virtual address

SBUS-SLOT0 obio SBUS-ID map-page       map into type1 (on-card I/O) space
SBUS-ID l@ .                           fetch and display sbus-id
```

NOTE *SBus slot 0 contains the DMA, Ethernet, and SCSI chips. Slot 1 contains SBus cards.*

17

P2 Bus Interface

17.1. P2 Bus Interface Overview

The DRAM Parity memory on-board the SPARCengine 1E CPU Card can be augmented with the addition of up to 4 SPARCengine 1E ECC Memory Cards. The CPU accesses this additional memory by means of a Sun-designed private bus (P2 Bus) whose signals are carried on the outer rows "A" and "C" of the 96-pin P2 backplane connector. This bus is an extension of the CPU's SBus in which address and data is multiplexed and only slave transfers are supported (i.e., an SBus master may not reside on the P2 Bus).

17.2. Non-Compatibility Announcement for the P2 Bus

The SPARCengine 1E P2 Bus is unique to the SPARCengine 1E. It is not compatible with the P2 Bus on any other Sun Microsystems card products. Use of the Sun Microsystems P2 Bus for development is discouraged by Sun Microsystems, who reserves the right to re-configure the P2 Bus for each product released. The P2 Bus is not documented for public use.

While the SPARCengine 1E card family will work in a Sun-3E compatible backplane, Sun-3E parity memory cards are not supported in a SPARCengine 1E system.

17.3. P2 Bus Interface Memory Map

The P2 Bus Interface supports type 1 slave transfers to access ECC Memory registers and type 0 slave transfers to access the ECC Memory itself. The following tables outline the mapping of the P2 Bus within the physical address space of the SBus. These addresses are said to reside within the P2 slot of the SBus.

Pins 1 & 2 of J0801

The first table below is applicable when pins 1 and 2 of jumper J0801 are jumpered on the SE1E CPU.

Table 17-1 *P2 Slot Addresses - J0801 Pin 1 Jumpered to Pin 2*

SBus Address	Type	Description
0xFC000000-0xFC0000FF	1	1E ECC Memory I/O Registers
0xFC000100-0xFFFFFFFF	1	Reserved P2 Bus type 1
0x0C080000-0x0FFFFFFF	0	Reserved P2 Bus type 0
0x10000000-0x1FFFFFFF	0	1E ECC Memory

Pins 2 & 3 of J0801

The table below is applicable when pins 2 and 3 of jumper J0801 are jumpered together.

Table 17-2 *P2 Bus Slot Addresses - J0801 Pin 2 Jumpered to Pin 3*

SBus Address	Type	Description
0xFC000000-0xFC0000FF	1	1E ECC Memory I/O Registers
0xFC000100-0xFFFFFFFF	1	Reserved P2 Bus type 1
0x00000000-0x0FFFFFFF	0	1E ECC Memory
0x10000000-0x1FFFFFFF	0	Reserved P2 Bus type 0

17.4. P2 Bus Connector Pinout List

The table below lists the pinout of the CPU P2 connector, which includes the P2 Bus on the two outside rows.

Table 17-3 *P2 Bus Pinout List -- Connector P1802*

Signal	Pin #	Signal	Pin #	Signal	Pin #
PAR<0>	1	VCC	33	PAR<1>	65
ACK8*	2	GND	34	GND	66
ACK32*	3	RESERVED	35	CLK	67
SIZ0	4	P1.A24	36	MEM_ERR*	68
SIZ1	5	P1.A25	37	PAR<3>	69
PAR<2>	6	P1.A26	38	IRQ5*	70
READ	7	P1.A27	39	GND	71
SIZ2	8	P1.A28	40	I/OSEL*	72
BUS_ERR*	9	P1.A29	41	RESET*	73
SELRAM*	10	P1.A30	42	PUR*	74
AS*	11	P1.A31	43	ERR_EN*	75
GND	12	GND	44	GND	76
DA0	13	VCC	45	DA16	77
DAI	14	P1.D16	46	DA17	78
DA2	15	P1.D17	47	DA18	79
DA3	16	P1.D18	48	DA19	80
GND	17	P1.D19	49	GND	81
DA4	18	P1.D20	50	DA20	82
DA5	19	P1.D21	51	DA21	83
DA6	20	P1.D22	52	DA22	84
DA7	21	P1.D23	53	DA23	85
GND	22	GND	54	GND	86
DA8	23	P1.D24	55	DA24	87
DA9	24	P1.D25	56	DA25	88
DA10	25	P1.D26	57	DA26	89
DA11	26	P1.D27	58	DA27	90
GND	27	P1.D28	59	GND	91
DA12	28	P1.D29	60	DA28	92
DA13	29	P1.D30	61	DA29	93
DA14	30	P1.D31	62	DA30	94
DA15	31	GND	63	DA31	95
GND	32	VCC	64	GND	96

NOTES Signals on the middle row of pins are for the VMEbus.

Signals on all outer row pins except for 1,6,10,65,69 and 74 are used for the P2 Bus.

17.5. P2 Bus Performance

The following table outlines the performance of 1E ECC memory access via the P2 Bus.

Table 17-4 *P2 Bus Performance*

Memory Access Cycle Type	Number of SBus Clock Cycles
Word Read	8
Word Write	6
Burst Read (16 bytes)	14
Burst Write (16 bytes)	11
Partial Write (less than 4 bytes)	10

17.6. P2 Bus Programming Operation

The example in this section show how to use type 0 and type 1 device spaces. The program are going to be used as modules of other programs, the stack pointer preserve calling window should only be used at the beginning of the program.

```
/*
 * P2_bus.s
 * you can put all data define in the include file, for instance.
 * #include   "data_definition.h"
 */
#define ECC_cmd_virt   0xff0000              /* ECC command/status virtual address */
#define ECC_cmd_phy   0xfc0000000            /* ECC cmd/status physical address */
#define TYPE1_ATTRIBUTES   0xd4000000        /* valid, writable, don't cache */

    .seg   "text"
    .global P2_bus_cmd
!-------------------------------------------------------------------------
! Read ECC command/status register in type 1 space
P2_bus_cmd:
        save    %sp, -MINFRAME,%sp          ! Preserve calling window.
        set     ECC_cmd_virt, %o0           ! Virtual address to be mapped.
        set     TYPE1_ATTRIBUTES, %o2       ! Page table entry attributes.
        set     ECC_cmd_phy, %o3            ! Physical address to be mapped.

        call    _pmap_memory                ! Map in specified block of memory.
        mov     %o0, %o1                    ! Upper virtual address to be mapped.
        set     ECC_cmd_virt, %l5           ! Address of cmd/status register.
        ld      [%l5], %l4                  ! Read cmd/status register
        nop
        ret
        restore

/*
 * P2-Bus ECC memory interface test.
 */
#define ECC_phy_phy   0x100000000            /* ECC memory physical address */
#define ECC_phy_virt   0x100000              /* ECC memory virtual address */
#define TEST_PATT   0xa5a5a5a5               /* Test pattern */
#define TYPE0_ATTRIBUTES   0xd0000000        /* valid, writable, don't cache */

    .seg   "text"
    .global P2_bus_memory
!-------------------------------------------------------------------------
```

```
! Map ECC memory in type 0 space and read/write ECC memory.
P2_bus_memory:
        save    %sp, -MINFRAME,%sp      ! preserve calling window
        clr     %o0                     ! Lower virtual address to be mapped.
        set     ECC_phy_virt, %o1       ! Upper virtual address to be mapped.
        set     TYPE0_ATTRIBUTES, %o2   ! Page table entry attributes.
        set     ECC_cmd_phy, %o3        ! Physical address to be mapped.

        call    _pmap_memory            ! Map in specified block of memory.
        nop
        set     ECC_phy_virt, %l4       ! Address of virtual memory.
        set     TEST_PATT, %l2          ! test pattern source
        st      %l2, [%l4]              ! put pattern into main RAM
        not     %l2                     ! invert pattern
        inc     4, %l4                  ! bump up to next word
        st      %l2, [%l4]              ! put invert pattern into main RAM

        ld      [%l4], %l1              ! get pattern from high location
        cmp     %l1, %l2                ! is data okay?
    be  3f                              ! if yes go check original location
        nop
        call    error                   ! print test failure on console or LED
        nop
3:
        dec     4, %l4                  ! return to original word
        ld      [%l4], %l1              ! get pattern from original location
        not     %l2                     ! re-invert source pattern
        cmp     %l1, %l2                ! is data okay?
        be      4f                      ! if yes go check original location
        nop
        call    error                   ! print test failure on console or LED
        b,a     1f                      ! jump 1f exit
4:
        call    test_passed             ! print test passed on console or LED
        nop
1:
        ret
        restore

/*
 *      PMAP.S
 */
#define SEGINCR         0x40000         /* offset between adjacent segments */
#define SGSHIFT         18              /* LOG2(NBSG) */
#define ASI_SM          0x3             /* segment map */
#define PAGINCR         0x2000          /* offset between adjacent pages */
#define PGSHIFT         13              /* LOG2(NBPG) */
#define ASI_PM          0x4             /* page map */

        .seg    "text"
        .global _pmap_memory
/*
 * Synopsis: status=pmap_memory(low_va, hi_va, attributes, low_pa);
 *      status                  : (int) don't care
 *      low_va          (%i0): (unsigned long) initial virtual address
 *      hi_va           (%i1): (unsigned long) final virtual address
 *      attributes      (%i2): (unsigned long) page table entry attributes
 *      low_p           (%i3): (unsigned long) initial physical address
 *
 * Function "pmap_memory" (physical map memory) linearly maps the range of
 * virtual addresses specified by 'low_va' and 'hi_va', beginning at physical
 * address 'low_pa', with page table entry attributes 'attributes'.
 */
_pmap_memory:
```

```
        save    %sp, -MINFRAME,%sp
    /*
     * Fill in segment table entries for a linear mapping of main memory.
     */
        set     SEGINCR, %l3            ! Adjacent segment table entry offset.
        sub     %l3, 0x1, %l4           ! Mask for segment table addressing.
        andn    %i0, %l4, %l2           ! Adjust initial virtual address.
3:      srl     %l2, SGSHIFT, %l0       ! Segment table entry (pmeg number).
        stha    %l0, [%l2]ASI_SM        ! Set segment table entry.
        add     %l2, %l3, %l2           ! Increment address by one segment.
        cmp     %l2, %i1                ! If current virtual address is not
        bgt     4f
        nop
        bleu    3b                      ! greater than the final virtual
        nop                             ! address, continue.
4:
        /*
         * Fill in page table entries for a linear mapping of main memory.
         */
        set     PAGINCR, %l3            ! Adjacent page table entry offset.
        sub     %l3, 0x1, %l4           ! Mask for page table addressing.
        andn    %i0, %l4, %l2           ! Adjust initial virtual address.
        /*
         * Generate the initial page table entry.
         */
        andn    %i3, %l4, %l1           ! Adjust initial physical address.
5:      srl     %l1, PGSHIFT, %l0       ! Page number of page table entry.
        or      %l0, %i2, %l0           ! Combine attributes and page number.
        sta     %l0, [%l2]ASI_PM        ! Set page table entry.
        add     %l1, %l3, %l1           ! Increment physical address by a page.
        add     %l2, %l3, %l2           ! Increment virtual address by a page.
        cmp     %l2, %i1                ! If current virtual address is not
        bgt     6f
        nop
        bleu    5b                      ! greater than the final virtual
        nop                             ! address, continue.
6:
        /*
         * Mapping completed.
         */
        ret
        restore
```

18

Serial Interface A

18.1. Required Reference Material for Serial Interface A

Reference material required for a complete definition of the Serial Interface A:

Z8030/Z8530 Serial Communications Controller (SCC) Technical Manual, Zilog, Inc., January 1983.

AmZ8030/AmZ8530 Serial Communications Controller (SCC) Technical Manual, Advanced Micro Devices, Inc., 1982.

EIA Standard -- RS-232-C, Electronic Industries Association, August, 1969.

EIA Standard -- RS-422-A, Electronic Industries Association, December, 1978

EIA Standard -- RS-423-A, Electronic Industries Association, December, 1978.

EIA Standard -- RS-449-A, Electronic Industries Association, December, 1977.

18.2. Serial Interface A Device Address

0xE2000000
On-Board Input/Output
(OBIO)

18.3. Serial Interface A Definition

The A serial port interface supports asynchronous RS-423 with full modem control lines, and supports RS-449 on selected lines. For most applications, the interface will be directly compatible with RS-232 equipment as well. In addition to the RS-423 interface logic, four of the signals are brought out to separate connector pins via RS-422 compatible drivers. and receivers. Transmit Data (TxD), Receive Data (RxD), Request to Send (RTS) and Clear to Send (CTS) signals are provided for use in electrically noisy environments or where longer cable lengths are required.

18.4. Serial Interface A Performance

Asynchronous Speed: 19.2 baud
Synchronous Speed: 9600 baud

The Synchronous Serial Communications Controller (Z8530) will support data rates up to 2M bits/second. However, RS-232 limits transmission rates in asynchronous mode to 20K bits/second, and RS-423 limits data rates to 100K bits/second. The maximum baud rate supported by the Boot PROM on the SPARCengine 1E is 19.2K bits/second and assumes a 1/16 bit clock divisor. Faster bit rates can be programmed for custom applications. Refer to the **Z8530 SCC Technical Manuals** for more details. For reference, the clock input to the SCC for baud rate generation is 4.9152 MHz.

18.5. The Connector

Because of space considerations in designing the SPARCengine 1E, standard 25 pin (DB-25) connectors would not fit on the rear panel. The connector style is a 15-pin 3-row female (or receptacle) connector housed in a standard DB-9 shell, commonly referred to as a "double-density" or "high-density" connector. Mating male (plug) connectors are available from a variety of vendors. Below are several that are compatible with the SPARCengine 1E serial connectors.

NOTE *The Serial A and B connectors are NOT DB9 connectors.*

Table 18-1 *Compatible Serial Connectors*

Manufacturer	Part Number
AMP	204501-3
ITT Cannon	ZDEA111981
Viking	DDS2MS1

18.6. Serial A Connector Pinout List

Table 18-2 *Serial A Pinout List -- Connector J1201*

Signal	Pin #	Comments
GND	1	Ground
TxD	2	Transmit Data (RS-423 only)
RxD-	3	Receive Data- (RS-423 and -422)
RTS	4	Request to Send (RS-423 only)
CTS-	5	Clear to Send- (RS-423 and -422)
DSR	6	Data Set Ready
DTR	7	Data Terminal Ready
DCD	8	Data Carrier Detect
RTS+	9	Request to Send+ (RS-422 only)
RTS-	10	Request to Send- (RS-422 only)
CTS+	11	Clear to Send+ (RS-422 only)
TxD+	12	Transmit Data+ (RS-422 only)
TxD-	13	Transmit Data- (RS-422 only)
RxD+	14	Receive Data+ (RS-422 only)
GND	15	Chassis Ground

18.7. Special Cabling Requirements

An adapter cable is required to connect the SPARCengine 1E serial ports to standard RS-423 or RS-232 compatible DTE or DCE equipment. Due to the variety of possible interface options, no single cable will be adequate for all configurations. Following are examples of serial interface configurations using the SPARCengine 1E serial A port.

Table 18-3 *Serial A Cabling Option: Asynchronous RS-232 DTE to DTE*

Signal	SPARCengine 1E Serial A Port (DTE) A Connector (15-pin)	RS-232 Device (DTE) RS-232 Connector (25-pin)
GND	1	7
TxD	2	3
RxD	3	2
RTS	4	5
CTS	5	4
DCD	8	20
DTR	7	8

Table 18-4 *Serial A Cabling Option: Asynchronous RS-449 DTE to DTE*

Signal	**SPARCengine 1E Serial A Port** (DTE) A Connector (15-pin)	**RS-449 Device** (DCE) RS-449 Connector (37-pin)
GND	1	19
TxD+	12	24
TxD-	13	6
RxD+	14	22
RxD-	3	4
RTS+	9	25
RTS-	10	7
CTS+	11	27
CTS-	5	9

19

Serial Interface B

19.1. Required Reference Material for Serial Interface B

Reference material required for a complete definition of the Serial Interface B:

Z8030/Z8530 Serial Communications Controller (SCC) Technical Manual, Zilog, Inc., January 1983.

AmZ8030/AmZ8530 Serial Communications Controller (SCC) Technical Manual, Advanced Micro Devices, Inc., 1982.

EIA Standard -- RS-232-C, Electronic Industries Association, August, 1969.

EIA Standard -- RS-422-A, Electronic Industries Association, December, 1978

EIA Standard -- RS-423-A, Electronic Industries Association, December, 1978.

EIA Standard -- RS-449-A, Electronic Industries Association, December, 1977.

19.2. Serial Interface B ~~Device Address~~

0xE2000000
~~On-Board Input/Output~~
(OBIO)

19.3. Serial Interface B Definition

The B serial port interface supports asynchronous and synchronous RS-423 with full modem control lines. For most applications the interface will be directly compatible with RS-232 equipment as well. The B port is not identical to the A port; synchronous clock lines (TxC, TxCO, and RxC) have been provided to support applications requiring SunLink capability via a synchronous modem connection.

19.4. Serial Interface B Performance

Asynchronous Speed: 19.2 baud
Synchronous Speed: 9600 baud

The Synchronous Serial Communications Controller (Z8530) will support data rates up to 1M bits/second. Refer to the **Z8530 SCC Technical Manual** for more details. For reference, the clock input to the SCC for baud rate generation is 4.9152 MHz.

19.5. The Connector

Because of space considerations in designing the SPARCengine 1E, standard 25 pin (DB-25) connectors would not fit on the rear panel. The connector style is a 3-row female (or receptacle) connector housed in a standard DB-9 shell, commonly referred to as a "double-density" or "high-density" connector. Mating male (plug) connectors are available from a variety of vendors. Below are several that are compatible with the SPARCengine 1E serial connectors.

Table 19-1 *Compatible Serial Connectors*

Manufacturer	Part Number
AMP	204501-3
ITT Cannon	ZDEA111981
Viking	DDS2MS1

19.6. Serial B Connector Pinout List

Table 19-2 *Serial B Pinout List -- connector J1202*

Signal	Pin #	Comments
GND	1	Ground
TxD	2	Transmit Data (RS-423, RS232)
RxD-	3	Receive Data (RS-423, RS232)
RTS	4	Request to Send
CTS	5	Clear to Send
DSR	6	Data Set Ready
DTR	7	Data Terminal Ready
DCD	8	Data Carrier Detect
TxC	9	Sync Transmit Clock (Input)
TxCO	10	Sync Transmit Clock (Output)
RxC	11	Sync Receive Clock (Input)
TxD+	12	Transmit Data+ (RS-422 only)
TxD-	13	Transmit Data- (RS-422 only)
RxD+	14	Receive Data+ (RS-422 only)
GND	15	Ground

19.7. Special Cabling Requirements

An adapter cable is required to connect the SPARCengine 1E serial ports to standard RS-423 or RS-232 compatible DTE or DCE equipment. Due to the variety of possible interface options, no single cable will be adequate for all configurations. Following are examples of serial interface configurations using the SPARCengine 1E serial B port.

Table 19-3 *Serial B Cabling Option: Asynchronous RS-232 DTE to DTE*

Signal	SPARCengine 1E Serial B Port (DTE) B Connector (15-pin)	RS-232 Device (DTE) RS-232 Connector (25-pin)
GND	1	7
TxD	2	3
RxD	3	2
RTS	4	5
CTS	5	4
DCD	8	20
DTR	7	8

Table 19-4 *Serial B Cabling Option: Synchronous RS-232 DTE to DTE*

Signal	SPARCengine 1E Serial B Port (DTE) B Connector (15-pin)	RS-232 Modem (DTE) RS-232 Connector (25-pin)
TDX	2	3
RDX	3	2
GND	1	7
DCD,DSR	8,6	20
DTR	7	8,6
RTS	4	5
CTS	5	4
TXCO	10	17
RXC	11	24

Table 19-5 *Serial B Cabling Option: Synchronous RS-232 DTE to DCE*

Signal	SPARCengine 1E Serial B Port (DTE) B Connector (15-pin)	RS-232 Modem (DCE) RS-232 Connector (25-pin)
GND	1	7
TxD	2	2
RxD	3	3
RTS	4	4
CTS	5	5
DSR	6	6
DTR	7	20
DCD	8	8
TxC	9	15
RxC	11	17

Table 19-6 *Serial B Cabling Option: Asynchronous RS-449 DTE to DTE*

Signal	SPARCengine 1E Serial B Port (DTE) A Connector (15-pin)	RS-449 Device (DCE) RS-449 Connector (37-pin)
GND	1	19
TxD+	12	24
TxD-	13	6
RxD+	14	22
RxD-	3	4

20

Keyboard/Mouse Interface

20.1. Required Reference Material for Keyboard/Mouse Interface

Reference material required for a complete definition of the Keyboard/Mouse Interface:

Z8030/Z8530 Serial Communications Controller (SCC) Technical Manual, Zilog, Inc., January 1983.

AmZ8030/AmZ8530 Serial Communications Controller (SCC) Technical Manual, Advanced Micro Devices, Inc., 1982.

20.2. Keyboard/Mouse Device Address

0xE0000000
On-Board Input/Output
(OBIO)

20.3. Keyboard/Mouse Interface Definition

The keyboard/mouse serial interfaces are implemented with a Z8530 Synchronous Serial Communications Controller. The SCC features two programmable serial channels with built in baud-rate generators. The clock input to the SCC for baud-rate generation is 4.9152 MHz and is independent of the IU clock.

Reset of the keyboard/mouse SCC is forced at power-up by asserting both read and write strobes simultaneously.

Mouse and keyboard data are transmitted and received over connector J1101 (DB-15). All data signals are TTL-compatible and cannot be direct-connected to RS-232, RS-423, or other non-TTL compatible equipment.

20.4. Specifications

The keyboard/mouse interfaces are designed to connect to Sun type-3 and type-4 keyboards. The Boot PROM will interrogate the interface upon power-up to determine if a keyboard is present. If not, the PROM expects a terminal to be connected to the Serial A port.

NOTE *The keyboard/mouse interface is intended for use with Sun keyboards only; custom serial expansion via this interface requires Boot PROM modifications and is not recommended without the support of Sun Consulting.*

20.5. Keyboard/Mouse Connector Pinout List

Table 20-1 *Keyboard/Mouse Pinout List*

Signal	Pin #	Comments
RxD0	1	Received Data 0
GND	2	Ground
TxD0	3	Transmitted Data 0
GND	4	Ground
RxD1	5	Received Data 1
GND	6	Ground
TxD1	7	Transmitted Data 1
GND	8	Ground
GND	9	Ground
+5V	10	
+5V	11	
+5V	12	
+5V	13	
+5V	14	
+5V	15	

A

Environmental Tests and Results

A.1. Tests Completed

As of this printing (March 31, 1990), all "standard" tests are complete, some "rugged" tests are complete, and the remaining "rugged" tests are expected to be completed by May 31, 1990. In the results presented in this chapter, those tests that are still to be completed are indicated with TBD (to be determined). For definitions of standard and rugged, see appropriate sections in this chapter.

A.2. Overview

This chapter describes the environmental tests performed on the SPARCengine 1E card set. The test series include both standard and rugged tests, which correspond to the Sun system-level Environmental Test Specification and Military Standard 810E, respectively.

NOTE *Because these test specifications are only applicable to full systems (boards within chassis, with power supplies, etc), they have been modified for the board-level SPARCengine 1E evaluation.*

A summary of the test configurations (board sets and test fixtures) is provided. After the summary, a general description of the test approach is given for both standard and rugged series, along with the test levels for the individual tests.

Results are given for all standard tests and those rugged tests already completed.

A.3. Test Configurations

For simplicity, the SPARCengine 1E boards were tested together as "systems" in a variety of special test fixtures. The weakest board in the set for each test will thus determine the performance level of the entire set; this allows a common specification for all boards and greatly reduces test time.

The boards included in the entire test series are:

SPARCengine 1E CPU
SPARCengine 1E Color Frame Buffer
SPARCengine 1E Analog Frame Buffer
SPARCengine 1E Memory 4MB
SPARCengine 1E Memory 16MB

A total of two complete sets of boards were used in these tests, to allow parallel testing and minimize the effects of transducer installation and removal.

