Digital Semiconductor SA-110 Microprocessor Evaluation Board

Reference Manual

Order Number: EC-QU5KA-TE

The EBSA-110 is an evaluation board for the SA-110 StrongARM® microprocessor. This manual is the single point-of-reference for all users of the EBSA-110. It is a configuration guide, a programmers' guide and a technical reference.

Revision/Update Information: Version 1.0.

March 1996

While Digital believes the information included in this publication is correct as of the date of publication, it is subject to change without notice.

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

© Digital Equipment Corporation 1996. All rights reserved.

The following are trademarks of Digital Equipment Corporation: Digital, Digital Semiconductor and the DIGITAL Logo.

Digital Semiconductor is a Digital Equipment Corporation business.

ABEL is a registered trademark of Data I/O Corporation.

Altera is a registered trademark of Altera Corporation.

ARM and StrongARM are registered trademarks of ARM Ltd.

Intel is a registered trademark of Intel Corporation.

PostScript is a registered trademark of Adobe Systems Incorporated.

Quickswitch is a registered trademark of Quality Semiconductor, Inc.

TimingDesigner is a registered trademark of Chronolgy.

VIEWlogic is a registered trademark of Viewlogic Systems Inc.

Windows is a trademark of Microsoft Corporation.

All other trademarks and registered trademarks are the property of their respective holders.

This document was prepared using VAX DOCUMENT, Version 2.1.

Contents

Preface xi

1 Getting Started

1.1 Physical Description 1–1
1.2 Handling Precautions 1–1
1.3 Visual Inspection 1–1
1.3.1 Cabling 1–3
1.3.2 Links 1–3
1.3.3 Socketed Components 1–3
1.3.3.1 The CPU 1–3
1.4 Power-On 1–4
1.5 Attaching the EBSA-110 to a Terminal or Host System 1–4

2 Functional Specification

2.1 CPU 2–1
2.2 Clocks 2–1
2.3 Reset 2–3
2.4 Power 2–3
2.4.1 Voltage Domains 2–3
2.4.2 Power Sequencing 2–4
2.5 Memory 2–4
2.5.1 ROM 2–4
2.5.2 SSRAM 2–4
2.5.3 DRAM 2–5
2.5.4 Memory Map Switching 2–5
2.6 I/O Sub-System 2–5
2.6.1 Serial Ports and Printer Port 2–6
2.6.2 Ethernet Port and UID ROM 2–6
2.6.3 PCMCIA Controller 2–6
2.6.4 Counter/Timer 2–6
2.6.5 Soft I/O 2–6
2.6.6 Architectural Compliance Verification Facilities 2–6
2.7 Interrupts 2–7
2.8 JTAG 2–7
2.9 Hardware Debug Support 2–7
2.10 Expansion 2–7
2.11 Control Logic 2–7
2.12 Endian Issues 2–7
2.13 LEDs 2–8
2.14 On-Board Software 2–8

3 Programmers' Guide

3.1 Memory Map . . . 3–1
3.1.1 Memory Map After Reset . . . 3–3
3.1.2 Characteristics of Memory . . . 3–4
3.2 Memory Map Decodes . . . 3–4
3.2.1 DRAM Space . . . 3–4
3.2.2 DRAM Configuration Space . . . 3–5
3.2.3 Synchronous SRAM (SSRAM) Space . . . 3–5
3.2.4 EPROM/Flash Space . . . 3–5
3.2.5 I/O Space . . . 3–6
3.2.6 RW_ABORT Space . . . 3–7
3.2.7 R_ABORT Space . . . 3–7
3.2.8 ISAMEM Space . . . 3–7
3.2.8.1 EBUFMEM Space . . . 3–8
3.2.8.2 PCMCIAMEM Space . . . 3–8
3.2.9 ISAIO Space . . . 3–8
3.3 The PIT Registers . . . 3–9
3.4 The SuperI/O Registers . . . 3–10
3.5 The Ethernet Controller Registers . . . 3–11
3.6 The PCMCIA Controller Registers . . . 3–12
3.7 The Soft Register . . . 3–12
3.8 Reset State . . . 3–13
3.9 Software Restrictions . . . 3–13
3.9.1 8-bit Accesses to Odd Addresses . . . 3–13

4 Interrupts

4.1 Distinguishing CTB_OS, CTB_ARCH Under Software Control . . . 4–2
4.2 CTB_OS . . . 4–2
4.2.1 The FIQ_MASK Register . . . 4–2
4.2.2 The IRQ_MASK Register . . . 4–3
4.2.3 The IRQ_MSET, IRQ_MCLR Registers . . . 4–3
4.2.4 The IRQ_RAW Register . . . 4–4
4.2.5 The IRQ_MSKD Register . . . 4–4
4.3 CTB_ARCH . . . 4–4
4.3.1 The FIQ_MASK Register . . . 4–4
4.3.2 The IRQ_MASK Register . . . 4–5
4.3.3 The IRQ_CNT Cycle Counter Register . . . 4–5
4.3.4 The FIQ_CNT Cycle Counter Register . . . 4–6

5 Configuration of Memory and VLSI Devices

5.1 Configuring Cacheable/Non-Cacheable Space . . . 5–1
5.2 Switching the Memory Map . . . 5–1
5.3 DRAM . . . 5–1
5.3.1 Disable Refresh Requests . . . 5–2
5.3.2 Initialize the DRAM . . . 5–2
5.3.3 Enable Refresh Requests . . . 5–3
5.3.4 Determine the DRAM Type . . . 5–3
5.3.5 Size the Memory . . . 5–3
5.3.6 Test the memory . . . 5–4
5.4 SSRAM . . . 5–4
5.5 EPROM . . . 5–4
5.6 Flash . . . 5–4

5.7 PCMCIA Controller 5–4
5.7.1 Setting the PCMCIA Socket Programming Voltage 5–5
5.7.2 Setting a PCMCIA Socket Memory Window 5–5
5.8 Ethernet Controller 5–7
5.8.1 Send the Initiation Key 5–7
5.8.2 Put the Device into ‘CONFIG’ State 5–7
5.8.3 Configure the Plug-and-play Registers 5–8
5.8.4 Disable the Plug-and-play Registers 5–8
5.9 Super I/O Controller 5–9
5.10 Programmable Interval Timer 5–9

6 Performance

6.1 Synchronous SRAM Accesses 6–1
6.2 EDO DRAM Accesses 6–1
6.3 BEDO DRAM Accesses 6–2
6.4 Performance Impact of DRAM Refresh 6–3
6.5 EPROM and Flash Accesses 6–3
6.6 I/O Accesses 6–4
6.6.1 Ethernet Buffer Memory Bandwidth 6–4
6.7 Overlap of Cycles 6–4

7 Software Development Environment

7.1 Loadable Debuggable Images 7–1
7.1.1 Building 7–1
7.1.2 Run Time Environment 7–2
7.1.2.1 Memory Map 7–2
7.1.2.2 C Library Support 7–2
7.1.2.3 Exception Vectors 7–2
7.1.2.4 Access to I/O Devices 7–2
7.2 Standalone Flash Images 7–2
7.2.1 Building 7–2
7.2.2 Run Time Environment 7–3
7.2.2.1 Memory Map 7–3
7.2.2.2 C Library Support 7–3
7.2.2.3 Exception Vectors 7–3
7.2.2.4 Access to I/O Devices 7–3

8 On-Board Software

8.1 The Primary Boot Loader 8–1
8.2 The Format of Images in Flash 8–2
8.3 The Startup EPROM 8–3
8.4 Diagnostics 8–4
8.4.1 Getting Ready to Run the Diagnostics 8–4
8.4.2 Description of Tests 8–4

9 Software Utilities

9.1 The Flash Management Utility 9–1
9.1.1 When to Specify the Block Number 9–3
9.1.2 When to Specify the 'NoBoot' Option 9–4
9.2 The Bootp Utility 9–4
9.2.1 Variants of the bootp Program 9–5

10 Theory of Operation

10.1 A Tour of the Schematics 10–1
10.1.1 Principal Buses 10–2
10.1.2 Power 10–3
10.1.3 Decoupling 10–3
10.1.4 Voltage Levels 10–4
10.1.5 Clocks 10–4
10.1.6 Reset 10–5
10.1.7 The CPU 10–6
10.1.8 Jumpers, Etch Links, Debug Connectors and Test Points 10–6
10.1.9 SSRAM Interface 10–7
10.1.10 Buffering 10–7
10.1.11 DRAM Interface 10–7
10.1.12 Control Logic 10–8
10.1.13 EPROM/Flash 10–8
10.1.14 SuperI/O Controller 10–9
10.1.15 Ethernet Controller 10–10
10.1.16 PCMCIA Controller 10–11
10.1.17 JTAG Port 10–11
10.1.18 Counter/Timer 10–12
10.2 Control Logic 10–12
10.2.1 Control of CPU Bus Cycles 10–14
10.2.2 Types of Cycles 10–15
10.2.3 Sub-Block Wrapping 10–15
10.2.4 The Burst Counter 10–15
10.2.5 The Packer Address Counter 10–17
10.2.6 Accesses to 16-bit Peripherals 10–18
10.2.7 Memory Map Switching After Reset 10–18
10.2.8 BEDO DRAM Configuration Cycles 10–19
10.2.9 Address Decoding 10–19
10.2.9.1 Decoding Within the SSRAM Quadrant 10–20
10.2.9.2 Decoding Within the DRAM Quadrant 10–20
10.2.9.3 Decoding Within the ROM Quadrant 10–20
10.2.9.4 Decoding Within the IO Quadrant 10–21
10.3 Timing Analysis 10–21
10.4 Expanding the EBSA-110 10–21
10.5 The Printed Circuit Board 10–22
10.6 Design Improvements 10–22

11 Simulation Waveforms

11.1 automap 11–1
11.2 ss_wcrd 11–3
11.3 ss_wcwr 11–3
11.4 ss_rdwrap 11–6
11.5 ss_rdall 11–8
11.6 ed_wcrd 11–10
11.7 ed_wcwr 11–12
11.8 ed_rdwrap 11–14
11.9 bd_wcrd 11–14
11.10 bd_wcwr 11–17
11.11 bd_rdwrap 11–19
11.12 bd_wrf 11–21
11.13 rfrsh 11–21
11.14 cbr 11–24
11.15 romrd1 11–26
11.16 romrd2 11–28
11.17 flashwr 11–30
11.18 io 11–30
11.19 iordy 11–34
11.20 iozws 11–34
11.21 iorfrdy 11–34
11.22 iotrick 11–38

A Configuration Guide

A.1 Default Configuration A–1
A.2 Description of All Jumpers A–1
A.2.1 Supported Clock Configurations A–5
A.3 Description of All Links A–5
A.4 Connectors A–6
A.5 Debug Connectors A–7
A.6 Debug Pick-up Points A–8
A.7 LEDs A–8
A.8 Cables Within the Enclosure A–9
A.8.1 Power Supply A–9
A.8.2 Serial Ports A–10
A.8.3 Parallel Port A–10
A.8.4 Reset Switch A–10
A.8.5 Turbo Switch A–10
A.8.6 Loudspeaker A–10
A.9 Cables for External Connection A–10
A.9.1 Serial Ports A–10
A.9.1.1 Serial Cable for SUN Workstation A–11
A.9.2 Parallel Port A–11
A.9.3 Parallel Port Loopback A–12
A.9.4 Ethernet Port A–12
A.9.5 JTAG Port A–12
A.10 Upgrading the DRAM SIMMs A–13

B Debugging a Broken Board

B.1 Basic Checks B–1
B.2 Checking the Board B–1
B.3 Diagnostic Failure B–2

C The Design Database

D SA-110 Bus Transactor Model User's Guide

D.1 Instantiating the Model D–2
D.2 Command Reference D–3
D.2.1 set_addr {address} D–3
D.2.2 set_page {offset} D–3
D.2.3 set_bytes {byte masks}, set_size {size} D–3
D.2.4 do_rd {expected read data} D–4
D.2.5 do_crd {expected read data} D–4
D.2.6 do_wr {write data} D–4
D.2.7 do_fwr {write data} D–4
D.2.8 do_idle {number of cycles} D–4
D.2.9 do_swap {expected read data} {write data} D–4
D.3 How It Works D–4
D.4 It Is Not Idiot-Proof! D–5
D.4.1 Completeness, Known Bugs and Model Support D–6
D.4.2 Porting, Modifying and Rebuilding D–6

E ABEL Tutorial

F Getting Started with an Uncased Board

F.0.1 Choosing a Power Supply F–1
F.0.2 Choosing an Enclosure F–1

G Technical Support and Ordering Information

Index

Figures

1–1 The EBSA-110 Board 1–2
1–2 Position of Debug LED 1–4
2–1 EBSA-110 Block Diagram 2–2
11–1 automap 11–2
11–2 ss_wcrd 11–4
11–3 ss_wcwr 11–5
11–4 ss_rdwrap 11–7
11–5 ss_rdall 11–9
11–6 ed_wcrd 11–11
11–7 ed_wcwr 11–13
11–8 ed_rdwrap 11–15

11–9 bd_wcrd 11–16
11–10 bd_wcwr 11–18
11–11 bd_rdwrap 11–20
11–12 bd_wrf 11–22
11–13 rfrsh 11–23
11–14 cbr 11–25
11–15 romrd1 11–27
11–16 romrd2 11–29
11–17 flashwr 11–31
11–18 io 11–32
11–19 iordy 11–35
11–20 iozws 11–36
11–21 iorfrdy 11–37
11–22 iotrick 11–39
A–1 EBSA-110 Configuration Links A–2
A–2 Position of LEDs A–9

Tables

3–1 Memory Map 3–1
3–2 Addresses in External-Decode Space 3–9
3–3 PIT Internal Registers 3–10
3–4 SuperI/O Registers 3–10
3–5 Ethernet Controller Registers 3–11
3–6 Bit Assignment of Soft Register 3–12
4–1 Interrupt Control Registers - CTB_OS Configuration 4–1
4–2 Interrupt Control Registers - CTB_ARCH Configuration 4–1
4–3 FIQ Mask Bit Positions 4–2
4–4 Interrupt Mask Bit Positions - CTB_OS Configuration 4–3
4–5 FIQ Mask Bit Positions 4–4
4–6 Interrupt Mask Bit Positions - CTB_ARCH Configuration 4–5
5–1 PCMCIA Controller Configuration Sequence 5–4
5–2 PCMCIA Programming Voltages 5–5
5–3 Ethernet Plug-and-play Register Configuration Sequence 5–8
5–4 Ethernet Plug-and-play Register Initial Values 5–8
6–1 Stalls Added During EDO DRAM Accesses 6–2
6–2 Stalls Added During BEDO DRAM Accesses 6–2
6–3 Stalls Added During EPROM and Flash Accesses 6–4
6–4 Stalls Added During I/O Accesses 6–4
6–5 Stalls Caused by Back-to-Back Cycles 6–5
8–1 Boot Image Selection 8–1
8–2 Flash Image Header 8–3
8–3 Selecting Diagnostics 8–4
10–1 Byte/Half-Word Decode Using SA0, SBHE_L 10–18
A–1 Jumpers A–3
A–2 Links A–5
A–3 Connectors A–6

A–4 Debug Connectors A–7
A–5 Pick-up point A–8
A–6 Null-MoDem Cable A–10
A–7 SUN Null-MoDem Cable A–11
A–8 Bidirectional Parallel Cable A–11
A–9 Parallel Port Loopback Connector A–12
A–10 JTAG Cable A–12
A–11 Suitable DRAM SIMMs A–13

Preface

Introduction

The EBSA-110 is an evaluation board for Digital Semiconductor's SA-110 microprocessor. It is designed to meet the following requirements:

- To provide a power-on vehicle for the SA-110 chip.
- To provide an environment in which to run the ARM® architectural compliance software test suite.
- To provide a non-proprietary example design.
- To provide a software development environment, including a fast memory sub-system on which to run software benchmarks.

This document is a single point-of-reference both for configuring and using the board and for engineers wishing to copy parts of its design. As such, it has the following scope:

- Functional specification
- Theory of operation (to be read in conjunction with the circuit schematics)
- Configuration guide (memory options, speed options, jumper and link options)
- Programmers' guide (memory maps, boot process, references to programmble I/O devices on the board)

This document does not aim to duplicate material to be found elsewhere. Specifically, it does not duplicate material that is to be found in vendor data sheets for components used in the design, nor does it document the ARM software development environment.

How to Use This Document

All readers should turn to Chapter 1 for information about how to connect and power-on the board, how to verify that it is working correctly and how to connect it to a terminal or host system.

All readers are advised to read Chapter 2 to get an understanding of the overall functionality of the board. Subsequent chapters assume a familiarity with the material in this chapter.

Thereafter, software engineers will probably want to refer to the following chapters:

- Chapter 3 is a guide to the memory map of the board and the address decoding of all I/O devices.
- Chapter 4 describes the interrupt structures.

- Chapter 5 is a guide to configuration of the memory and VLSI devices on the board.
- Chapter 7 is a brief introduction to the software development environment.
- Chapter 8 describes the on-board software, including the power-on sequence of the board, and the power-on diagnostics.
- Chapter 9 describes software utilities which are provided with the EBSA-110.
- Chapter 6 contains performance-related information, and documents the cycle times required for accessing various devices on the board.

Hardware engineers will probably want to refer to the following chapters:

- Chapter 10 is a detailed technical description of the hardware of the board, including a theory of operation.
- Chapter 11 describes a number of simulation waveforms, giving a deeper insight into the operation of the EBSA-110 control state machines.

A number of appendices provide general reference material:

- Appendix A describes all of the link and jumper options present on the board, and all of the cables that may be required for connection to the board.
- Appendix B provides hints on how to track down faults on the EBSA-110.
- Appendix C describes the machine-readable design databases for the EBSA-110 hardware and software.
- Appendix D describes the operation and usage of the SA-110 Bus Transaction Model, which is provided as part of the design database.
- Appendix E is a brief tutorial in the ABEL® PLD synthesis language used to describe the state machines in the EBSA-110 design.
- Appendix F describes how to choose a suitable enclosure and power supply for the board.
- Appendix G describes other relevant documents and services that are available from Digital and its partners.

Notation

All numbers are shown in decimal unless otherwise stated.

All hexadecimal numbers have an 0x prefix. 32-bit hex values have dots for ease of reading. Examples are: 0xfe0b.3004, 0xfb.

All binary number have an 0b prefix; long numbers include dots for ease of reading. Examples are: 0b00, 0b0000.0000.1010.0000.

This document refers to an 8-bit data unit as a byte, a 16-bit data unit as a half-word and a 32-bit data unit as a longword. †

This document uses the notation INT*n* to refer to a naturally-aligned block of *n* bytes. Thus, an INT4 is an aligned 32-bit value whilst an INT32 is eight 32-bit values on a naturally aligned address (this corresponds to the size and alignment of an SA-110 cache block).

† Standard ARM notation is to use the terms byte, half-word and word, respectively. Digital's convention is to use the terms byte, word and longword. Therefore, this document avoids use of the term 'word', which is ambiguous to different audiences.

Electrical signal names are shown thus: **cpu_wait_l**. An **_l** at the end of a signal name indicates that the signal is asserted (active) when it is low (close to 0V).

References

This section provides a selective bibliography and a reference to relevant manufacturers' data sheets. ARM-specific and SA-110-specific information is referenced in Appendix G.

1. SSRAM: Micron MT58LC32K36C4-LG (or MT58LC32K36D7-LG) data sheet.
2. PCMCIA controller: VADEM VG-468 PC Card Socket Controller Data Manual (December 1993, Rev 02, or later). VADEM, San Jose, CA. Tel +1 408 943-9301. Fax +1 408 943-9735. UK distributor: MMD. Tel +44 1734 633700.
3. Super I/O: PC87312 data sheet, National Semiconductor Corporation.
4. EDO DRAM: Micron MT16D232M-6 X DRAM Module data sheet.
5. Burst EDO DRAM: Micron MT4D232M-6 ES DRAM Module data sheet.
6. Flash ROM: 28F008SA data sheet, Intel order number 290429-004.
7. EPLD: EPM7096LC84-7, Altera 1995 Data Book.
8. Ethernet controller: Advanced Micro Devices Am79C961A data sheet (AMD publication number 19364 Rev. A (October 1994) with Amendment sheet 1)
9. High-Speed Digital Design - a handbook of black magic. (Howard W Johnson, Martin Graham, 1993 Prentice Hall ISBN 0-13-395724-1).

1
Getting Started

The EBSA-110 is provided built, tested and cased. This chapter provides a physical description of the board and then describes how to:

1. Perform a visual inspection of the EBSA-110.
2. Power-on the EBSA-110 for the first time.
3. Attach the EBSA-110 to a terminal or host system.

If you wish to use a different enclosure for the board, refer to Appendix F for details on choosing a suitable power supply and enclosure.

1.1 Physical Description

The EBSA-110 is shown in Figure 1–1. It is a single-board computer designed to match the form-factor of a baby-AT PC motherboard. This allows it to be mounted in a standard desktop or deskside PC cabinet. Flying leads connect the board to its I/O connectors. The I/O connectors are mounted in break-out holes that are standard on these systems. The board is powered from the cabinet's power supply using the standard PC power connectors.

1.2 Handling Precautions

The EBSA-110 contains components that are susceptible to permanent damage from electrostatic discharge ('static' electricity). Risk of damage can be alleviated by following a few simple handling precautions.

If the EBSA-110 was supplied cased and you remove the cover, ensure that the case is earthed and that you are wearing an antistatic wrist strap before making any adjustments to the board.

If the EBSA-110 is supplied as a bare board, it is supplied in an antistatic bag. Do not remove the board from the bag unless you are working on an antistatic earthed surface and wearing an antistatic wrist strap. Always adopt these precautions when handling the board.

1.3 Visual Inspection

When you receive your unit, you should perform these minimum checks:

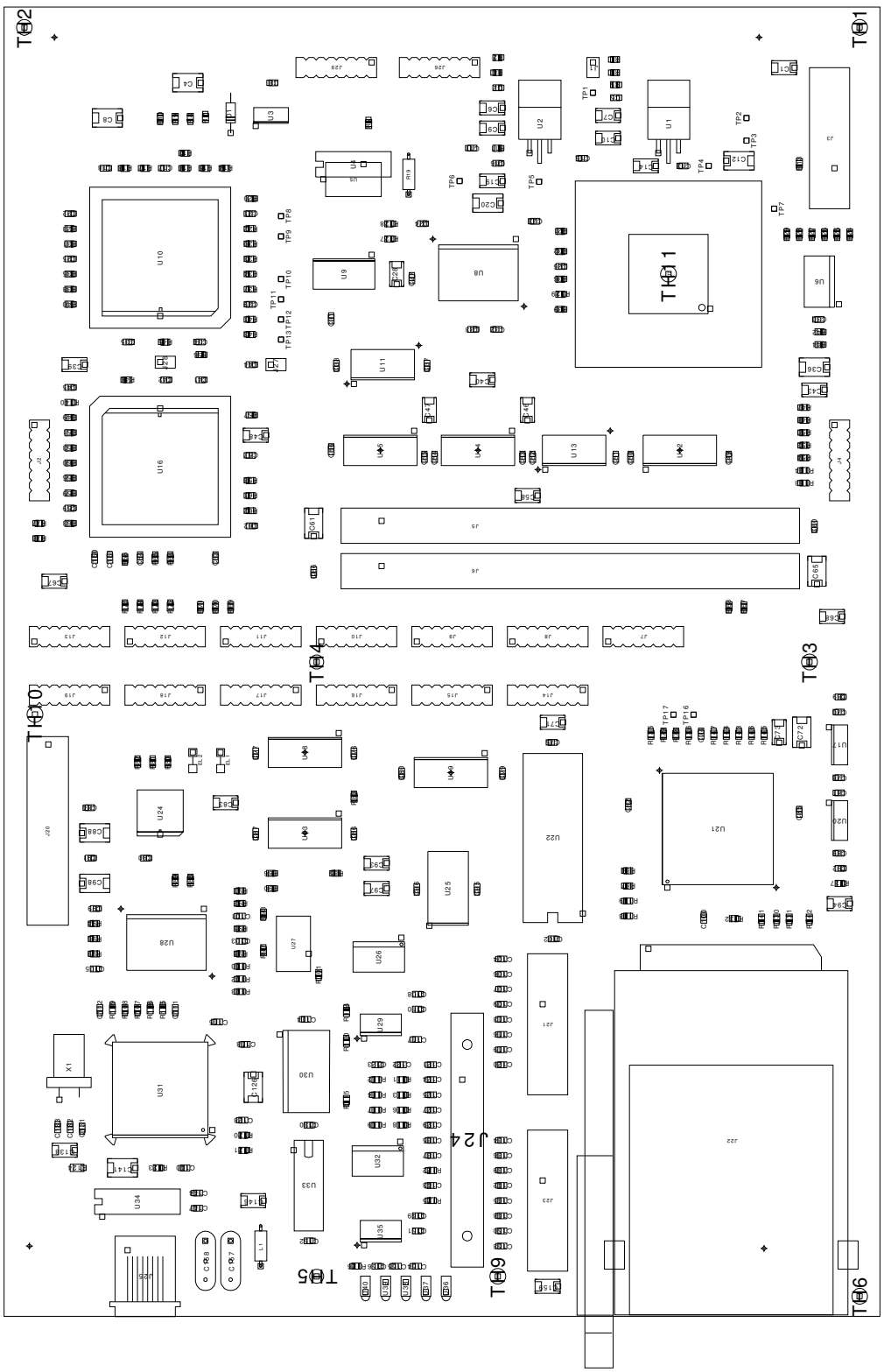
- Inspect the enclosure for physical damage.
- Check the power supply line input voltage is correct for your geography.

Even if your EBSA-110 was provided as a cased unit, you may still want to take the lid off and perform the following checks:

- Internal cables fully attached

Getting Started
1.3 Visual Inspection

Figure 1–1 The EBSA-110 Board



- Links attached and correctly set
- Socketed components properly seated

More details are provided below for this second set of checks.

1.3.1 Cabling

Identify each of the cables and ensure that it is correctly polarized and fully mated to the appropriate connector on the board:

- Power connectors: There are two 6-way power connectors. The 4 black cables align with the connector pins marked 'GND' on the board.
- Reset cable: This 2-way cable can be connected either way around. The board connector is marked 'RESET'.
- COM1 cable: This is a ribbon cable with pin 1 marked by a colored stripe. The board connector is marked 'COM1' and pin 1 is marked by a '1' and a pointer.
- COM2 cable: This is identical to the COM1 cable. The board connector is marked 'COM2' and pin 1 is marked by a '1' and a pointer.
- LPT cable: This is a ribbon cable with pin 1 marked by a colored stripe. The board connector is marked 'LPT1' and pin 1 is marked by a '1' and a pointer.

Refer to Appendix A if you cannot identify the connectors.

1.3.2 Links

Verify that all jumpers are pushed fully down on their mounting posts.

If any jumpers have come off or you are unsure about their positions, refer to Section A.1 for a description of the default settings.

1.3.3 Socketed Components

Verify that any socketed devices (the programmable devices, the DRAM SIMMs and the EPROM (if fitted)) are fully mated in their sockets.

1.3.3.1 The CPU

The EBSA-110 is designed so that the CPU can be soldered directly to the board, or fitted in a socket. Some boards have the CPU fitted in a socket. Do not tamper with the socket or remove the CPU unless you have a good reason to. The CPU is removed by pressing down on the socket frame that surrounds the CPU, then lifting the CPU out using a vacuum pencil. In the absence of a vacuum pencil, you can use something sticky on the blunt end of a pencil. Take care not to bend any CPU leg during this process, as it may result in an intermittent electrical contact when you replace the CPU.