To minimize test time, non-operating tests were performed on all boards at once (whenever possible). Operating tests were performed on two groups of boards (one group at a time), typically configured as follows:

Table A-1 *Testing Groups*

"M04" Group	"C16" Group
SPARCengine 1E CPU SPARCengine 1E Analog Frame Buffer SPARCengine 1E Memory 4MB	SPARCengine 1E CPU SPARCengine 1E Color Frame Buffer SPARCengine 1E Memory 16MB

Each group was on a common backplane, in a test fixture appropriate for the individual test.

Standard Test Series

The standard test series is the complete set of Sun Environmental Test Specifications (Sun 950-1316-02) for system-level products. The board set was tested to the requirements of the Class III (General Commercial/Industrial) environment. The series included tests which are not covered by MilStd 810E (such as system-oriented electrical tests) and others in which part or all of Sun's standard test levels exceed the MilStd (such as the high end of Non-Operating Temperature).

Most of the standard tests were conducted with the SPARCengine 1E boards installed in a card cage enclosure which consists of an off-the-shelf card cage, power supply, backplane, and fans, as well as a Sun-designed sheet metal shell. This enclosure is a six slot VME card cage in a Sun "shoebox" peripherial package. The "shoebox" is used by Sun for development and prototype testing of 6U format VMEbus boards. Since voltage margin and other DC power tests were not conducted by EnvTest on the SPARCengine 1E boards, this enclosure allowed the boards to be tested for response to AC power variations in a "typical" system. The enclosure also provided a convenient installation for all non-operating tests which do not require a special test fixture.

The standard test series consists of 18 individual tests.

Standard Test Results

The SPARCengine 1E board family has been successfully tested to the Class III requirements of Sun's Environmental Test Specification (950-1316-02). The Class III (General Commercial/Industrial) environment includes extremes of mechanical shock and vibration, climatic conditions, and electrical power variations.

Standard Reference Conditions

Temperature:	23 degrees C, +/- 5 degrees C
Humidity:	20% to 70% RH

Sun Standard Environmental Specifications

Mechanical Connection Repetitions	100 insertions at each I/O port.
Temperature	
Operating:	
Temperature:	0 to +40 degrees C
Humidity:	20% +/- 5% RH non-condensing at all temperatures.
Non-Operating:	
Temperature:	-40 to +75 degrees C
Humidity:	20% +/- 5% RH non-condensing at all temperatures.
Humidity	
Operating:	
Humidity/Temperature:	20% to 80% RH non-condensing @ 40 degrees C
Non-Operating:	
Humidity/Temperature:	95% RH non-condensing @ 40 degrees C
Altitude	
Operating:	
Altitude/Temperature:	10,000 ft (3048 meters) at 10 and 40 degrees C
Non-Operating:	
Altitude/Temperature:	40,000 ft (12,192 meters) at 0 degrees C
Shock	
Operating:	
Magnitude	5 G's (peak)
Duration	11 ms
Waveform	Half Sine
Repetitions	60 (10 times each on all six surfaces)
Non-Operating:	
Magnitude	30 G's (peak)
Duration	11 ms
Waveform	Half Sine
Repetitions	18 (3 times each on all six surfaces)
Vibration (Unpackaged)	
Operating:	
Frequency Range	5 to 500 to 5 Hz
Magnitude	0.03 inch (.76 mm) p-p disp. to .25 G's (peak) continue at 0.25 G's (peak)
Non-Operating:	
Frequency Range	5 to 500 to 5 Hz
Magnitude	0.15 inch (3.81 mm) p-p disp. to 1.0 G's (peak) continue at 1.0 G's (peak)

Rugged Test Series

The rugged test series is a subset of Military Standard 810E (Environmental Test Methods) for system-level devices. Tests from Military Standard 167-1 (Mechanical Vibrations of Shipboard Equipment) are also included. Rugged testing is not performed where the regular Sun environmental tests cover the MilStd requirements, or where other Sun groups do equivalent tests.

Testing of individual boards to 810E and 167-1 is complicated by the emphasis the Standards place on system testing. Component level testing is not supported by the Standards, and in some cases is not recommended. For such methods as shock and vibration, enclosures and/or shipping materials will modify the environmental excitation significantly; no reasonable test series could mimic the range of possible transformations. Methods such as altitude and temperature are also modified by an enclosure when the boards are in power on operating mode, since the airflow delivered by an enclosure design is a major influence on card performance.

Despite these drawbacks, conformance to the MilStd 810E is possible for some methods without significant modification. For other methods, special equipment has been developed and altered tests will be performed on the boards, to acquire data that can be used by enclosure engineers in the design of products which will meet the requirements of the Standards. For example, temperature and altitude tests will be run with equipment capable of providing and measuring airflow across the cards, so that the minimum airflow requirement can be determined. Another special fixture has been developed to rigidly support the cards in shock and vibration, so that the maximum allowable transmissibility of an enclosure can be determined.

For all methods below, it is assumed that the cards are not intended for combat applications; this restriction eliminates the extreme levels and test types which are required for such equipment. It is also assumed that failure of Sun cards in any test does not pose a direct hazard to personnel or equipment (as from explosion of components); this restriction reduces the required test time of certain tests and eliminates others.

Test Results for Military Standard Climatic Specifications

The SPARCengine 1E board set has been successfully tested to the requirements of Military Standard 810E for extremes of climatic environment to date. Those tests not completed are marked TBD (To Be Done). Operating tests were performed with the boards installed in a special test fixture which monitors airflow and component temperatures during the tests. The following table summarizes the minimum airflow requirement for the installed boards in each of the tested conditions:

Table A-2 *Powered Climatic Test Results*

810E Method	Description	Category	Condition (meters/sec)	Minimum Airflow
500.3	Low Pressure (Altitude)	Worst-Case	57.2 kPa (4570 m) (15,000 ft)	TBD
501.3	High Temperature	Ambient Basic Hot Ambient Hot Induced Basic Hot	43C 49C 63C	TBD TBD TBD
502.3	Low Temperature	TBD	TBD	TBD

NOTE *The airflow requirements shown represent the single worst case board in the product family; other boards may require less airflow.*