Note

The CPU socket is not polarized. When viewing the EBSA-110 so that the CPU is in the bottom right-hand corner of the board, the CPU is correctly orientated when its pin 1 (marked by a circle) is in the bottom left-hand corner of the chip.

1.4 Power-On

The initial test of the board should be performed with no cables attached to the system, apart from the power cable. Make a note of the jumpers fitted (if any) to J2 pins 9-10, 11-12, 13-14, 15-16. Remove any jumpers from these pins. This will force the board to execute its start-up software and then enter the ARM remote debug stub.

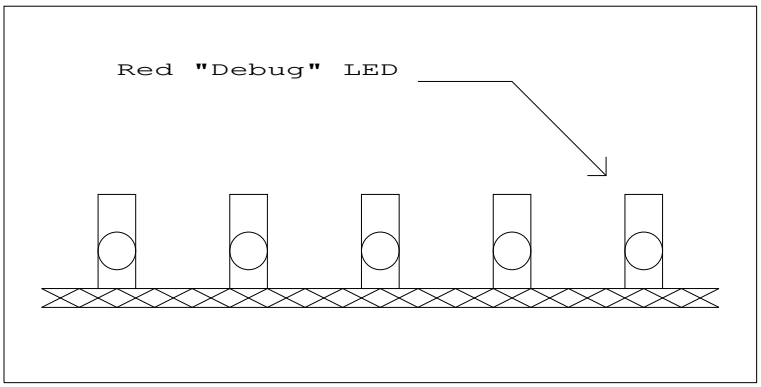
There are a group of 5 LEDs on the rear of the unit. Use Figure 1–2 to identify the 'debug' LED. Watch this LED as you power-on the board. The LED should be off whilst the board is reset, then on for about 0.5s whilst the ARM remote debug stub initializes, then turn off and remain off.

Now attach a terminal or terminal emulator to the COM1 port on the EBSA-110. The terminal should be configured for 9600 baud, 8-bit data, 1 stop bit, no parity, no flow control. After you reset or power-cycle the system, you should see a message like this on the terminal:

```
ARMa100, DEMON V1.1, 0x40020000 bytes RAM, ROM CRC OK, Little endian
```

If the system fails to behave as described, or you wish to perform more thorough testing of the system, run the system diagnostics using the procedure described in Section 8.4. If the system behaves correctly, read on.

Figure 1–2 Position of Debug LED



1.5 Attaching the EBSA-110 to a Terminal or Host System

When the EBSA-110 is used as a software development system, it is directly connected to a host system. The host system may be either a PC or a workstation. In either case, connection can be made in one of the following ways:

- Using a serial port

  In this configuration, a cable connects the COM1 port on the EBSA-110 to the host system. The debug environment uses the serial link as a bidirectional link for commands and responses, and to download images from the host to the EBSA-110.

- Using the Ethernet †

† The Ethernet option is not supported in the initial release of the software.

In this configuration, the EBSA-110 and the host system are both connected to an Ethernet LAN. The debug environment uses the LAN as a bidirectional link for commands and responses and to download images from the host to the EBSA-110.

Refer to Chapter 7 for more information on the software development environment.

When the EBSA-110 is running its power-on diagnostics, status and progress information are written to the COM1 port. The COM1 port should be connected to a terminal or terminal emulator configured for 9600 baud, 8-bit data, 1 stop bit, no parity, no flow control.

Refer to Chapter 8 for more information on the diagnostics.

If the EBSA-110 is used for some standalone application, that application may control all interfaces on the board. Refer to Section 7.2 for information on building standalone applications.

Refer to Section A.9 for details of the cables required in all these configurations.

2
Functional Specification

This chapter describes each functional element of the EBSA-110. More detailed information describing how the board works and how to program it can be found in later chapters of this document. Figure 2–1 is a block diagram of the board.

2.1 CPU

The EBSA-110 uses the SA-110 microprocessor. The board allows the CPU to be operated at any of its 16 core clock frequencies (between 88.3 MHz and 287.0 MHz with the upper limit determined by the speed grade of the CPU that is fitted) and either of its two core voltages (+1.5V or +2.0V). You can set the core voltage and frequency using jumpers on the board.

The EBSA-110 uses the SA-110 pin-bus in the following modes:

- Synchronous bus mode (SA-110 generates the bus clock)
- Enhanced bus mode (cache wrapping and write buffer merging)
- Fastbus mode (delayed address timing)

For special applications, these modes can be changed by rewiring etch links on the board. Refer to Section A.3.

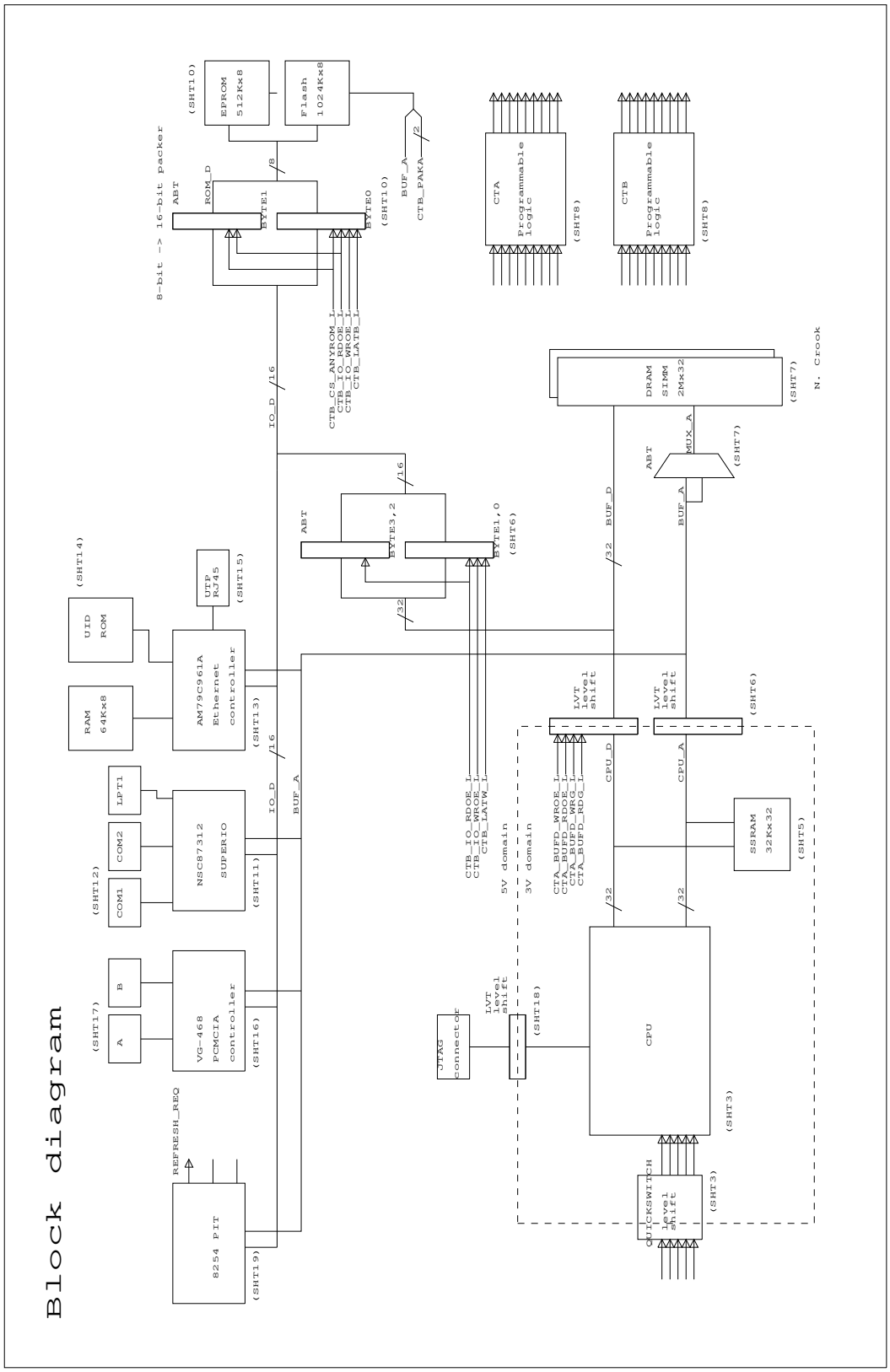
The CPU is packaged in a 144-pin thin quad flat-pack (TQFP). The EBSA-110 provides a dual-footprint layout to allow the CPU to be surface-mounted or to be fitted in a socket.

2.2 Clocks

The EBSA-110 uses the following oscillators:

- 3.6864 MHz (baud-rate) oscillator for the SA-110. The oscillator is a standard surface-mounted (SMT) part with a TTL output and it relies on an external level converter to generate the +3.3V switching levels that are required by the CPU. The CPU uses an internal phase-locked loop (PLL) to generate its core clock from this clock.
- 24.0 MHz oscillator for the SuperI/O controller. This is an SMT part with a TTL output.
- 20.0 MHz crystal for the Ethernet controller. This is a 2-pin through-hole part. The Ethernet controller contains the circuitry to bias this crystal into oscillation.

Figure 2–1 EBSA-110 Block Diagram



The SA-110 generates complementary output clocks **mclk**, **nmclk** by dividing down its core clock. The EBSA-110 is designed to run with a maximum **mclk** frequency of 55 MHz, and this corresponds to core clock divisors of between 2 and 5. **nmclk** is used to clock the on-board state machines and control logic.

Control logic on the board uses **nmclk** to generate a divided-by-seven clock (approximately 6 MHz) which is used in the I/O sub-system to clock the programmable interval timer and the PCMCIA controller.

All clocks have a low fanout, so no external clock buffering is used.

2.3 Reset

A power-on reset is generated from an RC network and schmitt trigger arrangement. A 2-pole 0.1" pitch connector is provided to allow an external normally-open reset switch to be attached. In a lab environment, you can generate reset by shorting these poles with a screwdriver or jumper.

2.4 Power

The EBSA-110 has the following power requirements:

- +5V +/- 5%, @ 1 Amp for the main logic
- +12V +/- 5%, @ 0.5 Amp for the Flash and PCMCIA sockets

The board will function without the +12V supply, with the restrictions that it will not be possible to program the Flash memory, and that there will be no +12V power available for plug-in PCMCIA cards †. If your power supply does not supply +12V, the +12V supply rail should be left disconnected.

Several devices on the board require +3.3V and this is generated from +5V using an adjustable linear regulator.

The CPU core requires +1.5V or +2.0V (depending upon the core clock frequency). This is generated from +3.3V using an adjustable linear regulator. You can select the regulator output using a jumper.

2.4.1 Voltage Domains

The SA-110 outputs use +3.3V switching levels, and the inputs are not +5V-tolerant (they cannot withstand +5V TTL switching levels). The synchronous SRAM's outputs use +3.3V switching levels, but the inputs *are* +5V-tolerant. All the remaining devices on the board are +5V-tolerant.

The following interfacing techniques are used on the board:

- 74LVT devices are used as voltage converters. These parts have a +3.3V supply and are +5V-tolerant. Their output switching range is within the TTL switching threshold and so they can drive TTL-level devices powered from +5V. 74LVT devices are used as interfaces on the CPU address and data buses.
- Output signals from +3.3V devices can be used to drive TTL-level inputs directly. This method is used to interface SA-110 outputs to the system control logic.

† This is most likely to affect PCMCIA disk drives and Flash cards.

- Quality Semiconductor 'QuickSwitch®' devices are used as voltage converters. These devices perform voltage conversion with 'zero' (actually, around 250ps) delay. These parts are used to interface clocks and system control logic outputs to SA-110 inputs.

2.4.2 Power Sequencing

The SA-110 requires two voltage supplies; a +3.3V supply to powers its primary input/output buffers (the pin bus) and a +1.5V or +2.0V supply to power its core. The +3.3V supply must become stable earlier than the core voltage supply. This requirement prevents any possibility of latch-up within parasitic structures on the SA-110. This requirement is satisfied on the EBSA-110 by deriving the +2.0V supply from the +3.3V supply, rather than directly from the +5.0V supply.

There are no other power-sequencing requirements.

2.5 Memory

The EBSA-110 provides three distinct memory regions:

- ROM: non-volatile storage for programs
- SSRAM: fast memory for time-critical code and data
- DRAM: for large code and data sets

2.5.1 ROM

Non-volatile storage for the bootstrap program is provided in an EPROM (512Kbyte) or a Flash ROM (1024Kbyte). Both are 8-bit devices. They are mapped into different regions of the memory map. A user-configurable jumper determines which of them is decoded at address 0 (and therefore supplies the reset vector to the CPU after reset). Normally, the Flash ROM would be used, and the EPROM (which is socketed) would not be fitted. The EPROM is provided for manufacturing use and as insurance against the user unwittingly deleting the primary bootstrap image from the Flash.

CPU reads from the ROM are automatically packed to provide INT4s to the CPU. This is achieved by a state sequence that reads four consecutive bytes from the ROM in response to the CPU access. This packing is transparent to the CPU, and does not affect the format or ordering of data programmed into the ROM.

During CPU writes, consecutive locations in the Flash are sparsely addressed. The mapping is described in Section 3.2.4.

CPU reads from the ROM can be sequential or non-sequential cycles.

CPU writes to the Flash (to program it) must be non-sequential cycles.

2.5.2 SSRAM

A region of fast RAM is implemented using a single 32Kx32 synchronous static RAM (SSRAM) device. This provides 128Kbytes. Alternatively, the circuit-board footprint for the SSRAM can accommodate the 64Kx36 device, when it becomes available. This memory region should be used for speed-critical code and data, when possible.

2.5.3 DRAM

Two 72-pin 5V DRAM memory SIMMs can be fitted. The DRAM memory controller supports Extended Data Out (EDO) and Burst EDO (BEDO) parts with an access time of 60ns or better. The first generation of BEDO parts are available as 2Mx32 SIMMs, so two SIMMs provide a total of 16Mbytes of DRAM.

The memory controller uses the ID signals from a DRAM SIMM to automatically accommodate SIMMs of various sizes.

The memory controller is soft-configurable between EDO and BEDO operation, and the bootstrap software automatically determines the memory type during the power-on sequence.

If two SIMMs are fitted, they must be of the same size and type.

The DRAMs are periodically refreshed using a CAS-before-RAS sequence. The refresh sequence is initiated by a timer which is initialized by the bootstrap software during the power-on sequence.

The DRAM controller is simple-minded in two ways:

- It will always satisfy the RAS precharge time between a pair of accesses, even if the accesses are to separate physical banks of DRAM.
- It does not interleave (neither between banks in a SIMM nor between SIMMs)

Sequential cycles from the CPU are always performed as page-mode DRAM accesses and conversely, the DRAM controller will always terminate a page mode cycle when the CPU terminates its sequential access (it does not speculatively keep the page open).

2.5.4 Memory Map Switching

Immediately after reset, the CPU fetches its reset vector from address 0. During normal operation, it is preferable to have RAM at address 0. The EBSA-110 control logic allows the memory map to be switched to accommodate these requirements. The process is described in Section 3.1.1.

2.6 I/O Sub-System

All I/O on the EBSA-110 is performed as programmed I/O under the control of the CPU. The I/O sub-system provides the following resources:

- 2 RS232 serial ports with PC-style 9-way D-connectors
- 1 bidirectional PC-style parallel printer port with 25-way D-connector
- Ethernet port with 10BaseT (twisted pair) media support on an RJ45 jack connector
- Ethernet Unique ID ROM
- PCMCIA controller
- 3-channel counter/timer
- Soft-programmable outputs
- Soft-readable inputs
- Architectural compliance verification facilities

2.6.1 Serial Ports and Printer Port

The serial and parallel ports are implemented using a National Semiconductor PC87312 SuperI/O III chip. This part also includes a floppy disk controller and decodes for an IDE interface but these functions are not used in this design.

2.6.2 Ethernet Port and UID ROM

The Ethernet port is implemented using a National Semiconductor Am79C961A 'ISA-net' controller operated in shared-memory mode. The UID ROM is accessed via the AM79C960. In shared-memory mode, the Am79C961A uses external RAM as temporary data storage. This external RAM is attached to the Am79C961A and is accessible to the CPU via the Am79C961A.

The Ethernet port only supports the 10BaseT (UTP) media. 10BaseT requires a hub-based topology but can also be used point-to-point between two nodes.

2.6.3 PCMCIA Controller

Two PCMCIA sockets are supported, using a Vadem VG468 controller. Socket A is the socket closer to the board. Both socket A and socket B can support Type I, II and III PCMCIA cards. A Type III card in Socket A will mechanically obscure Socket B, preventing it from being used.

2.6.4 Counter/Timer

The counter/timer is implemented using an Intel® 82C54 Programmable Interval Timer (PIT). The PIT is a three-channel device. One channel is dedicated to producing a periodic signal for the memory controller in order to initiate DRAM refresh. The other two channels are available for application software; their timeout outputs can be used to generate interrupts to the CPU.

2.6.5 Soft I/O

The soft I/O is implemented within programmble logic. There are 4 read/write outputs (the written value can be read back by the CPU) and 4 read-only inputs. These signals are used to control various on-board functions, including an LED. The bits are described in Section 3.7.

2.6.6 Architectural Compliance Verification Facilities

The programmable control logic implements some facilities which are used for architectural verification. This functionality duplicates facilities that exist in the silicon development environment for the SA-110. These facilities are:

- The ability to generate ABORTs to the CPU on certain read and write cycles.
- The ability to generate interrupts as the result of a timeout on a programmable down-counter that has been loaded by the CPU and then decremented at the rate of the system bus clock.

These facilities are described in more detail in Section 3.2.6, Section 3.2.7 and Chapter 4.

These facilities are not required during normal operation and the resources that they use within the programmble logic are reassigned to provide an interrupt controller.

2.7 Interrupts

The EBSA-110 includes a simple interrupt controller that can be used to support re-entrant interrupts and lowest-latency ‘priority levels’ on the flat interrupt structure that the single IRQ interrupt provides. The interrupt controller is described in Chapter 4.

2.8 JTAG

A 7x2 header is fitted to provide electrical access to the EBSA-110 JTAG port. The SA-110 only provides boundary scan access.

2.9 Hardware Debug Support

The EBSA-110 provides connectors and test points to make it easy to attach a logic analyzer to the board. The pickups are of two types:

- The buffered address and data buses and some other low-speed signals are routed to 16-pin 2x8 header plugs. These are suitable for direct connection to a Tektronix DAS logic state analyzer and can be connected to any other analyzer using ‘grabber’ probes.
- Various control signals have etch vias on their routing to allow a Harwin post to be soldered into the board. This allows a logic analyzer to be attached, but does not interfere with the high-speed signals by adding additional etch length and capacitance.

As always, the additional load imposed by test equipment may interfere with the normal operation of the board.

2.10 Expansion

The EBSA-110 has no standard expansion capability. However, the control signals present on the debug connectors are sufficient to allow I/O devices to be interfaced via a mezzanine PCB. This is discussed in Section 10.4.

2.11 Control Logic

All of the control logic for the CPU, memory and I/O sub-systems is implemented within two 84-pin PLCC programmable logic parts. The source files for these parts are provided as part of the design database, allowing you to modify them if required.

The control logic is described in detail in Section 10.2.

2.12 Endian Issues

The SA-110 can be configured as a little-endian or a big-endian machine †. The

† The terms ‘little-endian’ and ‘big-endian’ have been adopted by the computer industry to describe the way in which bytes are ordered within larger data units. Machines which treat the byte on the low-order data bus lines (the byte with the lowest address) as the least-significant byte are termed ‘little-endian’. Machines which use the opposite ordering (most-significant byte at the lowest address) are termed ‘big-endian’. The VAX is a little-endian machine, as is the Intel x86 family. The Motorola 68xx family is big-endian. Most modern RISC implementations are ambi-endian; they can be configured to run with either endian-ness. The term ‘endian’ comes from Jonathan Swift’s "Gulliver’s Travels". The two great empires of Lilliput and Blefuscu were engaged in a most obstinate war as the result of an edict published by the Emperor of Lilliput. In this edict, the Emperor did command "all his subjects, upon great penalties, to break the smaller end of their eggs". Many hundred large volumes have been published upon this controversy: but the books of the Big-Endians have been long forbidden. Swift’s

recommended mode of operation is little-endian, and this is the default state after reset. The EBSA-110 can be operated with either endian-ness, with no impact on the hardware. The on-board software is configured for little-endian operation. If big-endian operation is required, the following changes must be made:

- The definitions of all I/O addresses must be changed. For example, a byte-wide I/O device at address 0 on a little-endian machine will be at address 3 on a big-endian machine.
- The byte ordering within the ROM must be changed; the ROM packer hardware (refer to Section 2.5.1) packs bytes from the ROM into a little-endian order, so a big-endian image must be pre-scrambled to compensate (the alternative is to redesign the packer sequencer in the control logic).

2.13 LEDs

The EBSA-110 has 4 LEDs, which are used to provide information on the status of the Ethernet link, and a further 1 LED that is used to provide debug information. Section A.7 describes the LEDs.

2.14 On-Board Software

The EBSA-110 on-board software is programmed into the Flash ROM. The Flash can contain a number of independent images. At a minimum, the Flash contains a program called the Primary Bootstrap Loader (PBL). The PBL can load and start a specified image stored in Flash. By default, it starts up the ARM remote debugger stub. A power-on diagnostic suite is also programmed into the Flash and can be selected by changing jumpers on the board.

invention was a satire on the Spanish war of succession and a commentary on the history of religious controversy in England. The analogy to computer byte ordering is poor. Whilst it is true "That all true believers shall break their eggs at the convenient end", (in other words, endian-ness in egg consumption is irrelevant) the relative endian-ness of a pair of computers can be important when they want to exchange data.

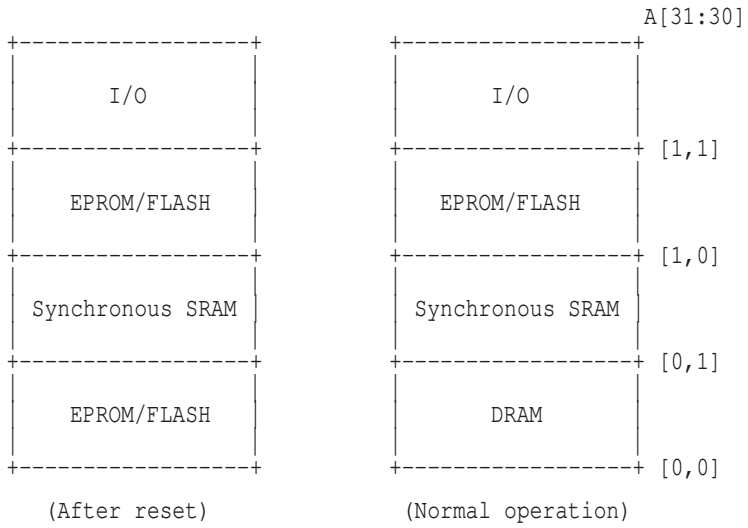
3
Programmers' Guide

This chapter is a reference for programmers. It describes the memory map of the board, the reset state of the system and software programming restrictions.

3.1 Memory Map

The SA-110 has a 32-bit address bus with byte addressability and a 32-bit data bus. The address space is divided into quadrants based on A[31:30] and devices have multiple aliases in each quadrant.

An overview of the memory map is shown below:



A full table of addresses within the memory space is shown in Table 3–1. Section 3.2 describes how the memory space is decoded.

Table 3–1 Memory Map

Address	Name	Function
0xffff.ffff	ISAIO_END	Last location in ISAIO space
0xf3c0.0000	TRICK7	Interrupt control
0xf380.0000	TRICK6	Interrupt control
0xf340.0000	TRICK5	Interrupt control
0xf300.0000	TRICK4	Interrupt control
0xf2c0.0000	TRICK3	Interrupt control

(continued on next page)

Table 3–1 (Cont.) Memory Map

Address	Name	Function
0xf280.0000	TRICK2	Interrupt control
0xf240.0000	TRICK1	Soft registers
0xf200.000d	PIT_CTLW	Control Word register
0xf200.0009	PIT_CNT2	Counter 2 register
0xf200.0005	PIT_CNT1	Counter 1 register
0xf200.0001	PIT_CNT0	Counter 0 register
0xf200.0000	TRICK0	PIT registers base
0xf200.0000	ISAIO_EDBASE	Start of external-decode ISAIO space
0xf000.0000	(free)	This area is free for other ISAIO stuff
0xf000.14f1	PNP_WRDATA	Plug-and-play Auto-Configuration write-data port (write-only)
0xf000.0ffc	SIO_COM1END	Last SuperI/O COM1 register
0xf000.0fe0	SIO_COM1BASE	Start of SuperI/O COM1 registers
0xf000.0e64	SIO_DATA	SuperI/O (configuration) data register
0xf000.0e60	SIO_INDEX	SuperI/O (configuration) index register
0xf000.0dfc	SIO_LPT2END	Last SuperI/O LPT2 register
0xf000.0de0	SIO_LPT2BASE	Start of SuperI/O LPT2 registers
0xf000.0bfc	SIO_COM2END	Last SuperI/O COM2 register
0xf000.0be0	SIO_COM2BASE	Start of SuperI/O COM1 registers
0xf000.07c1	PCMCIA_DATA	PCMCIA controller Data register
0xf000.07c0	PCMCIA_INDEX	PCMCIA controller Index register
0xf000.04f1	PNP_ADDRESS	Plug-and-play Auto-Configuration address port (write-only)
0xf000.046c	NET_IDP	ISACSR register data port
0xf000.0468	NET_RESET	Reset the controller
0xf000.0464	NET_RAP	Register address port (shared by RDP and IDP)
0xf000.0460	NET_RDP	Register data port
0xf000.0440	NET_UID	Ethernet UID address PROM (16 bytes)
0xf000.0405	PNP_RDDATA	Plug-and-play Auto-Configuration read-data port (read-only)
0xf000.0000	ISAIO_SDBASE	Start of self-decode ISAIO space
0xf000.0000	ISAIO_BASE	Start of ISAIO space
0xefff.ffff	ISAMEM_END	Last location in ISAMEM space
0xea00.0000	(free)	This area is free for other ISAMEM allocation
0xe9ff.ffff	PCMCIAMEM_END	Last location in PCMCIA reserved space
0xe800.0000	PCMCIAMEM_BASE	Start of PCMCIA reserved space
0xe7ff.ffff	EBUFMEM_ALIASE	End of last Ethernet buffer memory alias

(continued on next page)

Table 3–1 (Cont.) Memory Map

Address	Name	Function
0xe001.ffff	EBUFMEM_END	Last location of Ethernet buffer memory
0xe000.0000	EBUFMEM_BASE	Start of Ethernet buffer memory
0xe000.0000	ISAMEM_BASE	Start of ISAMEM space
0xdfff.ffff	RW_ABORT_END	Last location in Read/Write-Abort space
0xd000.0000	RW_ABORT_BASE	Start of Read/Write-Abort space
0xcfff.ffff	R_ABORT_END	Last location in Read-Abort space
0xc000.0000	R_ABORT_BASE	Start of Read-Abort space
0x8fff.ffff	ROM_ALIASE	End of last ROM alias
0xa007.ffff	EPROM_END	Last location in EPROM memory
0xa000.0000	EPROM_BASE	Start in EPROM memory
0x800f.ffff	FLASH_END	Last location in Flash memory
0x8000.0000	FLASH_BASE	Start of Flash memory
0x7fff.ffff	SSRAM_ALIASE	End of last SSRAM alias
0x4002.0000	SSRAM_ALIASS	Start of first SSRAM alias
0x4001.ffff	SSRAM_END	Last location of SSRAM memory
0x4000.0000	SSRAM_BASE	Start of SSRAM memory
0x3fff.ffff	DRAM_ALIASE	End of last DRAM alias
0x00ff.ffff	DRAM_16M	End of first 16Mbytes of DRAM
0x00bf.ffff	DRAM_12M	End of first 12Mbytes of DRAM
0x007f.ffff	DRAM_8M	End of first 8Mbytes of DRAM
0x003f.ffff	DRAM_4M	End of first 4Mbytes of DRAM
0x0000.0000	DRAM_BASE	Start of DRAM memory

3.1.1 Memory Map After Reset

After reset, the SA-110 fetches its reset vector from address 0. Therefore, it is necessary to have ROM at this address immediately after reset. This is achieved by decoding the EPROM/Flash in two quadrants immediately after reset. The memory map is switched under software control. The first write performed by the SA-110 (after reset) will switch the memory map to normal operation. The memory map switch occurs *after* the write cycle completes. Therefore, if the write is to address 0, the data will not be written to the DRAM, but will be written to EPROM/Flash space (that is, the write will be ignored †). If two writes to address 0 are performed, the second will successfully write data to the DRAM.

Due to the very low performance of ROM accesses (each 32-bit access is performed by packing data from 4 successive locations in the 8-bit ROM) the image to be executed should normally be copied into RAM first. Before starting a copy, the software should jump to the high-order alias of the ROM and switch the address map.

It is not possible to reverse this address map switching process under software control.

† The Flash ROM is sensitive to writes; that is how it is given commands and programmed. However, each of the commands requires a pair of bus cycles with specific (different) data so this switching mechanism is safe.

3.1.2 Characteristics of Memory

The SSRAM is small and fast (factor of 1), the DRAM is large and quite fast (factor of 1/2 - 1/3), the EPROM/Flash is very slow (factor of 1/16).

Because of the slow speed of the EPROM/Flash access, is is best to copy images from EPROM/Flash into SSRAM or DRAM and execute them from there.

3.2 Memory Map Decodes

This section describes how the memory space is decoded. It provides essentially the same information as Table 3–1, but in a greater level of detail.

The diagrams in the following sections use this key:

```
0 - must be 0
1 - must be 1
A - significant to address a location within device
a - significant to address optional locations within device (eg alternative
    memory size)
X - don't-care: significant to address aliases within device
- - not available (A1, A0 represent byte lanes on the 32-bit bus)
```

3.2.1 DRAM Space

```
 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X X X X X X X X X X X X X X X X X X X X X X X X - -|
+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X X X X X 0 A A A A A A A A A A A A A A A A A A A A - -| (B)EDO 1Mx32 SIM0
+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X X X X X 1 A A A A A A A A A A A A A A A A A A A A - -| (B)EDO 1Mx32 SIM1
+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X X X X 0 A A A A A A A A A A A A A A A A A A A A - -| (B)EDO 2Mx32 SIM0
+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X X X X 1 A A A A A A A A A A A A A A A A A A A A - -| (B)EDO 2Mx32 SIM1
+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X X X 0 A A A A A A A A A A A A A A A A A A A A - -| BEDO 4Mx32 SIM0
+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X X X 1 A A A A A A A A A A A A A A A A A A A A - -| BEDO 4Mx32 SIM1
+-------+-------+-------+-------+-------+-------+-------+---+---+
```

This decodes the DRAM. Zero, one or two DRAM SIMMs may be fitted. They must be 72-pin +5V types, either x32 or x36. The hardware automatically detects and accommodates 1Mx32, 2Mx32 and 4Mx32 SIMMs. This means that the board can accommodate up to 32Mbytes of DRAM. The DRAM controller can support EDO and BEDO parts. If two SIMMs are fitted, they must be the same size and type. The memory is contiguous and byte addressable. There are multiple aliases of the DRAM in the system address space. The hardware reads all bytes of a longword during reads, and performs byte masks during writes.

The DRAM is not accessible immediately after reset (see Section 3.1.1).

3.2.2 DRAM Configuration Space

```
 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X A A A A A X X X X X X X X X X X D D D D D D D D - -| DCBR=1
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

When BEDO DRAMs are fitted, a special pair of cycles, CBR (CAS-before-RAS) and WCBR (write CAS-before-RAS), must be performed in order to configure the DRAMs. EDO DRAMs require no configuration.

These cycles are performed by setting the DCBR (do CAS-before-RAS) bit in the Soft register and then performing read and write cycles to the normal DRAM space.

When performing these configuration cycles, the data written is irrelevant; the value on the low-order address lines configures the DRAMs. The following addresses should be used:

```
 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 X X X A A A A A X X X X X X X X X X X 0 0 1 0 0 0 0 0 - -| DCBR=1
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

The effect of this write is to set the BEDO DRAM burst ordering to 'linear'. After performing the write (or all writes) a read should be performed to the same address. This restores the DRAM to normal operation. After all CBR and WCBR cycles have been completed, the DCBR bit should be cleared.

The configuration process is described in detail in Section 5.3.

3.2.3 Synchronous SRAM (SSRAM) Space

```
 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 1 X X X X X X X X X X X a A A A A A A A A A A A A A A A - -|
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

This decodes the synchronous SRAM (SSRAM). There are multiple aliases. The SSRAM is contiguous and byte addressable. The hardware reads all bytes of a longword during reads, and performs byte masks during writes. The SSRAM is either 128Kbyte or 256Kbyte. 128Kbyte is the normal size.

3.2.4 EPROM/Flash Space

```
 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 0 S X X X X X X X X X X X X X X X X X X X X X X X X X X - -|
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 S X X X X X X X X X X X X X X X X X X X X X X X X X X - -| (at reset)
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 0 X X X X X b b X X A A A A A A A A A A A A A A A A A - -| (Flash)
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|0 0 1 X X X X X X X X X A A A A A A A A A A A A A A A A A - -| (EPROM)
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

This decodes the EPROM/Flash. In normal operation, S=0 decodes the Flash, and S=1 decodes the EPROM. Therefore, the Flash will be decoded at address 0 after reset and so the system will boot from Flash. If the jumper EPROM_ BOOT is fitted, the behavior of S is inverted, and so the system will boot from EPROM. Usually, there is no need to fit the EPROM. The EPROM socket and EPROM_BOOT facility are provided for manufacturing and as an 'emergency repair' mechanism in case the Flash is programmed with a bad image.

The EPROM is a 512Kbyte device, and is read-only. The Flash is a 1Mbyte device and is readable and sector writeable. Refer to the manufacturer's data sheet for programming details.

Since the EPROM and the Flash are byte-wide devices, they are accessed via a packer mechanism which assembles a 32-bit data unit from an aligned address. This is transparent to the software during reads, but affects the way in which byte accesses are performed when writing to the Flash.

When writing to the Flash, the data must always be supplied on the low-order byte lane. This corresponds to byte writes to addresses 0, 4, 8 and so on. The actual byte addressability is provided by the address lines marked 'b' above. Therefore, the first 9 byte locations in Flash are addressed as:

```
0x0000.0000
0x0040.0000
0x0080.0000
0x00c0.0000
0x0000.0004
0x0040.0004
0x0080.0004
0x00c0.0004
0x0000.0008
```

Writes to the Flash are always performed as non-sequential cycles, so the Flash must not be placed in cacheable space during programming and the write buffer must not be used for writes to the Flash.

3.2.5 I/O Space

```
 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 X X X X X X X X X X X X X X X X X X X X X X X X X X X X - -| IO quadrant
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 1 1 X X X X X X X X X X X X X X X X X X X X X X X X X X - -| ISAIO
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 1 0 X X X X X X X X X X X X X X X X X X X X X X X X X X - -| ISAMEM
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 0 1 X X X X X X X X X X X X X X X X X X X X X X X X X X - -| RW_ABORT
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 0 0 X X X X X X X X X X X X X X X X X X X X X X X X X X - -| R_ABORT
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

All devices in the I/O space quadrant are 8-bit or 16-bit devices and are physically attached to one or both of the two low-order byte lanes of the data bus. Since the data bus is actually 32-bits wide, addresses in I/O space are not contiguous.

Byte masks are only partially decoded in this space and so writes to incorrect addresses are more likely to corrupt a device than to be ignored.

In a little-endian system, all valid byte addresses in this space end in 0b00 or 0b01, and all valid half-word addresses end in 0b00.

The I/O space quadrant is further subdivided into quadrants:

- ISAIO space
- ISAMEM space
- RW_ABORT space
- R_ABORT space

3.2.6 RW_ABORT Space

The RW_ABORT space is an architectural compliance verification facility; it is unlikely to be useful in normal applications.

Any reads or writes within this address range will result in an abort exception. Sequential cycles to this address space will result in an abort exception for each data beat of the sequential cycle.

```
 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 X X X X X X X X X X X X X X X X X X X X X X X X X X X - -| IO quadrant
+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 0 1 X X X X X X X X X X X X X X X X X X X X X X X X X - -| RW_ABORT
+-------+-------+-------+-------+-------+-------+-------+---+---+
```

3.2.7 R_ABORT Space

The R_ABORT space is an architectural compliance verification facility; it is unlikely to be useful in normal applications.

Any reads within this space will result in an abort exception. Writes will be ignored. Sequential writes to this address space will be ignored. Sequential reads to this address space will result in an abort exception for each data beat of the sequential cycle.

```
 A A A A A A A A A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 X X X X X X X X X X X X X X X X X X X X X X X X X X X - -| IO quadrant
+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 0 0 X X X X X X X X X X X X X X X X X X X X X X X X X - -| R_ABORT
+-------+-------+-------+-------+-------+-------+-------+---+---+
```

3.2.8 ISAMEM Space

The ISAMEM space is used to access devices which behave like ISA-bus memory devices. This includes PCMCIA card resources (accessed through the PCMCIA controller) and the Ethernet controller's buffer memory.

The address ranges occupied by PCMCIA card resources and the Ethernet controller are software configurable.

Cycles in ISAMEM space are controlled using **memr_l**, **memw_l**, **zws_l** and **rdy**.

```
 A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 X X X X X X X X X X X X X X X X X X X X X X X X X X X X - -| IO quadrant
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 1 0 X X X X X X X X X X X X X X X X X X X X X X X X X X X - -| ISAMEM
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

3.2.8.1 EBUFMEM Space

The Ethernet controller maintains transmit and receive data structures in a piece of shared memory. This memory is decoded within the ISAMEM space in the address range where **a[27]**=0, that is:

```
 A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 1 0 0 X X X X X X X X X A A A A A A A A A A A A A A A A - -| EBUFMEM
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

The EBUFMEM is accessed using byte and half-word read/writes. It appears in the address space as 16-bit memory and so its locations are non-contiguous.

The algorithm to convert an offset *a* in EBUFMEM space into an address is:

```
address = (a & 1) | ((a & 0xffff.fffe) << 1) | 0xe000.0000
```

3.2.8.2 PCMCIAMEM Space

Resources on PCMCIA cards are accessed by configuring the PCMCIA controller to open windows in the ISAMEM space. The addresses must be selected to fall into an address range where **a[27]**=1 †, that is:

```
 A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 1 0 1 X X A A A A A A A A A A A A A A A A A A A A A A A A - -| PCMCIAMEM
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

Refer to Section 5.7.2 for a worked example of setting a PCMCIA window.

The PCMCIAMEM is accessed using byte and half-word read/writes. It appears in the address space as 16-bit memory and so its locations are non-contiguous.

The algorithm to convert an offset *a* in PCMCIAMEM space into an address is:

```
address = (a & 1) | ((a & 0xffff.fffe) << 1) | 0xe000.0000
```

3.2.9 ISAIO Space

The ISAIO space is used to access devices which behave like ISA-bus I/O devices. This includes the PIT, the Soft I/O register, the interrupt control registers, the SuperI/O registers, the PCMCIA internal control registers, PCMCIA card resources (accessed through the PCMCIA controller) and some of the Ethernet controller registers.

The address ranges occupied by PCMCIA card resources and the Ethernet controller are software configurable.

† There is no hardware checking of this restriction, because the PCMCIA controller cannot decode **a[27]**. If you try to map memory into the area of ISAMEM space where **a[27]**=0, you will cause bus contention with the EBUFMEM area.

Cycles in ISAIO space are controlled using **ior_l**, **iow_l**, **zws_l** and **rdy**.

```
 A A A A A A A A A A A A A A A A A A A A A A
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 A A A A A A A A A A
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 1 1 X X X X X X X X X X X X X X X X X X X X X X X X X X - -| ISAIO
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 1 1 X X 0 X X X X X X X X X X X X A A A A A A A A A A - -| self-decode
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
|1 1 1 1 X X 1 A A A X X X X X X X X X X X X X X X X X X X - -| external-decode
+-------+-------+-------+-------+-------+-------+-------+-------+---+---+
```

The ISAIO space is divided into two regions:

- The first region is for ISA-like devices that decode their own addresses based on the expectation that the whole I/O space is limited to a 10-bit decode (I/O addresses 0-0x3FF). **a[25]**=0 is used to select these devices.
- The second region is for devices that require external address decode logic.

The following devices sit in self-decoding space:

- SuperI/O controller
- Ethernet controller
- PCMCIA controller (including resources on PCMCIA cards)

The following devices sit in external-decode space:

- PIT
- Soft register
- Interrupt control registers

These devices are further decoded using **a[24:22]**. The addresses of these registers are shown in Table 3–2. For more information on the PIT, soft register and interrupt control registers, refer to Section 3.3, Section 3.7 and Chapter 4.

Table 3–2 Addresses in External-Decode Space

Address	Name	Function
0xf3c0.0000	TRICK7	Interrupt control
0xf380.0000	TRICK6	Interrupt control
0xf340.0000	TRICK5	Interrupt control
0xf300.0000	TRICK4	Interrupt control
0xf2c0.0000	TRICK3	Interrupt control
0xf280.0000	TRICK2	Interrupt control
0xf240.0000	TRICK1	Soft registers
0xf200.0000	TRICK0	PIT registers

3.3 The PIT Registers

The Programmable Interval Timer (PIT) is an Intel 82C54. It is physically attached to bits 15:8 of the data bus.

The PIT has three timer channels. All are clocked by **ctb_clkby7**, which has a nominal frequency of 7.6 MHz. Channel 0 is used to provide a refresh request for the DRAM; its configuration is described in Section 5.3. Channels 1 and 2 are uncommitted, and may be used to generate interrupts to the CPU.

The PIT is decoded in external-decode ISAIO space. It has 4 memory-mapped registers which are accessed by byte read/writes to the addresses shown in Table 3–3.

Table 3–3 PIT Internal Registers

Address	Name
0xf200.000d	Control Word register
0xf200.0009	Counter 2 register
0xf200.0005	Counter 1 register
0xf200.0001	Counter 0 register

3.4 The SuperI/O Registers

The SuperI/O controller is a National Semiconductor PC87312. It is physically attached to bits 7:0 of the data bus.

The SuperI/O controller has 4 groups of registers which are decoded in self-decoding ISAIO space. It is accessed using byte read/writes to the addesses shown in Table 3–4.

Table 3–4 SuperI/O Registers

Port Address	ISAIO Address	Name	Function
0x03f8	0xf000.0fe0	COM1	COM1 UART base address
0x02f8	0xf000.0be0	COM2	COM2 UART base address
0x0378	0xf000.0de0	LPT2	LPT2 printer port base address
0x0398	0xf000.0e60	INDEX	Configuration index register address
0x0399	0xf000.0e64	DATA	Configuration data register address

The positions of these registers are software configurable, but you are recommended to leave them at their power-on defaults.

For the SuperI/O controller, use this algorithm to convert an I/O address, *a*, into an ISAIO address:

```
ISAIO_address = (a << 2) | 0xf000.0000

'take the port address, left shift by 2 bits then OR with 0xf000.0000'
```

For programming information, refer to the manufacturer's data sheet.

3.5 The Ethernet Controller Registers

The Ethernet controller is an AMD Am79C961A. It is physically attached to bits 15:0 of the data bus.

The Ethernet controller is decoded in self-decoding ISAIO space. It has 2 groups of registers which are accessed by byte and half-word read/writes:

- 3 Auto-Configuration registers. These registers are used to access indirectly the Plug-and-play (PNP) configuration registers.
- Access registers. These registers are used to access directly the Unique ID (UID) PROM and to access indirectly the internal resources of the Ethernet controller.

The register addresses are shown in Table 3–5.

Table 3–5 Ethernet Controller Registers

Port Address	ISAIO Address	Name	Function
0x0279	0xf000.04f1	PNP_ADDRESS	Plug-and-play Auto-Configuration address port (write-only)
0x0a79	0xf000.14f1	PNP_WRDATA	Plug-and-play Auto-Configuration write-data port (write-only)
0x0203	0xf000.0405	PNP_RDDATA	Plug-and-play Auto-Configuration read-data port (read-only)
0x0220	0xf000.0440	NET_UID	Ethernet UID address PROM (16 bytes)
0x0230	0xf000.0460	NET_RDP	Register data port
0x0232	0xf000.0464	NET_RAP	Register address port (shared by RDP and IDP)
0x0234	0xf000.0468	NET_RESET	Reset the controller
0x0236	0xf000.046c	NET_IDP	ISACSR register data port

The positions of the registers are software configurable, and must be configured after reset using the process described in Section 5.8. The addresses shown assume that the default values in Section 5.8 are used.

The addresses shown in this table assume that 16-bit I/O is used to access the registers. If 8-bit I/O is used, registers at odd addresses must be accessed using longword stores and loads. For the Ethernet controller, this affects the plug-and-play registers during configuration. Refer to Section 3.9.1 for details.

This is the algorithm used to convert a memory address, *a*, to an ISAMEM address:

```
ISAMEM_address = (a & 1) | ((a & 0xffff.fffe) << 1) | 0xe000.0000
```

This is the algorithm used to convert an I/O address, *a*, into an ISAIO address:

```
ISAIO_address = (a & 1) | ((a & 0xffff.fffe) << 1) | 0xf000.0000
```

For programming information, refer to the manufacturer's data sheet.

3.6 The PCMCIA Controller Registers

The PCMCIA controller is a Vadem VG-468. It is register-compatible with the Intel 82365SL. It is physically attached to bits 15:0 of the data bus.

The PCMCIA controller has two internal registers which are decoded in self-decoding ISAIO space. They are accessed by byte read/writes to the addresses shown below:

- The Index register is at port address 0x03e0, corresponding to ISAIO address 0xf000.07c0.
- The Data register is at port address 0x03e1, corresponding to ISAIO address 0xf000.07c1.

The position of these registers is software configurable, but you are recommended to leave them at their power-on defaults.

PCMCIA cards plugged into the controller will require address space allocation in ISAMEM and/or ISAIO space. Addresses are allocated under software control, and must be selected so that they do not clash with any other devices in the system. Once a device has been allocated space, it can be accessed by byte and word read/writes.

The addresses shown above assume that 16-bit I/O is used to access the registers. If 8-bit I/O is used, registers at odd addresses must be accessed using longword stores and loads. For the PCMCIA controller, this affects all accesses to the data register. Refer to Section 3.9.1 for details.

For programming information, refer to the manufacturer's data sheet.

3.7 The Soft Register

The Soft register is used for board configuration and control. This register is accessed by byte reads and writes to address 0xf240.0000. All output bits are automatically reset to 0 after reset or power-on. The bit assignment of this register is shown in Table 3–6.

Table 3–6 Bit Assignment of Soft Register

Bit	Name	Type	Description
7	LED_L	Read/Write	Write a 0 to illuminate the red 'debug' LED, write a 1 to extinguish it.
6	SPKR	Read/Write	Write a 1 to this bit to enable the speaker output. When this bit is set, the speaker is driven from the output of PIT channel 1. This facility is only available when using CTB_OS. The state of this bit does not affect the ability of the PIT channel 1 to generate interrupts.
5	DCBR	Read/Write	Write a 1 to this bit so that accesses to DRAM space are performed as CAS-before-RAS cycles; used for DRAM configuration only. **This bit must only be set when the refresh counter is disabled.**
4	BURST	Read/Write	Write a 1 to this bit when the system is fitted with BEDO DRAMs.

(continued on next page)

Table 3–6 (Cont.) Bit Assignment of Soft Register

Bit	Name	Type	Description
3	JMP15	Read-only	When read as a '0', indicates that a jumper is fitted on J4 pins 15-16. Writes are don't care. This jumper is unassigned by on-board software and can be used by application software.
2	JMP13	Read-only	When read as a '0', indicates that a jumper is fitted on J4 pins 13-14. Writes are don't care. This jumper is used by the PBL.
1	JMP11	Read-only	When read as '0', indicates that a jumper is fitted on J4 pins 11-12. Writes are don't care. This jumper is used by the PBL.
0	JMP09	Read-only	When read as '0', indicates that a jumper is fitted on J4 pins 9-10. Writes are don't care. This jumper is used by the PBL.

3.8 Reset State

When the EBSA-110 is held in reset, the SA-110 PLL stops. During this time, the external state machines and the PIT will not be clocked. Therefore, the contents of DRAM will be UNKNOWN after reset.

Since the EBSA-110 can be reset asynchronously with respect to CPU bus activity, it is possible that a reset will also corrupt the contents of the SSRAM.

3.9 Software Restrictions

The EBSA-110 does not support sequential cycles (store multiple or load multiple) into I/O space or ROM/Flash space. The only exeception to this is the abort space, which does support sequential cycles.

3.9.1 8-bit Accesses to Odd Addresses

A problem occurs in some peripherals (or some registers within peripherals) which are only designed to accommodate 8-bit I/O cycles. For these accesses, the peripheral expects to transfer data on the low-order byte lane, byte lane 0. If the register address is even, this will work correctly. However, if the register address is odd, the CPU will expect to transfer data on byte lane 1.

The EBSA-110 supports 16-bit I/O to all devices. In 16-bit I/O, data is transferred on the natural byte lane. You can perform 16-bit I/O by using LDB/STB and LDH/STH instructions (8-bit and 16-bit loads and stores).

The EBSA-110 provides hardware support which also allows 8-bit I/O to odd register addresses in the Ethernet and PCMCIA controllers. You can perform 8-bit I/O cycles to odd addresses by using LD/ST instructions (32-bit loads and stores) to an address that is rounded down from the odd address. The significant data is presented in the low byte of the 32-bit data. On reads, the high-order bytes must be masked off under software control.

Accesses to even addresses are always achieved using LDB/STB and LDH/STH instructions; there is no difference between 8-bit I/O and 16-bit I/O in this case.

For example:

```
;; PCMCIA controller is accessed via index register (at 0x3E0) and data
;; register (at 0x3E1). The data register is at an odd address so
;; it must be accessed using the 8-bit I/O trick.
PCMCIA_INDEX16       EQU  0xf00007c0 ;; for 16-bit I/O to index register

PCMCIA_DATA8         EQU  0xf00007c1 ;; this is what it would be..
PCMCIA_DATA16        EQU  0xf00007c0 ;; for 8-bit I/O to data register

                     LDR  r0, =PCMCIA_INDEX16
                     LDR  r1, =PCMCIA_DATA8
                     MOV  r2, 0xff

                     MOV  r3, 0x03
                     STB  r3, [r0]   ;; select register 3 (write to INDEX)
                     MOV  r4, 0x47
                     ST   r4, [r1]   ;; set register to 47 (write to DATA)

                     LDB  r5, [r0]
                     CMP  r5, r3     ;; can check this directly
                     LD   r5, [r1]
                     AND  r5, r5, r2
                     CMP  r5, r4     ;; had to mask before comparing
```

In practice, 8-bit I/O cycles are only required for configuring the Ethernet controller and configuring the PCMCIA controller and so the overhead introduced by the masking process is minimal.

4
Interrupts

The SA-110 has two interrupt inputs: **irq** (interrupt request) and **fiq** (fast interrupt request). The EBSA-110 provides software-programmable interrupt control logic to route interrupts from I/O devices to one or other of the two CPU interrupt inputs.

There are two different configurations for the interrupt control logic. The first, referred to as CTB_OS, is optimized for the implementation of operating-system software on the EBSA-110. The second, referred to as CTB_ARCH, is optimized for architectural compliance verification of the SA-110 processor. These two configurations can be distinguished under software control.

Most systems use the CTB_OS configuration. The configuration is reported when the power-on diagnostics are run.

An EBSA-110 can be configured for one or other version by changing a single, socketed, programmable logic device.

This chapter concentrates on the CTB_OS configuration and then discusses the CTB_ARCH in terms of its differences from the CTB_OS configuration.

Table 4–1 shows the registers available in the CTB_OS configuration. Table 4–2 shows the registers available in the CTB_ARCH configuration.

Table 4–1 Interrupt Control Registers - CTB_OS Configuration

Address	Name	Read Function	Write Function
0xf280.0000	TRICK2	reserved	FIQ_MASK
0xf2c0.0000	TRICK3	IRQ_MASK	IRQ_MSET
0xf300.0000	TRICK4	IRQ_MSKD	IRQ_MCLR
0xf340.0000	TRICK5	IRQ_RAW	reserved
0xf380.0000	TRICK6	reserved	reserved
0xf3c0.0000	TRICK7	reserved	reserved

Table 4–2 Interrupt Control Registers - CTB_ARCH Configuration

Address	Name	Read Function	Write Function
0xf280.0000	TRICK2	reserved	FIQ_MASK
0xf2c0.0000	TRICK3	reserved	IRQ_MASK
0xf300.0000	TRICK4	reserved	reserved
0xf340.0000	TRICK5	reserved	reserved

(continued on next page)

Table 4–2 (Cont.) Interrupt Control Registers - CTB_ARCH Configuration

Address	Name	Read Function	Write Function
0xf380.0000	TRICK6	reserved	FIQ_CNT
0xf3c0.0000	TRICK7	reserved	IRQ_CNT

4.1 Distinguishing CTB_OS, CTB_ARCH Under Software Control

Software can distinguish the CTB_ARCH and the CTB_OS configurations using this sequence:

1. Write 0xff to the TRICK4 register.
2. Write 0x55 to the TRICK3 register.
3. Write 0x00 to the TRICK3 register†.

After this sequence of accesses, a read from the TRICK3 register will return 0x55 for a CTB_OS configuration, and 0x00‡ for a CTB_ARCH configuration.

4.2 CTB_OS

The CTB_OS configuration is designed for normal software applications. It provides an interrupt controller designed for use by an operating system. For IRQ, the functionality meets the requirements of the ARM "Reference Microcontroller" specification. For FIQ, somewhat simpler functionality is provided.

The CTB_OS configuration is intended to allow independent device drivers to support re-entrant interrupts and lowest-latency ‘priority levels’ on the flat interrupt structure that the single **irq** interrupt provides.

The CTB_OS configuration also provides a write-only mask register that allows any combination of interrupts to be routed to the SA-110 **fiq** input.

4.2.1 The FIQ_MASK Register

The FIQ_MASK register is write-only and is used to control which interrupt sources can generate an interrupt on FIQ. This register is accessed by byte writes. The bit assignment of this register is shown in Table 4–3. In all cases, writing a ‘1’ enables the interrupt source.

Table 4–3 FIQ Mask Bit Positions

Bit	Function
7	USER_IRQ interrupt
6	PCMCIA socket A interrupt OR† PCMCIA socket B interrupt

†This is a Boolean OR operation; the interrupt is asserted if either PCMCIA socket is generating an interrupt.