In addition, the SPARCengine 1E board set has been successfully tested in the following unpowered climatic tests:

Table A-3 *Unpowered Climatic Test Results*

810E Method	Description	Category	Condition
500.3	Low Pressure (Altitude)	Worst-Case	57.2 kPa (4570 m)
501.3	High Temperature	Induced Hot	71C
502.3	Low Temperature	Induced Severe Cold	-51C
503.3	Temperature Shock	Induced Hot to Induced Severe Cold	71C -51C
507.3	Humidity	Aggravated	TBD

Test Results for Military Standard Dynamic Mechanical Specifications

The SPARCengine 1E board set has been successfully tested to the requirements of Military Standard 810E for extremes of dynamic environment. Operating tests were performed with the boards installed in a rigid test fixture which transmits system level excitation inputs directly to the boards. The following table summarizes the maximum overtest condition applied for each test; this overtesting allows for amplification of nominal test conditions at chassis resonances:

Table A-4 *Dynamic Environment Test Results*

810E Method	Description	Category	Nominal Condition	Chassis Amplification
514.4	Vibration	Category 1 (1) Basic Trans Common Carrier	1.27 GRMS 10-500 Hz Random	3x-5x band-limited, depending on axis and band frequency limits
514.4	Vibration	Category 8 Ground Mobile Common Carrier	1.27 GRMS 10-500 Hz Random	3x-5x band-limited, depending onaxis and band frequency limits
516.4	Functional Shock	Ground Equipment	40G SRS 6-9 msec	2x

Test Results for Shipboard Vibration

Performance to Military Standard 167-1 for shipboard vibration environments is specified similarly:

Table A-5 *Shipboard Vibration Test Results*

167 Method	Description	Category	Nominal Condition	Chassis Amplification
Type I	Environmental Vibration	Shipboard Equipment	.003 to .03 in 4 to 50 Hz	TBD

Packaged Product Test Results

In addition, the SPARCengine 1E board family has been tested in the following packaged product test (no overtest applicable):

Table A-6 *Packaged Product Test Results*

810E Method	Description	Category	Nominal Condition
516.4	Transit Drop (Packaged)	Under 45.4 kg Under 91 cm	48 inches 26 surfaces

A.4. Disclaimer

Testing to the requirements of Military Standards 810E and 167-1 is intended for full systems only. The fixtured board-level tests performed by Sun were intended to provide information to system designers and integrators, so that enclosure systems which will survive the MilStd tests can be developed. While each specification above includes adequate information to allow practical enclosure design or selection, no guarantee can be made of board performance in a particular enclosure.

A.5. Thermal Mapping

The SPARCengine 1E CPU board was mapped running SunDiag using an infrared thermal imager. The test was performed at room temperature (23C) and still air in an open frame chassis. No component mapped during this testing measured higher than recommended maximum allowable temperature for the above test conditions.

Below is a table showing various component temperatures measured on the board. This table should be used for relative temperature comparisons only.

Table A-7 *Temperatures of SPARCengine 1E Components*

Component Location	Temperature (C)
U1101	57
U0102	52.5
U0101	53.5
U0903	73.5
U1201	68.5
U0602	68.5
U1202	68.5
U0401	54
U0402	63.5
U0302	64.5
U0901	63
U1605	64
U1602	64
U1410	73
U0801	54
U1607	56
RP1003/1004	61.2

B

Getting Help for the SPARCengine 1E CPU Cards

If you have problems installing or using the SPARCengine 1E CPU cards, call Sun Microsystems at the appropriate hotline number (see the next page).

You will be asked for the following information:

- Your name and electronic mail address (if any).
- Your company name, address, and phone number.
- The model and serial number of your SPARCengine 1E CPU card.
- Any information that may help to diagnose the problem.

If you prefer, you can direct questions by electronic mail to `sun!hotline`. Be sure to include the same information as above.

Call your sales representative if you have questions about Sun support services or your shipment.

Sun Hotline Numbers

Sun customers can call service hotlines throughout the world for answers to software-support and hardware-support questions. The service hotlines are listed below. If your country is not shown in the table, please phone your local Sun sales office.

Country	*Service Region*	*Hotline Number*
Australia	Sun Australia	(2) 436-4699
Canada	*Central Region:*	
	Ottawa	(613) 723-8112
	Ontario	(800) 263-1680 or (416) 477-6745
	Eastern Region:	
	New Brunswick, Newfoundland, Novia Scotia, Prince Edward Island, Quebec	(800) 361-1554 or (514) 738-4885
	Prairies Region:	
	Saskatchewan, Manitoba, Alberta	(800) 661-9256 or (403) 262-6722
	Vancouver:	
	British Columbia	(800) 663-0440 or (604) 684-4120
France	Paris Sun Microsystems France SA	1 4630 0231
Germany	Munich Sun Microsystems GmbH	89/95094-321
Hong Kong	Sun Hong Kong	(5) 865-1688
Japan	C. Itoh Data Systems Nihon Sun	(3) 497-4746 (3) 221-7021
The Netherlands	Soest Sun Microsystems Nederland BV	2155 24888
Sweden	Solna Sun Microsystems AB	8 764 78 10
Switzerland	Zurich Sun Microsystems (Schweiz) AG	1 828 9555
United Kingdom	*European Customer Service:*	
	Surrey Sun Microsystems UK Ltd	276 50183
	Albany Park Sun Microsystems UK Ltd	0276 691052
United States	All, including Puerto Rico	1-800-USA-4-SUN (1-800-872-4786)
Countries Not Listed	All countries outside the USA, Europe, and Northern Africa	(415) 496-6119

B.1. Getting Sun Help with Your Software Development

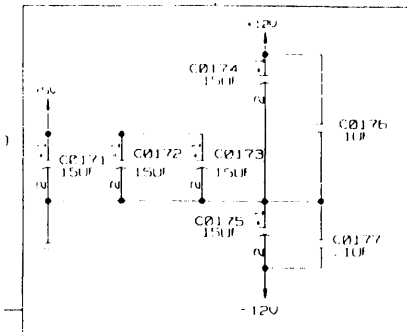
Sun's Professional Services organization is available to assist you in developing firmware and software for your SPARCengine 1E card family. If your development requires customization of Sun's standard software, or creation of drivers for the Sun private P2 Bus, Sun Consulting can help you identify the specific source code that you will need.

Contact your Sun Sales Representative.

C

The CPU Card Schematic Diagrams & Assembly Drawings

REV	DESCRIPTION	DATE	APPROVED
50	ENG. RELEASE PER ECO G781		



+5V

C0101 .1UF 50V 20%	C0102 .1UF 50V 20%	C0103 .1UF 50V 20%	C0104 .1UF 50V 20%	C0105 .1UF 50V 20%	C0106 .1UF 50V 20%	C0107 .1UF 50V 20%	C0108 .1UF 50V 20%	C0109 .1UF 50V 20%	C0110 .1UF 50V 20%
C0111 .1UF 50V 20%	C0112 .1UF 50V 20%	C0113 .1UF 50V 20%	C0114 .1UF 50V 20%	C0115 .1UF 50V 20%	C0116 .1UF 50V 20%	C0117 .1UF 50V 20%	C0118 .1UF 50V 20%	C0119 .1UF 50V 20%	C0120 .1UF 50V 20%	C0121 .1UF 50V 20%
C0122 .1UF 50V 20%	C0123 .1UF 50V 20%	C0124 .1UF 50V 20%	C0125 .1UF 50V 20%	C0126 .1UF 50V 20%	C0127 .1UF 50V 20%	C0128 .1UF 50V 20%	C0129 .1UF 50V 20%	C0130 .1UF 50V 20%	C0131 .1UF 50V 20%