(continued on next page)

† The final step of writing 0x00 is important. Without this step, the data bus could remain charged with the 0x55 data, giving a misleading result on the final step.

‡ In the implementation of the CTB_ARCH device, the data bus will turn on for the read, but no register will be selected because the IRQ_MASK is write-only, therefore the bus will be driven with 0.

Table 4–3 (Cont.) FIQ Mask Bit Positions

Bit	Function
5	PIT channel 2 interrupt
4	PIT channel 1 interrupt
3	Ethernet interrupt
2	COM2 interrupt
1	COM1 interrupt
0	LPT1 interrupt

The reset state of FIQ_MASK is UNKNOWN so a write of 0x00 is required to put the mask into a known state.

4.2.2 The IRQ_MASK Register

This register is read-only (but is at the same address as a write-only register) and allows software to determine what interrupt sources are currently enabled. This register is accessed by byte reads. The bit assignment of this register is shown in Table 4–4. In all cases, reading a '1' indicates that the interrupt source is enabled.

Interrupt sources are enabled and disabled using the IRQ_MSET and IRQ_MCLR registers.

Table 4–4 Interrupt Mask Bit Positions - CTB_OS Configuration

Bit	Function
7	USER_IRQ interrupt
6	PCMCIA socket A interrupt OR† PCMCIA socket B interrupt
5	PIT channel 2 interrupt
4	PIT channel 1 interrupt
3	Ethernet interrupt
2	COM2 interrupt
1	COM1 interrupt
0	LPT1 interrupt

†This is a Boolean OR operation; the interrupt is asserted if either PCMCIA socket is generating an interrupt.

4.2.3 The IRQ_MSET, IRQ_MCLR Registers

These two write-only registers allow bits in the IRQ_MASK register to be set and cleared. A byte write to IRQ_MSET will enable any interrupt source which has a '1' in its bit position. No other interrupt sources are affected. A byte write to IRQ_MCLR will disable any interrupt source which has a '1' in its bit position. No other interrupt sources are affected. These two registers allow interrupt sources to be enabled and disabled independently without the controlling software needing to know the current state of the interrupt mask. The bit assignments for writes to these registers are identical to those for the CTB_OS configuration of IRQ_MASK (shown in Table 4–4).

The current state of the interrupt mask can be read from IRQ_MASK.

The reset state of IRQ_MASK is UNKNOWN so a write of 0xff to IRQ_MCLR is required to put it into a known state (0x00).

4.2.4 The IRQ_RAW Register

The IRQ_RAW register is read-only and is accessed by byte reads. The IRQ_RAW register returns the unmasked state of the interrupt sources. A '1' indicates that the associated interrupt is asserted. The bit assignments are identical to those for the CTB_OS configuration of IRQ_MASK (shown in Table 4–4).

All interrupt sources are synchronized to provide data for the IRQ_RAW register. When the CPU starts a read from IRQ_RAW register the state of the synchronized interrupt sources is frozen until the read cycle has completed. This prevents the CPU from reading changing data (which would have indeterminate results).

4.2.5 The IRQ_MSKD Register

The IRQ_MSKD register is read-only and is accessed by byte reads. The IRQ_MSKD register shows which enabled interrupt sources are asserting an interrupt. A '1' indicates that the associated interrupt is both asserted and enabled. The bit assignments are identical to those for the CTB_OS configuration of IRQ_MASK (shown in Table 4–4). If the value of IRQ_MSKD is non-zero, the IRQ interrupt will be asserted at the CPU.

4.3 CTB_ARCH

The CTB_ARCH configuration is designed for architectural compliance verification testing. It provides a method of generating interrupts to the CPU under software control, using counters that are clocked at the CPU bus frequency.

The CTB_ARCH configuration provides a pair of write-only mask registers. One of these allows any combination of interrupts to be routed to the SA-110 **irq** input, and the other provides the same function for the **fiq** inputs.

The CTB_ARCH configuration does not provide the facility to read the current state of the interrupts or interrupt masks.

4.3.1 The FIQ_MASK Register

The FIQ_MASK register is write-only and is used to control which interrupt sources can generate an interrupt on FIQ. This register is accessed by byte writes. The bit assignment of this register is shown in Table 4–5. In all cases, writing a '1' enables the interrupt source.

The only difference between the CTB_ARCH and CTB_OS implementations of the FIQ_MASK register is the assignment of bit 7.

Table 4–5 FIQ Mask Bit Positions

Bit	Function
7	FIQ_CNT cycle counter interrupt

(continued on next page)

Table 4–5 (Cont.) FIQ Mask Bit Positions

Bit	Function
6	PCMCIA socket A interrupt OR† PCMCIA socket B interrupt
5	PIT channel 2 interrupt
4	PIT channel 1 interrupt
3	Ethernet interrupt
2	COM2 interrupt
1	COM1 interrupt
0	LPT1 interrupt

†This is a Boolean OR operation; the interrupt is asserted if either PCMCIA socket is generating an interrupt.

The reset state of FIQ_MASK is UNKNOWN so a write of 0x00 is required to put the mask into a known state.

4.3.2 The IRQ_MASK Register

This register is write-only and behaves in the same way as the FIQ_MASK register. The only difference is the assignment of bit 7. This register is accessed by byte writes. It is used to control which interrupt sources can generate an interrupt on IRQ. This register is accessed by byte writes. The bit assignment of this register is shown in Table 4–6. In all cases, writing a '1' enables the interrupt source.

Table 4–6 Interrupt Mask Bit Positions - CTB_ARCH Configuration

Bit	Function
7	IRQ_CNT cycle counter interrupt
6	PCMCIA socket A interrupt OR† PCMCIA socket B interrupt
5	PIT channel 2 interrupt
4	PIT channel 1 interrupt
3	Ethernet interrupt
2	COM2 interrupt
1	COM1 interrupt
0	LPT1 interrupt

†This is a Boolean OR operation; the interrupt is asserted if either PCMCIA socket is generating an interrupt.

The reset state of IRQ_MASK is UNKNOWN so a write of 0x00 is required to put the mask into a known state.

4.3.3 The IRQ_CNT Cycle Counter Register

The IRQ cycle counter register is write-only and is used to generate an interrupt after a certain number of bus clock (**mclk**) cycles. This register is accessed by byte writes. When a value of 0-0xfe is written to the register, it is decremented to 0 at a rate of one per **mclk**. When it reaches 0, it generates an interrupt on IRQ (provided its mask bit is set). On the next **mclk**, the counter decrements from 0 to 0xff, and remains at 0xff. The timer interrupt is cleared by a write to the IRQ_MASK register with D7=1.

The reset state of the counter is UNKNOWN so a write of 0x00 is required to initialize the counter before using it for the first time.

4.3.4 The FIQ_CNT Cycle Counter Register

The FIQ cycle counter register is write-only and is used to generate an interrupt after a certain number of bus clock (**mclk**) cycles. This register is accessed by byte writes. When a value of 0-0xfe is written to the register, it is decremented to 0 at a rate of one per **mclk**. When it reaches 0, it generates an interrupt on FIQ (provided its mask bit is set). On the next **mclk**, the counter decrements from 0 to 0xff, and remains at 0xff. The timer interrupt is cleared by a write to the FIQ Mask register with D7=1.

The reset state of the counter is UNKNOWN so a write of 0x00 is required to initialize the counter before using it for the first time.

5

Configuration of Memory and VLSI Devices

Software must perform a number of tasks to initialize the hardware. This section provides some guidelines on configuring the memory and VLSI devices on the board.

5.1 Configuring Cacheable/Non-Cacheable Space

In order to enable the CPU D-cache and write-buffer you must enable the MMU. The page-tables used by the MMU can control, on a page-by-page basis, whether a page is cacheable and/or bufferable.

The CPU caches and write buffers may be enabled for read and write accesses to on-board SSRAM and DRAM.

The CPU caches may be enabled for read accesses to the ROM and Flash. For writes, the Flash is essentially an I/O device; the CPU D-cache and write buffers should be disabled for writes to the Flash.

The CPU caches and write buffers must be disabled for accesses to I/O space (including the ISAMEM space).

5.2 Switching the Memory Map

After reset, the ROM is decoded at address 0 and the DRAM is not accessible. The mechanism for switching the address space is described in Section 3.1.1.

5.3 DRAM

The EBSA-110 supports zero, one or two DRAM SIMMs. If two are fitted, they must be of the same type.

After power-on or reset, on-board software must configure the DRAM using this sequence:

1. Disable refresh requests.
2. Assume the DRAM is BEDO; configure it, set the wrapping mode and ‘wake up’ all banks.
3. Enable refresh requests.
4. Determine DRAM type (EDO/BEDO) and configure accordingly.
5. (Optional) Size the memory non-destructively.
6. (Optional) Test the memory.

Each of these steps will now be described in detail.

5.3.1 Disable Refresh Requests

After reset, the PIT may be in an unknown state. To disable refresh requests whilst DRAM configuration is in progress † perform this sequence of PIT writes:

1. Write 0x3a to the PIT_CTLW register (load 16-bit count for channel 0 in mode 5).
2. Write 0x1 to the PIT_CTLW register (least-significant count is 1).
3. Write 0x0 to the PIT_CTLW register (most-significant count is 0).

This will make the counter generate a single refresh request and then no others.

5.3.2 Initialize the DRAM

The initialization sequence is:

1. Clear the SOFT_BURST bit in the Soft register.
2. Wait until the refresh generated when the refresh counter was disabled (see previous section) has been completed. If the code is running from ROM (EPROM or Flash) at this point in the initialization, no additional delay will be needed.

   At this stage, the refresh counter cannot generate any further refresh requests, and it is safe to set the SOFT_DCBR bit
3. Set the SOFT_DCBR bit in the Soft register.
4. At each of the following 32 addresses:

   ```
   for (i=0; i<32; i++)
     address := 0x0000.0080 & (i << 22);
   ```

   perform these accesses:

   1. Write the value 0x0000.0000 to the address (actually, the write data is irrelevant).

      This step in the configuration is only required for BEDO DRAMs, and it sets the wrapping mode to linear burst. Although it is only needed for BEDO DRAMs it is benign to EDO DRAMs and is a prerequisite to determining whether EDO or BEDO DRAMs (or no DRAMs at all) are fitted.
   2. Read from the address 9 times. The first read takes the DRAM out of programming mode. The next 8 reads will ‘wake up’ the DRAMs.

      This step is required for both EDO and BEDO DRAMs.

   When coding the write/reads sequence, ensure that all of the accesses are performed and that they are performed in order (beware of optimizing compilers).

   At this stage in the configuration, the DRAM memory size is not known, therefore the range of addresses ensures that every bank of DRAM that *could* be present gets initialized.
5. Clear the SOFT_DCBR bit.

† It is critical that refresh requests are disabled whilst the SOFT_DCBR bit is set. Failure to comply with this rule will mean that refresh cycles will spuriously terminate CPU cycles, leading to unpredictable behavior, which may include system lock-up.

5.3.3 Enable Refresh Requests

Set up channel 0 of the PIT to generate periodic refresh requests. The refresh period is calculated from this data:

- The slowest supported **mclk** frequency is 42.3 MHz (23.6ns period).
- The refresh counter is clocked at **mclk**/7 = 6.0 MHz (approx.), corresponding to a clock period of 165ns.
- The DRAMs require 2048 refresh cycles in 32ms. For a distributed refresh, this means 1 refresh every 15.625us.
- The minimum refresh interval must be reduced by an amount corresponding to the maximum latency to start a refresh. This maximum latency will occur if a refresh request coincides with a 16-longword load-multiple from DRAM, which requires a maximum of 54 **mclk** cycles; 1.28us. This changes the minimum refresh interval to 14.34us.

Refresh is configured and enabled by performing this sequence of PIT writes:

1. Write 0x36 to the PIT_CTLW register (load 16-bit count for channel 0 in mode 3).
2. Write 0x57 to the PIT_CTLW register (least-significant count is 87 (0x57) clock periods of 165ns, which equals 14.34us).
3. Write 0x0 to the PIT_CTLW register (most-significant count is 0).

5.3.4 Determine the DRAM Type

At this stage the DRAM is in EDO mode (SOFT_BURST is clear). Perform this sequence:

1. Non-sequential write to address 0x0000.0000 †, data 0xaaaa.aaaa.
2. Non-sequential write to address 0x0000.0004, data 0x5555.55555.
3. Non-sequential read from address 0x0000.0000. If the data is 0xaaaa.aaaa, the memory is EDO DRAM. If the data is not 0xaaaa.aaaa, the DRAM is faulty or not fitted or is BEDO DRAM. Set the SOFT_BURST bit and perform the read again.

When the SOFT_BURST bit is set correctly, ensure that bursts work correctly by using non-sequential writes to store data in 4 adjacent locations and then reading the data back twice; first by performing a load-multiple (sequential reads) and then by reading the same locations using non-sequential reads.

This technique works because a BEDO DRAM requires 2 **cas_l** pulses to access the first read data. If it only receives 1 (because the controller is configured for EDO) it will keep its data bus tristate.

5.3.5 Size the Memory

Many techniques exist. Memory is quantized in units of 4Mbytes, so it is only necessary to check on 4Mbyte boundaries. A non-destructive probe is preferable.

† This algorithm assumes that, if a single DRAM SIMM is fitted, it is fitted in the correct socket.

5.3.6 Test the memory

Use a technique of your choice. To test the memory properly, you should perform both sequential and non-sequential reads and writes.

At this point, the DRAM is fully configured.

5.4 SSRAM

The synchronous SRAM does not require any configuration.

5.5 EPROM

The EPROM does not require any configuration.

5.6 Flash

Reading from Flash does not require any configuration. Writing to Flash requires accesses to other registers within the device. Refer to the manufacturer's data sheet for details. Refer to Section 3.2.4 for the addressing sequence required to access sequential bytes during writes to Flash.

The Flash ROM on the EBSA-110 may have a vendor ID of 0x89,0xA2 (Intel 28F008SASA) or 0x89,0xA1 (Intel 28F008SA-L).

5.7 PCMCIA Controller

Before the PCMCIA controller can be configured, the Super I/O IDE device must be disabled, using the procedure described in Section 5.9.

The VADEM PCMCIA controller has some 'Unique Registers' (registers which differentiate it from previous-generation chips). These can be enabled by this code sequence:

- Perform a byte store of 0x0e to address PCMCIA_INDEX
- Perform a byte store of 0x37 to address PCMCIA_INDEX

The PCMCIA controller can be configured using the sequence shown in Table 5–1. For each register, the write involves the two-stage process of performing a byte store of the register number to address PCMCIA_INDEX and then performing a longword store of the associated data to address (PCMCIA_DATA - 1). The writes should be performed in the order shown in the table.

Table 5–1 PCMCIA Controller Configuration Sequence

Address	Value	Comment
0x38	0x12	A Async clock, card debounce delays
0x78	0x12	B Async clock, card debounce delays
0x39	0x00	A Timers off
0x79	0x00	B Timers off
0x3b	0x00	A GPIO configuration, external chip select disabled
0x7b	0x00	B GPIO configuration, external chip select disabled
0x3d	0x00	A Clear programmable chip select address

(continued on next page)

Table 5–1 (Cont.) PCMCIA Controller Configuration Sequence

Address	Value	Comment
0x7d	0x00	B Clear programmable chip select address
0x3e	0x00	A Programmable chip select not used
0x7e	0x00	B Programmable chip select not used
0x3f	0x00	A Disable ATA option
0x7f	0x00	A Disable ATA option
0x02	0x00	A Power and reset control; oe disabled, resume disabled, auto-power disabled, Vpp off
0x02	0x00	B Power and reset control; oe disabled, resume disabled, auto-power disabled, Vpp off
0x16	0x00	A Card detect and general control register; all off
0x56	0x00	B Card detect and general control register; all off
0x1e	0x00	A,B Global control register; active-high interrupts
0x03	0x04	Assign Card A interrupt to IRQ3
0x43	0x03	Assign Card B interrupt to IRQ4
0x05	0x48	A Interrupt enabled for card and status-change interrupts
0x45	0x38	B Interrupt enabled for card and status-change interrupts
0x06	0x20	A Enable MEMCS16, disable all memory and I/O windows
0x46	0x20	A Enable MEMCS16, disable all memory and I/O windows

5.7.1 Setting the PCMCIA Socket Programming Voltage

The programming voltages **VPP1, VPP2** are set using the Power and RESETDRV Control Register. This is at Index 0x02 for socket A and at Index 0x42 for socket B. The appropriate values for this register are shown in Table 5–2.

Table 5–2 PCMCIA Programming Voltages

Voltage	d[3:0]
0V	0x0
5V	0x8
12V	0x2
Off	0xa

5.7.2 Setting a PCMCIA Socket Memory Window

The PCMCIA controller provides facilites for mapping portions of the 64Mbyte (26-bit) PCMCIA address space into the 16Mbyte (24-bit) ISA address space. On the EBSA-110 the ISA address space is decoded in the ISAMEM and ISAIO areas of the address space.

The controller allows a number of windows to be defined. Each window maps a region of PCMCIA address space into a region of ISA address space. Each window is configured using a set of control registers. For each socket (socket A, socket B) there are 2 I/O windows and 5 memory windows. Memory windows are also used to access attribute space on the cards.

Setting up a memory or I/O window requires these steps:

- Choose the system address space range.

  The system address is the address in the 24-bit ISA address space. The lower 64kbytes of this space cannot be allocated to memory windows.

- Calculate the equivalent range in the EBSA-110 address space.

  The ISA address space is a subset of the EBSA-110 address space, and is sparsely mapped into EBSA-110 address space.

- Calculate the window register values.
- Select which window to use.
- Configure the window registers.
- Enable the window.

Here is a worked example of setting up a memory window. The process for setting up an I/O window is similar. Refer to the manufacturer's data sheet for more information.

```
Goal: set a window from system space into attribute space on Socket A using
memory window 1, so that attribute space from address 0x0 to 0xffff is
available.

* Choose the system address space range: The addresses from 0x0 to 0xffff are
not available, so choose addresses 0x1.0000 - 0x1.ffff.

* Calculate the address in EBSA-110 address space: The PCMCIA system address space
starts at PCMCIAMEM_BASE (0xe800.0000). The PCMCIA system address space is
sparsely mapped because it only occupies 16 bits of the 32-bit address
bus; its data path is accessible on the two low-order byte lanes. Therefore,
the equivalent EBSA-110 address range is:
    (2 * system address space range) + PCMCIAMEM_BASE
    = range 0xe802.0000 -> 0xe803.ffff

* Calculate the window register values:
start_address = 0x1.0000
stop_address = 0x1.ffff
offset = pcmcia_address - start_address = 0 - 0x1.0000 = 0xffff.f000

The pcmcia_address is the start address in PCMCIA address space that the
window is mapped to.

System Memory Address Mapping Start Reg. Low  =  (start_address >> 12) AND 0xff
System Memory Address Mapping Start Reg. High = ((start_address >> 20) AND 0x0f) OR 0x80
- the OR of 0x80 selects a 16-bit data path.

System Memory Address Mapping Stop Reg. Low   =  (stop_address >> 12) AND 0xff
System Memory Address Mapping Stop Reg. High  =  (stop_address >> 20) AND 0x0f

Card Memory Offset Address Reg. Low           =  (offset >> 12) AND 0xff
Card Memory Offset Address Reg. High          = ((offset >> 12) AND 0xff) OR 0x40
- the OR of 0x40 selects attribute space.

* Select card A window 0. Its registers are at:

System Memory Address Mapping Start Reg. Low  = 0x10
System Memory Address Mapping Start Reg. High = 0x11
System Memory Address Mapping Stop Reg. Low   = 0x12
System Memory Address Mapping Stop Reg. High  = 0x13
Card Memory Offset Address Reg. Low           = 0x14
Card Memory Offset Address Reg. High          = 0x15

* Configure the window registers:
```

```
write 0x10 to register at 0x10
write 0x80 to register at 0x11
write 0x1f to register at 0x12
write 0x00 to register at 0x13
write 0xf0 to register at 0x14
write 0x7f to register at 0x15

Each write is a 2-stage process that involves writing the register number to the
PCMCIA_ADDRESS register then writing the data to the PCMCIA_DATA register.

* Enable the window by reading the value of the register at address 0x06,
OR with 0x01 (to enable memory window 0) and write the value back.
```

5.8 Ethernet Controller

The Am79C961A is configured without an EEPROM. In this mode, it will enter Software Relocatable Mode after powerup or reset. On-board software must configure the device using this sequence:

1. Send the initiation key.
2. Put the device into 'CONFIG' state.
3. Configure the Plug-and-play registers.
4. Disable the Plug-and-play registers.

5.8.1 Send the Initiation Key

The initiation key is a specific byte sequence which must be written to the PNP_ADDRESS register†. This process takes the Am79C961A out of its 'Wait For Key' state. The pattern must be sequential; any other I/O cycles to the Ethernet controller will reset the state machine that is checking the pattern. After a reset, the Plug-and-play registers are configured for 8-bit I/O cycles, therefore these writes must be performed as longword stores to address (PNP_ADDRESS-1) (Refer to Section 3.9.1). The key‡ is:

```
6b, 35, 9a, cd, e6, f3, 79, bc,
5e, af, 57, 2b, 15, 8a, c5, e2,
f1, f8, 7c, 3e, 9f, 4f, 27, 13,
09, 84, 42, a1, d0, 68, 34, 1a
```

5.8.2 Put the Device into 'CONFIG' State

When the key has been written, the Ethernet controller Plug-and-play state machine can be transitioned to the 'CONFIG' state. This is achieved using the sequence shown in Table 5–3. For each register, the write involves the two-stage process of performing a longword store of the register number to address (PNP_ADDRESS-1) and then performing a longword store of the associated data to address (PNP_WRDATA-1). The writes should be performed in the order shown in the table.

† The addresses of all of these registers are described in Section 3.5.
‡ This is not the same as the key that is used when the Am79C961A is configured with an EEPROM.

Table 5–3 Ethernet Plug-and-play Register Configuration Sequence

Address	Name	Value	Comment
0x02	Configuration control	0x05	Reset CSN to 0
0x03	Wake[CSN]	0x00	Go to ISOLATION state
0x00	Set RD_DATA port	0x80	Set RD_DATA port
0x06	Card Select Number	0x01	Set CSN to 1 and go to CONFIG state

5.8.3 Configure the Plug-and-play Registers

The next stage in the initialization process is to write configuration values to the Plug-and-play registers. Each write requires the same 2-stage process described in the previous section. The recommended values are shown in Table 5–4; they should be written in the order shown.

Table 5–4 Ethernet Plug-and-play Register Initial Values

Address	Name	Value	Comment
0x43	Boot PROM base	0xfe	Disable boot PROM decode
0x48	SRAM base23:16	0x0c	Set SRAM base
0x49	SRAM base15:08	0x00	-
0x4a	SRAM Memcon	0x02	Set SRAM access width to 16-bit
0x4b	SRAM limit23:16	0xff	Set SRAM size to 64K
0x4c	SRAM limit15:08	0x00	-
0x60	IO base15:8	0x02	Set base address in I/O space to 0x220
0x61	IO base07:00	0x20	-
0x70	IRQ sel	0x03	Select interrupts on IRQ3
0x71	IRQ type	0x02	Select active-high edge-sensitive
0x74	DMA sel	0x00	No DMA channel
0xf0	Vendor-defined	0x04	8-bit I/O, enable address PROM
0x31	I/O Range Check	0x00	Disable I/O range check
0x30	Activate	1	Activate the logical device
0xf0	Vendor-defined	0x05	Switch to 16-bit I/O

At the end of this sequence, the Ethernet controller has been configured to use 16-bit I/O and so all subsequent accesses to the device can use byte and half-word loads and stores.

5.8.4 Disable the Plug-and-play Registers

The final stage in the initialization process is to disable the Plug-and-play registers. This step must be performed before accesses to the UID ROM or buffer memory are possible. Use this code sequence:

- Write 0x02 to the PNP_ADDRESS register to select the configuration control register.
- Write 0x02 to the PNP_WRDATA register to make the Plug-and-play state machine transition back to the WAIT_FOR_KEY state.

Since the Ethernet controller has now been configured to use 16-bit I/O, these accesses should be performed using half-word stores.

At this point, the Ethernet controller is configured and its registers can be accessed using the NET_IDP, NET_RAP and NET_RDP registers. The IEEE unique identification address can be read from the sequence of addresses starting at the NET_UID address.

5.9 Super I/O Controller

No configuration is required before accessing the COM1, COM2 or LPT1 ports. The SuperI/O UARTs and LPT registers should be left at their default addresses, so that the interrupt assignment is not changed (LPT2 interrupt on IRQ7, COM1 interrupt on IRQ4 and COM2 interrupt on IRQ3).

The addresses of the (unused) IDE logic within the Super I/O controller clash with the addresses used by the PCMCIA controller. Therefore, it is necessary to disable the IDE logic. Use this code sequence:

1. Write 0x00 to the SIO_INDEX register to select the Function Enable register (all of these accesses should be byte loads and stores).
2. Read the SIO_DATA register.
3. AND the value read with 0xBF to clear bit 6; this disables the IDE function.
4. Write the new value back to the SIO_DATA register.
5. Write the same value back to the SIO_DATA register a second time. The SIO requires this double write before it will update its registers. If you code this sequence using a high-level language, make sure that your compiler does not optimize out this second write.

5.10 Programmable Interval Timer

Since the interrupt controller expects level-sensitive interrupts, the PIT timer channels 1 and 2 must be operated in Mode0.

Channel 0 is used as the refresh timer and its initialization is described in Section 5.3.

6
Performance

This section discusses the performance of the memory and I/O sub-systems. Performance is discussed in terms of the number of stall cycles that are inserted into a bus transaction during accesses to the various devices on the EBSA-110. The waveforms described in Chapter 11 show most of the scenarios described here.

6.1 Synchronous SRAM Accesses

During CPU write cycles to synchronous SRAM, no stall cycles are inserted. This is true for both non-sequential and sequential cycles.

During CPU read cycles from synchronous SRAM, one stall cycle is introduced at the start of each (sequential or non-sequential) bus cycle. This stall cycle is required to fill the read pipeline of the SSRAM. During read sequential cycles, an additional two stall cycles are inserted whenever a new address must be loaded into the SSRAM. A reload occurs either after 4 beats of data have been read (for aligned accesses or wrapped cache line fills) or when the address of the read crosses an INT16 boundary (of all other reads). The first stall cycle is required to load the new address into the SSRAM and the second wait state is required to fill the read pipeline of the SSRAM.

Cache line fills always wrap around INT16 boundaries, making the SSRAM reads very efficient. Cache line fills will experience a total of 3 inserted stall cycles.

A worst-case sequential read of 8 INT4s would cross an INT16 boundary three times during the read, and would therefore experience a total of 5 inserted stall cycles.

6.2 EDO DRAM Accesses

Accesses to EDO DRAM require stall cycles to be inserted to meet the access time of the DRAMs.

Sequential accesses to EDO DRAM are always performed in page mode, so that the overall access time is lower than the equivalent non-sequential accesses.

The performance of EDO DRAM accesses is shown in Table 6–1.

Table 6–1 Stalls Added During EDO DRAM Accesses

Cycle Type	Total Stalls Inserted
Non-sequential read	5
2-beat sequential read	8
3-beat sequential read	11
4-beat sequential read	14
8-beat sequential read	26
Non-sequential write	4
2-beat sequential write	6
3-beat sequential write	8
4-beat sequential write	10
8-beat sequential write	18

6.3 BEDO DRAM Accesses

Accesses to BEDO DRAM require stall cycles to be inserted to meet the access time of the DRAMs. Read accesses incur an overhead because (relative to a normal DRAM access) an extra CAS pulse is required to start the fill of the data pipeline. Write accesses incur an overhead because (relative to a normal DRAM access) an extra recovery cycle is required at the end of the cycle, prior to negating RAS. This recovery cycle only incurs a performance penalty when back-to-back DRAM cycles are performed (refer to Section 6.7).

Accesses to BEDO are most efficient when aligned blocks of data are being read and written. This makes it well suited to the bus traffic generated in systems with caches.

When unaligned blocks of data are being read and written a performance penalty is incurred by the extra cycles needed to abort a burst and reload the column address.

The performance of BEDO DRAM accesses is shown in Table 6–2.

Table 6–2 Stalls Added During BEDO DRAM Accesses

Cycle Type	Total Stalls Inserted
Non-sequential read	7
Aligned 2-beat sequential read	8
Aligned 3-beat sequential read	9
Aligned 4-beat sequential read	10
Aligned 8-beat sequential read	19
Unaligned 2-beat sequential read	12
Unaligned 3-beat sequential read	13
Unaligned 4-beat sequential read	14
Unaligned 8-beat sequential read	22

(continued on next page)

Table 6–2 (Cont.) Stalls Added During BEDO DRAM Accesses

Cycle Type	Total Stalls Inserted
Non-sequential write	4
Aligned 2-beat sequential write	6
Aligned 3-beat sequential write	8
Aligned 4-beat sequential write	10
Aligned 8-beat sequential write	18
Unaligned 2-beat sequential write	6
Unaligned 3-beat sequential write	8
Unaligned 4-beat sequential write	10
Unaligned 8-beat sequential write	18
Aligned 4-beat sequential full write	7
Aligned 8-beat sequential full write	12

Aligned accesses never incur the performance penalty of aborting the DRAM burst. The unaligned accesses are designed to cross as many block boundaries as possible (once for 2-beat, 3-beat and 4-beat, twice for 8-beat).

BEDO writes only run at the same speed as EDO writes. The reason for this is that the cycle must be stalled to allow the byte masks to become valid for each beat in turn. In some circumstances, the SA-110 is able to determine that all byte masks will be asserted for all beats of a sequential cycle. Such a cycle is called a ‘full write’ and corresponds to a merged write buffer write or the eviction (cast-out) of a dirty cache block. In these circumstances, it is not necessary to decode address information and this allows the cycle time of full writes to be reduced.

6.4 Performance Impact of DRAM Refresh

The bandwidth required by refresh is calculated as follows: DRAMs require 2048 refresh cycles in 32ms. The refresh sequence (shown in Figure 11–13) takes 9 clocks at 18ns. Therefore, in a 32ms period, the percentage of time occupied by DRAM refresh is (2048*9*18ns/32ms)*100 = 1.03%.

In practice, the impact of DRAM refresh will be lower than that figure. CPU cache accesses, I/O accesses and SSRAM accesses can all take place in parallel with refresh cycles. The only time that a refresh cycle will use system bandwidth is if the CPU attempts to access DRAM whilst a refresh cycle is in progress.

6.5 EPROM and Flash Accesses

Accesses to the EPROM and the Flash run at the same speed, even though the Flash device actually has a shorter access time. Reads from these devices have stall cycles inserted so that a sequence of bytes can be packed into a 32-bit data unit, and so that the access time of the devices is satisfied. Writes to the Flash have stall cycles inserted so that the access time is satisfied.

The performance of ROM (EPROM and Flash) accesses is shown in Table 6–3.

Table 6–3 Stalls Added During EPROM and Flash Accesses

Cycle Type	Total Stalls Inserted
Non-sequential read	44
2-beat sequential read	87
3-beat sequential read	130
8-beat sequential read	345
Non-sequential write	20

6.6 I/O Accesses

Accesses to I/O devices have stall cycles inserted in order to meet address setup, cycle time and address hold requirements for the slowest device. The PCMCIA controller allows plug-in cards to reduce the cycle time on an access-by-access basis, using **zws_l**. The PCMCIA controller and the Ethernet controller allow the cycle time to be extended on an access-by-access basis, using **rdy**. **rdy** allows the cycle time to be extended infinitely. **zws_l** allows the cycle time to be reduced, but there is a predetermined minimum cycle time.

The performance of I/O accesses is shown in Table 6–4.

Table 6–4 Stalls Added During I/O Accesses

Cycle Type	Total Stalls Inserted
Normal read	20
Normal write	20
Fastest read (using ZWS_L)	13
Fastest write (using ZWS_L)	13

6.6.1 Ethernet Buffer Memory Bandwidth

The Ethernet memory bandwidth cannot be simulated accurately and so was measured experimentally.

With the Ethernet controller idle, the bandwidth into Ethernet buffer memory was measured with the I-cache on, and the write buffer and D-cache off. A tight CPU write loop sustained a bandwidth of 3.3E6 bytes/second. A tight CPU read loop sustained a bandwidth of 3.7E6 bytes/second.

With the Ethernet controller transmitting and receiving continuously (in internal loopback) the bandwidth into the Ethernet buffer memory was reduced by 25%.

6.7 Overlap of Cycles

At the end of a DRAM cycle, there is a period of time called the RAS precharge period. A new DRAM access cannot start during this time. The state machines on the EBSA-110 are designed in such a way that an SSRAM access is not stalled due to the RAS precharge period of a previous DRAM cycle. This saves clock cycles in some situations. However, if a new DRAM access starts before the precharge for a previous cycle has completed, stall cycles must be inserted. Table 6–5 shows how many additional stall cycles are inserted when the second cycle type follows the first cycle type as a back-to-back cycle (that is, 1 idle cycle between the two bus transactions).

Table 6–5 Stalls Caused by Back-to-Back Cycles

First Cycle	Second Cycle	Additional Stalls
SSRAM access	Any access	0
I/O access	SSRAM access	0
I/O access	I/O access	0
I/O access	Any DRAM access	0
EDO read/write	SSRAM access	0
EDO read/write	I/O access	0
EDO read/write	Any DRAM access	1
BEDO read	SSRAM access	0
BEDO read	I/O access	0
BEDO read	Any DRAM access	1
BEDO write	SSRAM access	0
BEDO write	I/O access	0
BEDO write	Any DRAM access	2

7
Software Development Environment

This chapter describes the types of images that may be built for the EBSA-110, and how to use ARM's software development toolkit to build these images. The toolkit itself is described in the ARM Software Development Toolkit Reference Manual.

Two types of image are described:

- Loadable debuggable images
- Standalone Flash images

Flash images may be programmed into Flash using the FMU utility described in Section 9.1.

Note

This chapter assumes that the board is using the Demon debug server. Future versions of the EBSA-110 are expected to replace this with the Angel debug server. When this happens an addendum or technical note will be issued describing the differences.

7.1 Loadable Debuggable Images

These images are run under the control of the Demon debug agent held in ROM communicating with either armsd or the ARM Windowing debugger.

7.1.1 Building

These images may be written in C or assembler. No special options are needed when assembling or compiling. Debuggable C programs (and optionally debuggable assembler programs) should be linked with the Demon (semi-hosted) C library. To allow debugging they should be linked using either the -AIF or the -AIF -BIN options, although images linked with the -BIN option can be debugged at the machine code level.

Images that are to be loaded across the serial line using the debugger's load command may be linked to use any base address in SSRAM or DRAM except addresses below 0x8000.

Images that are to be loaded using bootp must also avoid loading into the memory containing the bootp program and its buffers. Two versions of the bootp utility are provided; one loads at 0x8000 and the other loads at 0x40000000, so this should not normally be a problem.

Images that use the C library to create their user stack should be linked to addresses in the first alias (i.e. the one starting at 0x0 or 0x40000000) of the area of RAM they are using.

7.1.2 Run Time Environment

7.1.2.1 Memory Map

All RAM except address 0 to 0x8000 is available to the program. DRAM will have been initialized before entry to the program. The MMU, write buffer, and caches will not have been initialized unless you have done this by running a previous program or by writing to the system coprocessor using debugger commands.

The C heap will be placed directly above the text segment of the program. If the program is running from DRAM the C library initialization functions will place the user stack at the top of DRAM. If it is running from SSRAM the user stack will be placed at the top of SSRAM.

7.1.2.2 C Library Support

The C library is described in the toolkit manual. All standard C functions are supported. All reference to files (including references to standard input and output) refers to these files on the host. This means that, for example, a call to printf( ) prints a string to the host that is running the debugger.

7.1.2.3 Exception Vectors

The Demon debug monitor uses the Undef, SWI and FIQ exception vector entries. The program can safely modify any other exception vector to jump to its own exception handlers. The program can also install its own handlers using SWI_ InstallHandler. This is described in the Demon documentation.

7.1.2.4 Access to I/O Devices

Demon uses the COM1 serial port. The program must not access this device. All other devices may be used by the program.

7.2 Standalone Flash Images

These images boot directly from Flash.

7.2.1 Building

These images may be written in C or assembler. No special options are needed when assembling or compiling. You must provide startup code and the code of any library functions used (refer to Section 7.2.2.2). There are two ways of linking such images:

- -AIF -BIN -BASE n
  - If the base address is outside of the address range of the Flash, the PBL will copy the image to its base address in system RAM (removing the header in the process) and execute it from its entry point; the image will execute from RAM.

    In this case, the image may occupy non-contiguous blocks in Flash.
  - If the base address is equal to the Flash block address + 0xc0, the PBL will execute the image by branching to its entry point; the image will execute from Flash.

    In this case, the image must occupy contiguous blocks in Flash.
  - If the address does not meet either of these requirements, the FMU will report an error and will not program the image into Flash.

  Images linked with this option may use any base address in RAM.

- -AIF -BASE n—The image will execute from Flash. Requirements are:
  - The image must occupy contiguous blocks in Flash.
  - The image must not contain any writable initialized data.
  - The address of the first Flash block to be used for the image must be known at link time.
  - The base 'n' must be the address of the Flash block + 0x40.

  In this case, the image is started by branching to the BL instruction that is the first longword of the AIF header. The FMU does not validate the entry point.

  This option should normally be avoided (except for programs that relocate themselves to RAM during initialization) since accesses to Flash are much slower than access to RAM.

7.2.2 Run Time Environment

7.2.2.1 Memory Map

All of RAM is available to the program. If the program is run from RAM, then DRAM will have been initialized before entry to the program. If it is run directly from Flash, then nothing will have been initialized. The boot time memory map will still be in use although the PC will be in the first alias above 0x80000000 of the Flash block (not in a low alias). The MMU and caches will not have been initialized. If the program is running from DRAM or Flash, the C library initialization functions will place the user stack at the top of DRAM. If it running from SRAM, the user stack will be placed at the top of SRAM.

7.2.2.2 C Library Support

ARM's software development toolkit includes sources and porting information for two run-time libraries; a minimum standalone library and an ANSI C library. EBSA-110 ports of these libraries may be supplied as part of the firmware database in the hardware developer's kit †.

7.2.2.3 Exception Vectors

The program may modify and use the exception vectors without restriction.

7.2.2.4 Access to I/O Devices

If a C library is used, it will provide routines to access some devices (for example, the COM1 serial port) and it will expect exclusive access to the associated underlying hardware. Other than this, the program may access any device.

† Early versions of the HDK are unlikely to provide this.

8
On-Board Software

When the EBSA-110 is reset or powered up, code execution commences with a fetch from the reset vector at location 0. Depending upon a jumper setting on the board, the reset vector can be supplied from the EPROM or from a Flash ROM.

Usually, the system will boot from an image called the Primary Boot Loader (PBL), which is stored in the Flash ROM.

A newly-manufactured system, or a system in which the Flash has become corrupted, cannot boot from Flash. In this case, a special EPROM, called the Startup EPROM, is used. The Startup EPROM performs power-on diagnostics and programs the PBL into the Flash.

This chapter describes the PBL, the Startup EPROM and the EBSA-110 diagnostics.

8.1 The Primary Boot Loader

The Primary Boot Loader (PBL) is a special image that is programmed into the first block (block 0) of the Flash. Normally, the PBL is the first code executed when the EBSA-110 comes out of reset.

The Flash can contain a number of different images; the main function of the PBL is to determine which image to execute and to execute the image. If necessary, the PBL will load the image from Flash into system memory.

Images are programmed into Flash using the Flash Management Utility (FMU) described in Section 9.1.

The format of the images in Flash is described in Section 8.2.

When the PBL is executed, it performs these tasks:

- Read the value of the boot jumpers to determine which image to boot. The boot jumpers are on J4, and 8 images can be selected using the settings shown in Table 8–1.

Table 8–1 Boot Image Selection

J2:13-14	J2:11-12	J2:13-14	Action
-	-	-	Enter ARM remote debug stub within PBL image
-	-	fit	Boot image 1 - normally the diagnostics
-	fit	-	Boot image 2
-	fit	fit	Boot image 3

(continued on next page)

Table 8–1 (Cont.) Boot Image Selection

J2:13-14	J2:11-12	J2:13-14	Action
fit	-	-	Boot image 4
fit	-	fit	Boot image 5
fit	fit	-	Boot image 6
fit	fit	fit	Boot image 7

- If Image 0 is selected then enter the ARM remote debug stub within the PBL image.
- If any other image is selected:
  - Search for the image in Flash, and verify that the checksum is correct.

    If image is not found or is corrupt (bad checksum), behave as though the selected image is image 0.
  - If the image is in executable AIF format, jump to the image (the system memory map has not been changed and the DRAM has not been initialized).
  - If the image is in non-executable AIF format, then:
    * Switch the memory map
    * Initialize DRAM
    * Load the image into memory at the addresses defined in the AIF header
    * Jump to the image's entry point

8.2 The Format of Images in Flash

The Flash ROM is a 1Mbyte part, organized as sixteen 64Kbyte blocks. Block 0 (at address 0x0000.0000, after reset) is reserved for the PBL. The remaining 15 blocks can be used to hold other images.

Each image, apart from the PBL, has an image header that allows it to be stored across non-contiguous blocks. Only the first block used by the image has an image header. Any individual block is only used by none or one image. Any block that is not in use will be in its erased state.

The format of an image stored in the Flash is basically AIF (ARM Image Format), with a few additional bytes prepended. The format is shown in Table 8–2.

When the FMU is used to program an image into Flash, the FMU will create and prepend the header information onto the image.

Note

You may write an alternative Flash programming utility, but it should follow the defined Flash structure so that the PBL can load the image.

Table 8–2 Flash Image Header

Offset (bytes)	Size (bytes)	Name	Description
0	4	Type	BL to AIF header (for executable AIF) or BL to image entry point (for non-executable AIF on image to be executed from Flash) or NOP (for non-executable AIF executed from RAM)
4	1	Number	Unique image number (0 to 0xff)
5	3	Sig	0x55 0xaa 0x00
8	4	Map	Allocation map. Bit 0 represents block 0, bit 31 represents block 31 (only bits 15:0 are required for the current Flash part)
12	4	Checksum	Checksum of image including headers, using the algorithm described below
16	4	Length	Image length (including all headers) - used to determine what gets checksummed
20	16	Name	ASCII string identifying name of image. Unused characters should be set to 0x20 (ASCII space)
36	4	Bootflags	Bit 0 is NoBoot. When set for an image, the PBL will load the image but then pass control to the ARM remote debug stub within the PBL.
40	24	Reserved	Reserved for future use
64	128	AIF header	AIF header for image

The headers use a total of 192 bytes. The first free byte is at offset 192 (0xc0).

The checksum is formed by taking the 2's complement of the 32-bit sum (ignoring carry) of all longwords of the header and image, excluding the checksum itself, as specified by the length field. If the length is not an integral number of longwords, the 'missing' bytes are set to 0xff (the unprogrammed state of bytes in Flash).

When the checksum is correct, a 32-bit sum (ignoring carry) of all longwords of the header and image, including any bytes required to round the length up to an integral number of longwords, will be 0.

Block 0 of the Flash will always contain image 0, the PBL image. It is undefined whether this image contains an image header.

Images can have an image number between 0 and 0xff, but the PBL can only load and start image numbers 0-7.

Software that deletes an image in Flash should erase all the blocks used by that image. Software that programs an image in Flash should determine which blocks are free by checking each block for an image header and then ORing the allocation maps of all the valid image headers.

8.3 The Startup EPROM

The Startup EPROM performs manufacturing diagnostics on the board. This process includes programming the PBL image into the Flash. After initial programming of the Flash (which is done during manufacture), you should not normally boot the board from the startup EPROM unless Flash image 0 (the PBL) is missing or has been corrupted. Section 8.4.2 describes the diagnostic tests.

To execute the Startup EPROM:

- Check that the correct EPROM is fitted.
- Fit jumper J4 pin 6-8 (this selects booting from the EPROM).
- Reset or power-cycle the board.

During the tests, the red 'debug' LED and the COM1 port provide progress information. Refer to Section 8.4.2.

Once the tests have been completed successfully, remove the jumper from J4 pin 6-8 and reset the board.

8.4 Diagnostics

There are 2 versions of the diagnostic tests:

- Power-on diagnostics
- Manufacturing diagnostics

The manufacturing diagnostics are a superset of the power-on diagnostics. The manufacturing diagnostics are run when the 'startup EPROM' is used (see Section 8.3). The power-on diagnostics are normally programmed into the Flash ROM as Image 1 and executed when Image 1 is selected.

8.4.1 Getting Ready to Run the Diagnostics

To run the diagnostics:

1. Use null-MoDem cables to connect terminals (or virtual terminals running on a PC or workstation), to the board's COM1 and COM2 ports. Configure both terminals for 9600 baud, 8-bit data, 1 stop bit, no parity, no flow control.
2. (Optionally) Connect an Ethernet loopback connector to the Ethernet port.
3. (Optionally) Connect the parallel port loopback connector to the parallel port. A suitable loopback connector is described in Section A.9.3.
4. Set the jumpers to select either the manufacturing diagnostics or the power-on diagnostics in accordance with Table 8–3.

Table 8–3 Selecting Diagnostics

To select..	J2:2-4	J2:13-14	J2:11-12	J2:9-10
Power-on diagnostics	remove	remove	remove	fit
Manufacturing diagnostics	fit	don't-care	don't-care	don't-care

5. Reset or power-cycle the system.

8.4.2 Description of Tests

This section describes the diagnostic tests, in order of execution, and highlights differences between the power-on diagnostics and the manufacturing diagnostics. It also includes a sample output from running the tests.

The tests performed are:

1. LED test: flash debug LED 8 times. This demonstrates that the tests have started, and that some I/O path is working.

2. Memory map decode test: jump to high-order image of ROM, flip memory map and check that low-order image disappears.
3. Write banner messages to COM1 and COM2. If these banner messages are not seen within 15 seconds of the tests starting, then one of the following problems has occurred:
   - i The ROM cannot be accessed at its high address location.
   - ii The UART cannot be accessed or is not functioning.
   - iii The terminals (or virtual terminals) attached to the ports are wrongly configured.

   From here on, a progress message is written to COM1 each time a test is started or completed, and any errors detected by the program are reported to COM1.
4. Soft I/O test: verify all read/write bits function correctly.
5. Size and test SSRAM. The SSRAM tests performed are:
   - i Write the address of each 32-bit location to itself (using word writes), then read them all back (using word reads) and check that they contain the correct values.
   - ii Write the address of each 32-bit location shifted right by 16 to itself, then read them all back and check that they contain the correct values. This tests that the upper bits of each word are being written and read correctly.
   - iii Write a value to each byte of SSRAM (using byte writes), then read them all back using byte reads checking the values read.
   - iv Read back the values written by the previous test using word reads.
   - v Store multiples of 1 to 5 words are done at each possible alignment in a 4-word block. After each store multiple, the program tests that the correct values have been stored, that no other memory (close to the 4-word block) has been corrupted, and that the registers have not been corrupted.
   - vi Load multiples of 1 to 5 words are done at each possible alignment in a 4 word block. After each load multiple, the program tests that the correct values have been read and that memory close to the 4-word block has not been corrupted.
6. Copy remaining tests to SSRAM, and jump to SSRAM.
7. Test PIT counter 0 (refresh timer).
8. Configure DRAM and identify the type of DRAM fitted.
9. Size and test DRAM. The SSRAM tests are repeated on the DRAM. Test 3 (writing and reading every byte of DRAM) may take up to 1 minute.
10. Identify CTB type (OS or ARCH).

Note

Any errors detected before this point are regarded as fatal; the tests are aborted. Any errors detected after this point are regarded as recoverable, and the tests attempt to continue.

11. Test PIT counters 1 and 2, including the interrupt paths associated with these PIT counters. This also tests the FIQ and IRQ control registers (or at least the bits associated with these interrupts) in the trick box logic.
12. Test the Ethernet controller.
13. Test the Ethernet UID ROM.
14. Test the Ethernet buffer RAM.
15. Test Ethernet input (using internal loopback), and the Ethernet interrupt paths.
16. Test Ethernet input using external loopback. This test will generate a warning if an external loopback is not detected.
17. Test the parallel port control registers.
18. Test the parallel port using external loopback. This test will generate a warning if an external loopback is not detected.
19. Test PCMCIA controller, and the associated interrupt paths. If a card is fitted, print the card's attribute space.
20. Test trick box aborts.
21. (Architectural logic only) Test the trick box cycle counters.
22. Test Flash control registers. (This does *not* reprogram any location in the Flash. It identifies the vendor and checks that the programming voltage is correct.)
23. Calculate the processor and bus speeds of the board.
24. Test input from COM1 and COM2 and the COM1 and COM2 interrupt paths. This test requires user input on the terminals attached to COM1 and COM2; follow the instructions printed to COM1 and COM2.
25. Read from Flash and report if any valid images are found.
26. You will be asked whether an integrity check should be performed on the Flash. If you agree (by answering 'y'), then this test performs an integrity test of blocks 1–15 of the Flash. Any blocks that contain valid images will be left unchanged. Any other blocks will be erased.
27. Read the jumper values from the soft register.
28. Print a summary of the detected configuration of the board. The information printed includes:
    i The processor's clock speeds.
    ii The type of logic fitted to the board.
    iii The amount of SSRAM fitted.
    iv The amount and type of DRAM fitted.
    v The jumper values read from the soft register.
    vi The MAC address read from the Ethernet UID ROM.
29. You will be asked if this summary is correct.
30. Print a "Tests passed" or "Tests failed" message. If the tests failed, print a summary of the failures.

31. (Manufacturing diagnostics only) If some of the tests failed, or there is a valid image in Flash block 0, ask whether Flash block 0 should be reprogrammed. If you respond by typing ‘n’, then the tests are terminated at this point.
32. (Manufacturing diagnostics only) Program the PBL into block 0 of the Flash.
33. (Manufacturing diagnostics only) Program the power-on diagnostics into Flash as image 1.

The processor is explictly *not* tested by the self-tests, although the processor may be assumed to be working more or less correctly if the tests run at all.

If the EBSA-110 fails its power-on diagnostic tests, refer to Appendix B.

An example of the output produced (on COM1) by the diagnostics is shown below. The output on your system may vary slightly from this, due to later additions to the diagnostics or a different board configuration.

```
Starting EBSA110 selftests V0.0; this is COM1. Results will be reported here
*** Testing Soft register R/W bits ***
Testing all bits
Test Passed
*** Soft register R/W bit tests complete ***
128Kbytes of SSRAM detected
*** Starting SSRAM tests ***
Testing word writing each word's address to itself
Test Passed
Testing word writing each word's address to its top halfword
Test Passed
Testing byte writing each byte; contents of each byte should be
address mod 255
Test Passed
Testing reading the data written by the previous test as words
Test Passed
Store multiple tests starting
Test Passed
Load multiple tests starting
Test Passed
*** SSRAM tests complete ***
*** Copying remaining tests to SSRAM ***
*** Now executing tests from SSRAM ***
*** Testing Refresh Timer ***
Testing counter 0 without interrupts
Test Passed
*** Refresh Timer Tests Complete ***
*** Starting DRAM tests ***
Initializing DRAM
Either no DRAM is fitted or EDO DRAMs are fitted
Sizing DRAM
DRAM size is  0x1000000
Testing word writing each word's address to itself
Test Passed
Testing word writing each word's address to its top halfword
Test Passed
Testing byte writing each byte; contents of each byte should be
address mod 255
Test Passed
Testing reading the data written by the previous test as words
Test Passed
Store multiple tests starting
Test Passed
Load multiple tests starting
Test Passed
*** DRAM tests complete ***
Getting CBT type
```

```
CTB configured as ARCH
*** Testing PIT counters 1 and 2 ***
Testing counter 1 without interrupts
Test Passed
Testing counter 2 without interrupts
Test Passed
Testing IRQ path for counter 1
Test Passed
Testing IRQ path for counter 2
Test Passed
Testing FIQ path for counter 1
Test Passed
Testing FIQ path for counter 2
Test Passed
*** Tests for PIT counters 1 and 2 complete ***
*** Testing ethernet controller and associated devices ***
Plug and play register access:
Test Passed
MAC address (08-00-2b-95-1d-7b)
Test Passed
Checking Checksum
Test Passed
Testing CSR0 access
Test Passed
Testing CSR1 writes
Test Passed
Testing shared RAM
Testing half word writes of each word's in shared RAM address to itself
Test Passed
Testing byte writes and reading each byte of shared RAM
Test Passed
Testing reading the data written by the previous test as halfwords
Test Passed
Shared RAM tests completed,
Testing internal loopback.
Test Passed
Testing IRQ path
Interrupt flag in device set OK
Test Passed
Testing IRQ path with no interrupt asserted
Test Passed
Testing FIQ path
Interrupt flag in device set OK
Test Passed
Testing FIQ path with no interrupt asserted
Test Passed
*** Ethernet controller tests complete ***
*** Testing PCMCIA controller and associated devices ***
PCMCIA device test starting...
Disable IDE Registers
Test Passed
Verify controller ID
Test Passed
Bit test of Socket A register
Test Passed
Bit test of Socket B register
Test Passed
Testing socket A IRQ assertion
Test Passed
Testing socket A FIQ assertion
Test Passed
Testing socket A IRQ negation
Test Passed
Testing socket A FIQ negation
Test Passed
```

```
Testing socket B IRQ assertion
Test Passed
Testing socket B FIQ assertion
Test Passed
Testing socket B IRQ negation
Test Passed
Testing socket B FIQ negation
Test Passed
*** PCMCIA controller tests complete ***
*** Testing parallel port ***
Testing parallel port register access
Test Passed
*** Parallel port tests complete ***
*** Testing trick box aborts ***
Testing reads from RW_Abort space:
Test Passed
Testing writes to RW_Abort space:
Test Passed
Testing reads from R_Abort space:
Test Passed
Testing writes to R_Abort space:
Test Passed
*** Trick box abort tests complete ***
*** Testing trick box cycle counters ***
Testing IRQ_CNT
Test Passed
Testing FIQ_CNT
Test Passed
*** Trick box cycle counter tests complete ***
*** Testing Flash control registers ***
Testing Flash Id
Test Passed
Testing Programming voltage
Test Passed
*** Flash control register tests complete ***
*** Calculating processor and bus speed ***
CPU core clock frequency is 213.4 MHz
System bus frequency is (CPU core frequency)/5
*** Testing COM port input and interrupts ***
Testing COM1 input
############################################################
Please type some characters on COM1, followed by the return key
############################################################
>> The quick brown fox
Did the characters echo correctly (y/N)?
>> y
COM1 input OK.
Testing COM2 input
############################################################
Please type some characters on COM2, followed by the return key
############################################################
Did the characters echo correctly (y/N)?
>> y
Test Passed
Testing COM1 interrupt paths
############################################################
Please press the return key on the terminal attached to COM1
############################################################
Testing IRQ path
Test Passed
Testing FIQ path
Test Passed
Testing IRQ path with no device interrupt asserted
Test Passed
Testing FIQ path with no device interrupt asserted
```

```
Test Passed
Testing COM2 interrupt paths
#############################################################
Please press the return key on the terminal attached to COM2
#############################################################
Testing IRQ path
Test Passed
Testing FIQ path
Test Passed
Testing IRQ path with no device interrupt asserted
Test Passed
Testing FIQ path with no device interrupt asserted
Test Passed
Testing COM2 interrupt paths
*** COM port tests complete ***
*** Testing Flash ***
Searching for bootable images in Flash
No images found
###########################################################
WARNING:  Performing the Flash integrity check will delete
          all blocks that are not part of valid images
###########################################################
Should the Flash integrity check be performed (y/N)?
>> y
Flash integrity test starting
Testing block 0
Test Passed
Testing block 1
Test Passed
Testing block 2
Test Passed
Testing block 3
Test Passed
Testing block 4
Test Passed
Testing block 5
Test Passed
Testing block 6
Test Passed
Testing block 7
Test Passed
Testing block 8
Test Passed
Testing block 9
Test Passed
Testing block 10
Test Passed
Testing block 11
Test Passed
Testing block 12
Test Passed
Testing block 13
Test Passed
Testing block 14
Test Passed
Testing block 15
Test Passed
Flash integrity tests passed
*** Flash tests complete ***
```

```
Summary of board configuration detected
=======================================
CPU identification is 0x4401a100
CPU core clock frequency is 213.4 MHz
System bus frequency is (CPU core frequency)/5
CTB configured as ARCH
SSRAM size is  0x20000
Two EDO DRAM SIMMs fitted (SIMM size is 0x8 Mbytes)
DRAM size is  0x1000000
MAC address 08-00-2b-95-1d-7b
Jumper settings:
 J2 pins  9-10 Not fitted
 J2 pins 11-12 Fitted
 J2 pins 13-14 Fitted
 J2 pins 15-16 Not fitted
No images were found in Flash

Is this summary correct(y/N)?
>> y

*** TESTS PASSED ***
(No errors detected)

*** Programming Flash image 0 (Primary boot loader) ***
*** Flash image 0 programmed ***
*** Programming Flash image 1 (Power-on diagnostics) ***
*** Flash image 1 programmed ***
*** Verifying Flash images ***
Flash image 0 correct
Flash image 1 correct
*** Flash images correct ***
=========================================================
Selftests complete, please reset jumpers before rebooting
=========================================================
```

9
Software Utilities

This chapter describes two software utilities supplied as part of the design database. These utilities are:

- The Flash management utility (FMU)
- The bootp Ethernet load utility

These programs are supplied in source form and as ARM Image Format (AIF) files.