C0132 .1UF 50V 20%	C0133 .1UF 50V 20%	C0134 .1UF 50V 20%	C0135 .1UF 50V 20%	C0136 .1UF 50V 20%	C0137 .1UF 50V 20%	C0138 .1UF 50V 20%	C0139 .1UF 50V 20%	C0140 .1UF 50V 20%	C0141 .1UF 50V 20%
C0142 .1UF 50V 20%	C0143 .1UF 50V 20%	C0144 .1UF 50V 20%	C0145 .1UF 50V 20%	C0146 .1UF 50V 20%	C0147 .1UF 50V 20%	C0148 .1UF 50V 20%	C0149 .1UF 50V 20%	C0150 .1UF 50V 20%	C0151 .1UF 50V 20%
C0152 .1UF 50V 20%	C0153 .1UF 50V 20%	C0154 .1UF 50V 20%	C0155 .1UF 50V 20%	C0156 .1UF 50V 20%	C0157 .1UF 50V 20%	C0158 .1UF 50V 20%	C0159 .1UF 50V 20%	C0160 .1UF 50V 20%	C0161 .1UF 50V 20%
C0162 .1UF 50V 20%	C0163 .1UF 50V 20%	C0164 .1UF 50V 20%	C0165 .1UF 50V 20%	C0166 .1UF 50V 20%	C0167 .1UF 50V 20%	C0168 .1UF 50V 20%	C0169 .1UF 50V 20%	C0170 .1UF 50V 20%	

NOTES - UNLESS OTHERWISE SPECIFIED:

1. PARTIAL REFERENCE DESIGNATORS ARE SHOWN; FOR COMPLETE DESIGNATION, PREFIX WITH UNIT NUMBER AND ASSEMBLY DESIGNATIONS.
2. RESISTANCE VALUE IN OHMS.
3. CAPACITANCE VALUE IN UF.
4. P/O INDICATES PART OF.
5. UNUSED CONNECTOR PINS NOT SHOWN.
6. * FOLLOWING SIGNAL NAME INDICATES LOW OR NOT FUNCTION.

-1 AS SHOWN

L. RODRIGUES

sun microsystems

SCHEMATIC DIAGRAM - SPARC ENGINE 1E WITHOUT FPA

SIZE	FSCM NO.	DWG. NO.
D	7X430	050-8058-05

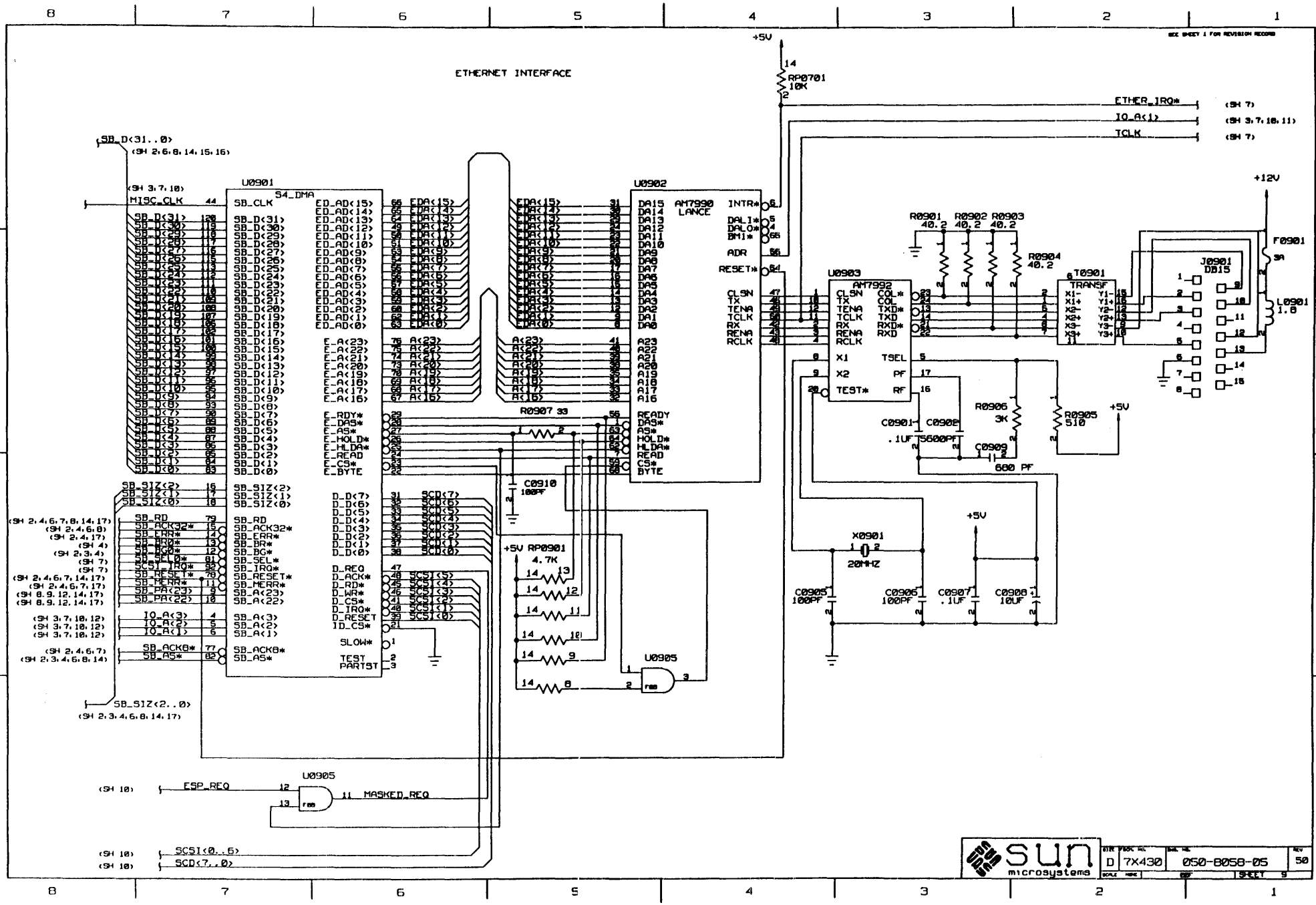
SCALE NONE | SHEET 1 OF 18

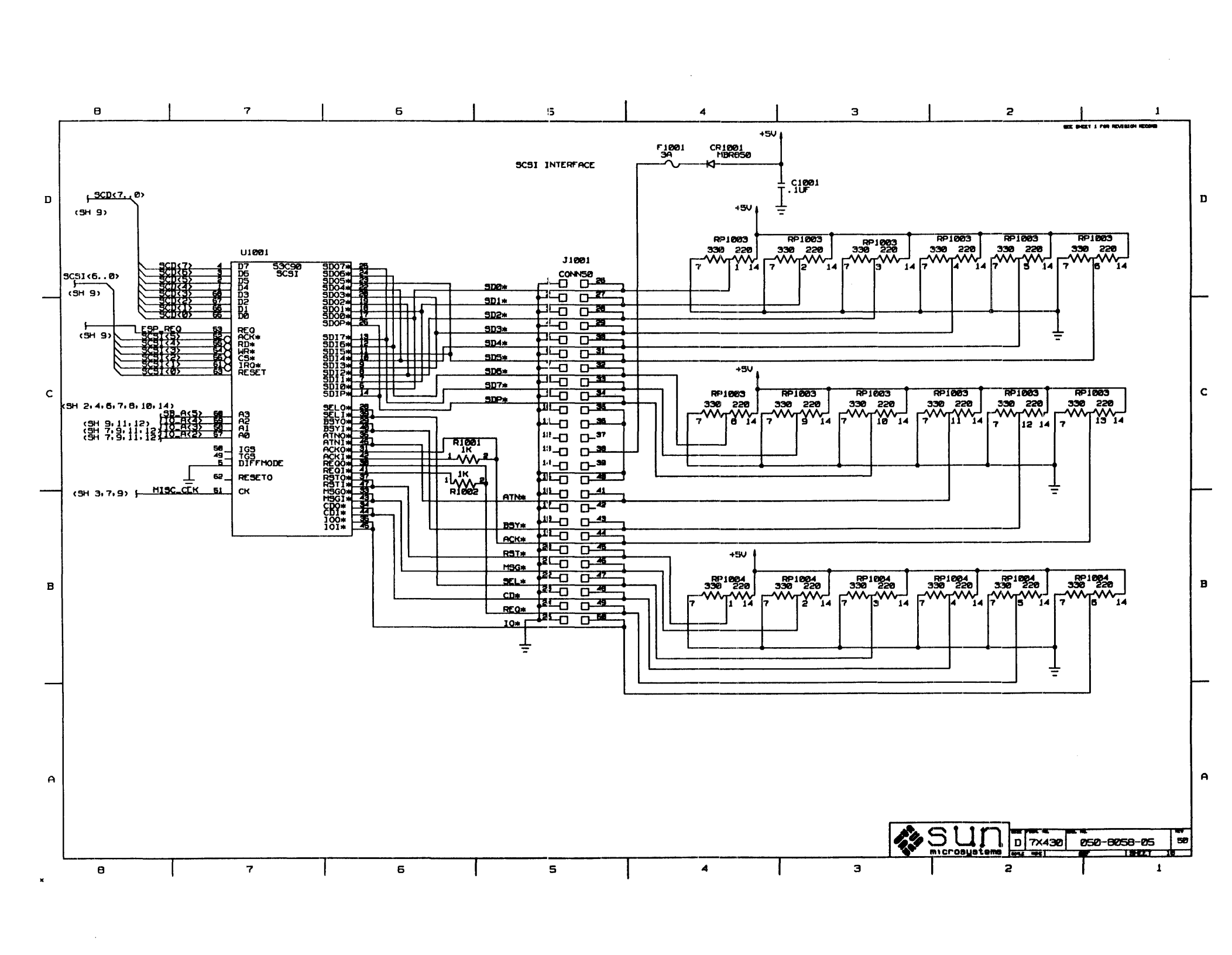
S BUS CONNECTOR

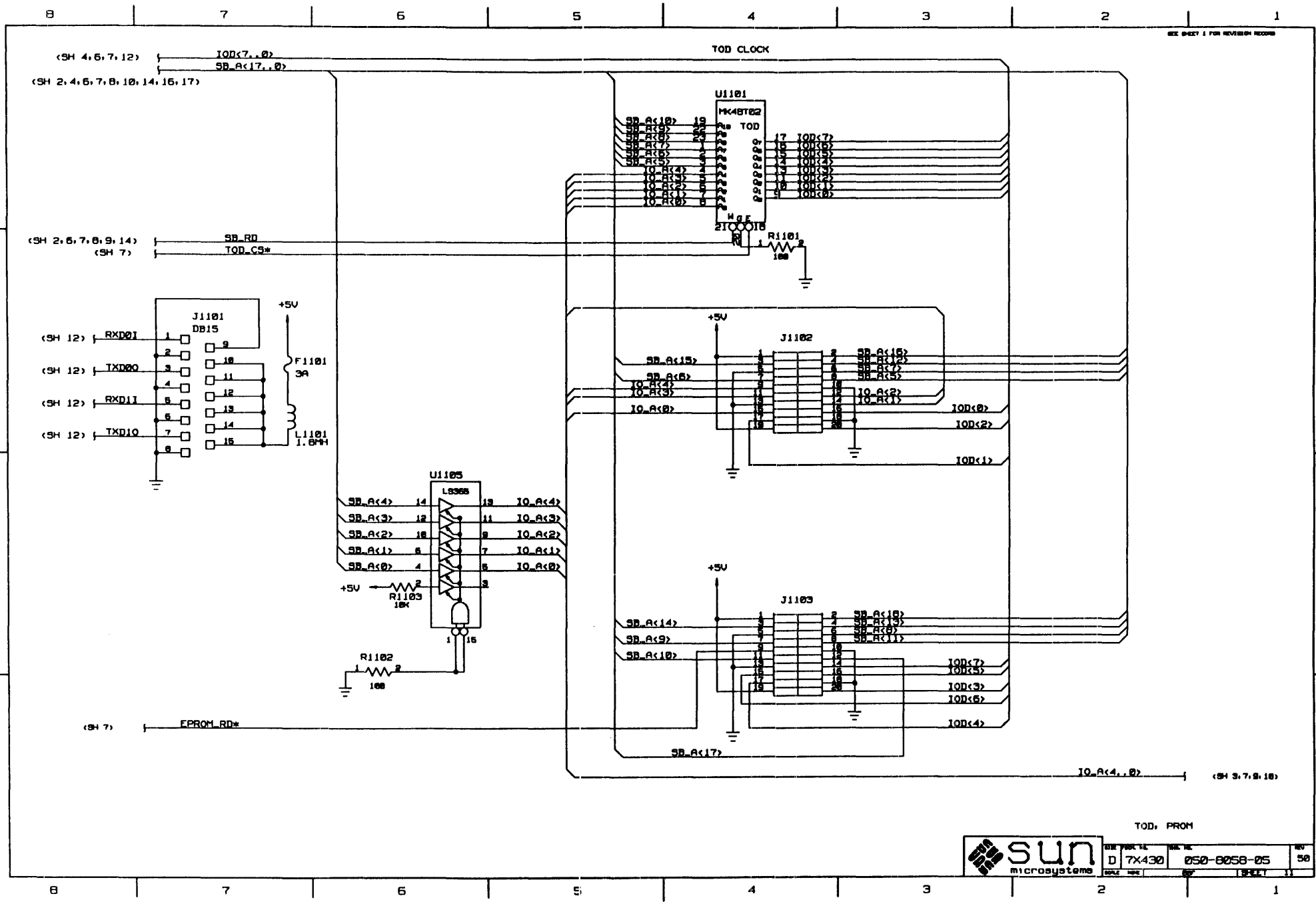
+5V

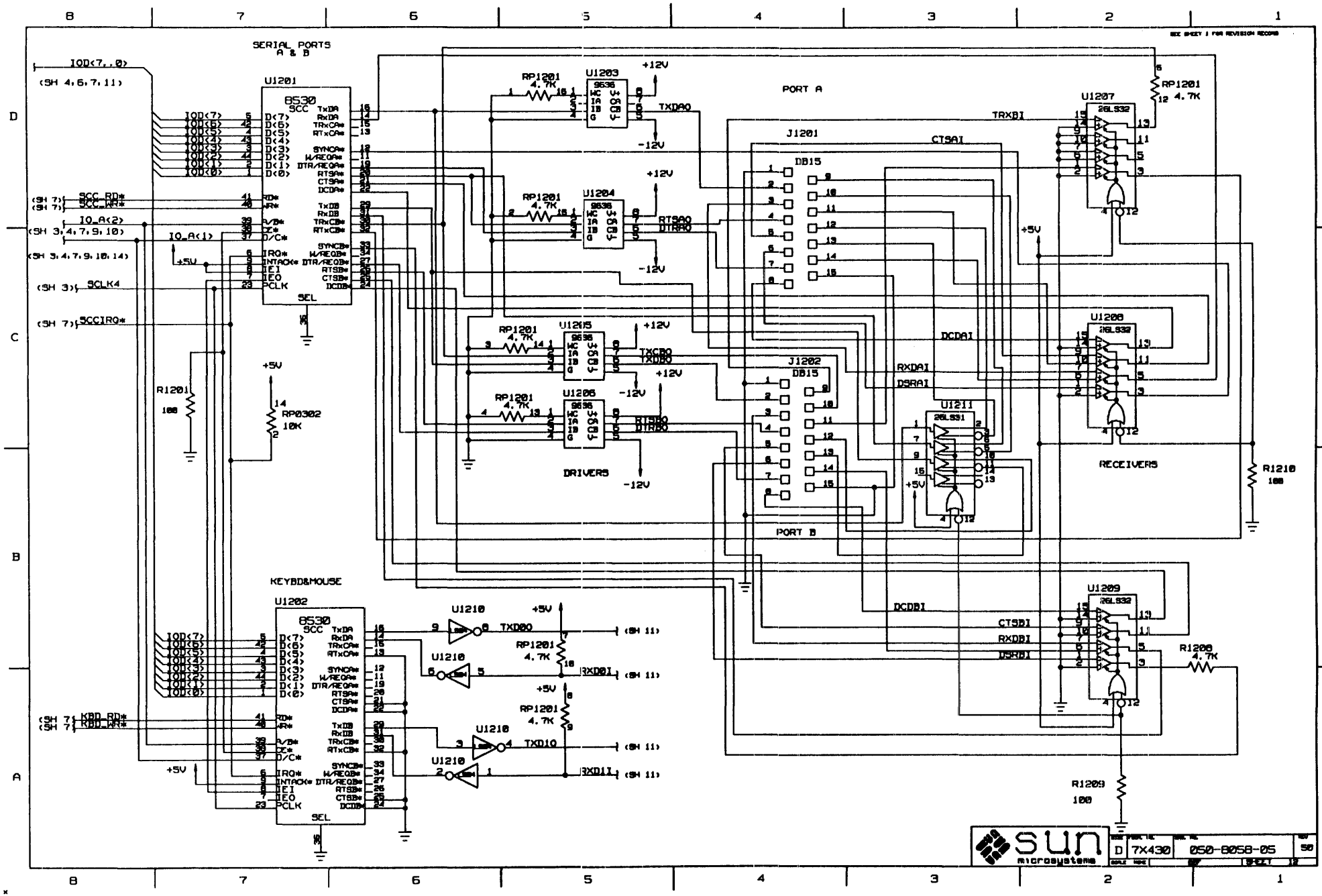
R0201 10K
R0202 10K
R0203 10K
R0204 10K
R0205 10K
R0206 10K
R0207 10K
R0208 10K

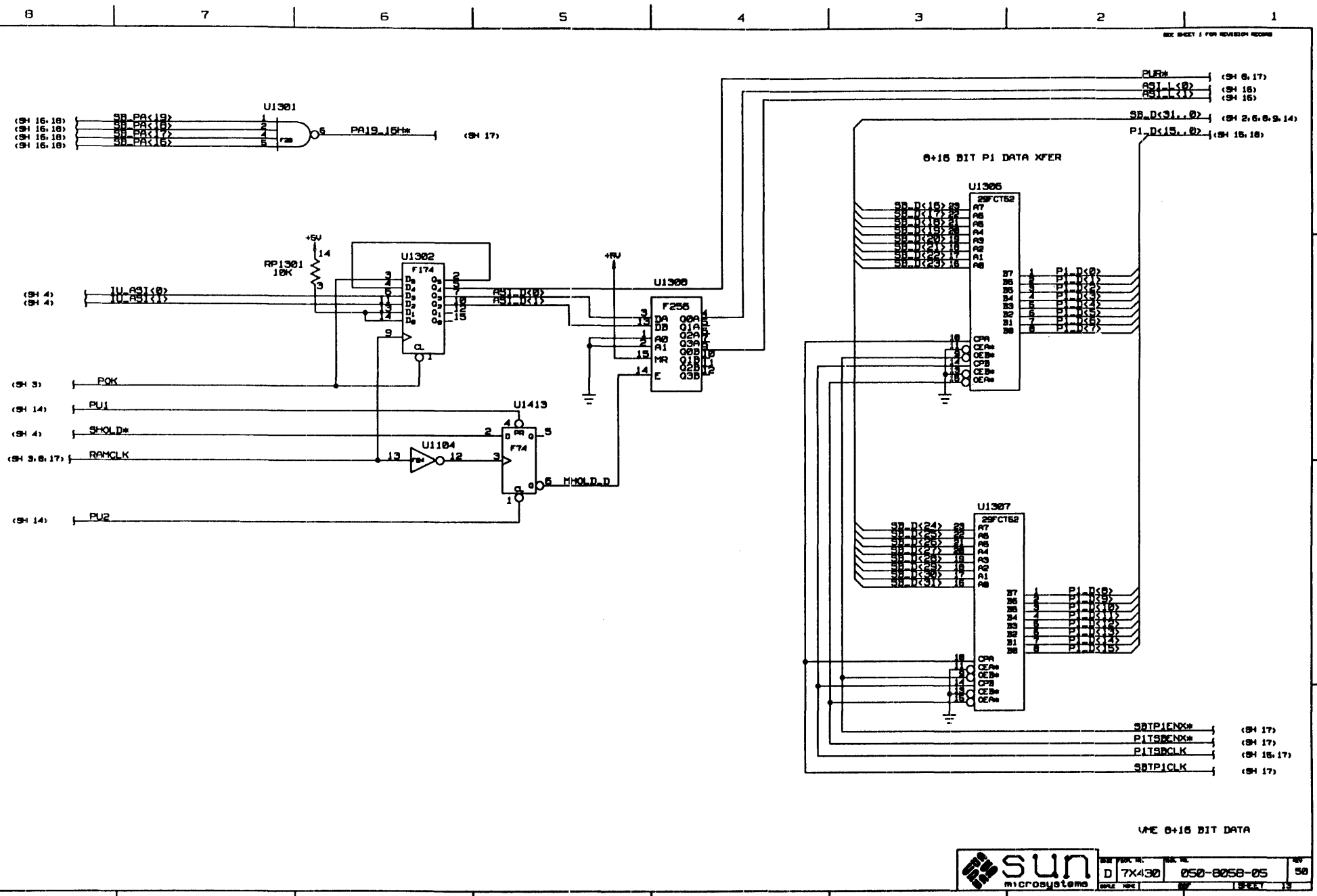
P3 - 1	GND/L	P3 - 25	SB_D<31>	P3 - 49	SB_CLK	P3 - 73	SB_D<30>
P3 - 2	SB_BR1*	P3 - 26	SB_SIZ<0>	P3 - 50	SB_BG1*	P3 - 74	SB_SIZ<1>
P3 - 3	SB_SEL1*	P3 - 27	SB_SIZ<2>	P3 - 51	SB_AS*	P3 - 75	SB_RD
P3 - 4	SB_IRQ<1>*	P3 - 28	SB_IRQ<7>*	P3 - 52	GND/L	P3 - 76	GND/L
P3 - 5	SB_D<0>	P3 - 29	SB_A<0>	P3 - 53	SB_D<1>	P3 - 77	SB_A<1>
P3 - 6	SB_D<2>	P3 - 30	SB_A<2>	P3 - 54	SB_D<3>	P3 - 78	SB_A<3>
P3 - 7	SB_D<4>	P3 - 31	SB_A<4>	P3 - 55	SB_D<5>	P3 - 79	SB_A<5>
P3 - 8	SB_IRQ<2>*	P3 - 32	SB_MERR*	P3 - 56	VCC	P3 - 80	VCC
P3 - 9	SB_D<6>	P3 - 33	SB_A<6>	P3 - 57	SB_D<7>	P3 - 81	SB_A<7>
P3 - 10	SB_D<8>	P3 - 34	SB_A<8>	P3 - 58	SB_D<9>	P3 - 82	SB_A<9>
P3 - 11	SB_D<10>	P3 - 35	SB_A<10>	P3 - 59	SB_D<11>	P3 - 83	SB_A<11>
P3 - 12	SB_IRQ<3>*	P3 - 36	SB_ERR*	P3 - 60	GND/L	P3 - 84	GND/L
P3 - 13	SB_D<12>	P3 - 37	SB_A<12>	P3 - 61	SB_D<13>	P3 - 85	SB_PA<13>
P3 - 14	SB_D<14>	P3 - 38	SB_PA<14>	P3 - 62	SB_D<15>	P3 - 86	SB_PA<15>
P3 - 15	SB_D<16>	P3 - 39	SB_PA<16>	P3 - 63	SB_D<17>	P3 - 87	SB_PA<17>
P3 - 16	SB_IRQ<4>*	P3 - 40	SB_ACK8*	P3 - 64	VCC	P3 - 88	VCC
P3 - 17	SB_D<19>	P3 - 41	SB_PA<18>	P3 - 65	SB_D<18>	P3 - 89	SB_PA<19>
P3 - 18	SB_D<21>	P3 - 42	SB_PA<20>	P3 - 66	SB_D<20>	P3 - 90	SB_PA<21>
P3 - 19	SB_D<23>	P3 - 43	SB_PA<22>	P3 - 67	SB_D<22>	P3 - 91	SB_PA<23>
P3 - 20	SB_IRQ<5>*	P3 - 44	SB_ACK32*	P3 - 68	GND/L	P3 - 92	GND/L
P3 - 21	SB_D<25>	P3 - 45	SB_PA<24>	P3 - 69	SB_D<24>	P3 - 93	SB_PA<25>
P3 - 22	SB_D<27>	P3 - 46	SB_PA<26>	P3 - 70	SB_D<26>	P3 - 94	SB_PA<27>
P3 - 23	SB_D<29>	P3 - 47	SB_SEL2	P3 - 71	SB_D<28>	P3 - 95	SB_RESET*
P3 - 24	SB_IRQ<6>*	P3 - 48	-12V/L	P3 - 72	VCC	P3 - 96	+12V/L

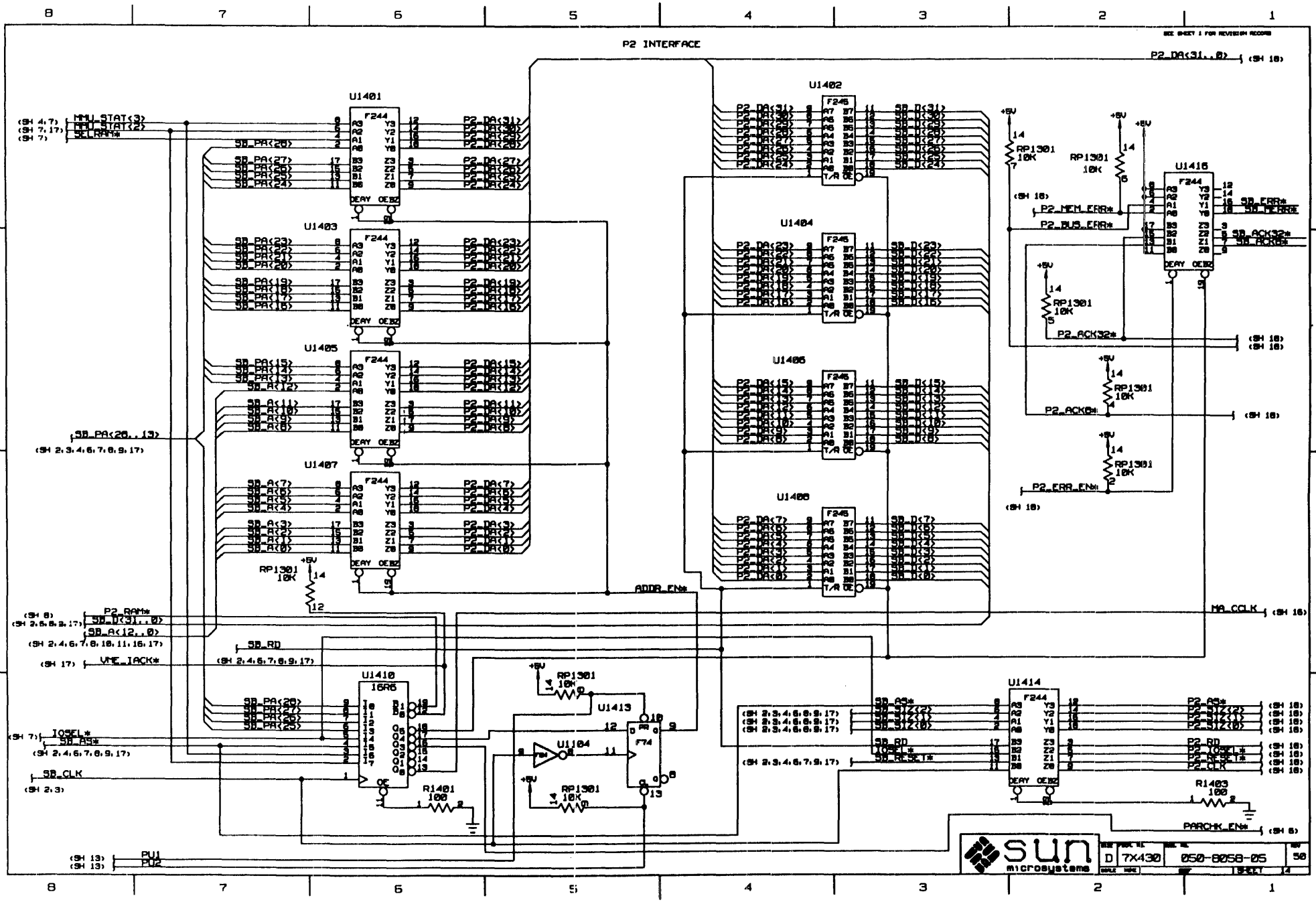

ETHERNET INTERFACE
SEE SHEET 1 FOR REVISION RECORD
+5V
+12V
RP0701
10K
ETHER_IRQ*
IO_A(1)
TCLK
SB_D(31..0)
MISC_CLK
U0901
S4_DMA
U0902
AM7990
LANCE
U0903
AM7992
R0901 40.2
R0902 40.2
R0903 40.2
R0904 40.2
T0901
TRANSF
J0901
DB15
F0901
3A
L0901
1.8
R0907 33
C0910
100PF
RP0901
4.7K
R0906
3K
R0905
510
C0901
.1UF
C0902
5600PF
C0909
600 PF
X0901
20MHZ
C0905
100PF
C0906
100PF
C0907
.1UF
C0908
10UF
U0905
ESP_REQ
MASKED_REQ
SCSI(0..6)
SCD(7..0)
SB_SIZ(2..0)
SUN MICROSYSTEMS
D
7X430
050-8058-05
50
SHEET 9

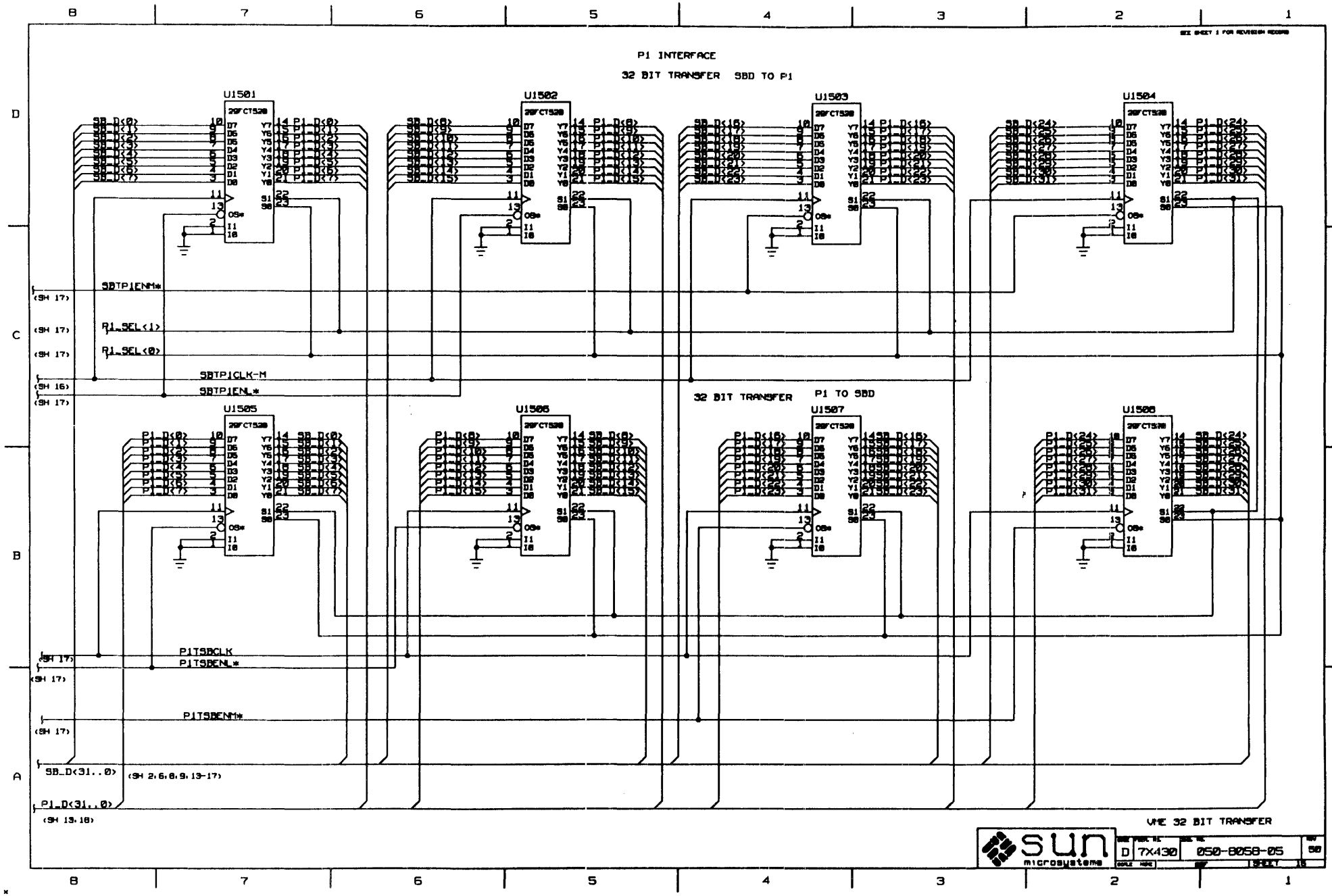

SCSI INTERFACE
U1001
53C90
SCSI
J1001
CONN50
F1001
3A
CR1001
MBR050
+5V
C1001
.1UF
R1001
1K
R1002
1K
RP1003
330 220
RP1004
330 220
SCD<7..0>
SCSI<6..0>
MISC_CLK
ATN*
BSY*
ACK*
RST*
MSG*
SEL*
CD*
REQ*
IO*

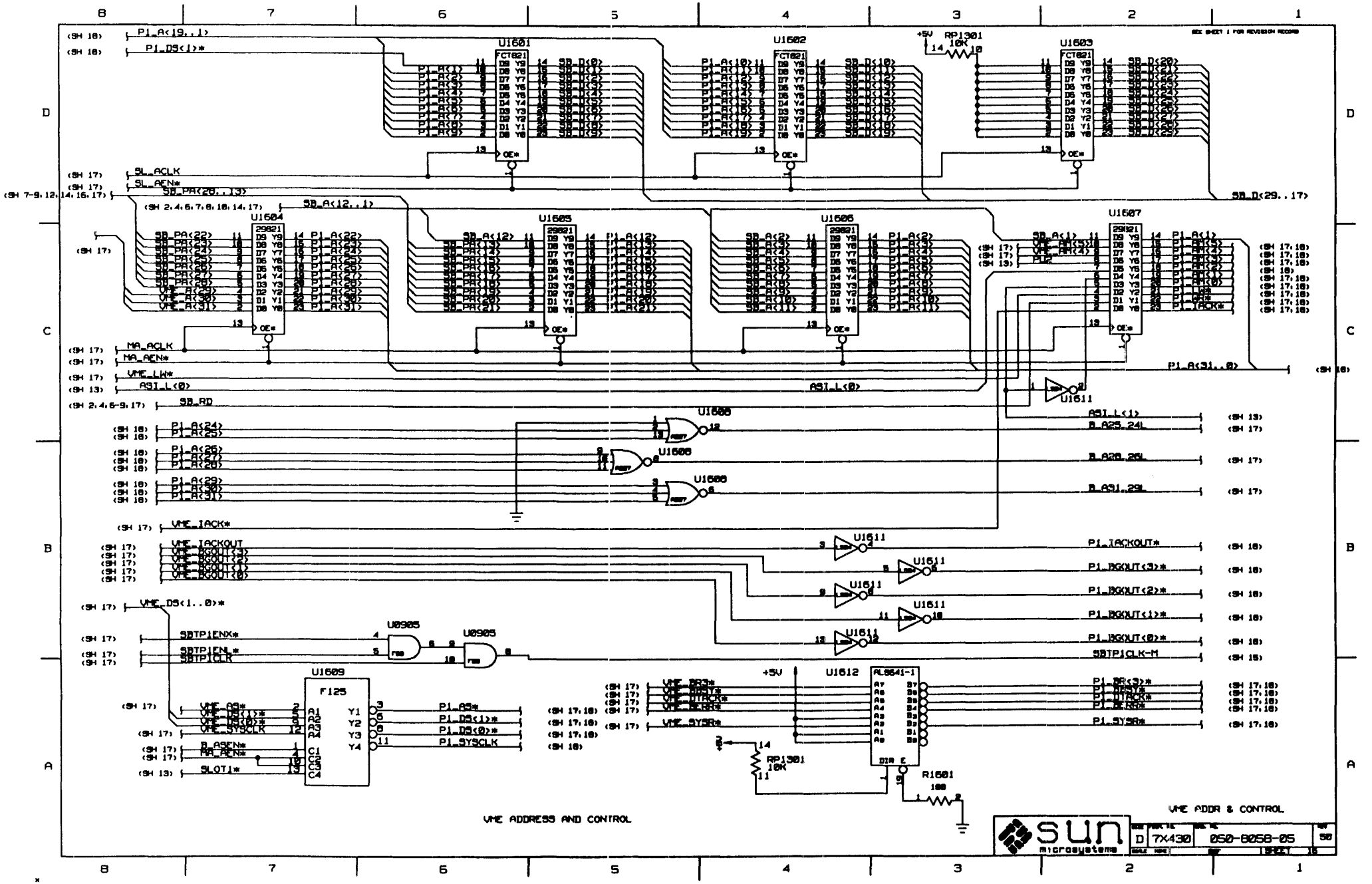

TOD CLOCK
(SH 4,6,7,12) IOD<7..0>
SB_A<17..0>
(SH 2,4,6,7,8,10,14,16,17)
U1101
MK48T02
TOD
2IOOOI8
(SH 2,6,7,8,9,14) SB_RD
(SH 7) TOD_CS*
R1101
J1101
DB15
(SH 12) RXD0I
(SH 12) TXD0O
(SH 12) RXD1I
(SH 12) TXD1O
+5V
F1101
3A
L1101
1.8MH
U1105
LS365
R1103
10K
R1102
J1102
J1103
(SH 7) EPROM_RD*
IO_A<4..0>
(SH 3,7,9,18)
TOD, PROM
sun microsystems
D
7X430
050-8058-05
50
SHEET 11


SERIAL PORTS A & B
KEYBD&MOUSE
PORT A
PORT B
DRIVERS
RECEIVERS
U1201
U1202
U1203
U1204
U1205
U1206
U1207
U1208
U1209
U1210
U1211
J1201
J1202
DB15
8530 SCC
IOD<7..0>
SCC_RD*
SCC_WR*
IO_A<2>
IO_A<1>
SCLK4
SCCIRQ*
KBD_RD*
KBD_WR*
RP0302 10K
R1201 100
R1209 100
R1210 100
R1208 4.7K
RP1201 4.7K
+12V
-12V
+5V
TXDA0
RTSA0
DTRA0
TXCB0
TXDB0
RTSB0
DTRB0
TRXBI
CTSAI
DCDAI
RXDAI
DSRAI
DCDBI
CTSBI
RXDBI
DSRBI
TXD00
RXD01
TXD10
RXD11
SUN microsystems
7X430
050-0058-05


U1301
PA19_15H*
U1302
F174
RP1301
10K
+5V
+RV
U1308
F256
U1413
F74
U1104
RAMCLK
SHOLD*
PU1
PU2
POK
MHOLD_D
IU_AS1<0>
IU_AS1<1>
AS1_D<0>
AS1_D<1>
PUR*
AS1_L<0>
AS1_L<1>
SB_D<31..0>
P1_D<15..0>
8+16 BIT P1 DATA XFER
U1306
U1307
25FCT52
SBTP1ENX*
P1TSBENX*
P1TSBCLK
SBTP1CLK
VME 8+16 BIT DATA
sun microsystems
7X430
050-8058-05
50


P2 INTERFACE
P2_DA(31..0)
U1401
U1402
U1403
U1404
U1405
U1406
U1407
U1408
U1410
U1413
U1414
U1416
F244
F245
F74
16R6
RP1301
10K
R1401
R1403
100
SB_PA(20..13)
SB_A(12..0)
SB_D(31..0)
SB_RD
P2_RAM*
VME_IACK*
ADDR_EN*
MA_CCLK
P2_MEM_ERR*
P2_BUS_ERR*
P2_ACK32*
P2_ACK8*
P2_ERR_EN*
IOSEL*
SB_AS*
SB_CLK
SB_RESET*
P2_CLK
PARCHK_EN*
PU1
PU2
sun microsystems
D
7X430
050-8058-05
50
SHEET 14


P1 INTERFACE
32 BIT TRANSFER SBD TO P1
U1501
U1502
U1503
U1504
U1505
U1506
U1507
U1508
29FCT520
SBTP1ENM*
P1_SEL<1>
P1_SEL<0>
SBTP1CLK-M
SBTP1ENL*
32 BIT TRANSFER P1 TO SBD
P1TSBCLK
P1TSBENL*
P1TSBENM*
SB_D<31..0>
P1_D<31..0>
VME 32 BIT TRANSFER
sun microsystems
D 7X430
050-0058-05

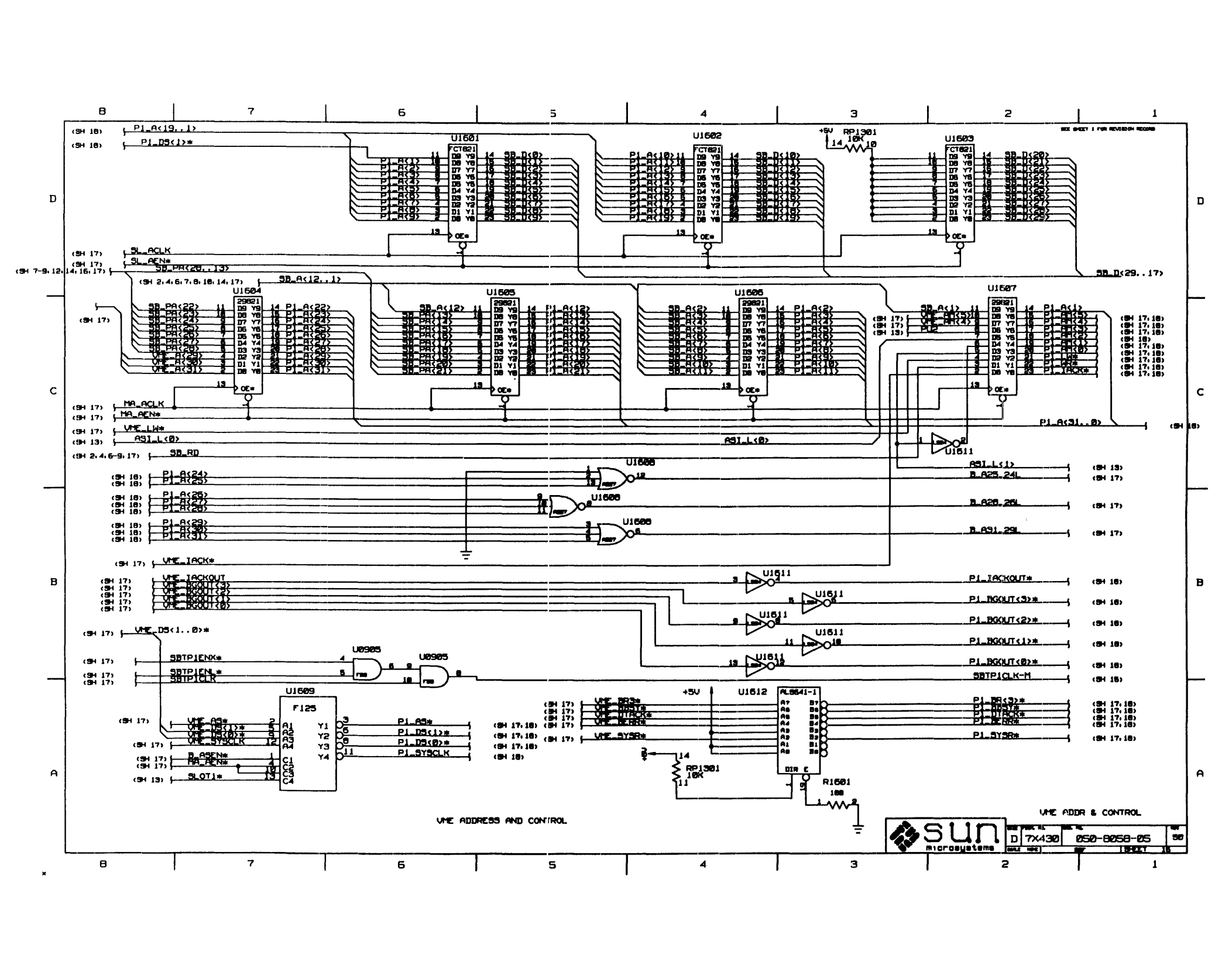

VME ADDRESS AND CONTROL
VME ADDR & CONTROL
sun microsystems
D
7X430
050-0058-05
SHEET 16

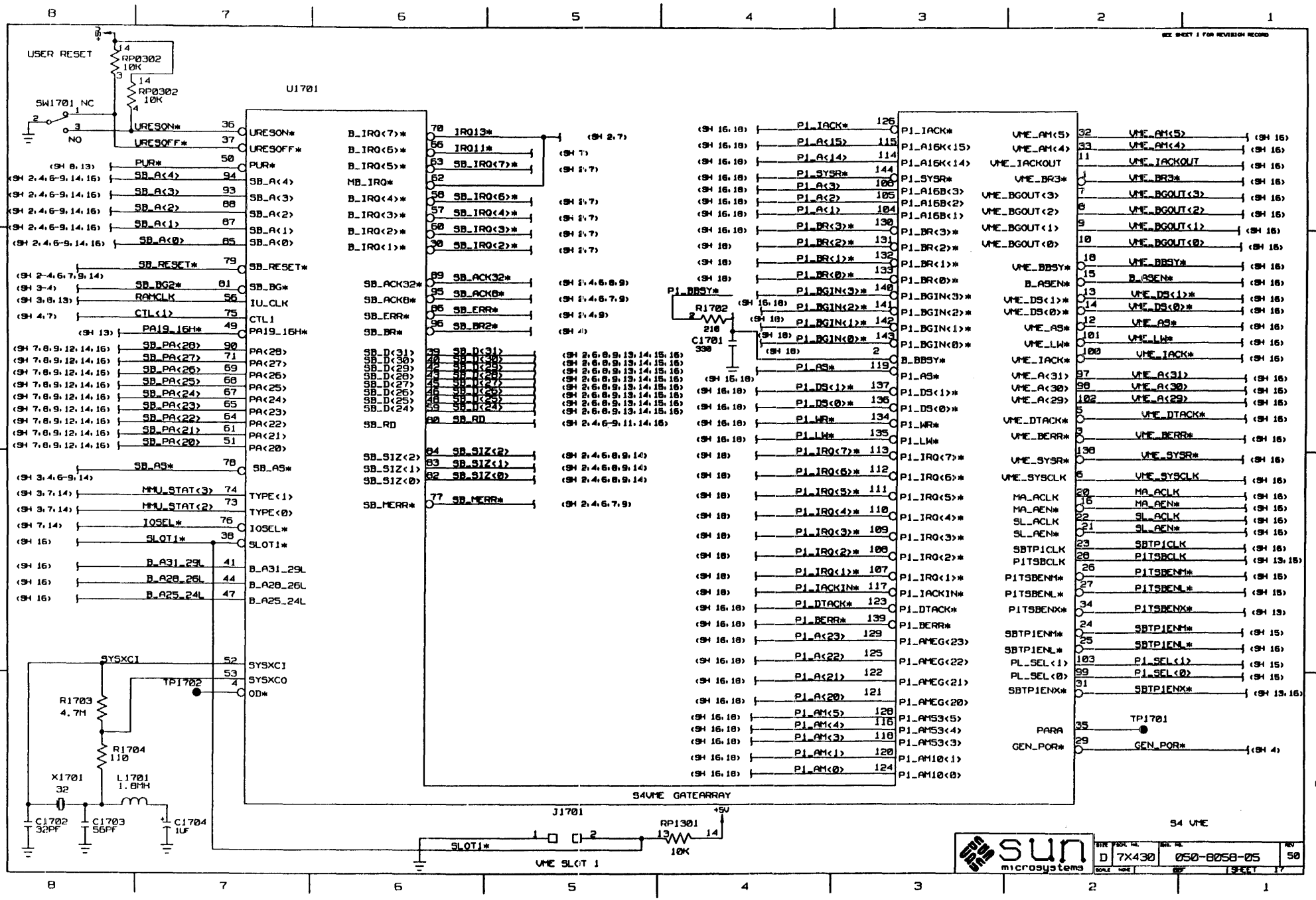
VME P1 CONNECTOR

P1 - 1	P1_D<0>	P1 - 33	P1_BBSY*	P1 - 65	P1_D<8>
P1 - 2	P1_D<1>	P1 - 34	NC	P1 - 66	P1_D<9>
P1 - 3	P1_D<2>	P1 - 35	NC	P1 - 67	P1_D<10>
P1 - 4	P1_D<3>	P1 - 36	P1_BGIN<0>*	P1 - 68	P1_D<11>
P1 - 5	P1_D<4>	P1 - 37	P1_BGOUT<0>*	P1 - 69	P1_D<12>
P1 - 6	P1_D<5>	P1 - 38	P1_BGIN<1>*	P1 - 70	P1_D<13>
P1 - 7	P1_D<6>	P1 - 39	P1_BGOUT<1>*	P1 - 71	P1_D<14>
P1 - 8	P1_D<7>	P1 - 40	P1_BGIN<2>*	P1 - 72	P1_D<15>
P1 - 9	GND/L	P1 - 41	P1_BGOUT<2>*	P1 - 73	GND/L
P1 - 10	P1_SYSCLK	P1 - 42	P1_BGIN<3>*	P1 - 74	NC
P1 - 11	GND/L	P1 - 43	P1_BGOUT<3>*	P1 - 75	P1_BERR*
P1 - 12	P1_DS<1>*	P1 - 44	P1_BR<0>*	P1 - 76	P1_SYSR*