9.1 The Flash Management Utility

Images are programmed into Flash using the Flash Management Utility (FMU). The executable, fmu.aif, is loaded and started using any of the ARM debuggers. The FMU uses the ARM debugger I/O services to provide a command-line interface. When you start the FMU, it checks for the presence of a Flash ROM, issues a start-up message and then prompts for user input:

```
Flash Management Utility [1.0]
Searching for flash device
Flash found at 0x80000000 (16 blocks of size 0x10000)
Scanning Flash blocks for usage
FMU>
```

The FMU Provides these commands:

- Help - List all of the available commands

```
FMU> help
FMU command summary:

List                - List images in flash
ListBlocks          - List how each Flash block is being used
TestBlock <block-number>
                    - Write a test pattern to a particular flash block
Delete <image-number>
                    - Delete an image in flash
DeleteBlock <block-number>
                    - Deletes a block that appears not to be in an image
DeleteAll           - Deletes all blocks except block 0
Program <image-number> <image-name> <file-name> [<block-number>] [NoBoot]
                    - Program the given image into flash
Quit                - Quit
Help                - Print this help text
```

- List - List the images in Flash. For example:

```
FMU> list
Listing images in Flash
Image 0 "BootLd          " Length 45232 bytes, Map 0x00000001
Image 1 "LedLoop         " Length 536 bytes, Map 0x00000002 Noboot
Image 2 "eForth:12-feb   " Length 69848 bytes, Map 0x0000000c
```

  - You supply the image number and name when you program the image.

  - The length shown is the size of the image including all headers.
  - The map is a bit map showing which blocks of the Flash are occupied by the image; bit 0 of the map corresponds to block 0 of the Flash, and the image's header is in the lowest block occupied by the image.
  - You optionally supply the NoBoot option when you program the image.

- ListBlocks - List how each Flash block is being used. The first few bytes of the Flash block are listed. If the block contains an image, its image number is given. For example:

```
FMU> listblocks
 0: (Image  0) 0x2e 0x00 0x00 0xeb 0x00 0x55 0xaa 0x00
 1: (Image  1) 0x02 0x00 0x00 0x00 0xe0 0xdd 0x21 0xc6
 2: (Image  2) 0xd8 0x10 0x01 0x00 0x65 0x46 0x6f 0x72
 3: (Image  2) 0x4c 0x0a 0x00 0x40 0x10 0x03 0x00 0x00
 4: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
 5: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
 6: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
 7: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
 8: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
 9: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
10: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
11: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
12: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
13: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
14: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
15: (Unused)   0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
```

- TestBlock <block-number> - Test a particular Flash block by writing a test pattern to the block and then verifying it. For example:

```
FMU> testblock 15
Do you really want to do this (y/N)? y
Writing test pattern to block 15
Reading test pattern from block 15
Flash test of block 15 worked
```

- Delete <image-number> - Delete an image in Flash. You cannot normally delete the Flash image that starts in Flash block 0 (the primary boot loader). The only time that the FMU utility permits you to do this is if the ARM remote debugger stub is executing from EPROM, rather than Flash. For example:

```
FMU> delete 3
Do you really want to do this (y/N)? y
Deleting flash blocks: 4
Scanning Flash blocks for usage
FMU>
FMU> delete 0
WARNING: Deleting flash boot block
Do you really want to do this (y/N)? y
Deleting flash blocks: 0
Scanning Flash blocks for usage
```

  If you are running an ARM remote debugger stub from an image other than image 0, then you can delete that image, but the FMU will be terminated during the delete. If you restart the system, it will execute the PBL and run correctly.

- DeleteBlock <block-number> - Delete a block that is not part of an image. This may be used to clean out corrupt blocks, or blocks that have been programmed by the TestBlock command. The FMU will not allow you to delete a block that is part of a valid image. For example:

```
FMU> deleteblock 15
Do you really want to do this (y/N)? y
Delete flash block 15
Scanning Flash blocks for usage
```

- DeleteAll - Delete all blocks except block 0.
- Program <image-number> <image-name> <file-name> [<block-number>] [NoBoot] - Program the image with name <image-name> into the the Flash as image number <image-number>. The image is read from the host from file <file-name> (which may include a directory name). Refer to Section 9.1.1 for details on the block-number option and to Section 9.1.2 for details on the NoBoot option. The Program command will fail with an error if:
  - The image number is already in use
  - There is insufficient free space in the Flash
  - The specified blocks are not free
  - The file does not exist or cannot be opened

  For example:

```
FMU> program 3 ledloop2 d:\users\crook\ledloop.aif noboot
Writing d:\users\crook\ledloop.aif into flash block 4
Deleting blocks ready to program:
Deleting block 4
Calculating checksum
Writing flash image header
Image is non-executable AIF file
The bootloader will copy this image to 40000000 before executing it
Writing image file
Scanning Flash blocks for usage
```

- Quit - exit from the FMU. When this command is executed, the FMU will return control to the debugger.

9.1.1 When to Specify the Block Number

By default, the FMU 'Program' command will program an image into Flash using any free blocks allocated in ascending block order. This can result in an image occupying non-contiguous blocks within the Flash.

When an image is a non-executable image (an image that will be loaded into system memory by the PBL prior to execution) the PBL will load an image from non-contiguous Flash blocks into contiguous system memory. Therefore, allowing an image to occupy non-contiguous Flash blocks makes efficient use of the Flash by avoiding fragmentation problems.

When an image is an executable image, it must occupy contiguous blocks within the Flash. In general, it must also have been linked to excute from a specific address (and therefore block) in the Flash.

Therefore, when using the FMU to program an executable image, you must specify the block-number when you issue the 'Program' command.

When a block-number is specified, the FMU will program the image into contiguous Flash blocks, starting from the specified block. The command will fail if insufficient unused contiguous blocks are available.

Refer to Section 8.1 for information on the PBL, and to Section 7.2 for information on building images that can be executed from Flash.

9.1.2 When to Specify the 'NoBoot' Option

The usual reason to program an image into Flash is so that it can be automatically executed after reset or power-on. If the image number is less than 7, the boot jumpers can be set so that the PBL will load and execute the image after a reset or power-on.

Sometimes, it is desirable to have the PBL load the image into system memory but then drop into the ARM remote debug stub. This process allows the image to be started up under the control of the debugger, to use the I/O facilities of the debugger and ultimately to pass control back to the debugger when the image terminates.

If you use the the NoBoot option when programming an image into Flash, the PBL will load the image into system memory but will not execute it; instead, control will pass to the ARM remote debug stub within image 0.

There is no way to change the state of the NoBoot flag for an image, once it has been programmed; you must delete the image and reprogram it with the NoBoot flag changed.

Refer to Section 8.1 for information on the PBL and the boot jumpers.

9.2 The Bootp Utility

The bootp utility provides a way of quickly loading large test programs using an Ethernet LAN. To use it, you need:

- Access to an Ethernet LAN
- A bootp server
- An IP address for your EBSA-110. It is a restriction of the bootp protocol that this IP address must be in the same subnet as the bootp server.

Before your bootp server will respond to load requests from your EBSA-110, you must configure it to recognize the IP address of your EBSA-110. Consult your local documentation for details. It will probably require you to add entries to your /etc/bootptab and /etc/hosts files, for example:

```
in /etc/bootptab:

TWIST:ht=ethernet:ha=08002b951d75:ip=16.36.0.30:\
:hd=/usr/users/tester/boot/arm/:bf=flash_test.aif_dram:vm=auto:

in /etc/hosts: for TWIST as:

# Evaluation Boards
16.36.0.30 twist.reo.dec.com twist TWIST
```

The EBSA-110 hardware address (08002b951d75 in the example above) is displayed when running the diagnostics. You can also find it out by running the bootp program.

The executable, bootp.aif, is loaded and started using any of the ARM debuggers. It uses the ARM debugger I/O services to provide a command-line interface. When you start the program, it checks the Ethernet interface and issues some start-up messages.

The /etc/bootptab allows you to specify a default image to be loaded (flash_ test.aif_dram in the /etc/bootptab example above). The bootp program loads images into memory starting at 0x8000 by default. If you use both of these defaults, the session will look similar to this:

```
Starting bootp/tftp test; initializing networking components
Am79C961 driver loaded:         Ethernet Device: 0
MAC address:  08-00-2b-95-1d-75
Please enter file name (or just CR for default)

Please enter, in hex, the load address in memory (or CR for default, 0x8000)

Attempting BOOTP.
..
 My IP address:     16.36.0.30
 Server IP address: 16.36.0.188

Loading /usr/users/tester/boot/arm//flash_test.aif_dram at 0x40000000

File loaded
```

If you choose to specify the filename and load address explicitly, the session will look similar to this:

```
Starting bootp/tftp test; initializing networking components
Am79C961 driver loaded:         Ethernet Device: 0
MAC address:  08-00-2b-95-1d-75
Please enter file name (or just CR for default)
/usr/users/tester/boot/arm/test
Please enter, in hex, the load address in memory (or CR for default, 0x8000)
1ff00
Attempting BOOTP.
....
 My IP address:     16.36.0.30
 Server IP address: 16.36.0.188

Loading /usr/users/tester/boot/arm/test at 0x1ff00

File loaded
```

Once the image has been loaded, the bootp program returns control to the ARM remote debug stub.

You can use the ARM debugger to start or debug the loaded image, but you do not have access to the image's symbolic information.

9.2.1 Variants of the bootp Program

Two variants of the bootp program are supplied:

- bootp.aif - linked to load and run at address 0x0000.8000 (the first available location in DRAM).
- bootp_4.aif - linked to load and run at address 0x4000.0000 (the first location in SSRAM).

Both versions are linked -aif -bin. They can either be loaded using the debugger or programmed into Flash (using the NoBoot flag) and loaded automatically by the PBL.

10
Theory of Operation

This chapter provides a technical description of the EBSA-110 hardware. It should be read in conjunction with the EBSA-110 schematic set, programmable logic listings and timing diagrams (all of these are provided as part of the EBSA-110 Design Database - refer to Appendix C). You should read this chapter if you wish to gain a detailed understanding of the operation of the board. You are assumed to:

- Have a background in high-speed digital design
- Have some familiarity with the ARM architecture and the SA-110 bus interface
- Have access to the manufacturer's data sheets for the memories and VLSI devices used on the EBSA-110

Specific pages of the schematic set are referenced by sheet number (for example, SHT6). The sheet number is shown in the bottom right-hand corner of the schematic.

This chapter includes:

- A topic-by-topic tour of the EBSA-110 schematics, including a description of the principal buses.
- A description of the control logic, which is implemented in two programmable logic devices referred to as 'CTA' and 'CTB'.
- A discussion on how an expansion board could be designed for the EBSA-110.
- A summary of the design rules used for the PCB layup and routing.

Simulation waveforms for all the important state machine sequences, together with descriptive commentaries, can be found in Chapter 11.

10.1 A Tour of the Schematics

This section describes the principal buses in the EBSA-110 design, and then describes the implementation and operation of each functional block, whilst cross-referencing to the relevant pages of the schematic set.

The block diagram (SHT1 of the schematics, included as Figure 2–1 in Chapter 2) shows the VLSI devices and the connection of the principal buses, and also provides a cross-reference to the location of any particular functional block within the schematic set.

On the schematics, every signal has a three-letter prefix † which indicates the origin (driver) of the signal. For bidirectional signals, the 'most important' driver of the signal determines the prefix.

† There are a few exceptions, but they should not cause confusion.

10.1.1 Principal Buses

The principal buses are:

- **cpu_a[31:2]** - the CPU address bus. This 30-bit bus has +3.3V switching levels (it is not 5V tolerant) and drives the SSRAM directly. Some bits are driven into CTA, CTB where they are used for address space decoding. The address bus provides a longword address. Byte resolution is provided by the byte lane enables, **be[3:0]_l**.
- **buf_a[29:2]** - buffered address bus. This 30-bit bus is generated from **cpu_a** by two 74LVT16244 buffers on SHT6. It therefore has +3.3V switching levels but is 5V tolerant. The LVT buffers are permanently enabled. **buf_a** drives the DRAM address multiplexer, the EPROM/Flash and all I/O devices.
- **cta_trick_a[2:0]** - trick-address bus. This 3-bit bus is generated in CTA on SHT8 and used solely in CTB, also on SHT8. It is a modified version of **cpu_a[24:22]** - each address line is ANDed with **cpu_a25**. **trick_a** is used to decode accesses to the PIT and to the interrupt control registers within CTB.
- **mux_a[10:0]** - multiplexed row/column address bus. This 11-bit bus is generated from **buf_a** by a 74ABT162260 on SHT7. It has 5V switching levels. It is used as the address bus for the DRAMs.
- **cpu_d[31:0]** - the CPU data bus. This 32-bit bus has +3.3V switching levels (it is not 5V tolerant) and drives the SSRAM directly.
- **buf_d[31:0]** - buffered data bus. This 32-bit bus is generated from **cpu_d** by two 74LVT16543 latching buffers on SHT6. It therefore has +3.3V switching levels but is 5V tolerant. The LVT buffers are bidirectional and the control signals are generated by CTA, on SHT8. The buffers have independent OE control and (transparent) data latches on each port. **buf_d** drives the DRAMs and the I/O data bus buffers.
- **io_d[15:0]** - I/O data bus. This 16-bit bus is generated from the 32-bit **cpu_d** by two 74ABT16543 latching buffers on SHT6. It has 5V switching levels. The ABT buffers are bidirectional and the control signals are generated by CTB, on SHT8. **io_d** drives all I/O devices and the ROM data buffer. In the write direction (**buf_d** driving **io_d**) the **io_d** is always driven from the two low-order bytes of **buf_d** (**buf_d[15:0]**). This is the data path used for all I/O writes, and for writes to the Flash. In the read direction (**io_d** driving **buf_d**) the value of **io_d[15:0]** can be latched in the two high-order or the two low-order bytes of **buf_d**. This means that a 2-stage process can allow the 16-bit **io_d** bus to drive a 32-bit value on **buf_d**. This 2-stage process is used for EPROM/Flash reads. All I/O reads return data to the CPU on **d[15:0]** via **buf_d[15:0]** and **io_d[15:0]**.
- **rom_d[7:0]** - ROM data bus. This 8-bit bus connects the Flash/EPROM to the 16-bit **io_d** bus via a 74ABT16543 latching buffer on SHT10. It has 5V switching levels. The ABT buffer is bidirectional and the control signals are generated by CTB, on SHT8. In the write direction (used for Flash writes) **cpu_d[7:0]** drives **rom_d[7:0]** via **buf_d[7:0]** and **io_d[7:0]**. In the read direction (used for Flash and EPROM reads) **rom_d[7:0]** can be latched in the high-order or low-order byte of **io_d**. This means that a 2-stage process can allow the 8-bit ROM to drive a 16-bit value on **io_d**. When used in conjunction with the **io_d** bus latches, this process is used to pack data from four successive ROM addresses into a 32-bit value on **cpu_d** (via **buf_d**).

10.1.2 Power

Power comes onto the board through a PC-style 12-way connector (SHT21). The board uses +5V and +12V straight from this connector. On-board regulators generate +3.3V and +2V.

Most devices on the board use +5V power.

The +3.3V power is used for the I/O buffers of the SA-110, for the SSRAM and for the LVT buffers. +3.3V is regulated directly from +5V using a low-dropout regulator, a Linear Technology LT1086 (SHT20). This is an adjustable regulator set to provide a +3.3V output.

The following devices require current at +3.3V:

- 2 LVT data bus buffers (5mA each)
- 2 LVT address bus buffers (5mA each)
- 1 quickswitch level converter (10mA)
- 1 SSRAM (170mA)
- SA-110 I/O cells
- Current sourced by the outputs of all these devices

The first elements of this list sum to 0.26A, so a 1A regulator provides sufficient margin to account for the output drive requirements.

The +2V power is used for the core of the SA-110. +2V is regulated from the +3.3V rail. This ensures that the +3.3V rail is stable earlier than the +2V rail during power-on, and therefore protects the SA-110 against latch-up. The +3.3V rail has enough power to meet the additional load imposed by the +2V regulator. A second LT1086 is used (SHT20). The voltage output can be adjusted to +1.5V by adding a jumper on the board. This lower voltage is used by the SA-110 at lower core clock frequencies.

The power dissipation of the SA-110 is between 290mW and 860mW, depending upon the core voltage and frequency. Therefore the core power consumption will not exceed 430mA. A 1A adjustable regulator provides sufficient margin.

The +12V power is used for programming the Flash ROM and is available for use by plug-in PCMCIA cards.

The RS232 drivers do not require +12V or -12V because they have integral bias generator logic, as described in Section 10.1.14.

10.1.3 Decoupling

The EBSA-110 uses tantalum electrolytic capacitors for bulk decoupling of the power rails, and ceramic capacitors for decoupling of individual ICs.

The bulk decoupling capacitors, which are a mixture of 10uF and 47uF parts, are evenly distributed around the board. They are shown on SHT19 and SHT21. In addition, bulk decouplers are located physically close to:

- The connector that brings power onto the board (one capacitor for +5V and one for +12V).
- The input and output of each voltage regulator.

0.1uF ceramic decoupling capacitors are located physically close to the power and ground pins of the chip they are intended to decouple. The decoupling capacitors are shown on the same schematic sheet as the device they decouple. Exceptions to this rule affect the CPU (SHT3) and the programmable logic (SHT8), and are marked on the schematics.

10.1.4 Voltage Levels

The SA-110 I/O pins switch at +3.3V and are not +5V tolerant. Since many of the devices on the EBSA-110 are +5V parts, they must be connected via level-translation circuitry.

The SA-110 interfaces directly to the SSRAM. The SSRAM is a +3.3V part.

The SA-110 address bus, data bus and byte enables are all buffered using LVT parts. These switch at +3.3V but are +5V tolerant. This allows the SA-110 to be interfaced to the +5V parts on the rest of the board.

SA-110 outputs, such as **cpu_mreq_l** are used directly as inputs to the control logic; CTA and CTB. These devices have CMOS (high-impedance) inputs with TTL input switching thresholds which means that a +3.3V CMOS (10%-90% swing) output can be interfaced directly. Since the inputs are CMOS, there is no danger of a current path back from the control logic to SA-110 (for example, during power-on).

Signals that are used as inputs to the SA-110 but which have been generated with 5V switching levels must be level converted before they can drive the SA-110 pins. This affects 10 signals and the level-conversion is performed using a Quickswitch QS3384 (equivalent pin-compatible devices are available from Texas Instruments and National Semiconductors). This part is shown on SHT3. On the schematics, signals that have been level-shifted to +3.3V have a **3V3_** prefix. For example, **cta_wait_l** is converted to **3V3_wait_l**. The QS3384 acts as a set of bidirectional FET switches. It introduces negligible delay (25ps). Since the FET switches saturate, the switching level can be controlled by controlling the saturation (supply rail) voltage. With the QS3384 powered at +4.3V †, the driven output will be limited to +3.3V, even under light loading.

Note that these devices can also be used as bidirectional converters, since they simply act as low-impedance switches (they introduce some resistance and have no gain).

10.1.5 Clocks

The EBSA-110 uses 6 clocks:

- SA-110 PLL input clock, **osc3**
- SA-110 output clock **cpu_mclk**
- SA-110 output clock **cpu_mclk_l**
- Ethernet controller clock
- Super I/O controller clock
- PCMCIA controller/PIT clock

† Quality Semiconductor recommend using a diode to get this voltage drop, but experiment showed that this did not work. The EBSA-110 uses a pair of resistors as a voltage divider.

The SA-110 has an internal PLL which is driven from a 3.68 MHz input clock. This clock is generated from a TTL baud-rate oscillator (SHT3) as **osc3** and is then level-shifted to generate **3V3_osc3** (SHT3). The clock circuitry shown on SHT3 is intended to allow an off-board signal generator to be used to drive either the PLL input clock or a full-speed test clock, bypassing the PLL. Both of these options are only intended for chip verification.

The SA-110 generates **cpu_mclk** and **cpu_mclk_l** which are used to synchronize bus interface operations. For the EBSA-110, this clock has a nominal frequency of 55 MHz. The EBSA-110 uses **cpu_mclk_l** to clock the SSRAM and the control logic; CTA and CTB (3 loads in total).

The SA-110 uses the signal from the **cpu_mclk** (output) pin to control its bus interface unit internally. Therefore, use of **cpu_mclk_l** externally can cause a skew between the bus timing produced by the CPU and the bus timing generated by the external control logic. The solution to this problem † when using **cpu_mclk_l** is to add a dummy load to **cpu_mclk** so that the load on both clocks is the same. On the EBSA-110 this is achieved by adding an etch stub to **cpu_mclk** (which would otherwise be unconnected) and matching the length of this stub to the length required to route **cpu_mclk_l** to the SSRAM and control logic. The etch stub is terminated in a capacitor, which mimics the pin-load imposed by the devices on **cpu_mclk**.

Note

Designs that do not use **cpu_mclk_l** can leave **cpu_mclk_l** unconnected; no analagous termination is required on this signal.

The Ethernet controller (SHT13) generates its clock from a 20 MHz crystal.

The Super I/O controller (SHT11) uses a TTL, 24 MHz oscillator to generate its input clock, **osc24**.

The PCMCIA controller (SHT16) and the PIT (SHT19) both use a low-frequency clock **ctb_clkby7** which is generated (in CTB) by dividing the **mclk** by 7. The result is an asymmetric clock with a high-time of 4 **mclk** periods and a low-time of 3 **mclk** periods. For the supported **mclk** frequencies, this gives a nominal **ctb_clkby7** frequency of 7.6 MHz. This clock is *not* phase-synchronized with I/O cycles.

10.1.6 Reset

Reset can be generated:

- Automatically at power-on, by a resistor-capacitor-diode network (SHT18)
- By a push-button (connected to J2, SHT22)
- Under remote control, by a debug box attached to the JTAG connector (**srst_l**, from J3, SHT18)

† This problem arises because, for historical reasons, the bus clock is the inverse of what you would expect; the control logic and the SSRAM must be clocked on the falling edge of **cpu_mclk**. For low frequency designs, this is a problem that can be solved trivially using an inverter. At high frequencies, the skew and delay introduced by an inverter would be unacceptable.

These three reset sources are combined and then debounced by a schmitt trigger circuit to generate **rst_reset_l** (SHT18). This is level-shifted (SHT3) to generate **3V3_reset_l** which resets the CPU. After system power-on or a CPU power-on, it takes many microseconds for the CPU PLL to become stable. Therefore, the CPU generates a reset output, **cpu_reset_l**, which is used to keep external circuitry in a reset state until the **cpu_mclk** is stable‡. **cpu_reset_l** is buffered (SHT18) to generate **buf_reset_l**, **buf_reset1** and **buf_reset0** and these are used to reset the VLSI devices on the board (SHT18 of the schematics details which reset is used for which device). Series source terminations are used on these resets to prevent ringing, since they potentially drive long etch lengths.

10.1.7 The CPU

The SA-110 (SHT3) has a number of configuration options. On the EBSA-110 all of the configuration pins are wired to +3.3V or 0V via outer-layer etch links on the PCB. This allows other modes to be selected in special applications. These options are selected by default:

- Synchronous bus mode: the CPU generates **mclk** as an output. All bus operations are synchronous to **mclk**, and **mclk** is a sub-multiple of the core clock.
- Fastbus mode: the address timing is pipelined.
- Enhanced bus mode: cache line fetches can be wrapped, the write buffer can merge operations and the CPU supplies byte masks.

The SA-110 core clock and bus clock frequencies can be selected using jumpers on SHT4. The core clock can be run at any of the frequencies supported by the SA-110 and the **mclk** divisor must be set to give a maximum **mclk** frequency of 55 MHz.

The SA-110 is always the bus master, and so it never needs to tristate its address, data or control signals. Therefore, the **abe**, **dbe** and **mse** inputs are tied asserted (via etch links).

The SA-110 powerdown capability is not used by the EBSA-110 and so the **pwrslp_l** input is tied negated (via an etch link).

The high-drive critical signals from the CPU (**cpu_mreq_l**, **cpu_mclk** and **cpu_mclk_l**) have source series terminations to prevent ringing.

Chapter 11 contains detailed descriptions of the bus cycles performed by the CPU.

10.1.8 Jumpers, Etch Links, Debug Connectors and Test Points

All of the configuration options on the EBSA-110 are controlled either by plug-in 2-pin 0.1" jumpers (for options that you may wish to change) or by outer-layer etch links on the PCB (for options that will only change under exceptional circumstances).

Since the EBSA-110 is a debug vehicle for the SA-110 it provides a number of connectors to help debug. The less speed-critical signals are routed to 16-pin 0.1" pitch connectors (SHT4). These connectors are wired with odd-numbered pins connected to 0V. They are designed for direct connection to Tektronix DAS logic state analyzer pickup pods. Speed-critical signals (where it is not acceptable to increase the etch length for the purpose of debug) have an in-line PCB via clear

‡ It is not strictly necessary to use this signal for the EBSA-110 design, because the power-on reset pulse will be long enough to ensure that the PLL is stable. Designs that power-down the CPU to save power will need to use the CPU's reset.

of any component footprint. These vias are populated with Harwin test pins to allow an oscilloscope or logic state analyzer to be attached easily.