P1 - 13	P1_DS<0>*	P1 - 45	P1_BR<1>*	P1 - 77	P1_LL*
P1 - 14	P1_WR*	P1 - 46	P1_BR<2>*	P1 - 78	P1_AM<5>
P1 - 15	GND/L	P1 - 47	P1_BR<3>*	P1 - 79	P1_A<23>
P1 - 16	P1_DTACK*	P1 - 48	P1_AM<0>	P1 - 80	P1_A<22>
P1 - 17	GND/L	P1 - 49	P1_AM<1>	P1 - 81	P1_A<21>
P1 - 18	P1_AS*	P1 - 50	P1_AM<2>	P1 - 82	P1_A<20>
P1 - 19	GND/L	P1 - 51	P1_AM<3>	P1 - 83	P1_A<19>
P1 - 20	P1_IACK*	P1 - 52	GND/L	P1 - 84	P1_A<18>
P1 - 21	P1_IACKIN*	P1 - 53	NC	P1 - 85	P1_A<17>
P1 - 22	P1_IACKOUT*	P1 - 54	NC	P1 - 86	P1_A<16>
P1 - 23	P1_AM<4>	P1 - 55	GND/L	P1 - 87	P1_A<15>
P1 - 24	P1_A<7>	P1 - 56	P1_IRQ<7>*	P1 - 88	P1_A<14>
P1 - 25	P1_A<6>	P1 - 57	P1_IRQ<6>*	P1 - 89	P1_A<13>
P1 - 26	P1_A<5>	P1 - 58	P1_IRQ<5>*	P1 - 90	P1_A<12>
P1 - 27	P1_A<4>	P1 - 59	P1_IRQ<4>*	P1 - 91	P1_A<11>
P1 - 28	P1_A<3>	P1 - 60	P1_IRQ<3>*	P1 - 92	P1_A<10>
P1 - 29	P1_A<2>	P1 - 61	P1_IRQ<2>*	P1 - 93	P1_A<9>
P1 - 30	P1_A<1>	P1 - 62	P1_IRQ<1>*	P1 - 94	P1_A<8>
P1 - 31	-12V/L	P1 - 63	NC	P1 - 95	+12V/L
P1 - 32	VCC	P1 - 64	VCC	P1 - 96	VCC

VME P2 CONNECTOR

P2 - 1	PAR<0>	P2 - 33	VCC	P2 - 65	PAR<1>
P2 - 2	P2_ACK0*	P2 - 34	GND/L	P2 - 66	GND/L
P2 - 3	P2_ACK32*	P2 - 35	RESERVED	P2 - 67	P2_CLK
P2 - 4	P2_SIZ<0>	P2 - 36	P1_A<24>	P2 - 68	P2_MEM_ERR*
P2 - 5	P2_SIZ<1>	P2 - 37	P1_A<25>	P2 - 69	PAR<3>
P2 - 6	PAR<2>	P2 - 38	P1_A<26>	P2 - 70	SB_IRQ<5>*
P2 - 7	P2_RD	P2 - 39	P1_A<27>	P2 - 71	GND/L
P2 - 8	P2_SIZ<2>	P2 - 40	P1_A<28>	P2 - 72	P2_IOSEL*
P2 - 9	P2_BUS_ERR*	P2 - 41	P1_A<29>	P2 - 73	P2_RESET*
P2 - 10	SELRAM*	P2 - 42	P1_A<30>	P2 - 74	PUR*
P2 - 11	P2_AS*	P2 - 43	P1_A<31>	P2 - 75	P2_ERR_EN*
P2 - 12	GND/L	P2 - 44	GND/L	P2 - 76	GND/L
P2 - 13	P2_DA<0>	P2 - 45	VCC	P2 - 77	P2_DA<16>
P2 - 14	P2_DA<1>	P2 - 46	P1_D<16>	P2 - 78	P2_DA<17>
P2 - 15	P2_DA<2>	P2 - 47	P1_D<17>	P2 - 79	P2_DA<18>
P2 - 16	P2_DA<3>	P2 - 48	P1_D<18>	P2 - 80	P2_DA<19>
P2 - 17	GND/L	P2 - 49	P1_D<19>	P2 - 81	GND/L
P2 - 18	P2_DA<4>	P2 - 50	P1_D<20>	P2 - 82	P2_DA<20>
P2 - 19	P2_DA<5>	P2 - 51	P1_D<21>	P2 - 83	P2_DA<21>
P2 - 20	P2_DA<6>	P2 - 52	P1_D<22>	P2 - 84	P2_DA<22>
P2 - 21	P2_DA<7>	P2 - 53	P1_D<23>	P2 - 85	P2_DA<23>
P2 - 22	GND/L	P2 - 54	GND/L	P2 - 86	GND/L
P2 - 23	P2_DA<8>	P2 - 55	P1_D<24>	P2 - 87	P2_DA<24>
P2 - 24	P2_DA<9>	P2 - 56	P1_D<25>	P2 - 88	P2_DA<25>
P2 - 25	P2_DA<10>	P2 - 57	P1_D<26>	P2 - 89	P2_DA<26>
P2 - 26	P2_DA<11>	P2 - 58	P1_D<27>	P2 - 90	P2_DA<27>
P2 - 27	GND/L	P2 - 59	P1_D<28>	P2 - 91	GND/L
P2 - 28	P2_DA<12>	P2 - 60	P1_D<29>	P2 - 92	P2_DA<28>
P2 - 29	P2_DA<13>	P2 - 61	P1_D<30>	P2 - 93	P2_DA<29>
P2 - 30	P2_DA<14>	P2 - 62	P1_D<31>	P2 - 94	P2_DA<30>
P2 - 31	P2_DA<15>	P2 - 63	GND/L	P2 - 95	P2_DA<31>
P2 - 32	GND/L	P2 - 64	VCC	P2 - 96	GND/L


U1601
U1602
U1603
U1604
U1605
U1606
U1607
U1608
U1609
U1610
U1611
U1612
U0905
U0905
RP1301
RP1301
R1601
F125
AS641-1
FCT821
29821
P1_A<19..1>
P1_DS<1>*
SL_ACLK
SL_AEN*
SB_PA<29..13>
SB_A<12..1>
SB_D<29..17>
MA_ACLK
MA_AEN*
VME_LW*
ASI_L<0>
SB_RD
P1_A<31..0>
VME_IACK*
VME_IACKOUT
VME_BGOUT<3>
VME_BGOUT<2>
VME_BGOUT<1>
VME_BGOUT<0>
VME_DS<1..0>*
SBTP1ENX*
SBTP1ENL*
SBTP1CLK
VME_AS*
VME_DS<1>*
VME_DS<0>*
VME_SYSCLK
B_ASEN*
MA_AEN*
SLOT1*
P1_AS*
P1_DS<1>*
P1_DS<0>*
P1_SYSCLK
VME_BR3*
VME_BBSY*
VME_DTACK*
VME_BERR*
VME_SYSR*
P1_BR3*
P1_BBSY*
P1_DTACK*
P1_BERR*
P1_SYSR*
+5V
ASI_L<1>
B_A25_24L
B_A26_26L
B_A31_29L
P1_IACKOUT*
P1_BGOUT<3>*
P1_BGOUT<2>*
P1_BGOUT<1>*
P1_BGOUT<0>*
SBTP1CLK-M
P1_IACK*
VME ADDRESS AND CONTROL
VME ADDR & CONTROL
sun microsystems
D
7X430
050-0058-05
50
SHEET 16

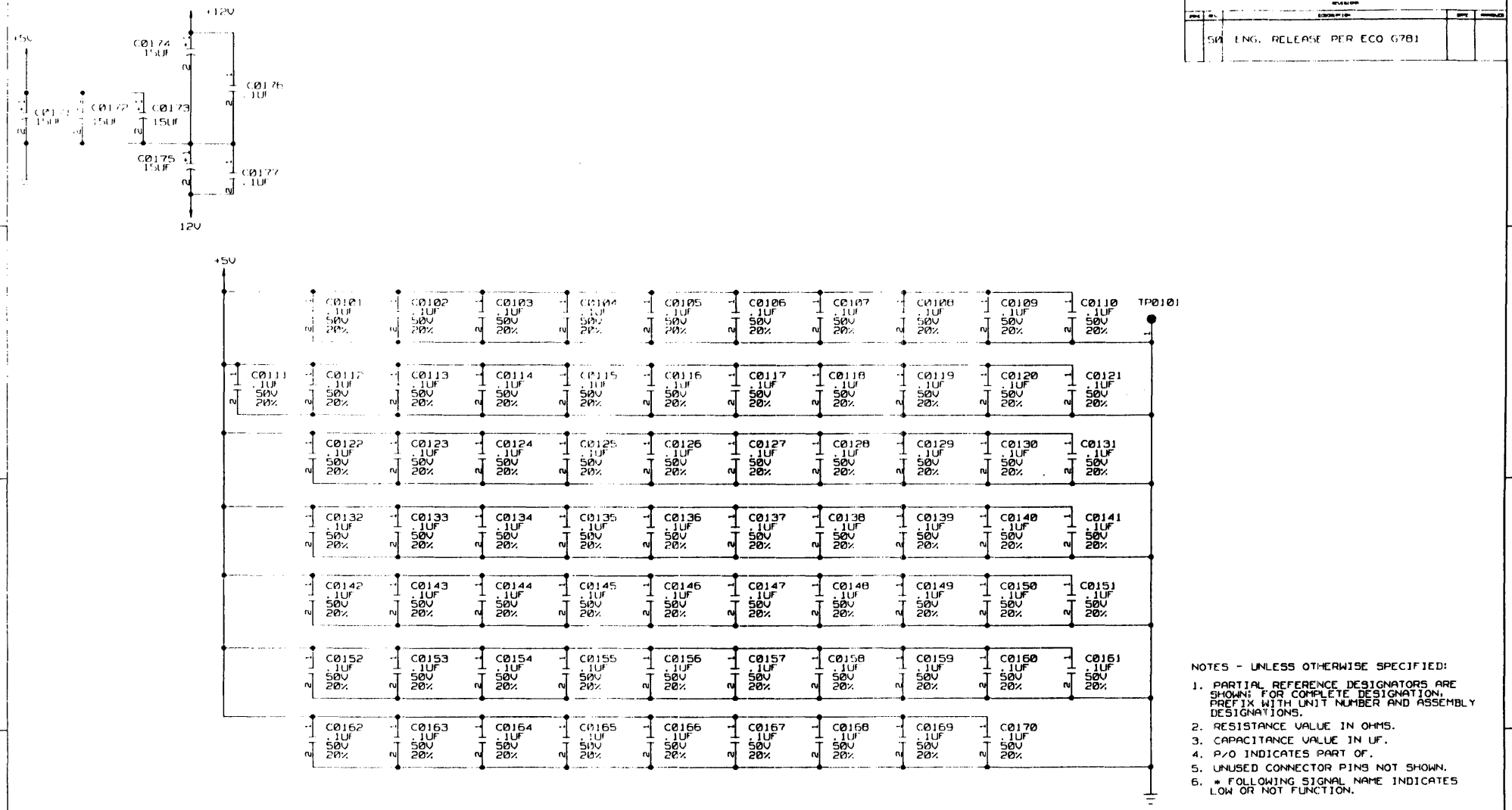

USER RESET
RP0302 10K
SW1701 NC
NO
U1701
S4VME GATEARRAY
J1701
VME SLOT 1
SYSXCI
SYSXCO
TP1702
R1703 4.7M
R1704 110
X1701 32
L1701 1.8MH
C1702 32PF
C1703 56PF
C1704 1UF
R1702 220
C1701 330
RP1301 10K
TP1701
PARA
GEN_POR*
S4 VME
SUN microsystems
D
7X430
050-8058-05
50
SHEET 17

VME P1 CONNECTOR

P1 - 1	P1_D<0>	P1 - 33	P1_BBSY*	P1 - 65	P1_D<8>
P1 - 2	P1_D<1>	P1 - 34	NC	P1 - 66	P1_D<9>
P1 - 3	P1_D<2>	P1 - 35	NC	P1 - 67	P1_D<10>
P1 - 4	P1_D<3>	P1 - 36	P1_BGIN<0>*	P1 - 68	P1_D<11>
P1 - 5	P1_D<4>	P1 - 37	P1_BGOUT<0>*	P1 - 69	P1_D<12>
P1 - 6	P1_D<5>	P1 - 38	P1_BGIN<1>*	P1 - 70	P1_D<13>
P1 - 7	P1_D<6>	P1 - 39	P1_BGOUT<1>*	P1 - 71	P1_D<14>
P1 - 8	P1_D<7>	P1 - 40	P1_BGIN<2>*	P1 - 72	P1_D<15>
P1 - 9	GND/L	P1 - 41	P1_BGOUT<2>*	P1 - 73	GND/L
P1 - 10	P1_SYSCLK	P1 - 42	P1_BGIN<3>*	P1 - 74	NC
P1 - 11	GND/L	P1 - 43	P1_BGOUT<3>*	P1 - 75	P1_BERR*
P1 - 12	P1_DS<1>*	P1 - 44	P1_BR<0>*	P1 - 76	P1_SYSR*
P1 - 13	P1_DS<0>*	P1 - 45	P1_BR<1>*	P1 - 77	P1_LI*
P1 - 14	P1_WR*	P1 - 46	P1_BR<2>*	P1 - 78	P1_AM<5>
P1 - 15	GND/L	P1 - 47	P1_BR<3>*	P1 - 79	P1_A<23>
P1 - 16	P1_DTACK*	P1 - 48	P1_AM<0>	P1 - 80	P1_A<22>
P1 - 17	GND/L	P1 - 49	P1_AM<1>	P1 - 81	P1_A<21>
P1 - 18	P1_AS*	P1 - 50	P1_AM<2>	P1 - 82	P1_A<20>
P1 - 19	GND/L	P1 - 51	P1_AM<3>	P1 - 83	P1_A<19>
P1 - 20	P1_IACK*	P1 - 52	GND/L	P1 - 84	P1_A<18>
P1 - 21	P1_IACKIN*	P1 - 53	NC	P1 - 85	P1_A<17>
P1 - 22	P1_IACKOUT*	P1 - 54	NC	P1 - 86	P1_A<16>
P1 - 23	P1_AM<4>	P1 - 55	GND/L	P1 - 87	P1_A<15>
P1 - 24	P1_A<7>	P1 - 56	P1_IRQ<7>*	P1 - 88	P1_A<14>
P1 - 25	P1_A<6>	P1 - 57	P1_IRQ<6>*	P1 - 89	P1_A<13>
P1 - 26	P1_A<5>	P1 - 58	P1_IRQ<5>*	P1 - 90	P1_A<12>
P1 - 27	P1_A<4>	P1 - 59	P1_IRQ<4>*	P1 - 91	P1_A<11>
P1 - 28	P1_A<3>	P1 - 60	P1_IRQ<3>*	P1 - 92	P1_A<10>
P1 - 29	P1_A<2>	P1 - 61	P1_IRQ<2>*	P1 - 93	P1_A<9>
P1 - 30	P1_A<1>	P1 - 62	P1_IRQ<1>*	P1 - 94	P1_A<8>
P1 - 31	-12V/L	P1 - 63	NC	P1 - 95	+12V/L
P1 - 32	VCC	P1 - 64	VCC	P1 - 96	VCC

VME P2 CONNECTOR

P2 - 1	PAR<0>	P2 - 33	VCC	P2 - 65	PAR<1>
P2 - 2	P2_ACK8*	P2 - 34	GND/L	P2 - 66	GND/L
P2 - 3	P2_ACK32*	P2 - 35	RESERVED	P2 - 67	P2_CLK
P2 - 4	P2_SIZ<0>	P2 - 36	P1_A<24>	P2 - 68	P2_MEM_ERR*
P2 - 5	P2_SIZ<1>	P2 - 37	P1_A<25>	P2 - 69	PAR<3>
P2 - 6	PAR<2>	P2 - 38	P1_A<26>	P2 - 70	SB_IRQ<5>*
P2 - 7	P2_RD	P2 - 39	P1_A<27>	P2 - 71	GND/L
P2 - 8	P2_SIZ<2>	P2 - 40	P1_A<28>	P2 - 72	P2_IOSEL*
P2 - 9	P2_BUS_ERR*	P2 - 41	P1_A<29>	P2 - 73	P2_RESET*
P2 - 10	SELRAM*	P2 - 42	P1_A<30>	P2 - 74	PWR*
P2 - 11	P2_AS*	P2 - 43	P1_A<31>	P2 - 75	P2_ERR_EN*
P2 - 12	GND/L	P2 - 44	GND/L	P2 - 76	GND/L
P2 - 13	P2_DA<0>	P2 - 45	VCC	P2 - 77	P2_DA<16>
P2 - 14	P2_DA<1>	P2 - 46	P1_D<16>	P2 - 78	P2_DA<17>
P2 - 15	P2_DA<2>	P2 - 47	P1_D<17>	P2 - 79	P2_DA<18>
P2 - 16	P2_DA<3>	P2 - 48	P1_D<18>	P2 - 80	P2_DA<19>
P2 - 17	GND/L	P2 - 49	P1_D<19>	P2 - 81	GND/L
P2 - 18	P2_DA<4>	P2 - 50	P1_D<20>	P2 - 82	P2_DA<20>
P2 - 19	P2_DA<5>	P2 - 51	P1_D<21>	P2 - 83	P2_DA<21>
P2 - 20	P2_DA<6>	P2 - 52	P1_D<22>	P2 - 84	P2_DA<22>
P2 - 21	P2_DA<7>	P2 - 53	P1_D<23>	P2 - 85	P2_DA<23>
P2 - 22	GND/L	P2 - 54	GND/L	P2 - 86	GND/L
P2 - 23	P2_DA<8>	P2 - 55	P1_D<24>	P2 - 87	P2_DA<24>
P2 - 24	P2_DA<9>	P2 - 56	P1_D<25>	P2 - 88	P2_DA<25>
P2 - 25	P2_DA<10>	P2 - 57	P1_D<26>	P2 - 89	P2_DA<26>
P2 - 26	P2_DA<11>	P2 - 58	P1_D<27>	P2 - 90	P2_DA<27>
P2 - 27	GND/L	P2 - 59	P1_D<28>	P2 - 91	GND/L
P2 - 28	P2_DA<12>	P2 - 60	P1_D<29>	P2 - 92	P2_DA<28>
P2 - 29	P2_DA<13>	P2 - 61	P1_D<30>	P2 - 93	P2_DA<29>
P2 - 30	P2_DA<14>	P2 - 62	P1_D<31>	P2 - 94	P2_DA<30>
P2 - 31	P2_DA<15>	P2 - 63	GND/L	P2 - 95	P2_DA<31>
P2 - 32	GND/L	P2 - 64	VCC	P2 - 96	GND/L



DWG NO	SH	REV
050-8035-05	1	50

REV	SH	DESCRIPTION	DATE	APPROVED
	50	ENG. RELEASE PER ECO 6781		

NOTES - UNLESS OTHERWISE SPECIFIED:

1. PARTIAL REFERENCE DESIGNATORS ARE SHOWN; FOR COMPLETE DESIGNATION, PREFIX WITH UNIT NUMBER AND ASSEMBLY DESIGNATIONS.
2. RESISTANCE VALUE IN OHMS.
3. CAPACITANCE VALUE IN UF.
4. P/O INDICATES PART OF.
5. UNUSED CONNECTOR PINS NOT SHOWN.
6. * FOLLOWING SIGNAL NAME INDICATES LOW OR NOT FUNCTION.

-1 AS SHOWN

DRAWN	L. RODRIGUES

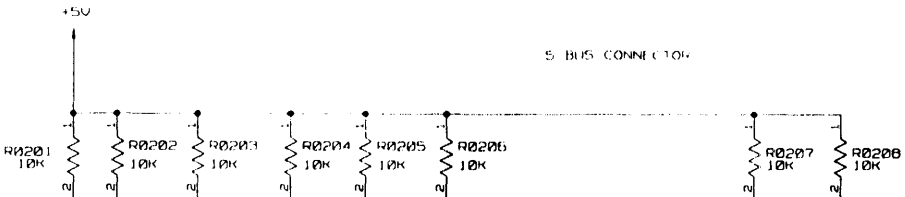
sun microsystems

SCHEMATIC DIAGRAM - SPARC ENGINE 1E WITH EXP & FPA

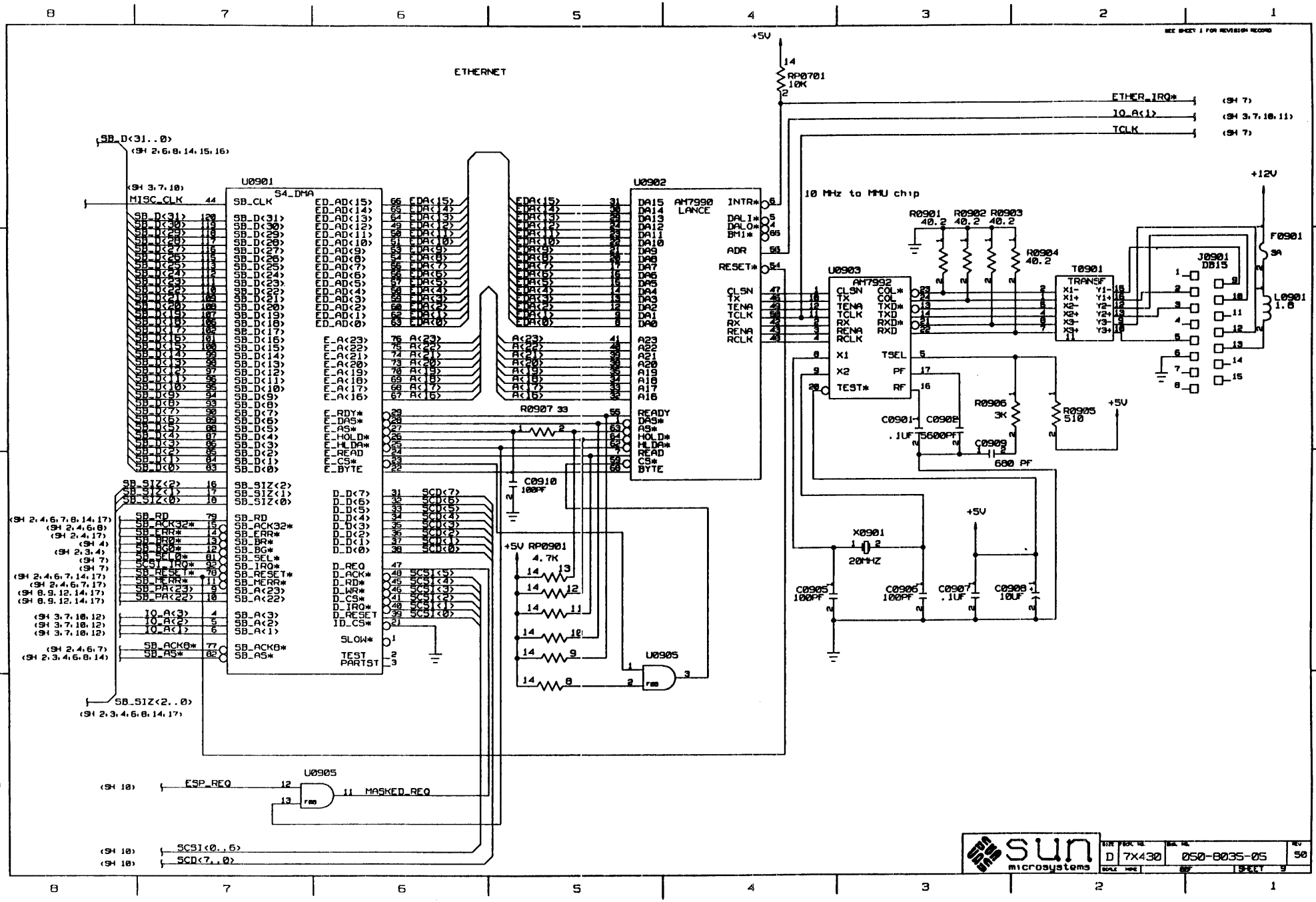
SIZE	BOX NO.	DWG NO.
D	7X430	050-8035-05

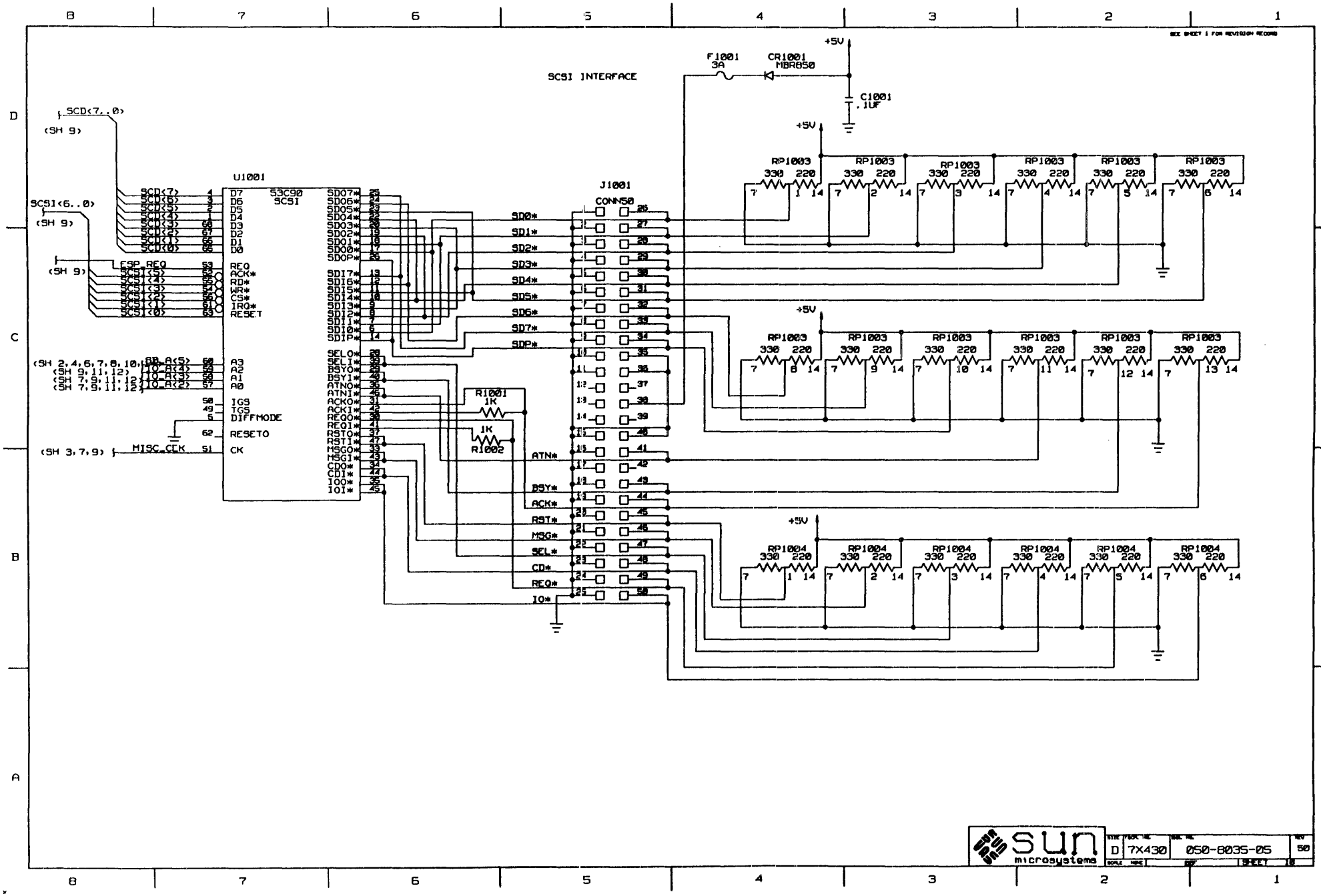
SCALE NONE | SHEET 1 OF 18

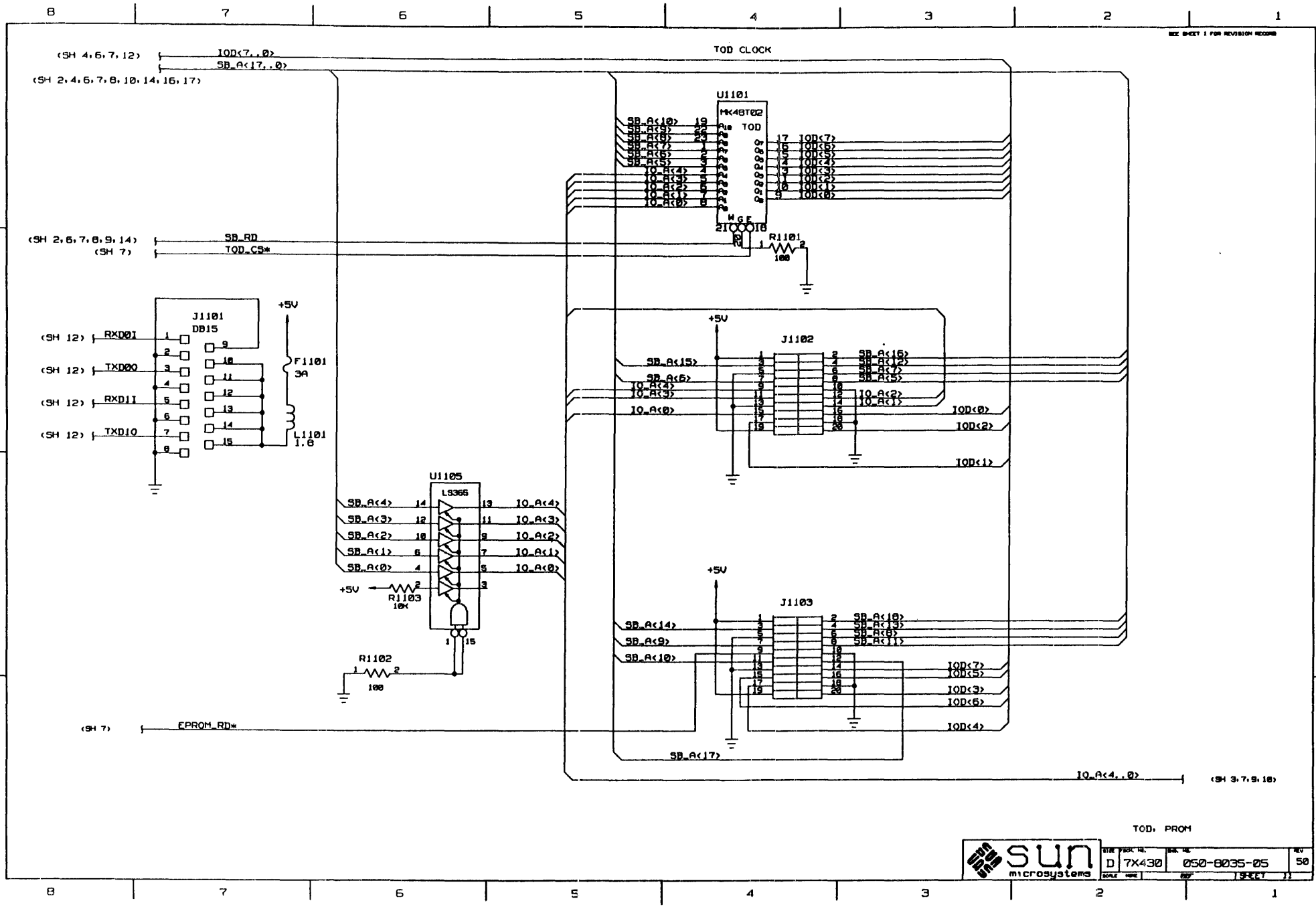
S BUS CONNECTOR

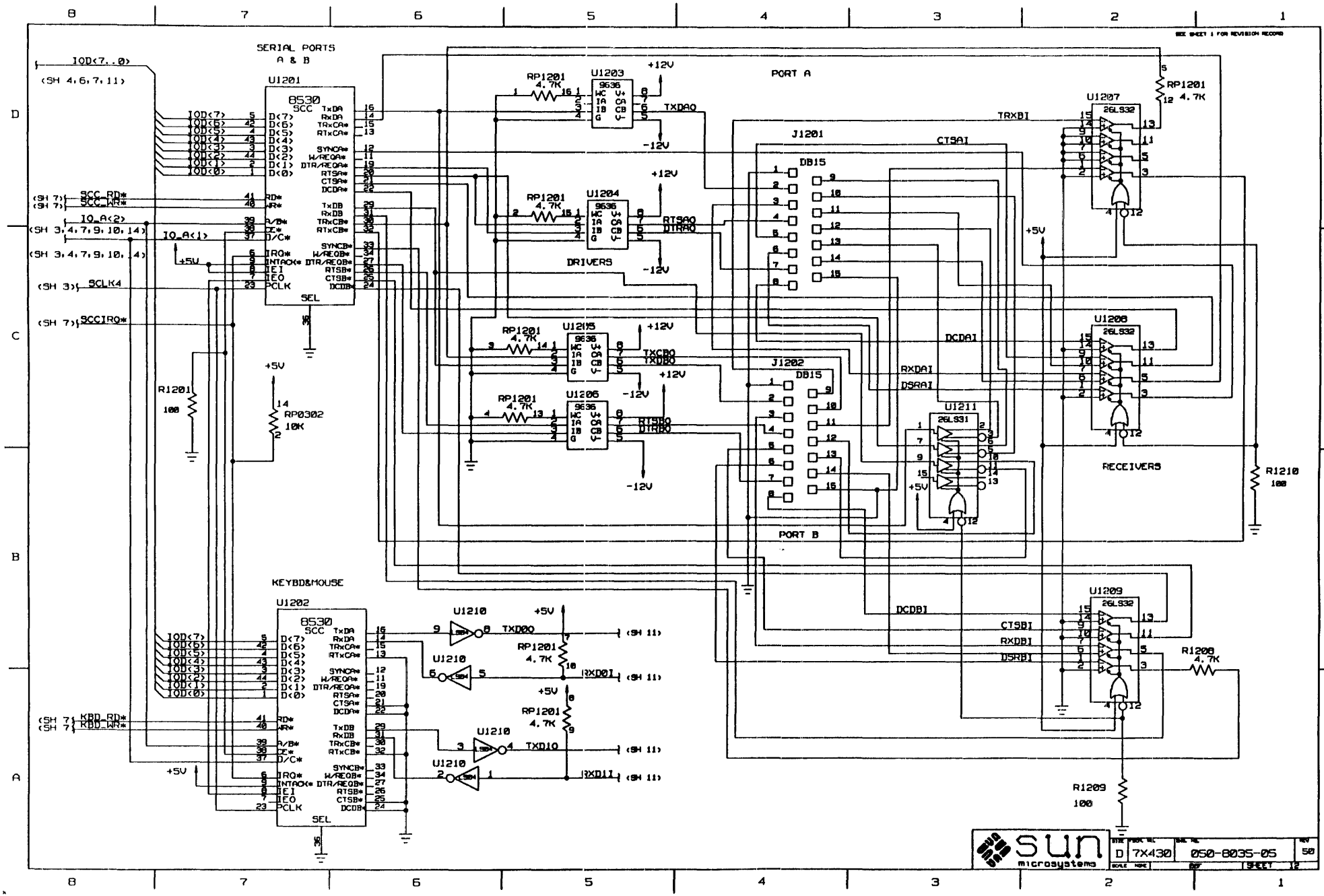


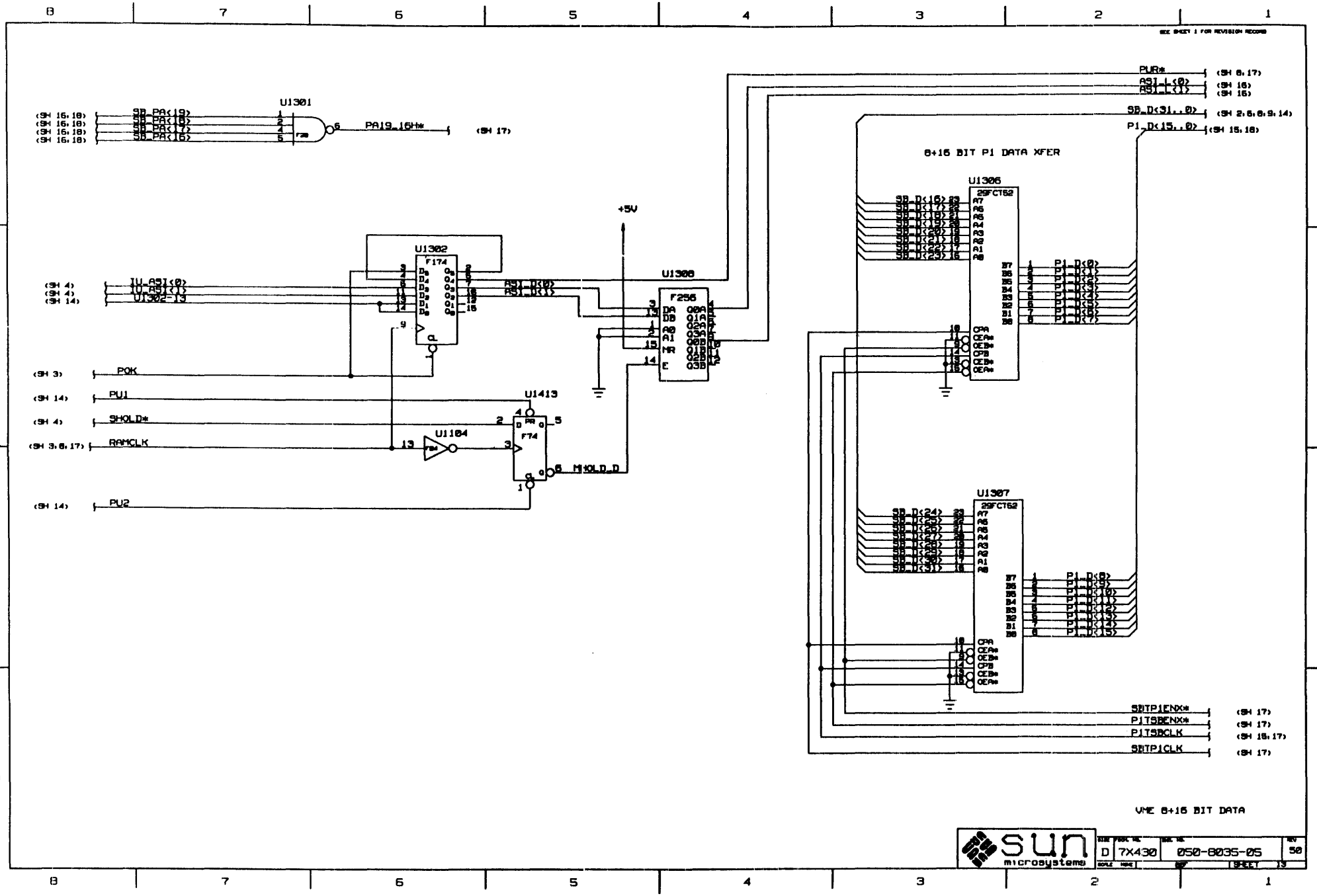
P3 - 1	GND	P3 - 25	SB_D<31>	P3 - 49	SB_CLK	P3 - 73	SB_D<30>
P3 - 2	SB_BG1*	P3 - 26	SB_SIZ<0>	P3 - 50	SB_BG1*	P3 - 74	SB_SIZ<1>
P3 - 3	SB_SEL1*	P3 - 27	SB_SIZ<2>	P3 - 51	SB_AS*	P3 - 75	SB_RD
P3 - 4	SB_IRQ<1>*	P3 - 28	SB_IRQ<7>*	P3 - 52	GND	P3 - 76	GND
P3 - 5	SB_D<1>	P3 - 29	SB_A<0>	P3 - 53	SB_D<1>	P3 - 77	SB_A<1>
P3 - 6	SB_D<2>	P3 - 30	SB_A<2>	P3 - 54	SB_D<3>	P3 - 78	SB_A<3>
P3 - 7	SB_D<4>	P3 - 31	SB_A<4>	P3 - 55	SB_D<5>	P3 - 79	SB_A<5>
P3 - 8	SB_IRQ<2>*	P3 - 32	SB_MERR*	P3 - 56	VCC	P3 - 80	VCC
P3 - 9	SB_D<6>	P3 - 33	SB_A<6>	P3 - 57	SB_D<7>	P3 - 81	SB_A<7>
P3 - 10	SB_D<8>	P3 - 34	SB_A<8>	P3 - 58	SB_D<9>	P3 - 82	SB_A<9>
P3 - 11	SB_D<10>	P3 - 35	SB_A<10>	P3 - 59	SB_D<11>	P3 - 83	SB_A<11>
P3 - 12	SB_IRQ<3>*	P3 - 36	SB_ERR*	P3 - 60	GND	P3 - 84	GND
P3 - 13	SB_D<12>	P3 - 37	SB_A<12>	P3 - 61	SB_D<13>	P3 - 85	SB_PA<13>
P3 - 14	SB_D<14>	P3 - 38	SB_PA<14>	P3 - 62	SB_D<15>	P3 - 86	SB_PA<15>
P3 - 15	SB_D<16>	P3 - 39	SB_PA<16>	P3 - 63	SB_D<17>	P3 - 87	SB_PA<17>
P3 - 16	SB_IRQ<4>*	P3 - 40	SB_ACK8*	P3 - 64	VCC	P3 - 88	VCC
P3 - 17	SB_D<19>	P3 - 41	SB_PA<18>	P3 - 65	SB_D<18>	P3 - 89	SB_PA<19>
P3 - 18	SB_D<21>	P3 - 42	SB_PA<20>	P3 - 66	SB_D<20>	P3 - 90	SB_PA<21>
P3 - 19	SB_D<23>	P3 - 43	SB_PA<22>	P3 - 67	SB_D<22>	P3 - 91	SB_PA<23>
P3 - 20	SB_IRQ<5>*	P3 - 44	SB_ACK32*	P3 - 68	GND	P3 - 92	GND
P3 - 21	SB_D<25>	P3 - 45	SB_PA<24>	P3 - 69	SB_D<24>	P3 - 93	SB_PA<25>
P3 - 22	SB_D<27>	P3 - 46	SB_PA<26>	P3 - 70	SB_D<26>	P3 - 94	SB_PA<27>
P3 - 23	SB_D<29>	P3 - 47	SB_SEL2	P3 - 71	SB_D<28>	P3 - 95	SB_RESET*
P3 - 24	SB_IRQ<6>*	P3 - 48	-12V/L	P3 - 72	VCC	P3 - 96	+12V/L

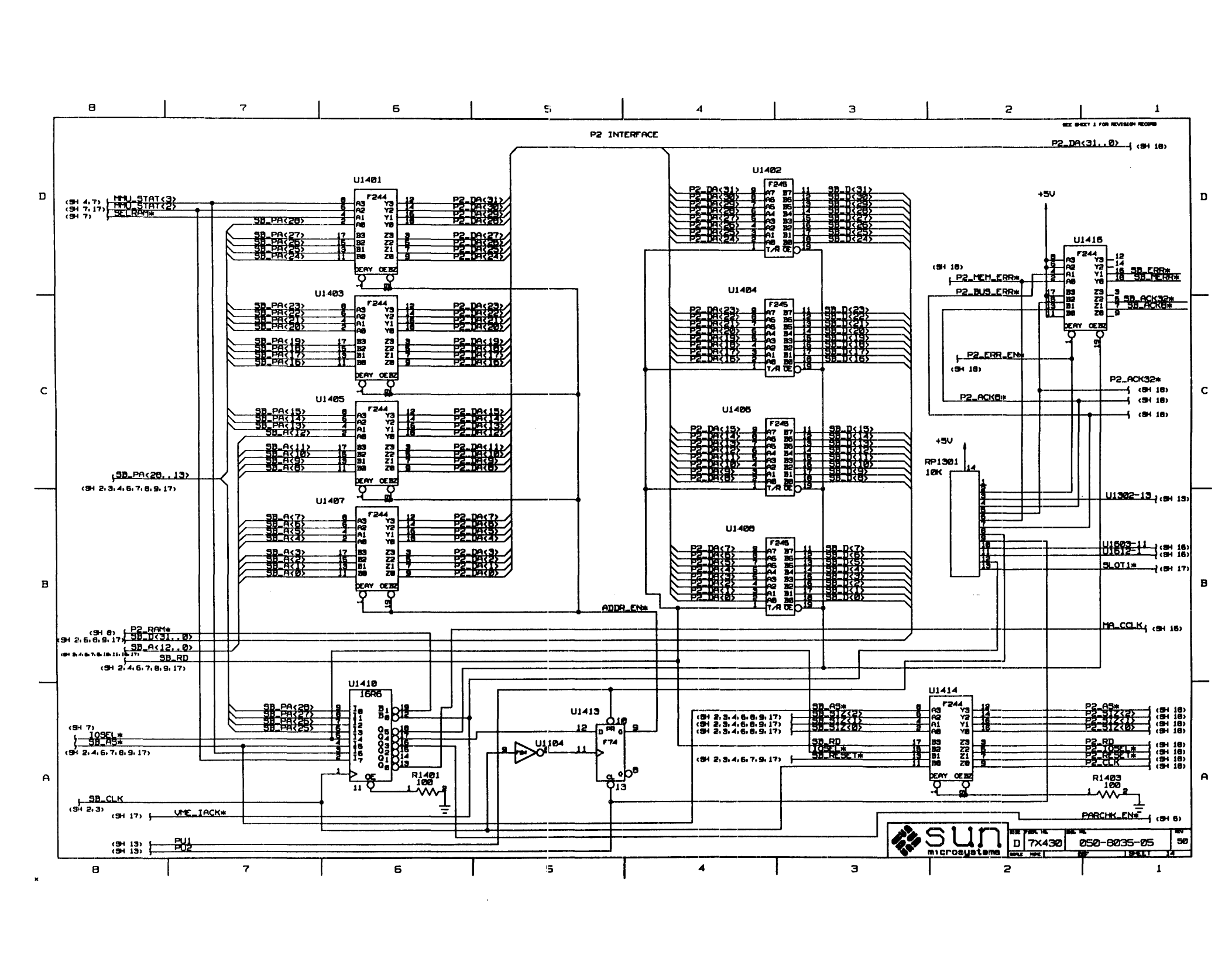

ETHERNET
U0901
S4_DMA
U0902
AM7990 LANCE
U0903
AM7992
U0905
U0905
T0901
TRANSF
J0901
DB15
RP0701
10K
RP0901
4.7K
X0901
20MHZ
R0907 33
R0901 40.2
R0902 40.2
R0903 40.2
R0904 40.2
R0905 510
R0906 3K
C0901 .1UF
C0902 5600PF
C0903 680 PF
C0905 100PF
C0906 100PF
C0907 .1UF
C0908 10UF
C0910 100PF
F0901 3A
L0901 1.8
+5V
+12V
10 MHz to MMU chip
SB_D<31..0>
SB_SIZ<2..0>
ETHER_IRQ*
IO_A<1>
TCLK
ESP_REQ
MASKED_REQ
SCSI<0..6>
SCD<7..0>
SB_CLK
MISC_CLK
SB_RESET*
SB_IRQ*
SB_BR*
SB_BG*
SB_ACK32*
SB_ACK8*
SB_AS*
SB_RD
SB_ERR*
SB_MERR*
SB_PA<23>
SB_PA<22>
SB_A<23>
SB_A<2>
SB_A<1>
E_A<23>
E_A<22>
E_A<21>
E_A<20>
E_A<19>
E_A<18>
E_A<17>
E_A<16>
E_RDY*
E_DAS*
E_AS*
E_HOLD*
E_HLDA*
E_READ
E_CS*
E_BYTE
D_REQ
D_ACK*
D_RD*
D_WR*
D_CS*
D_IRQ*
D_RESET
ID_CS*
SLOW*
TEST
PARTST
INTR*
DALI*
DALO*
BMI*
ADR
RESET*
CLSN
TX
TENA
TCLK
RX
RENA
RCLK
READY
DAS*
AS*
HOLD*
HLDA*
READ
CS*
BYTE
SUN microsystems
050-8035-05
7X430
50

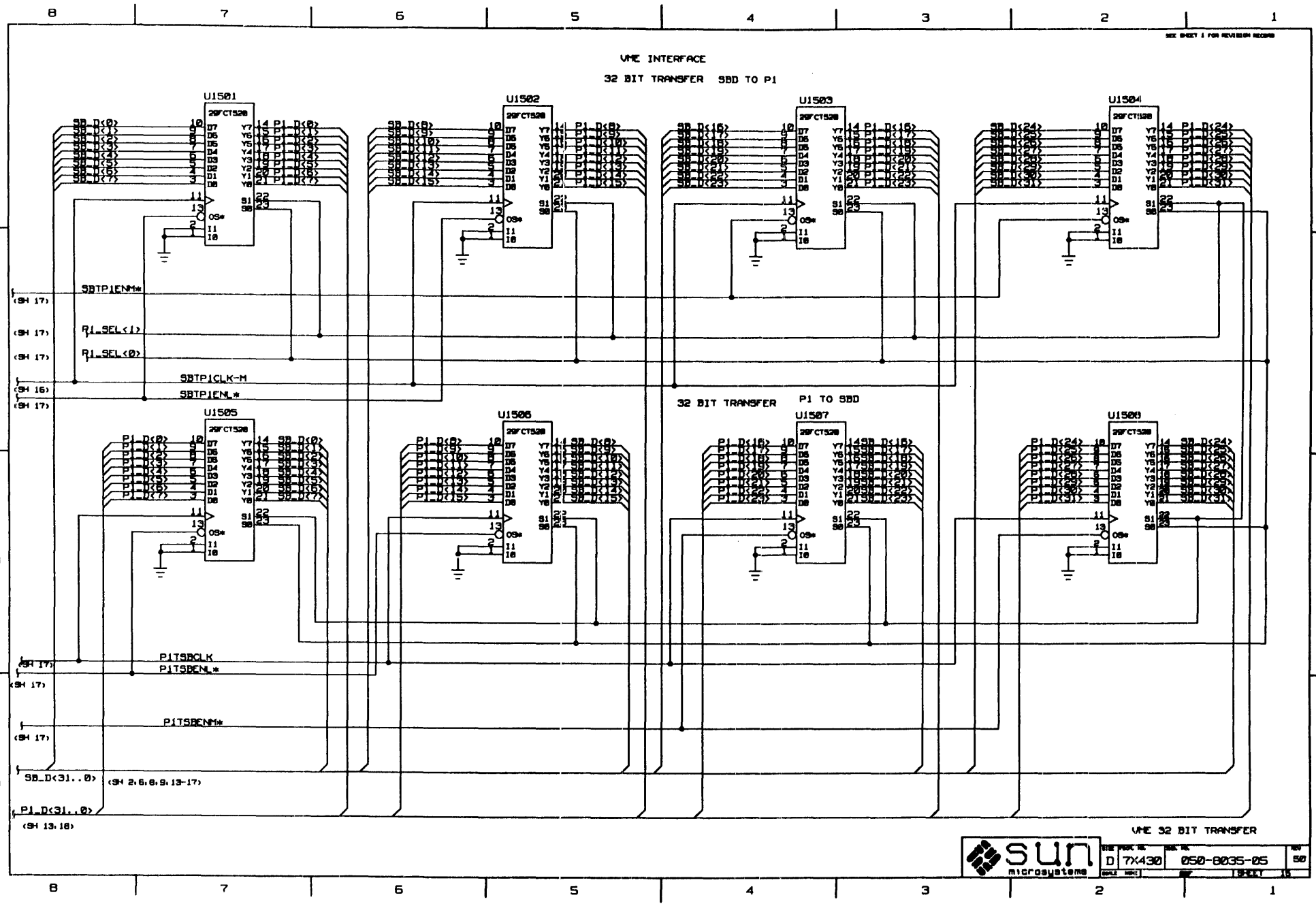

SCSI INTERFACE
F1001
3A
CR1001
MBR850
+5V
C1001
.1UF
U1001
53C90
SCSI
J1001
CONN50
R1001
1K
R1002
1K
RP1003
RP1004
330 220
SCD<7..0>
SCSI<6..0>
ESP_REQ
MISC_CLK
DIFFMODE
RESETO
IGS
TGS
SD0*
SD1*
SD2*
SD3*
SD4*
SD5*
SD6*
SD7*
SDP*
ATN*
BSY*
ACK*
RST*
MSG*
SEL*
CD*
REQ*
IO*

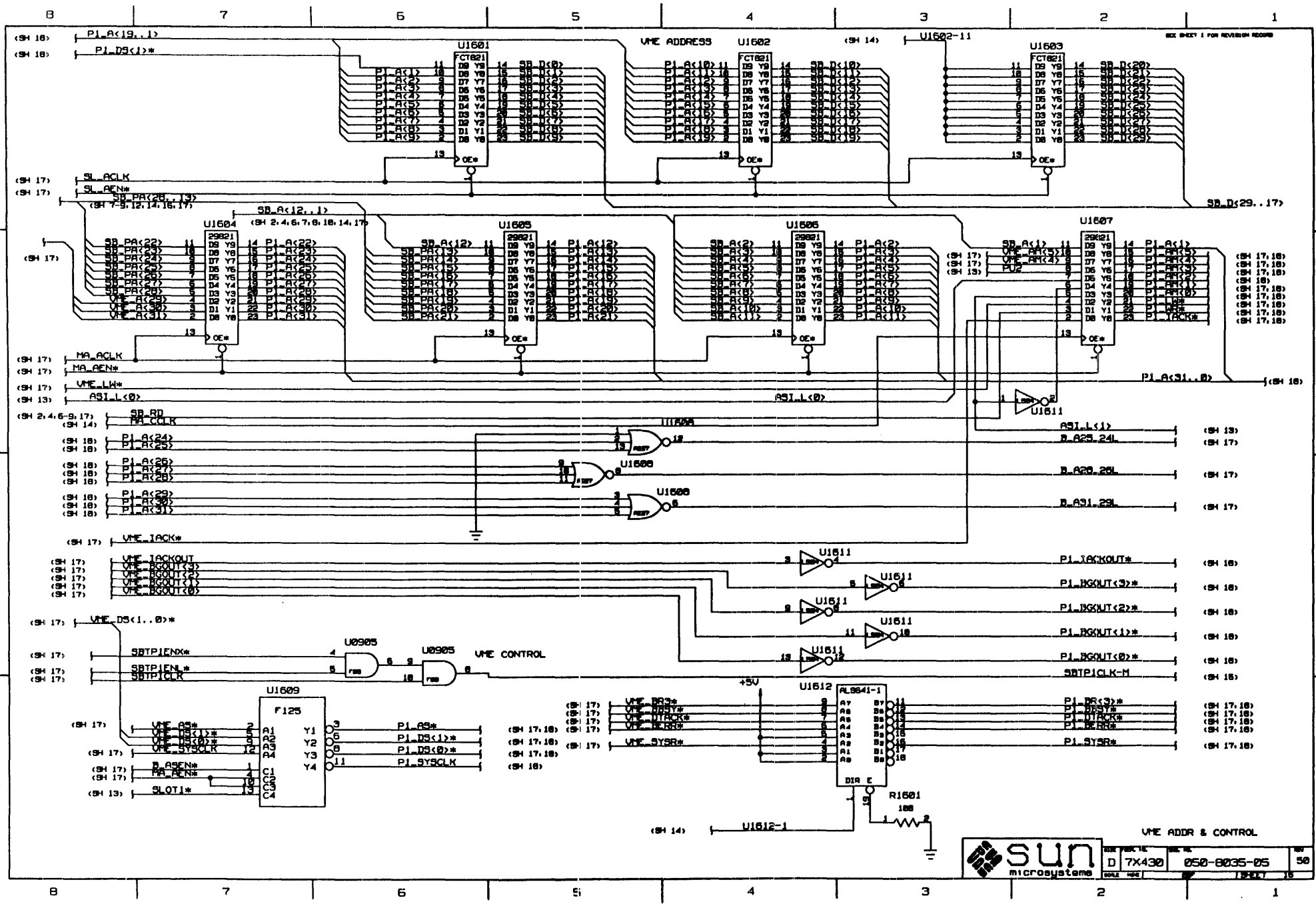

TOD CLOCK
(SH 4,6,7,12)
IOD<7..0>
SB_A<17..0>
(SH 2,4,6,7,8,10,14,16,17)
U1101
MK48T02
TOD
(SH 2,6,7,8,9,14)
SB_RD
(SH 7)
TOD_CS*
R1101
100
J1101
DB15
(SH 12) RXD0I
(SH 12) TXD0O
(SH 12) RXDII
(SH 12) TXDIO
+5V
F1101
3A
L1101
1.8
J1102
SB_A<15>
SB_A<6>
IO_A<4>
IO_A<3>
IO_A<0>
IO_A<2>
IO_A<1>
IOD<0>
IOD<2>
IOD<1>
U1105
LS366
SB_A<4>
SB_A<3>
SB_A<2>
SB_A<1>
SB_A<0>
IO_A<4>
IO_A<3>
IO_A<2>
IO_A<1>
IO_A<0>
R1103
10K
R1102
100
J1103
SB_A<14>
SB_A<9>
SB_A<10>
IOD<7>
IOD<5>
IOD<3>
IOD<6>
IOD<4>
SB_A<17>
(SH 7) EPROM_RD*
IO_A<4..0>
(SH 3,7,9,16)
TOD, PROM
sun microsystems
D
7X430
050-8035-05
50

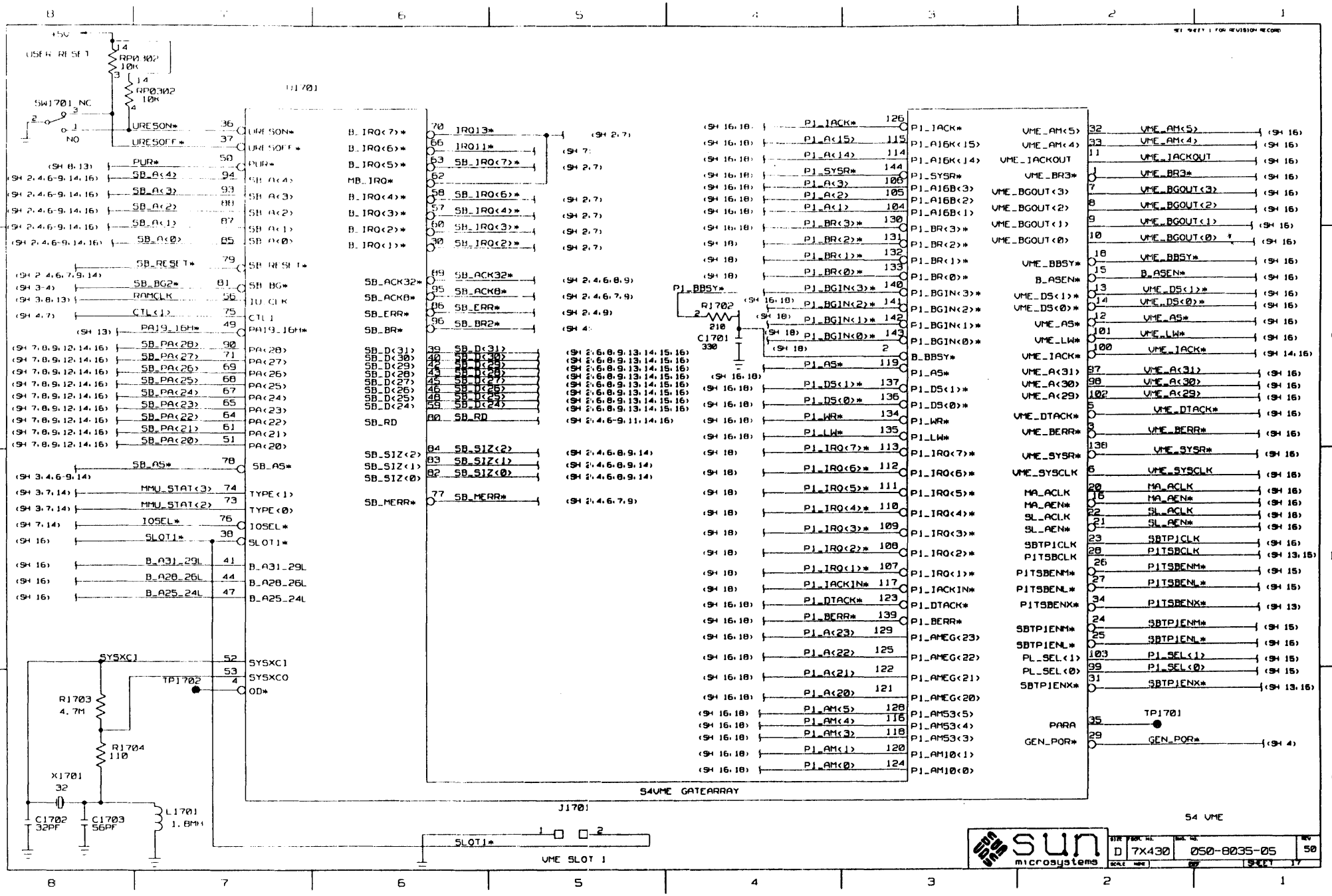

SERIAL PORTS
A & B
KEYBD&MOUSE
PORT A
PORT B
DRIVERS
RECEIVERS
SCC
8530
SEL
J1201
J1202
DB15
IOD<7..0>
(SH 4,6,7,11)
SCC_RD*
SCC_WR*
IO_A<2>
IO_A<1>
SCLK4
SCCIRQ*
KBD_RD*
KBD_WR*
+5V
-5V
+12V
-12V
R1201
100
RP0302
10K
RP1201
4.7K
R1210
R1209
R1208
TXDA0
TXDB0
RTSA0
DTRA0
RTSB0
DTRB0
TXD00
TXD01
TXD10
TXD11
RXD01
RXD11
TRXBI
CTSAI
DCDAI
RXDAI
DSRAI
DCDBI
CTSBI
RXDBI
DSRBI
U1201
U1202
U1203
U1204
U1205
U1206
U1207
U1208
U1209
U1210
U1211
9636
26LS32
26LS31
SEE SHEET 1 FOR REVISION RECORD
sun microsystems
D
7X430
050-8035-05
50
SHEET 12


SEE SHEET 1 FOR REVISION RECORD
U1301
SB_PA<19>
SB_PA<18>
SB_PA<17>
SB_PA<16>
PA19_16H*
(SH 17)
U1302
F174
IV_ASI<0>
IO_ASI<1>
U1302-13
ASI_D<0>
ASI_D<1>
+5V
U1308
F256
POK
PU1
SHOLD*
RAMCLK
U1104
U1413
F74
MHOLD_D
PU2
PUR*
ASI_L<0>
ASI_L<1>
SB_D<31..0>
P1_D<15..0>
8+16 BIT P1 DATA XFER
U1306
U1307
25FCT62
SBITP1ENX*
P1TSBENX*
P1TSBCLK
SBITP1CLK
VME 8+16 BIT DATA
sun microsystems
D
7X430
050-0035-05
50
SHEET 19


P2 INTERFACE
U1401
U1402
U1403
U1404
U1405
U1406
U1407
U1408
U1410
U1413
U1414
U1415
U1104
R1401
100
R1403
100
RP1301
10K
+5V
SB_PA<28..13>
P2_DA<31..0>
SB_D<31..0>
SB_A<12..0>
SB_RD
P2_RAM*
ADDR_EN*
MA_CCLK
SLOT1*
U1302-13
P2_MEM_ERR*
P2_BUS_ERR*
P2_ERR_EN*
P2_ACK32*
P2_ACK0*
SB_ERR*
SB_ACK32*
SB_ACK0*
IOSEL*
SB_AS*
SB_RESET*
P2_AS*
P2_RD
P2_IOSEL*
P2_RESET*
P2_CLK
SB_CLK
VME_IACK*
PARCHK_EN*
PU1
PU2
SEE SHEET 1 FOR REVISION RECORD
sun microsystems
D
7X430
050-8035-05
50
SHEET 14


VME INTERFACE
32 BIT TRANSFER SBD TO PI
32 BIT TRANSFER PI TO SBD
U1501
U1502
U1503
U1504
U1505
U1506
U1507
U1508
SBTPIENM*
PI_SEL<1>
PI_SEL<0>
SBTPICLK-M
SBTPIENL*
PITSBCLK
PITSBENL*
PITSBENM*
SB_D<31..0>
PI_D<31..0>
VME 32 BIT TRANSFER
sun microsystems
D 7X430
050-8035-05


VME ADDRESS
U1601
U1602
U1603
U1604
U1605
U1606
U1607
U1608
U1609
U1611
U1612
U0905
VME CONTROL
SB_D<29..17>
P1_A<31..0>
SB_A<12..1>
SB_PA<28..13>
SL_ACLK
SL_AEN*
MA_ACLK
MA_AEN*
VME_LW*
AS1_L<0>
SB_RD
MA_CCLK
VME_IACK*
VME_IACKOUT
VME_BGOUT<3>
VME_BGOUT<2>
VME_BGOUT<1>