The EBSA-110 schematic directory (SHT1) provides a reference to the whereabouts of all jumpers, pickups and links. Appendix A is the single reference point within this document for all jumpers, etchlinks and connectors on the board.

10.1.9 SSRAM Interface

The EBSA-110 uses a Micron MT58LC32K36C4 32Kx32 synchronous SRAM. This part is pipelined and so, on reads, it takes one cycle between the address being sampled and the read data being supplied. All operations are sychronized to **cpu_mclk_l**.

The SSRAM is a +3.3V part, but its I/O is +5V tolerant. Its address and data signals, together with the byte enables, are directly connected to the SA-110 and use +3.3V switching. The control signals are generated by CTA (SHT8) and use +5V switching, relying on the +5V tolerance of the SSRAM's inputs.

The SSRAM is a burst-mode part. Once given an address, it uses an internal address counter to generate the addresses of the other three locations in the same INT16 block. If the initial address is not aligned to an INT16 boundary, the address will wrap at some point. The SSRAM is configured to use linear wrapping (by negating the **mode** input). If it is unnecessary to access all 4 locations of the block, a new address can be loaded at any point. Loading a block start address is always an explicit process, and is initiated by SSRAM control inputs.

cta_oe_l, **cta_ce_l** and **cta_adsc_l** are used to control accesses to the SSRAM. **cta_adv_l** is used to stall the burst rate. **cta_bwe_l** is used to enable the byte masks during write operations.

The SSRAM is never put into powerdown mode. By default, every address presented on the address bus by the CPU will be latched into the SSRAM as a potential read start address. This technique reduces the access latency when the CPU performs an access to the SSRAM. Examples of SSRAM accesses are shown in the simulation waveforms in Chapter 11.

The SSRAM footprint on the EBSA-110 is designed to accommodate either a 32-bit or a 36-bit part. It can also accommodate the next-generation 64Kx36 part.

10.1.10 Buffering

The CPU address and data buses are buffered. The low-order 16-bits of the buffered data bus is buffered again to generate an I/O data bus. This buffering is shown on SHT6. The buffered buses are described in Section 10.1.1.

10.1.11 DRAM Interface

The EBSA-110 uses +5V, 32-bit, 72-pin DRAM SIMMs and can accommodate 2 parts (SHT7). It accommodates Extended Data-Out (EDO) or Burst Extended Data-Out (BEDO) parts. The DRAM state machine uses a software-configurable input to control state transitions to ensure correct operation for either type of DRAM.

An EDO DRAM is like an ordinary DRAM, except that the read data does not tristate when CAS negates. This simple modification allows greater memory bandwidth, since CAS can be negated (starting the CAS precharge) sooner.

A BEDO DRAM has an internal 2-bit address counter. Like an SSRAM, it takes an address and uses an internal counter to generate the addresses of the other three locations in the same INT16 block. If the initial address is not aligned to an INT16 boundary, the address will wrap at some point. The BEDO uses a special write-CAS-before-RAS cycle to configure whether the wrapping mode is either linear or interleaved. If it is unnecessary to access all 4 locations of the block, a new address can only be loaded after explicitly stopping the current burst. (With an SSRAM every address load is explicit, and burst termination is implicit. With BEDO DRAMS it is the other way around; address load is normally implicit, and so burst termination must be explicit.)

A BEDO DRAM is controlled using the usual **ras_l**, **cas_l** and **we_l** control signals. BEDO DRAM chips also have an **oe_l** input, to facilitate page-mode read-write cycles. This signal is not available when SIMMs are used.

The row/column address multiplexing is performed by a 74ABT162260 (SHT7).

Sequential cycles from the CPU are always performed as page-mode DRAM accesses. For this to work correctly, sequential accesses must not be able to cause a transition in any address line that is used for the DRAM ROW address.

The CPU cannot perform a sequential access that crosses a 2048 byte boundary. Therefore, the least-significant bit that is guaranteed not to change during a sequential access is **a[11]**. The least-significant bit that is used as a DRAM ROW address is **a[12]**. Therefore, page-mode DRAM accesses will always work correctly for sequential cycles.

The DRAM controller always terminates a page mode cycle when the CPU terminates its sequential access (it does not speculatively keep the page open).

Byte operations are performed by decoding the CPU byte enables (within CTA) and using them to qualify the four **cas** signals during write operations.

The **ras** signals are decoded from CPU address lines. The decoding is a function of the SIMM size fitted, and this is determined by monitoring the **sim_id[2:1]** outputs of the SIMM. Refer to the CTA source file for details of the decoding.

10.1.12 Control Logic

The control logic is described in Section 10.2. The control logic is on SHT8. Each of the control signals generated by the control logic has a series source-termination resistor, and these are on SHT9. For example, the SSRAM output enable signal is generated by CTA as **un_oe_l** on SHT8 and goes through a series resistor on SHT9 to generate **cta_oe_l**. The EBSA-110 PCB is designed so that these series resistors are physically close (shortest possible etch length) to their drivers.

10.1.13 EPROM/Flash

The EBSA-110 accommodates a 512Kbyte EPROM (which is socketed) and a 1024Kbyte Flash ROM (SHT10). These are both mapped into the memory map simultaneously, and they are normally decoded within the memory quadrant where **cpu_a[31:30]**=[1,0]. After reset, they are also decoded in the bottom quadrant of memory (**cpu_a[31:30]**=[0,0]), which allows the CPU to perform its inital opcode fetches from ROM. There is a jumper input to the control logic **lnk_eprom_boot_l** to control whether the EPROM or the Flash is decoded in the lower half of the ROM quadrant. When the jumper is removed (the default), the Flash will be decoded in the bottom half of the quadrant, and the system will attempt to boot from the image in Flash. The EPROM is only necessary for

manufacturing use (to allow the Flash to be programmed easily on a new system) and may not be fitted in production boards.

Read and write accesses to the ROMs are controlled by the IO state machine in CTB. The ROMs are byte-wide devices. During reads, the IO state machine supplies the two low-order address lines, **ctb_paka[1:0]**, from an internal 2-bit counter. It reads four consecutive locations and latches (packs) the data in data bus buffers, so that 32-bit data is supplied to the CPU.

The byte packing is facilitated by the **rom_d** data buffer (SHT10) and the **io_d** buffer (SHT6). The structure of the data buses is described in Section 10.1.1. The behavior of the byte packer is described in Section 11.15.

Writes (to the Flash) must be byte writes, on the low-order byte lane (the exact mechanism is described in Section 3.2.4, and the waveforms are shown in Section 11.17). It would be possible to allow 32-bit writes and then perform a byte unpacking sequence, but this would add complexity to the hardware for no benefit. In order to allow byte addressability of the Flash, the **ctb_paka[1:0]** counter is jam-loaded with address line information at the start of a Flash write sequence. The **cpu_a[23:22]** address lines are used to provide the low-order (byte addressing) information. The selection of address lines is arbitrary (provided they do not overlap the **buf_a[19:2]** used to address the higher-order address lines of the Flash) and **cpu_a[23:22]** are used because they are already required in CTB, where the **paka** counter is implemented.

10.1.14 SuperI/O Controller

The National Semiconductor 87312 Super I/O controller (SHT11) is used to provide two serial ports and one parallel (printer) port. This part also contains a floppy disk controller and IDE decode logic, but these functions are not used.

The 87312 is interfaced to the **io_d** and **buf_a** buses. The device is I/O-mapped, and is mapped into the EBSA-110 self-decoding ISAIO space (Section 3.2.9). Accesses to the 87312 are controlled by the IO state machine in CTB. This state machine imitates ISA-bus I/O cycles and generates **ctb_ior_l** and **ctb_iow_l**. The ISA-bus interface is self-clocked. **buf_a25** is connected to the **aen** input of the 87312. **aen** is used on the ISA bus to instruct an I/O device not to respond to the I/O command signals. Therefore, it can be used as a kind of active-low chip select. In the EBSA-110 design, it is used to differentiate between self-decoding ISAIO space and external-decoded ISAIO space. When **buf_a25** is ‘1’, external-decoded ISAIO space is decoded, and the 87312 will ignore the **ctb_ior_l** and **ctb_iow_l** strobes.

The 87312 is a byte-wide device, and is wired to the low-order byte lane.

The 87312 interfaces to the parallel port via a set of series termination resistors (SHT11). There are pull-up resistors on the control signals, and grounded 220pF capacitors on the data signals. The capacitors are intended to reduce electromagnetic radiation from these signals.

The 87312 interfaces to the serial ports via a pair of Maxim MAX211E RS232 driver/receivers (SHT12). These devices have integral switch mode power converters to generate +12V and -12V from their +5V supply rail. There are a bank of 0.1uF capacitors (SHT12) that act as reservoir capacitors for the power converters. RS232 Output signals are routed through inductors, to remove any switch-mode noise that may have cross-coupled onto them. This is intended to reduce electromagnetic radiation. All of the signals also have grounded 220pF capacitors on them, intended to reduce electromagnetic radiation from these signals.

10.1.15 Ethernet Controller

The AMD 79C961A Plug-and-play ISA Ethernet controller (SHT13) is used to provide an Ethernet interface.

The 79C961A can be configured to operate in bus master mode or shared memory mode. In the EBSA-110 design, it is configured to run in shared memory mode. In this mode, received packets are written to a piece of memory local to the 79C961A. Frames for transmission must be written into this memory by the CPU. This buffer memory (SHT14) is 64Kbytes in size (it is implemented using a 128Kbyte part, but the 79C961A is only capable of accessing 64Kbytes). The buffer memory is on a private address and data bus, and all accesses to it are controlled by the 79C961A. The CPU accesses this memory through a window in the 79C961A address space, and the 79C961A performs arbitration between its own accesses, and those performed on behalf of the CPU.

The 79C961A is interfaced to the **io_d** and **buf_a** buses. It has both memory-mapped and I/O-mapped resources and is mapped into the self-decoding ISAIO space (Section 3.2.9) and the self-decoding ISAMEM space (Section 3.2.8). Accesses to the 79C961A are controlled by the IO state machine in CTB. This state machine imitates ISA-bus cycles and generates **ctb_ior_l**, **ctb_iow_l**, **ctb_memr_l** and **ctb_memw_l**. **buf_a25** is connected to the **aen** input of the 79C961A to prevent it from responding to external-decoded ISAIO accesses.

The 79C961A generates **rdy** and can negate this signal during accesses in order to extend the cycle time. This mechanism is used to resolve arbitration conflicts during accesses to the buffer memory. **rdy** is an open-collector signal which is also driven by the VG468 PCMCIA controller. It has a pullup resistor (SHT9) and is synchronized within CTB. The IO state machine uses the synchronized version of this signal as an indication that it should extend the I/O cycle time (by keeping the strobe asserted). When the synchronized version of **rdy** asserts, the IO state machine terminates the I/O cycle by negating the strobe and then completing the cycle normally.

The 79C961A is a 16-bit device and supports byte and half-word accesses. The logic used to decode these accesses is described in Section 10.2.6.

The 79C961A uses an external 20 MHz crystal to generate all of its internal clocks. Its ISA-bus interface is self-clocked. It contains some sensitive analogue circuitry, and so has specific decoupling requirements, which are noted on the schematics (SHT13).

The 79C961A drives four LEDs which provide an indication of network activity and link state.

A 32-byte ROM (SHT14) is used to provide a unique ID for the Ethernet controller (its IEEE Ethernet address). The ROM interfaces to the system **buf_a** bus, but to the 79C961A private data bus, **net_d**. The 79C961 performs address decoding for the ROM and generates its chip-select, **net_cs_aprom_l**.

The analogue circuitry for the Ethernet interface (SHT15) provides a 10-baseT connection. The Vadem interface transformer (FL1020) provides filtering, common-mode chokes and the equalization resistor network. This area of the design is layout-sensitive, and we used these layout techniques:

- Orient the Ethernet controller so that the analogue connections are close to the interface transformer.
- Use wide etch trace for the analogue signals.

- Route each differential pair of signals so that the etch runs parallel where possible, and make the traces similar lengths. This maximizes the common mode rejection.
- Cut away the power and ground planes under the analogue signals, to reduce noise pickup.
- Use an isolated power plane for the chassis earth of the analogue circuitry, and only connect to the system chassis at a single point, using a low-inductance connection.

10.1.16 PCMCIA Controller

The Vadem VG468 PCMCIA controller (SHT16) is used to provide an interface to 2 PCMCIA sockets (SHT17).

The VG468 is interfaced to the **io_d** and **buf_a** buses. It is a 16-bit device and supports byte and half-word accesses. The technique used to decode these accesses is described in Section 10.2.6. The VG468 has both memory-mapped and I/O-mapped resources and is mapped into the self-decoding ISAIO space (Section 3.2.9) and the self-decoding ISAMEM space (Section 3.2.8). Its connection to the EBSA-110 is very similar to the 79C961A's interface. The differences are:

- The VG468 requires a clock. The **ctb_clkby7** is used. Because this has an unknown phase relationship to **ctb_iow_l** and **ctb_memr_l**, the VG468 must be configured, under software control, to operate in asynchronous mode.
- As well as extending bus cycles using **rdy**, the VG468 can also truncate bus cycles by asserting **zws_l**. This open-collector signal has a pullup resistor (SHT9) and is synchronized within CTB. The VG468 is the only driver of this signal. The IO state machine uses the synchronized version of **zws_l** as an indication that it should terminate the I/O cycle (negate the strobe) as soon as possible, but complete the cycle normally (maintain the same address hold time with respect to strobe negation).
- The VG468 has a **bale** input, which is normally used to control an internal transparent latch that latches high-order address lines. This is required on PC AT systems, because the high-order address lines are only valid at the start of the cycle. On the EBSA-110, all address lines are stable for the duration of the cycle, therefore this signal is tied permanently asserted, so that the latch is held open (transparent).

The VG468 directly connects to the pins of the PCMCIA sockets (SHT17) and generates control signals for the power-switching circuitry (SHT17).

Each PCMCIA socket is supplied with +12V and +5V power via a software-controlled power switch. These power switches are Linear Technology LTC1472 devices. The LTC1472 +3V3 switch is actually used to switch the +5V power, because this makes the interface to the VG468 easier; the LTC1472 is intended to interface to another Vadem part, and the power enable outputs have the opposite polarity on that part.

10.1.17 JTAG Port

The JTAG port (SHT18) is only used to connect to the CPU JTAG interface. It is designed to interface to an existing ARM debug unit. A 74ACT244 is used to avoid directly connecting the CPU to external signals. The TTL outputs from this device are interfaced to the CPU via level shifters (SHT3).

10.1.18 Counter/Timer

The Intel 82C54 three-channel programmable interval timer (PIT) is used to provide a refresh counter and two timer interrupts (SHT19). Channel 1 is dedicated to generating periodic refresh requests but the other two channels are unassigned. They can be used to generate interrupts to the CPU.

All three channels are clocked from **ctb_clkby7**, which has a nominal frequency of 7.6 MHz (refer to Section 10.1.5). The PIT outputs are effectively asynchronous. The **pit_do_rfrsh** output is synchronized to **mclk** in CTA. **pit_irq1** and **pit_irq2** are synchronized to **mclk** in CTB, if necessary.

The PIT is interfaced to the **io_d** and **buf_a** buses. The device is I/O-mapped, and is mapped into the EBSA-110 external-decoded ISAIO space (Section 3.2.9). Accesses to the PIT are controlled by the IO state machine in CTB. This state machine imitates ISA-bus I/O cycles and generates **ctb_ior_l** and **ctb_iow_l**. Since the PIT is not self-decoded, CTB generates a chip select, **ctb_pit_cs_l** during PIT accesses. The PIT bus interface is self-clocked.

The PIT is a byte-wide device, and is wired to the high-order byte lane of **io_d**.

10.2 Control Logic

The control logic is partitioned into two programmable devices. These are called CTA, CTB. Each is an Altera® EPM7096LC86-7. The hardware description is expressed using the Data I/O ABEL language. Source files are provided as part of the design database. These parts can be redesigned and reprogrammed to meet your special needs. Refer to Appendix E for a brief tutorial on the ABEL language.

CTA, CTB are both clocked from **cpu_mclk_l** and contain mostly synchronous logic. In particular, it is a tight constraint on each that only one PLD propagation delay is permitted between flops.

Each device contains a mixture of state machines and random logic.

CTA contains:

- Main state machine - main interface to the CPU. Directly controls CPU, main data bus buffers, SSRAM accesses and the interfaces/handshakes for IO and DRAM state machines.
- DRAM state machine - software-selectable to generate BEDO or EDO DRAM Cycles. Can accommodate various SIMM sizes by automatic sensing of SIMM ID signals. Could be reprogrammed to support non-EDO DRAMs if required.
- RFRSH state machine - takes refresh requests from external timer and arbitrates with the DRAM state machine to make it generate a CAS-before-RAS refresh sequence.
- 2-bit Burst counter - tracks the position within a burst during SSRAM and EDO and BEDO DRAM accesses.
- Memory remapping state bits - cleared at reset and set once the first CPU write has been performed.
- Address bus decoding - generates chip selects for some devices.
- Address bus decoding - generates RAS for the appropriate memory bank during DRAM accesses.

CTA generates these outputs:

- SSRAM control signals; **cta_oe_l**, **cta_ce_l**, **cta_adsc_l**, **cta_adv_l**, **bwe_l**.
- CPU cycle length control; **cta_wait_l**
- MUX_A control; **cta_sel_col**
- DRAM controls; **cta_ras_s0b0_l**, **cta_ras_s0b1_l**, **cta_ras_s1b0_l**, **cta_ras_s1b1_l**, **cta_cas[3:0]_l**, **cta_dram_we_l**
- Buffer control for BUF_D buffer; **cta_bufd_wroe_l**, **cta_bufd_rdoe_l**, **cta_bufd_rdg_l**, **cta_bufd_wrg_l**.
- Chip selects for the IO_D buffer and for the EPROM, FLASH and PIT.
- CPU ABORT signal; **cta_abort**.
- The TRICK_A bus; **cta_trick[2:0]**.
- Handshake from Main state machine to IO state machine; **cta_do_io**.
- Internal observation signals for debug; **cta_obs[4:0]**.

CTB contains:

- IO state machine - controls access to I/O devices and to the EPROM/Flash (specifically the byte-to-longword packer). The I/O sub-system is based on IBMPC peripheral chips and so the IO State machine synthesizes an asynchronous ISA-like bus. All of the I/O is performed by CPU reads and writes; there is no DMA support.
- CLKBY state machine - generates a divide-by-seven clock for clocking some I/O devices. This clock is synchronous but has no defined phase relationship with the I/O strobes.
- 2-bit address counter - used by the EPROM/Flash byte-to-longword packer.
- Interrupt mask register - allows any interrupt source to be routed to the IRQ pin under software control.
- Fast Interrupt mask register - allows any interrupt source to be routed to the FIQ pin under software control.
- Fast counter - allows IRQ interrupts to be generated under software control.
- Fast counter - allows FIQ interrupts to be generated under software control.
- 2-bit counter - used as a general-purpose resource by the IO state machine to slow down some sequences without suffering state explosion.

CTB generates these outputs:

- Interrupts to the CPU; **ctb_fiq_l**, **ctb_irq_l**.
- Software-programmable outputs; **ctb_soft_burst**, **ctb_soft_dcbr**, **ctb_softo2**, **ctb_soft_led_l**.
- 8-bit data bus for connection to IO_D; **d[7:0]**.
- Divided-by-7 output clock for I/O devices; **ctb_clkby7**.
- I/O space memory and I/O strobes; **ctb_memr_l**, **ctb_memw_l**, **ctb_ior_l**, **ctb_iow_l**.
- Latch controls for the EPROM byte packer; **ctb_latw_l**, **ctb_latb_l**.
- Data-path controls for the IO_D bus; **ctb_io_wroe_l**, **ctb_io_rdoe_l**.

- Low-order address lines for EPROM/Flash; **ctb_paka[1:0]**.
- Handshake from IO state machine to Main state machine; **ctb_io_ack**.
- Write strobe for Flash; **ctb_flash_wr_l**.
- Internal observation signals for debug; **ctb_obs[4:0]**.

10.2.1 Control of CPU Bus Cycles

The CPU starts a bus cycle by driving an address and control information (write, byte masks) on the bus, and then asserting **cpu_mreq_l**. The Main state machine loops in its idle state, waiting for **cpu_mreq_l** to assert.

When **cpu_mreq_l** asserts, the Main state machine uses the high-order address lines to determine how the cycle will be completed. The possibilities are:

- SSRAM access. SSRAM accesses are handled by the Main state machine, with no help from external state machines. There are separate flows in the state machine for read and write cycles.
- I/O or ROM accesses. These accesses are performed by the IO state machine. The Main state machine asserts a handshake signal, **cta_do_io**, to the IO state machine in CTB. This causes the IO state machine to perform the cycle and to acknowledge the handshake with **ctb_io_ack**. The Main state machine loops waiting for **io_ack**. Whilst looping, it keeps **cta_do_io** asserted, and controls the data bus buffers. When it receives **io_ack**, it updates **cta_do_io**, latches the data in the data bus buffers (for reads) and terminates the beat by negating **cta_wait_l** to the CPU. For sequential cycles (which, in this address space, can only be ROM reads), **cta_do_io** will remain asserted and the IO state machine will perform a further cycle.
- DRAM accesses. These accesses are performed by the DRAM state machine. The interface between the Main state machine and the DRAM state machine uses handshakes **do_dram** and **dram_ack**, which behave in exactly the same way as the interface with the IO state machine. Since the DRAM state machine and the Main state machine are both in the same physical device (CTA), these handshakes are implemented as internal nodes and are not visible on the pins of CTA.

The DRAM state machine arbitrates between performing bus cycles for the CPU and performing refresh cycles for the RFRSH state machine. Refresh cycles always take priority. If a refresh cycle is in progress when the CPU asserts **do_dram**, then the DRAM state machine will simply ignore **do_dram** until the refresh cycle has completed. This process is invisible to the Main state machine, which simply loops waiting for **dram_ack**. In this case, **dram_ack** will arrive later than usual, because the refresh cycle must complete first.

A similar situation occurs when the CPU performs back-to-back (but non-sequential) cycles to the DRAM. When a DRAM cycle has completed, the DRAM state machine takes several clock cycles between delivering **dram_ack** to the CPU and returning to its idle loop. It is only sensitive to **do_dram** when it is in its idle loop. This ensures that back-to-back cycles cannot cause DRAM timing parameters like the RAS precharge period to be infringed. Section 6.7 shows which back-to-back cycle combinations are slowed down as a result of this.

10.2.2 Types of Cycles

The CPU can perform read, write and lock cycles. The read and write cycles can be non-sequential (single data beat) or sequential (multiple data beats).

A lock cycle is a read cycle followed by a write cycle. The **cpu_lock** signal is asserted throughout the pair of accesses. The lock cycle is intended to allow atomic access to a location. Since the SA-110 is the only device on the **ebsarm** that can initiate bus cycles †, the **cpu_lock** signal is ignored.

The control state machines support sequential and non-sequential read accesses to the SSRAM, the DRAM and the Flash/EPROM. They only support non-sequential accesses to the I/O devices.

The control state machines support sequential and non-sequential write accesses to the SSRAM, and the DRAM. They only support non-sequential accesses to the Flash and I/O devices.

10.2.3 Sub-Block Wrapping

The SA-110 performs wrapped accesses for some cache fetch cycles. These are performed as 8-beat sequential reads with **cpu_clf** (cache line fill) asserted. The assertion of **clf** indicates that the addresses may wrap during the sequential access.

The addresses for wrapped accesses are sequenced to provide the critical longword first. Rather than wrap around a modulo-8 address, the addresses are wrapped modulo-4. This wrapping order, called sub-block wrapping, is the optimum wrapping order pattern for interfacing to memories that have a burst size of 4.

Two examples of the (seven possible) address sequences are:

first address	0x0000.0001	0x0000.0007
	0x0000.0002	0x0000.0004
	0x0000.0003	0x0000.0005
	0x0000.0000	0x0000.0006
	0x0000.0005	0x0000.0003
	0x0000.0006	0x0000.0000
	0x0000.0007	0x0000.0001
last address	0x0000.0004	0x0000.0002

10.2.4 The Burst Counter

Consider the case where the CPU starts a sequential read at address 0x0000.000c from the SSRAM. If a 2-beat read is performed, the CPU will expect to receive data from locations 0x0000.000c and 0x0000.0010. However, the SSRAM is a burst device with a block size of 4. Its internal (2-bit) address counter will be loaded with 0b11 for the first access and will wrap to 0b00 for the second access. This corresponds to address 0x0000.0000.

The correct behavior in this case is for the Main state machine to terminate the burst after the first read, and to start a new burst at address 0x0000.0010.

† There is a somewhat esoteric exception to this statement. The Ethernet controller arbitrates for access to the Ethernet buffer memory. The CPU could read a location and change it simultaneously with the Ethernet controller writing the same location; even if the CPU were to perform a lock cycle, there is no way for the EBSA-110 control logic to guarantee atomic access to the Ethernet buffer memory. Correct sharing of data structures in the Ethernet buffer memory is architected by the Ethernet controller's buffer ownership protocols.

These boundary-crossing situations are detected by a 2-bit counter called **bcnt** (burst counter), implemented within CTA. This counter is used by the Main state machine for tracking progress during burst transactions to SSRAM and BEDO DRAM. The counter is controlled by the Main state machine and the count value is monitored by both the Main and the DRAM state machines.

Two control lines, **bcnt_ctl[1:0]**, are used to control the counter; they specify one of three operations:

- HOLD - leave the counter at its current value. This is the default operation, when neither control signal is asserted.
- DECREMENT - decrement the counter by 1. When the counter reaches zero, a decrement will cause it to wrap around to 3.
- LOAD_DEP - load the counter. The value loaded depends upon some other signals, explained below.

The counter is loaded at the start of an access, using LOAD_DEP. The **cpu_write** and **cpu_clf** signals determine what value is loaded by LOAD_DEP:

- **!cpu_write**, **!cpu_clf** - non-wrapped read at arbitrary address alignment. In this situation, LOAD_DEP loads the counter with the 1's complement of the low-order address lines. For example, if the low-order address is 00b, the address is aligned to a burst boundary and the burst size is 4, so the counter is loaded with 3. If the low-order address is 01b, the counter is loaded with 2, corresponding to a maximum allowable burst length of 3.
- **!cpu_write**, **cpu_clf** - wrapped cache line read at arbitrary address alignment *or* non-wrapped cache line read at block-aligned address. In either case, the burst size will be 4, so LOAD_DEP loads the counter with 3, regardless of the low-order address lines.
- **cpu_write**, **!cpu_clf** - non-wrapped write at arbitrary address alignment. In this situation, LOAD_DEP acts exactly like the **!cpu_write**, **!cpu_clf** state.
- **cpu_write**, **cpu_clf** - write with hint that address is block-aligned *and* that it is a full write (all byte masks will be asserted for all beats). This is used to optimize the BEDO DRAM write flow. Since the address is block-aligned, it is legitimate to load the counter to 3, so this behaves just like the **!cpu_write**, **cpu_clf** case.

As the transaction proceeds, the 'default' command is HOLD; the count value is unchanged. As each data beat occurs, a command of DECREMENT is issued, and the counter decrements towards zero.