VME_BGOUT<0>
VME_DS<1..0>*
SBTPIENX*
SBTPIENL*
SBTPICLR
SBTPICLK-M
VME_AS*
VME_SYSCLR
B_ASEN*
SLOT1*
P1_AS*
P1_DS<1>*
P1_DS<0>*
P1_SYSCLK
VME_BR3*
VME_BBSY*
VME_BTACK*
VME_BERR*
VME_SYSR*
P1_BR<3>*
P1_BBSY*
P1_BTACK*
P1_BERR*
P1_SYSR*
P1_IACKOUT*
P1_BGOUT<3>*
P1_BGOUT<2>*
P1_BGOUT<1>*
P1_BGOUT<0>*
R1601
VME ADDR & CONTROL
sun microsystems
D
7X430
050-8035-05
50


USER RESET
RP0302 10K
SW1701 NC
NO
U1701
URESON*
URESOFF*
PUR*
SB_A<4>
SB_A<3>
SB_A<2>
SB_A<1>
SB_A<0>
SB_RESET*
SB_BG2*
RAMCLK
CTL<1>
PA19_16M*
SB_PA<28>
SB_PA<27>
SB_PA<26>
SB_PA<25>
SB_PA<24>
SB_PA<23>
SB_PA<22>
SB_PA<21>
SB_PA<20>
SB_AS*
MMU_STAT<3>
MMU_STAT<2>
IOSEL*
SLOT1*
B_A31_29L
B_A28_26L
B_A25_24L
B_IRQ<7>*
B_IRQ<6>*
B_IRQ<5>*
MB_IRQ*
B_IRQ<4>*
B_IRQ<3>*
B_IRQ<2>*
B_IRQ<1>*
SB_ACK32*
SB_ACK8*
SB_ERR*
SB_BR*
SB_D<31>
SB_D<30>
SB_D<29>
SB_D<28>
SB_D<27>
SB_D<26>
SB_D<25>
SB_D<24>
SB_RD
SB_SIZ<2>
SB_SIZ<1>
SB_SIZ<0>
SB_MERR*
IRQ13*
IRQ11*
SB_IRQ<7>*
SB_IRQ<6>*
SB_IRQ<4>*
SB_IRQ<3>*
SB_IRQ<2>*
SB_ACK32*
SB_ACK8*
SB_ERR*
SB_BR2*
SB_D<31>
SB_D<30>
SB_D<29>
SB_D<28>
SB_D<27>
SB_D<26>
SB_D<25>
SB_D<24>
SB_RD
SB_SIZ<2>
SB_SIZ<1>
SB_SIZ<0>
SB_MERR*
P1_IACK*
P1_A<15>
P1_A<14>
P1_SYSR*
P1_A<3>
P1_A<2>
P1_A<1>
P1_BR<3>*
P1_BR<2>*
P1_BR<1>*
P1_BR<0>*
P1_BGIN<3>*
P1_BGIN<2>*
P1_BGIN<1>*
P1_BGIN<0>*
P1_AS*
P1_DS<1>*
P1_DS<0>*
P1_WR*
P1_LW*
P1_IRQ<7>*
P1_IRQ<6>*
P1_IRQ<5>*
P1_IRQ<4>*
P1_IRQ<3>*
P1_IRQ<2>*
P1_IRQ<1>*
P1_IACKIN*
P1_DTACK*
P1_BERR*
P1_A<23>
P1_A<22>
P1_A<21>
P1_A<20>
P1_AM<5>
P1_AM<4>
P1_AM<3>
P1_AM<1>
P1_AM<0>
P1_BBSY*
R1702 210
C1701 330
P1_IACK*
P1_A16K<15>
P1_A16K<14>
P1_SYSR*
P1_A16B<3>
P1_A16B<2>
P1_A16B<1>
P1_BR<3>*
P1_BR<2>*
P1_BR<1>*
P1_BR<0>*
P1_BGIN<3>*
P1_BGIN<2>*
P1_BGIN<1>*
P1_BGIN<0>*
B_BBSY*
P1_AS*
P1_DS<1>*
P1_DS<0>*
P1_WR*
P1_LW*
P1_IRQ<7>*
P1_IRQ<6>*
P1_IRQ<5>*
P1_IRQ<4>*
P1_IRQ<3>*
P1_IRQ<2>*
P1_IRQ<1>*
P1_IACKIN*
P1_DTACK*
P1_BERR*
P1_AMEG<23>
P1_AMEG<22>
P1_AMEG<21>
P1_AMEG<20>
P1_AM53<5>
P1_AM53<4>
P1_AM53<3>
P1_AM10<1>
P1_AM10<0>
VME_AM<5>
VME_AM<4>
VME_IACKOUT
VME_BR3*
VME_BGOUT<3>
VME_BGOUT<2>
VME_BGOUT<1>
VME_BGOUT<0>
VME_BBSY*
B_ASEN*
VME_DS<1>*
VME_DS<0>*
VME_AS*
VME_LW*
VME_IACK*
VME_A<31>
VME_A<30>
VME_A<29>
VME_DTACK*
VME_BERR*
VME_SYSR*
VME_SYSCLK
MA_ACLK
MA_AEN*
SL_ACLK
SL_AEN*
SBTPICLK
PITSBCLK
PITSBENM*
PITSBENL*
PITSBENX*
SBTPIENM*
SBTPIENL*
PL_SEL<1>
PL_SEL<0>
SBTPIENX*
PARA
GEN_POR*
TP1702
R1703 4.7M
R1704 110
X1701 32
C1702 32PF
C1703 56PF
L1701 1.8MH
SYSXC1
SYSXC0
OD*
TP1701
J1701
SLOT1*
VME SLOT 1
S4VME GATEARRAY
S4 VME
SUN microsystems
D
7X430
050-8035-05
50
SHEET 17

VME P1 CONNECTOR

P1 - 1	P1_D<0>	P1 - 33	P1_BBSY*	P1 - 65	P1_D<8>		
P1 - 2	P1_D<1>	P1 - 34	NC	P1 - 66	P1_D<9>		
P1 - 3	P1_D<2>	P1 - 35	NC	P1 - 67	P1_D<10>		
P1 - 4	P1_D<3>	P1 - 36	P1_BGIN<0>*	P1 - 68	P1_D<11>		
P1 - 5	P1_D<4>	P1 - 37	P1_BGOUT<0>*	P1 - 69	P1_D<12>		
P1 - 6	P1_D<5>	P1 - 38	P1_BGIN<1>*	P1 - 70	P1_D<13>		
P1 - 7	P1_D<6>	P1 - 39	P1_BGOUT<1>*	P1 - 71	P1_D<14>		
P1 - 8	P1_D<7>	P1 - 40	P1_BGIN<2>*	P1 - 72	P1_D<15>		
P1 - 9	GND	P1 - 41	P1_BGOUT<2>*	P1 - 73	GND		
P1 - 10	P1_SYSCLK	P1 - 42	P1_BGIN<3>*	P1 - 74	NC		
P1 - 11	GND	P1 - 43	P1_BGOUT<3>*	P1 - 75	P1_BERR*		
P1 - 12	P1_DS<1>*	P1 - 44	P1_BR<0>*	P1 - 76	P1_SYSR*		
P1 - 13	P1_DS<0>*	P1 - 45	P1_BR<1>*	P1 - 77	P1_LWR*		
P1 - 14	P1_WR*	P1 - 46	P1_BR<2>*	P1 - 78	P1_AM<5>		
P1 - 15	GND	P1 - 47	P1_BR<3>*	P1 - 79	P1_A<23>		
P1 - 16	P1_DTACK*	P1 - 48	P1_AM<0>	P1 - 80	P1_A<22>		
P1 - 17	GND	P1 - 49	P1_AM<1>	P1 - 81	P1_A<21>		
P1 - 18	P1_AS*	P1 - 50	P1_AM<2>	P1 - 82	P1_A<20>		
P1 - 19	GND	P1 - 51	P1_AM<3>	P1 - 83	P1_A<19>		
P1 - 20	P1_IACK*	P1 - 52	GND	P1 - 84	P1_A<18>		
P1 - 21	P1_IACKIN*	P1 - 53	NC	P1 - 85	P1_A<17>		
P1 - 22	P1_IACKOUT*	P1 - 54	NC	P1 - 86	P1_A<16>		
P1 - 23	P1_AM<4>	P1 - 55	GND	P1 - 87	P1_A<15>		
P1 - 24	P1_A<7>	P1 - 56	P1_IRQ<7>*	P1 - 88	P1_A<14>		
P1 - 25	P1_A<6>	P1 - 57	P1_IRQ<6>*	P1 - 89	P1_A<13>		
P1 - 26	P1_A<5>	P1 - 58	P1_IRQ<5>*	P1 - 90	P1_A<12>		
P1 - 27	P1_A<4>	P1 - 59	P1_IRQ<4>*	P1 - 91	P1_A<11>		
P1 - 28	P1_A<3>	P1 - 60	P1_IRQ<3>*	P1 - 92	P1_A<10>		
P1 - 29	P1_A<2>	P1 - 61	P1_IRQ<2>*	P1 - 93	P1_A<9>		
P1 - 30	P1_A<1>	P1 - 62	P1_IRQ<1>*	P1 - 94	P1_A<8>		
P1 - 31	-12V/L	P1 - 63	NC	P1 - 95	+12V/L		
P1 - 32	VCC	P1 - 64	VCC	P1 - 96	VCC		

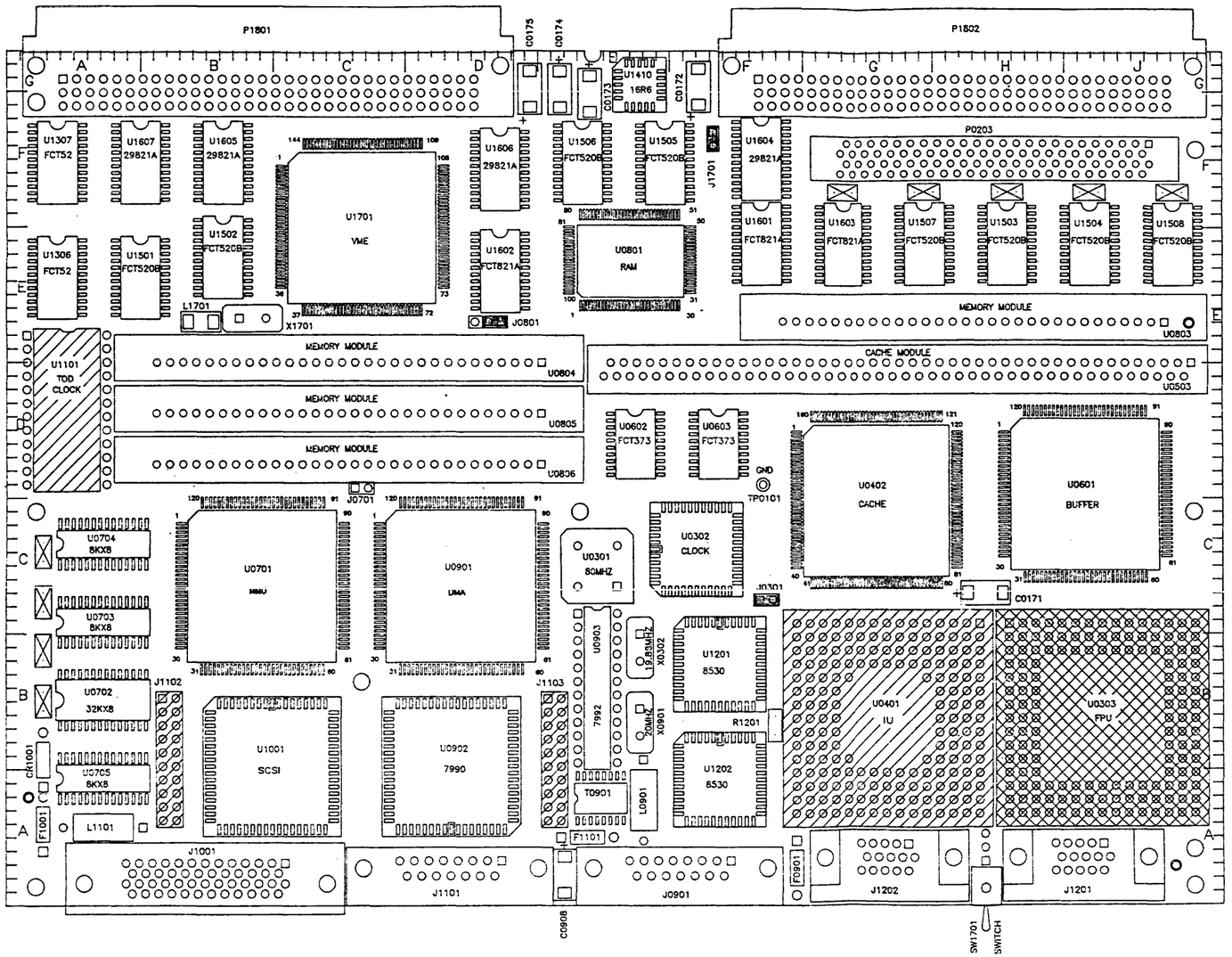
VME P2 CONNECTOR

P2 - 1	PAR<0>	P2 - 33	VCC	P2 - 65	PAR<1>		
P2 - 2	P2_ACK0*	P2 - 34	GND	P2 - 66	GND		
P2 - 3	P2_ACK32*	P2 - 35	RESERVED	P2 - 67	P2_CLK		
P2 - 4	P2_SIZ<0>	P2 - 36	P1_A<24>	P2 - 68	P2_MEM_ERR*		
P2 - 5	P2_SIZ<1>	P2 - 37	P1_A<25>	P2 - 69	PAR<3>		
P2 - 6	PAR<2>	P2 - 38	P1_A<26>	P2 - 70	SB_IRQ<5>*		
P2 - 7	P2_AD	P2 - 39	P1_A<27>	P2 - 71	GND		
P2 - 8	P2_SIZ<2>	P2 - 40	P1_A<28>	P2 - 72	P2_IOSEL*		
P2 - 9	P2_BUS_ERR*	P2 - 41	P1_A<29>	P2 - 73	P2_RESET*		
P2 - 10	SELRAM*	P2 - 42	P1_A<30>	P2 - 74	PUR*		
P2 - 11	P2_AS*	P2 - 43	P1_A<31>	P2 - 75	P2_ERR_EN*		
P2 - 12	GND	P2 - 44	GND	P2 - 76	GND		
P2 - 13	P2_DA<0>	P2 - 45	VCC	P2 - 77	P2_DA<16>		
P2 - 14	P2_DA<1>	P2 - 46	P1_D<16>	P2 - 78	P2_DA<17>		
P2 - 15	P2_DA<2>	P2 - 47	P1_D<17>	P2 - 79	P2_DA<18>		
P2 - 16	P2_DA<3>	P2 - 48	P1_D<18>	P2 - 80	P2_DA<19>		
P2 - 17	GND	P2 - 49	P1_D<19>	P2 - 81	GND		
P2 - 18	P2_DA<4>	P2 - 50	P1_D<20>	P2 - 82	P2_DA<20>		
P2 - 19	P2_DA<5>	P2 - 51	P1_D<21>	P2 - 83	P2_DA<21>		
P2 - 20	P2_DA<6>	P2 - 52	P1_D<22>	P2 - 84	P2_DA<22>		
P2 - 21	P2_DA<7>	P2 - 53	P1_D<23>	P2 - 85	P2_DA<23>		
P2 - 22	GND	P2 - 54	GND	P2 - 86	GND		
P2 - 23	P2_DA<8>	P2 - 55	P1_D<24>	P2 - 87	P2_DA<24>		
P2 - 24	P2_DA<9>	P2 - 56	P1_D<25>	P2 - 88	P2_DA<25>		
P2 - 25	P2_DA<10>	P2 - 57	P1_D<26>	P2 - 89	P2_DA<26>		
P2 - 26	P2_DA<11>	P2 - 58	P1_D<27>	P2 - 90	P2_DA<27>		
P2 - 27	GND	P2 - 59	P1_D<28>	P2 - 91	GND		
P2 - 28	P2_DA<12>	P2 - 60	P1_D<29>	P2 - 92	P2_DA<28>		
P2 - 29	P2_DA<13>	P2 - 61	P1_D<30>	P2 - 93	P2_DA<29>		
P2 - 30	P2_DA<14>	P2 - 62	P1_D<31>	P2 - 94	P2_DA<30>		
P2 - 31	P2_DA<15>	P2 - 63	GND	P2 - 95	P2_DA<31>		
P2 - 32	GND	P2 - 64	VCC	P2 - 96	GND		

DWG NO.		SH	REV

REVISION

ZONE	REV	DESCRIPTION	DATE	APPROVED
	50	ENGRG REL – ECO G740	2/20/90	JAM
	51	REV'D PER – ECO G773	2/26/90	JAM



COMPONENT SIDE

NOTES: (UNLESS OTHERWISE INDICATED)

1. FINISHED BOARDS SHALL MEET SUN'S WORKMANSHIP STANDARD.

   (A) PRINTED WIRE ASSY WORKMANSHIP STANDARD 910–1021–XX.

   (B) MECHANICAL WORKMANSHIP STANDARD 910–1019–XX.

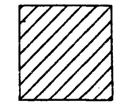
5–2 6–2 ATTACH BAR CODE LABEL IN APPROXIMATE AREA SHOWN. MARK DASH AND REVISION LEVEL WITH INDELIBLE INK WITHIN ONE INCH OF BAR CODE LABEL.

3. LEAD LENGTH MAXIMUM IS 1.50 mm.

2–4 INSTALL CAPACITORS .1uf P/N 110–1192–01 QTY 71 AT THESE LOCATIONS ☒ ON SOLDER SIDE, AND 9 ON COMPONENT SIDE.

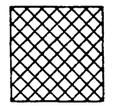
5–5 ALL HARDWARE TO BE TORQUED TO 4.5 INCH LBS +/– 0.5 INCH LBS.

6. INSTALL CHIP SOCKETS AT THESE LOCATIONS.



7. INSTALL COMPLETED PROM BOARD 501–8013–XX WITH HARDWARE AT FINAL ASSEMBLY.

8. DO NOT INSTALL SOCKET (190–1072–XX) AND CHIP (100–1807–XX) AT THIS LOCATION FOR 501/500–8058–XX.



2–3 INSTALL LED'S WITH CATHODE PIN (+) ON PAD INDICATED (SEE DETAIL).

10. INSTALL SHUNTS (130–0272) ACROSS PINS 1 AND 2 AT THESE LOCATIONS.

11. TAPE MEMORY AND CASHE MODULES TOWARD STIFFENER BAR PRIOR TO WAVE SOLDER.

PART LIST

ITEM NO.	QTY REQD	PART OR IDENTIFYING NO.	NOMENCLATURE OR DESCRIPTION	MATERIALS SPECIFIED

THIRD ANGLE PROJECTION

UNLESS OTHERWISE SPECIFIED DIMENSIONS ARE IN MILLIMETERS
TOLERANCES ARE:
DECIMALS .X ± N/A .XX ± N/A
ANGLES ± N/A
MATERIAL N/A
FINISH N/A
DO NOT SCALE DRAWING

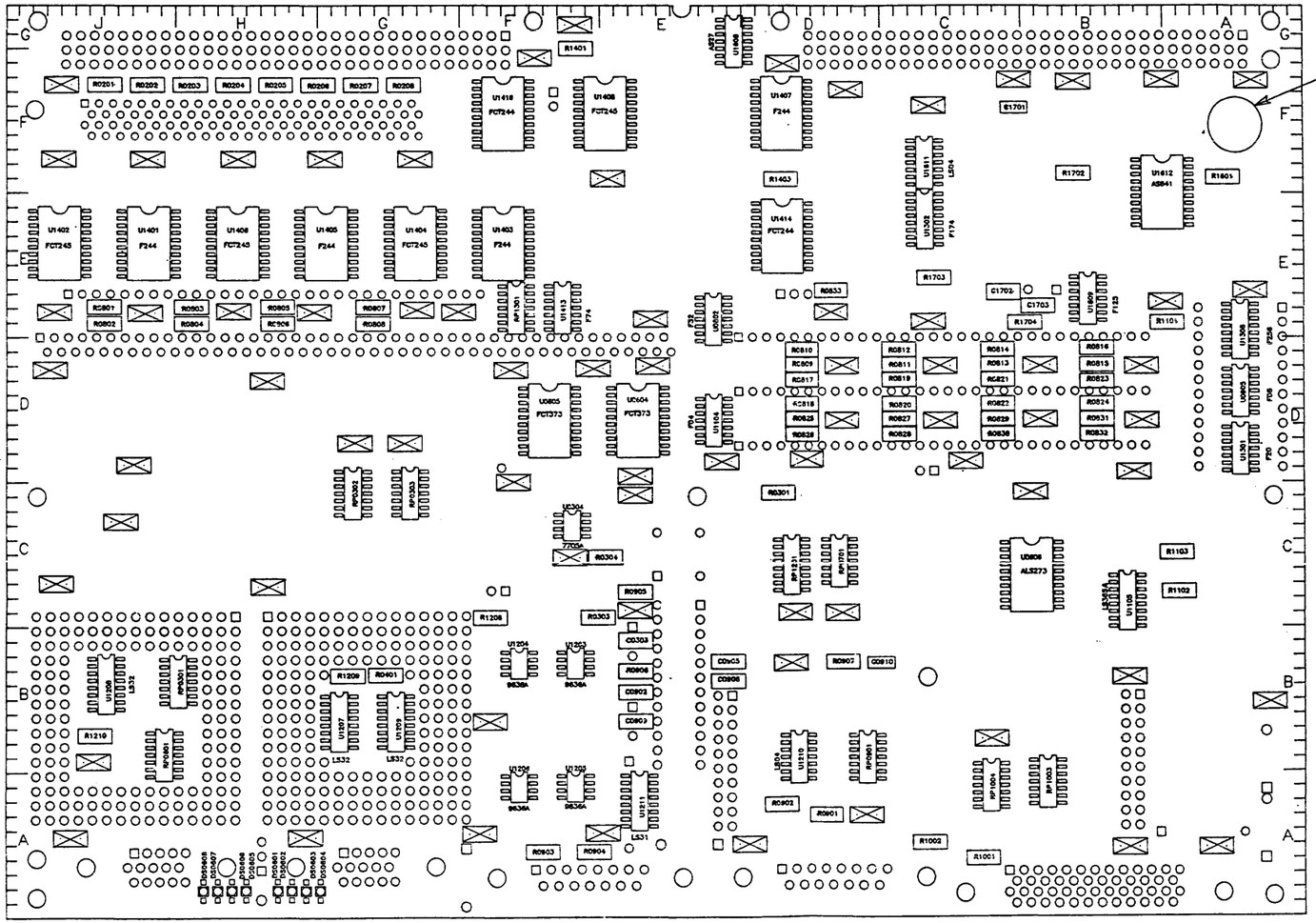
APPROVALS	DATE
DRAWN L.FORBES	2-15-90
CHECKED	2-16-90
ENGR	2/20/90
RELEASED	2/21/90

SUN MICROSYSTEMS

PC ASSY, SPARCengine 1E, CPU BOARD

SIZE	DWG NO.	REV.
D	504–8035–03	51

SCALE: SHEET 1 OF 6

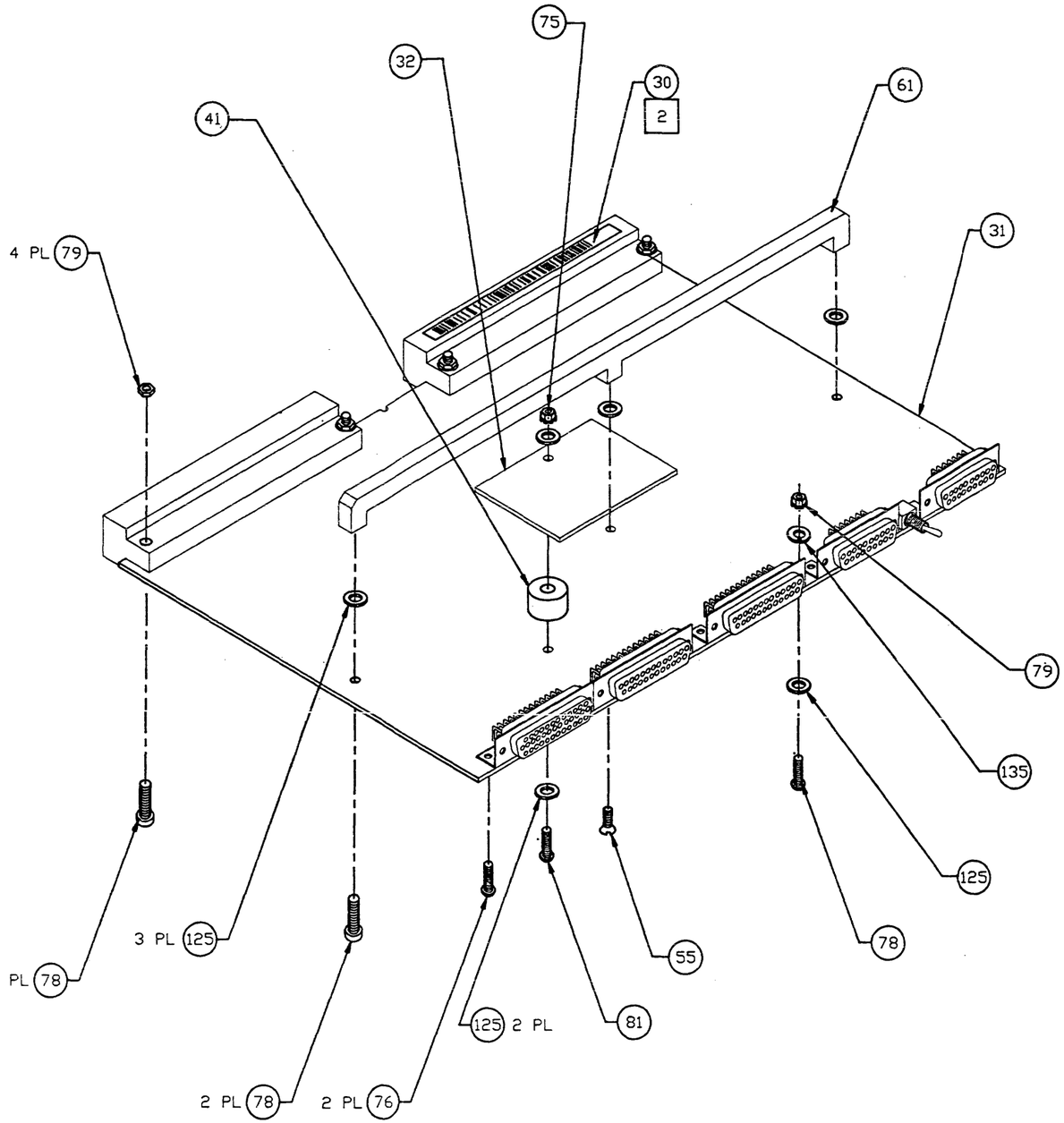


LED POLARITY

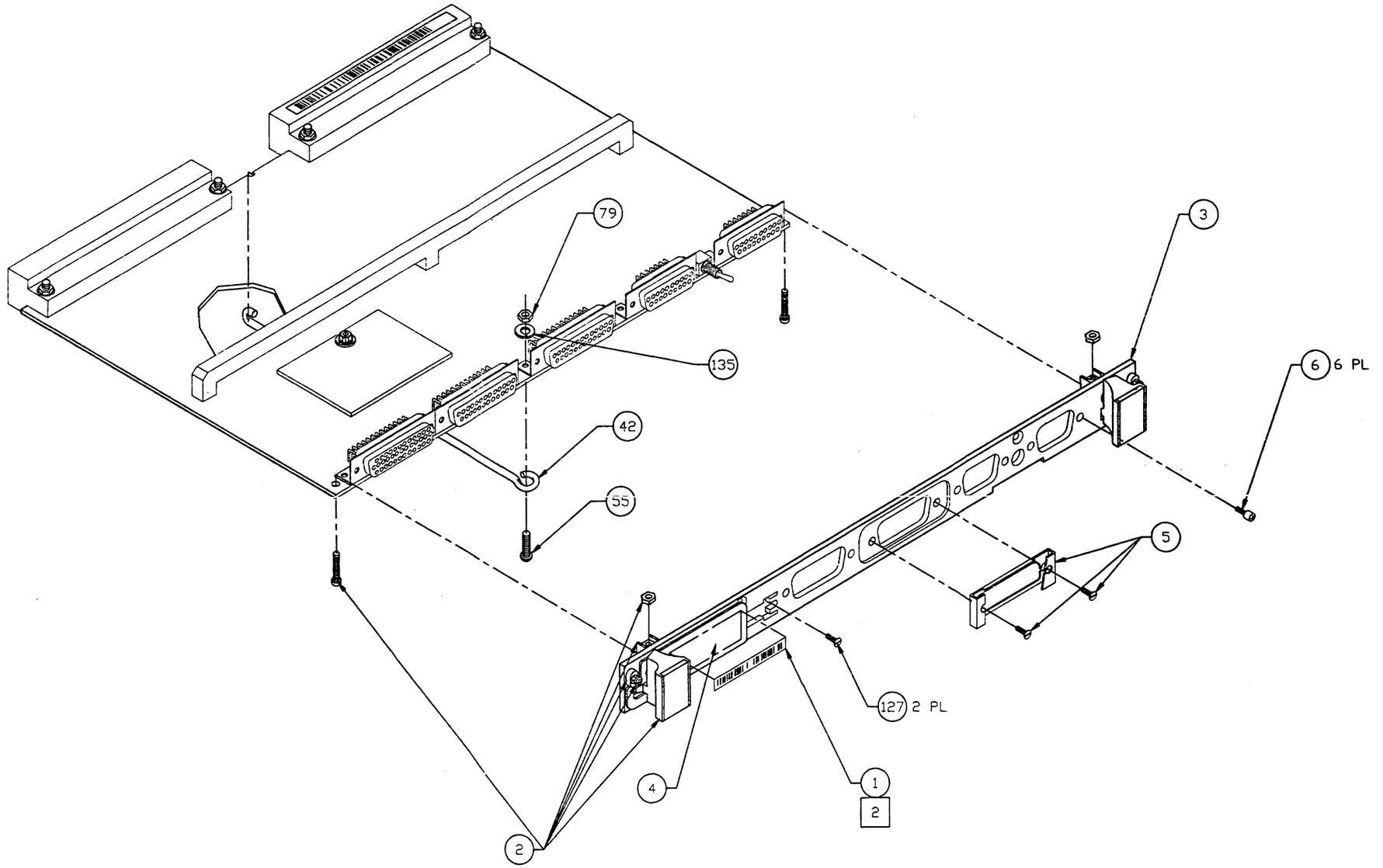
(CATHODE)+

SOLDER SIDE

SIZE	DWG NO.	REV.
D	504-8035-03	51
SCALE:		SHEET 2 OF 6



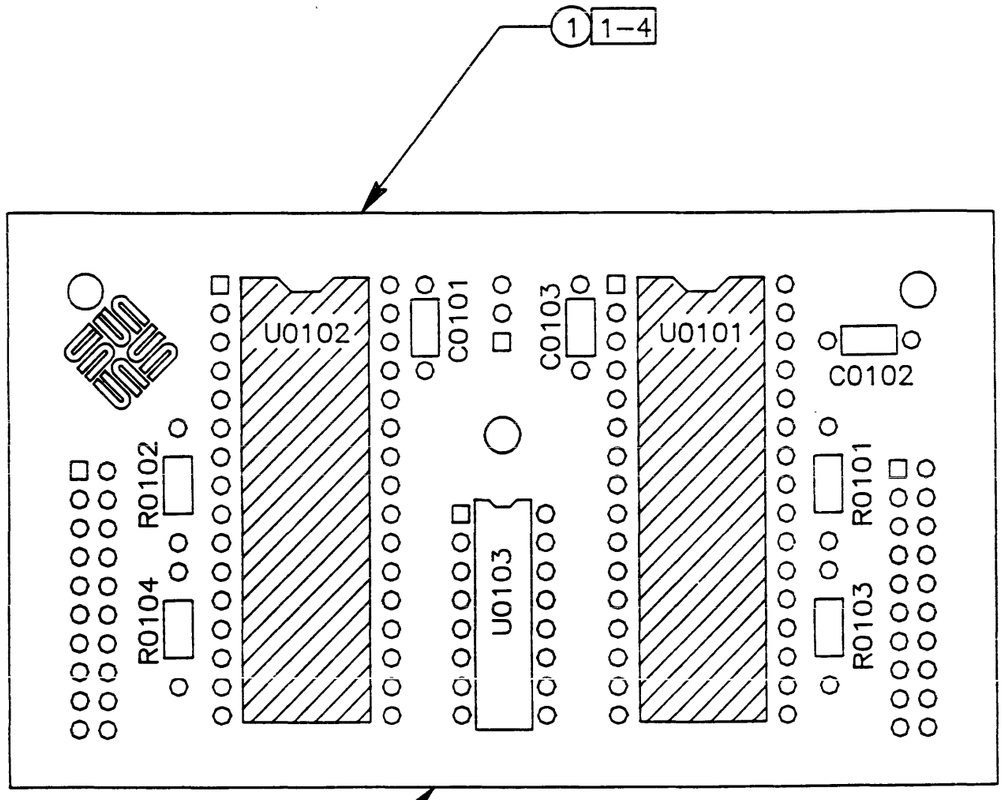
sun microsystems	SIZE D	DWG NO. 504-8035-03	REV. 51
	SCALE 1/1		SHEET 5 OF 6



sun microsystems	SIZE D	DWG NO. 504-8035-03	REV. 51
	SCALE 1/1		SHEET 6 OF 6

DIETRICH POST PROCESS NO. 130938

REVISION				
DASH	REV	DESCRIPTION	DATE	APPROVED
	50	RELEASE PER ECO – G725		



NOTES: (UNLESS OTHERWISE SPECIFIED)

1. FINISHED BOARD SHALL MEET SUN'S WORKMANSHIP STANDARD.

   (A) PRINTED WIRE ASSY WORKMANSHIP STANDARD 910–1021–XX.

   (B) MECHANICAL WORKMANSHIP STANDARD 910–1019–XX.

1–4 PLACE BAR CODE LABEL ALONG EDGE OF BOARD, MARK DASH AND REVISION LEVEL WITH INDELIBLE INK WITHIN ONE INCH OF BAR CODE LABEL ON COMPONENT SIDE.

2–1 INSTALL 2 EA. 130–1664–XX ON SOLDER SIDE.

3. INSTALL SOCKETS AT THESE LOCATIONS.

4. ASSY DRAWING CONSIST OF SINGLE BOARD ILLUSTRATION FROM 9 BOARD PANELIZED.

5. AFTER BOARD IS DEPANELIZED EDGES MUST BE CLEANED AND FLAT FOR BAR CODE PLACEMENT.

SEE SEPARATE BILL OF MATERIAL: 501–8013–XX ME/MA
500–8013–XX MA

UNLESS OTHERWISE SPECIFIED DIMENSIONS ARE IN INCHES TOLERANCES ARE:
DECIMALS ANGLES
.X ± N/A ± N/A
.XX ± N/A

MATERIAL N/A

FINISH N/A

DO NOT SCALE DRAWING

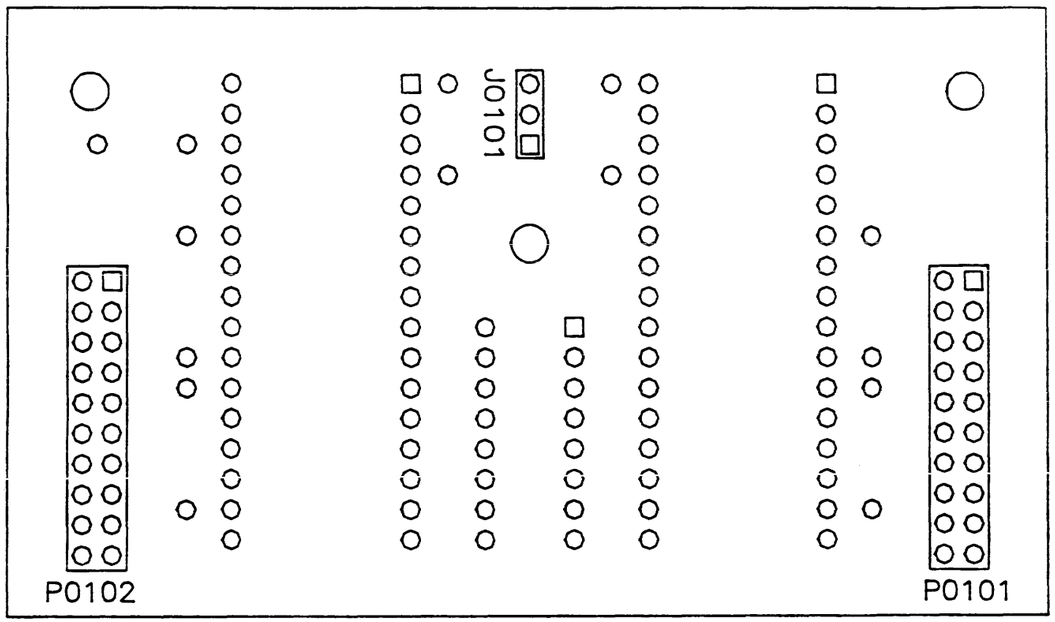
APPROVALS	DATE
DRAWN L.FORBES	1–25–90
CHECKED	
ENGR	
RELEASED	

SUN MICROSYSTEMS

PC ASSEMBLY,
SPARCengine 1E,
PROM DAUGHTER CARD

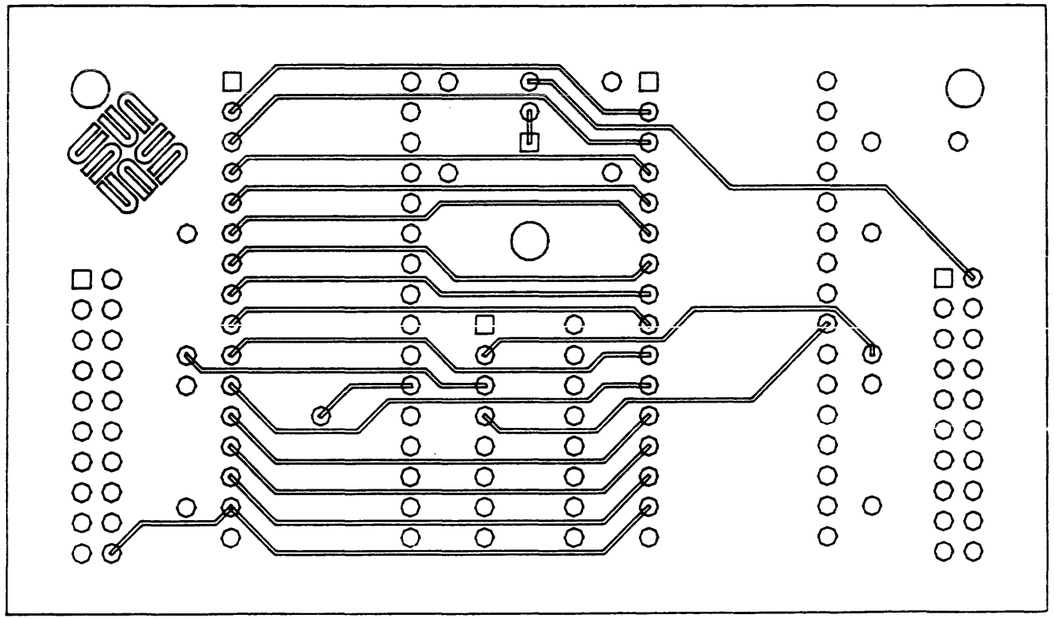
SIZE B | DWG NO. 504–8013–02 | REV. 50

SCALE: | SHEET 1 OF 4



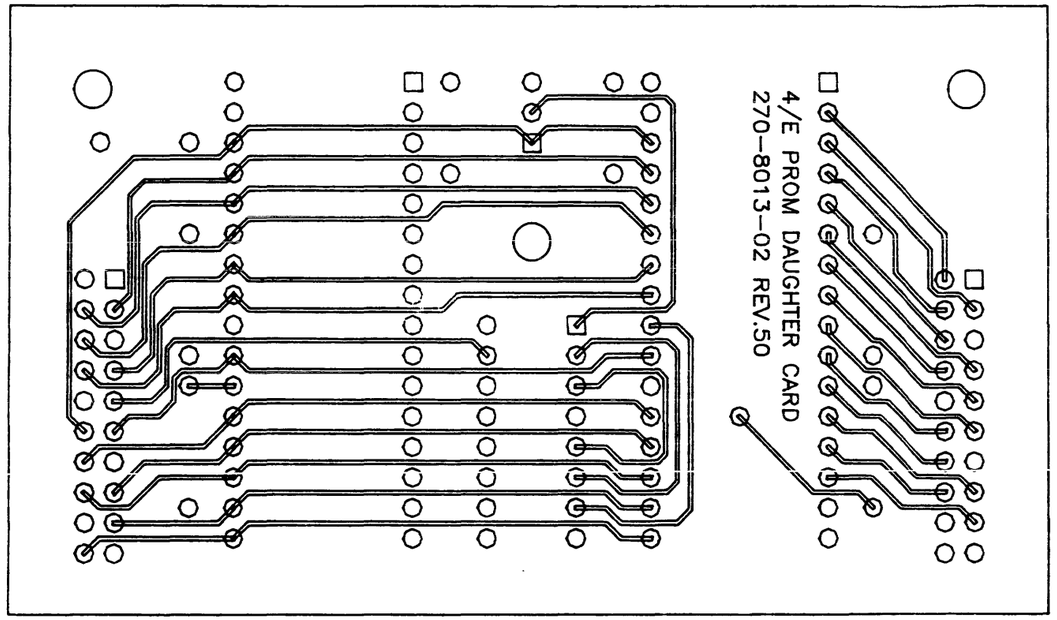
SOLDER SIDE

SIZE	DWG NO.	REV.
B	504-8013-02	50
SCALE:		SHEET 2 OF 4



COMPONENT SIDE

SIZE	DWG NO.	REV.
B	504-8013-02	50
SCALE:		SHEET 3 OF 4



SOLDER SIDE

SIZE	DWG NO.	REV.
B	504-8013-02	50
SCALE:		SHEET 4 OF 4