If the counter reaches zero during a transaction, it is used as an indication that a new address needs to be loaded into the SSRAM or BEDO DRAM. The counter state is detected in the Main state machine (for SSRAM accesses) and in the DRAM state machine (for BEDO DRAM accesses). Once a transaction has started, it is never necessary to reload the counter. If it reaches zero, it is sufficient to decrement it once more, to 3. This is correct for both wrapped and non-wrapped transactions and has three advantages:

1. It makes the decision easier.
2. It makes the counter logic simpler (no FULL LOAD required).
3. It avoids having to sample the address lines, which are slow and may not meet the setup into the programmable logic.

The counter value changes on the clock after the control signals are issued. Using **cpu_clf** directly in the counter avoids the need to have it valid sooner.

The burst counter logic looks like this:

```
BCNT := (BCNT       & (BCNT_CTL == B_HOLD))           #
        ([!A3,!A2]  & (BCNT_CTL == B_DEP_LD) & !CLF ) #
        ([ 1 , 1 ]  & (BCNT_CTL == B_DEP_LD) & CLF )  #
        ((BCNT - 1) & (BCNT_CTL == B_DECR));
```

10.2.5 The Packer Address Counter

CTB contains a 2-bit counter called **paka** (packer address). This counter is used to generate the two low-order address lines for the Flash and EPROM. It is controlled by the IO state machine.

Two control lines, **paka_ctl[1:0]**, are used to control the counter; they specify one of four operations:

- HOLD - leave the counter at its current value. This is the default operation, when neither control signal is asserted.
- LD - load the counter with the value from two high-order address lines.
- INCR - increment the counter value. If the counter reaches 3, an increment will cause it to wrap back to 0.
- CLR - clear the counter value to 0.

The counter is used during Flash and EPROM reads, and during Flash writes. Whilst the IO state machine is in its idle state, the counter is speculatively held clear, by asserting CLR, in case the next cycle is a ROM read.

If a Flash write starts, LD is asserted as the sequence starts, and no further control of the counter is required. No packing facility is provided during Flash writes, and the two high-order address lines provided by LD contribute to form a full byte address for the write cycle.

If a Flash or EPROM read starts, the IO state machine generates 4 reads to incrementing addresses, in order to provide a 32-bit value to the CPU. The IO state machine loops in a sequence that reads a byte from the ROM and then increments the ROM address by asserting INCR. The two **paka** signals directly drive the ROM address lines. As successive values are read, they are latched into external buffers. The IO state machine monitors the **paka** count to determine when all four bytes have been read, and terminates the data beat. For a sequential read (such as a cache line fill), the IO state machine will continue to assert INCR so that the count wraps back to 0 in time for the next beat of the sequence.

The pack address logic looks like this:

```
PAKA := (PAKA       & (PAKA_CTL == PAKA_HOLD)) #
        ([0,0]      & (PAKA_CTL == PAKA_CLR))  #
        ([A23, A22] & (PAKA_CTL == PAKA_LD))   #
        ((PAKA + 1) & (PAKA_CTL == PAKA_INCR));
```

10.2.6 Accesses to 16-bit Peripherals

The Am79C961A Ethernet controller and VG468 PCMCIA controller are 16-bit peripherals and support byte and half-word accesses. These devices are designed for interfacing to PCs.

On a PC, byte/half-word accesses are controlled by the low-order address line (**sa0** input to the peripheral) and the signal **sbhe_l**. The possible combinations of these signals dynamically accommodates PCs with 8-bit expansion buses (the original IBM PC, the IBM XT and clones) and those with 16-bit expansion buses (IBM PC-AT and clones). The EBSA-110 I/O bus always mimics a 16-bit expansion bus, and the possible options are shown in Table 10–1.

Table 10–1 Byte/Half-Word Decode Using SA0, SBHE_L

R/W	A0	SBHE_L	D7:0	D15:8	Description
READ	0	1	Slave drives	Float	Low byte read
READ	1	0	Float	Slave drives	High byte read
READ	0	0	Slave drives	Slave drives	16-bit read
WRITE	0	1	CPU drives	Float	Low byte write
WRITE	1	0	Float	CPU drives	High byte write
WRITE	0	0	CPU drives	CPU drives	16-bit write

The behavior of **sa0** and **sbhe_l** shown here can be implemented directly by the SA-110, using its byte masks. **buf_be1_l** is connected to **sbhe_l** and **buf_be0_l** is connected to **sa0**.

A problem occurs in some peripherals (or some registers within peripherals) which are only designed to accommodate 8-bit I/O cycles. For these accesses, the peripheral expects to transfer data on the low-order byte lane, byte lane 0. If the register address is even, this will work correctly. However, if the register address is odd, the CPU will expect to transfer data on byte lane 1.

The EBSA-110 uses a trick to allow 8-bit cycles to odd addresses to transfer data on the low-order byte lane. Instead of using **cpu_be0_l** to drive **sa0** on the peripheral directly, a modified version of this signal is used:

```
BYTE_BE0 = CPU_BE0 & !CPU_BE2 & !CPU_BE3
```

The effect of this modification is that **cta_byte_be0_l** asserts normally for byte and half-word accesses, but is negated for longword accesses (normally all the byte enables would be asserted for longword accesses). The negation of **cta_byte_be0_l** leads to **sa0** being asserted, causing the peripheral to perceive an odd byte address.

10.2.7 Memory Map Switching After Reset

After reset, the ROM is decoded at address space 0, in order to provide the reset vector. The first CPU-initiated write operation causes the address map to switch, so that RAM is decoded at address 0.

This is achieved by 2 state bits within the CTB logic. These state bits are **prime_map** and **normal_map**.

Both of these state bits are asynchronously cleared at reset. When the first write operation starts, **prime_map** asserts, and remains asserted. When the first write operation ends (**cpu_wait_l** negates), **normal_map** asserts and remains asserted.

This two-stage process ensures that the address map does not switch part-way through the write cycle; it switches *after* the completion of the first write cycle.

10.2.8 BEDO DRAM Configuration Cycles

The BEDO DRAMs must be programmed to use a linear wrapping order, before any other accesses are performed. This is achieved using a write-CAS-before-RAS (WCBR) cycle. During this cycle, mode information is passed into the DRAM on the address inputs, and is latched by the assertion of **ras_l**. After the DRAM has been programmed, a CBR (with the DRAM's **we_l** input negated) must be used to take the DRAM out of program mode.

CBR and WCBR cycles are performed under software control. The **soft_dcbr** signal, which is controlled by the Soft register, is used to modify the behavior of the DRAM state machine. When **soft_dcbr** is asserted, all write accesses to the DRAM space will generate WCBR cycles, and all reads from the DRAM space will generate CBR cycles.

The DRAM state machine uses the normal refresh state flow to implement the CBR and WCBR cycles. This has the advantage that it avoids adding extra states to the state machine. A disadvantage is that normal refreshes *must* be disabled whilst **soft_dcbr** is asserted. At the exit of the refresh state flow, the assignment 'DRAM_ACK := SOFT_DCBR;' in a particular state causes **dram_ack** to assert during CPU-initiated cycles. If refresh was enabled whilst **soft_dcbr** was asserted, the assignment would cause a pulse on **dram_ack**. If this occurred whilst a CPU DRAM access was being held-off (by the refresh) the cycle would be terminated prematurely.

sel_col is asserted during a WCBR cycle, and so the mode information is passed into the DRAM using the *column* address lines, though it is latched into the DRAM by the assertion of **ras_l**.

10.2.9 Address Decoding

Section 3.1 describes how the different memory and I/O devices are decoded within the memory map. This section describes how the address decoding is achieved.

The address decoding function is split between the CTA and CTB control blocks.

CTA uses **a[31:30]** and **normal_map** (see Section 10.2.7) to divide the physical address space into four quadrants:

- DRAM quadrant. This moves according to the state of **normal_map**. After reset, **normal_map** is negated and the DRAM quadrant is inaccessible. Once the memory map has been switched (**normal_map** is asserted) the DRAM quadrant is decoded when **a[31:30]** = [0,0]. Accesses to the DRAM quadrant are decoded by the Main state machine which generates a handshake, **do_dram**, to the DRAM state machine. This handshake initiates a DRAM access.
- SSRAM quadrant. This is always decoded when **a[31:30]** = [0,1]. Accesses to the SSRAM quadrant are completed under the control of the Main state machine.

- ROM quadrant. This moves according to the state of **normal_map**. After reset, **normal_map** is negated and the ROM quadrant is decoded both when **a[31:30]** = [0,0] and when **a[31:30]** = [1,0]. Once the memory map has been switched (**normal_map** is asserted) the ROM quadrant is only decoded when **a[31:30]** = [1,0]. Accesses to the ROM quadrant are decoded by the Main state machine which generates a handshake, **do_io**, to the IO state machine. This handshake initiates a ROM access.
- IO quadrant. This is always decoded when **a[31:30]** = [1,1]. Accesses to the IO quadrant are decoded by the Main state machine which generates a handshake, **do_io**, to the IO state machine. This handshake initiates an I/O access.

Further decoding of the address space is distributed into the logic that controls the specific accesses.

10.2.9.1 Decoding Within the SSRAM Quadrant

There is no further decoding within the SSRAM quadrant. The SSRAM is multiply aliased throughout this region.

10.2.9.2 Decoding Within the DRAM Quadrant

CTA decodes 4 separate **ras_l** signals within this quadrant. The decode is a function of CPU address lines, DRAM SIMM size and type (EDO or BEDO). Because of the complexity of this decode, four intermediate **pipe_ras_l** nodes are generated. The DRAM state machine asserts the correct **ras_l** outputs by copying the states of the **pipe_ras_l** nodes to the **ras_l** outputs.

10.2.9.3 Decoding Within the ROM Quadrant

CTA decodes the ROM quadrant to generate chip selects for the EPROM, the FLASH and the ROM_D buffer.

This decode is a function of **a[30:29]** and the signal **lnk_eprom_boot_l**, which is set by a jumper on the board.

When the jumper is fitted, the system is intended to boot from EPROM and therefore the EPROM is decoded (**cta_cs_eprom_l** asserted) when **a[30:29]**=[0,0]. The Flash is decoded (**cta_cs_flash_l** asserted) when **a[30:29]**=[0,1].

When the jumper is removed (the default), the system is intended to boot from Flash and therefore the decoding is reversed.

The chip selects are only asserted when **a[30]**=[0]. This stops the ROMs from driving the IO_D bus during an I/O access †.

The chip selects are not qualified with **a[31]**. The ROMs can be decoded at **a[31]**=[0] and **a[31]**=[1], depending upon the state of **normal_map**. The decode of **a[31]** and **normal_map** is implicit in the assertion of **do_io** (which is a prerequisite for a ROM access) and therefore they do not need to qualify the chip selects.

cta_cs_anyrom_l is a chip select for the ROM_D buffer, and is asserted when either **cta_cs_eprom_l** is asserted or **cta_cs_flash_l** is asserted.

cta_cs_anyrom_l is also used as an input to CTB, where it is used to determine whether a CPU read cycle should perform an ISAIO/ISAMEM sequence or a ROM packing sequence.

† Actually, only **cta_cs_anyrom_l** needs to be qualified with **a[30]**=[0]. That is enough to prevent the ROM_D buffer from driving IO_D when **ctb_io_rdoe_l** is asserted.

These three chip selects are simply clocked decodes of CPU address lines; they are not qualified by any control signals.

The Flash write strobe is generated during all IO state machine write cycles in which **cta_cs_anyrom_l** is asserted. This means that all the appropriate address lines are implicitly decoded.

10.2.9.4 Decoding Within the IO Quadrant

The IO quadrant is itself subdivided into four areas:

- Abort space. This is decoded in CTA. Certain accesses in this space generate a **cta_abort** to the CPU. This space is decoded by **a[31:28]** = [X,1,0,X]. **a[31]** is [X] because an abort is only generated during an I/O access (qualified by **io_ack**). **a[30]** must be [1] to avoid generating aborts during ROM accesses (since these also generate an **io_ack**). **a[29:28]** are used to split the IO quadrant into four areas; the Abort space occupies the lower two of these areas.
- ISAMEM space. This is decoded in CTB by **a[29:28]** = [1,0]. The higher-order address lines are implicitly decoded since any ISAMEM cycle must be initiated by assertion of **cta_do_io**.
- ISAIO space. This is decoded in CTB by **a[29:28]** = [1,1]. The higher-order address lines are implicitly decoded since any ISAIO cycle must be initiated by assertion of **cta_do_io**.

  The ISAIO space is itself divided into two areas:

  - Self-decoding space. This is the subset of ISAIO space for which **a[25]**=[0]. Self-decoded I/O devices are inhibited from responding to addresses in which **a[25]**=[1] (refer to Section 3.2.9 and Section 10.1.14).
  - External-decoded space. This is the subset of ISAIO space for which **a[25]**=[1]. This space is used to access the PIT, and to access the Trickbox registers within CTB. All external-decoded devices are selected by decoding **cta_trick_a[2:0]**. **cta_trick_a[2:0]** is generated in CTA by using **a[25]**=[1] to qualify 3 other arbitrary address lines that are already used in CTA. Accesses to trickbox registers 1-7 in ISAIO space decode **a[25]** explicitly. Trickbox register 0 is an exception, since **cta_trick_a[2:0]**=[0,0,0] does not guarantee that **a[25]**=[1]. Trickbox register 0 space is used to access the PIT, and it is decoded in CTA, using **a[25:22]** = [1,0,0,0], to generate the **cta_cs_pit_l**.

10.3 Timing Analysis

Worst-case min/max timing analysis was performed on the EBSA-110 design using Chronology's TimingDesigner® tool. The source files and plots of this analysis are supplied as part of the design database.

10.4 Expanding the EBSA-110

The EBSA-110 can be expanded by adding a mezzanine card which plugs into the debug connectors.

These connectors supply all the signals required to interface logic to the I/O bus, in ISAMEM or ISAIO space. The signals include **rdy** and **zws_l**, so that the cycle length can be controlled.

Power is not supplied on these connectors, and so a separate power connector will be required from the system's power supply. The connectors do supply the 0V reference (ground return) for signals on the connectors.

The debug connectors include 5 unassigned signals from each of the two programmable logic parts. These are **cta_obs[4:0]** and **ctb_obs[4:0]**. By default, these signals are used to provide observability of internal state information. They could be reassigned to some other purpose.

The debug connectors also provide access to **usr_irq**, which is a spare interrupt reserved for use by an expansion board. It is an active-high signal and has an associated pull-down resistor.

The pinout of the debug connectors is shown in Section A.5.

10.5 The Printed Circuit Board

The EBSA-110 printed circuit board is a 6-layer controlled impedance board using 0.005" track and 0.005" gap routing rules. The mechanical drawing of the board shows the board's layer construction and dimensions. The mechanical drawing is supplied as a PostScript® file as part of the design database, see Appendix C.

When the board was routed, all nets were daisy-chained except for the data and address buses, which were bussed. The clocks and strobes were hand-routed first, to minimize the etch lengths.

Many of the clock and strobe nets include series resistors. These act as source terminations to reduce ringing on the signals. In all cases, the series resistors are placed so that the etch from the signal's driver to the series resistor (nets with the **un_** prefix) is as short as possible.

10.6 Design Improvements

This section describes some areas where the performance or implementation efficiency of the design could be improved. These opportunities arise with the benefit of hindsight.

- Worst-case timing analysis of the design with characterized CPU timings indicate that it is possible to use slower pipelined SSRAMs (the board currently uses -6 parts, but -8 parts would work correctly).
- Worst-case timing analysis of the design with characterized CPU timings indicate that it is possible to use non-pipelined SSRAMs (12ns or faster). This would allow one stall cycle to be saved on some SSRAM reads, but would require modification to the main state machine.
- The LVT data bus buffering could be implemented using cheaper unlatched parts; the latches were necessary in an earlier version of the design but are not necessary in the final design.
- Some DRAM accesses could be speeded up by having CAS negate on the falling clock (so that CAS is asserted for half a clock period, rather than a whole clock period). This would allow CAS to assert on successive clock edges; at the moment, CAS can only assert on alternate clock edges.

11
Simulation Waveforms

This chapter provides cycle-by-cycle descriptions of a number of simulation waveforms which show the major functions of all of the state machines and control logic on the EBSA-110.

These descriptions are intended to be read sequentially - detail presented in early sections is not repeated in later sections.

All of the waveforms in this section are included (in PostScript format) in the design database. When printed separately, they will be slightly larger than they are reproduced here (since these pages have margins). Each of them can be reproduced in a simulation environment, by running the specified test scripts on the design. The whole set can be produced using the simulation script 'do_specwave.cmd'.

All of the simulations show a system clock with a period of 18.75ns.

11.1 automap

This waveform, shown in Figure 11–1, was produced using the simulation script 'do_automap.cmd'.

This simulation shows the way in which the address map decoding changes after reset. After reset, the EPROM/Flash is decoded in the first and the third quadrant of the physical address space (base addresses 0, 0x8000.0000). The first write performed by the CPU (address is don't care) causes the address map to change so that the EPROM/Flash is no longer decoded in the first quadrant; the DRAM is decoded there.

The waveform shows the sequence:

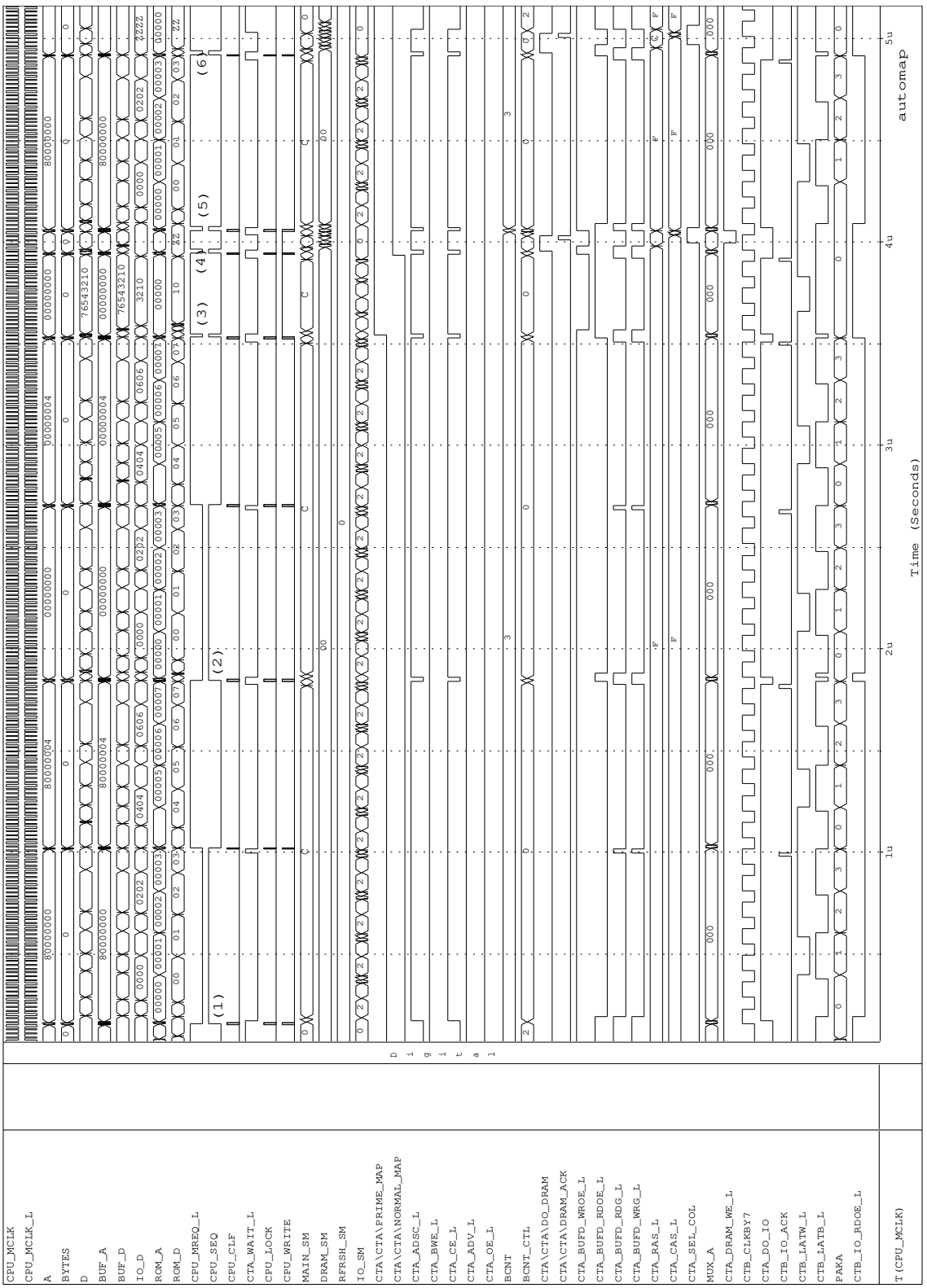
1. 2-beat sequential EPROM read; shows that EPROM is decoded at address 0x8000.0000 (the EPROM packing sequence is described in Section 11.15).
2. 2-beat sequential EPROM read; this shows that EPROM is also decoded at address 0.
3. Write to address 0 (performed as a write to the EPROM).

   This write causes the address map to change. The write does *not* affect the contents of the DRAM.

   Notice that **prime_map** asserts at the start of the write cycle and that the asserted state of **prime_map** causes **normal_map** to assert at the end of write cycle (when **cta_wait_l** negates). The assertion of **normal_map** causes the address map decoding to change.
4. Write to address 0 (performed as a write to DRAM). This shows that the address map has now changed for write cycles.

Simulation Waveforms
11.1 automap

Figure 11–1 automap



5. Write to EPROM at address 0x8000.0000; this shows that the EPROM is still decoded in high address space.
6. Read from address 0 (performed as a read from DRAM). This shows that the address map has now changed for read cycles.

11.2 ss_wcrd

This waveform, shown in Figure 11–2, was produced using the simulation script 'do_ss_wcrd.cmd'.

This simulation shows an SSRAM worst-case read sequence. It performs a 4-beat sequential read starting at address 0x4000.0008. Reads are performed from addresses 0x4000.0008, 0x4000.000c, 0x4000.0010 and 0x4000.0014. The first two locations are in one SSRAM burst block and the second two locations are in the next SSRAM burst block. Therefore, the Main state machine must cross an SSRAM block boundary during the access.

The waveform shows the sequence:

1. The first address (0x4000.0008) is loaded into the SSRAM. It is the address of the third location in a block. The burst counter, **bcnt**, loads the value 1, to show that there is 1 more piece of data available in this block. The **bcnt** value is the 1's complement of the low-order CPU address lines.
2. The state machine introduces 1 stall cycle to account for the access latency of the SSRAM.
3. Two beats of data are returned to the CPU, back-to-back (0x8899.aabb, 0xccdd.eeff) on adjacent clock cycles.
4. **bcnt** reaches 0, forcing a new address (0x4000.0010) to be loaded into the SSRAM. The access address is now aligned to a block boundary, so **bcnt** is decremented and wraps back to 3.
5. The Main state machine introduces 2 stall cycles; the first allows the address to be latched in, the second accounts for the access latency of the SSRAM.
6. Two more beats of data are returned to the CPU, back-to-back (0x1122.3344, 0x0011.2233).
7. The cycle ends because **cpu_mreq_l** negates, and the Main state machine goes back to state 0 (idle).

11.3 ss_wcwr

This waveform, shown in Figure 11–3, was produced using the simulation script 'do_ss_wcwr.cmd'.

This simulation shows an SSRAM worst-case write sequence. This is a 4-beat sequential write across an SSRAM block boundary; the write equivalent of ss_wcrd.

Simulation Waveforms
11.3 ss_wcwr

Figure 11–2 ss_wcrd

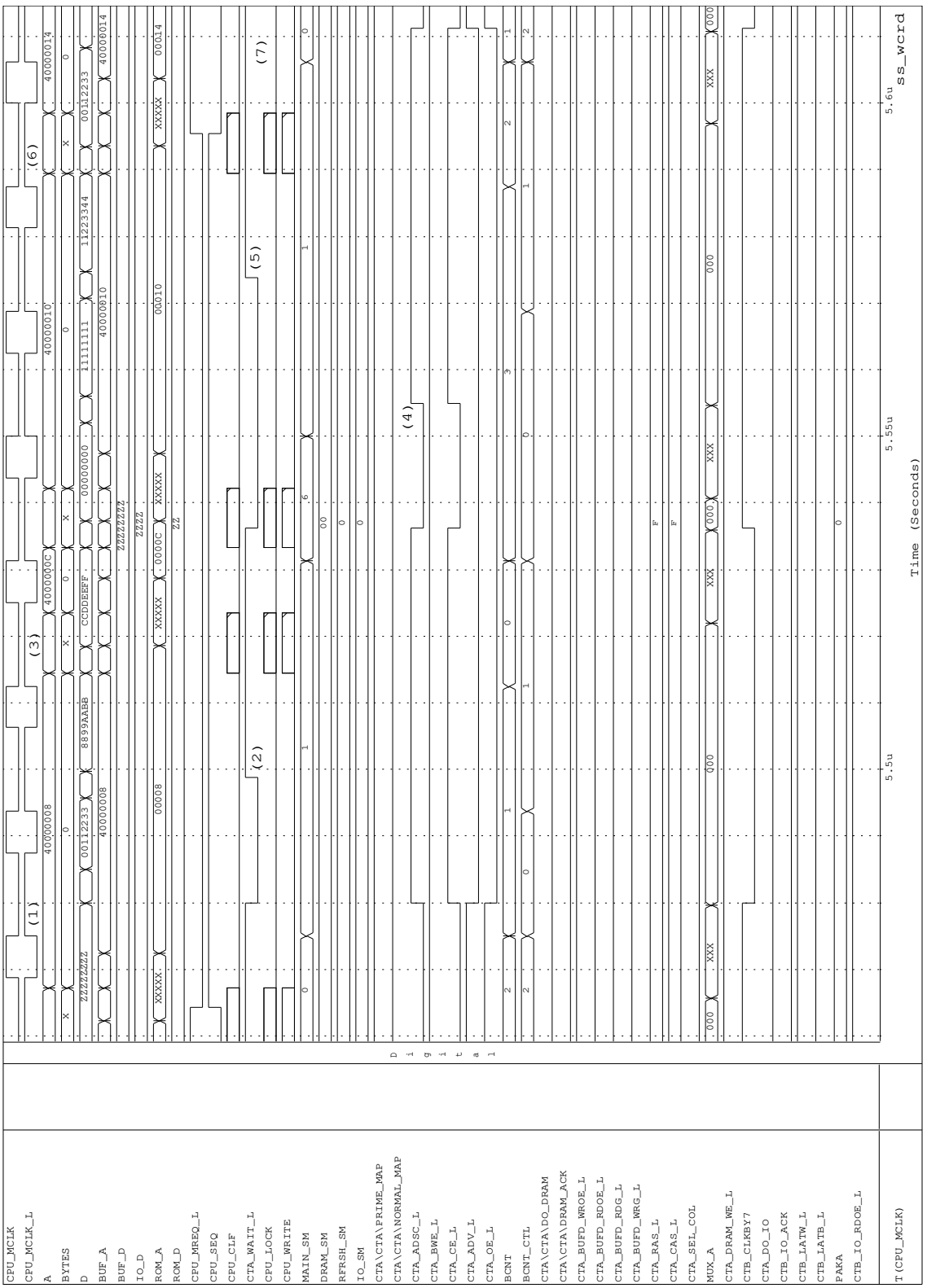
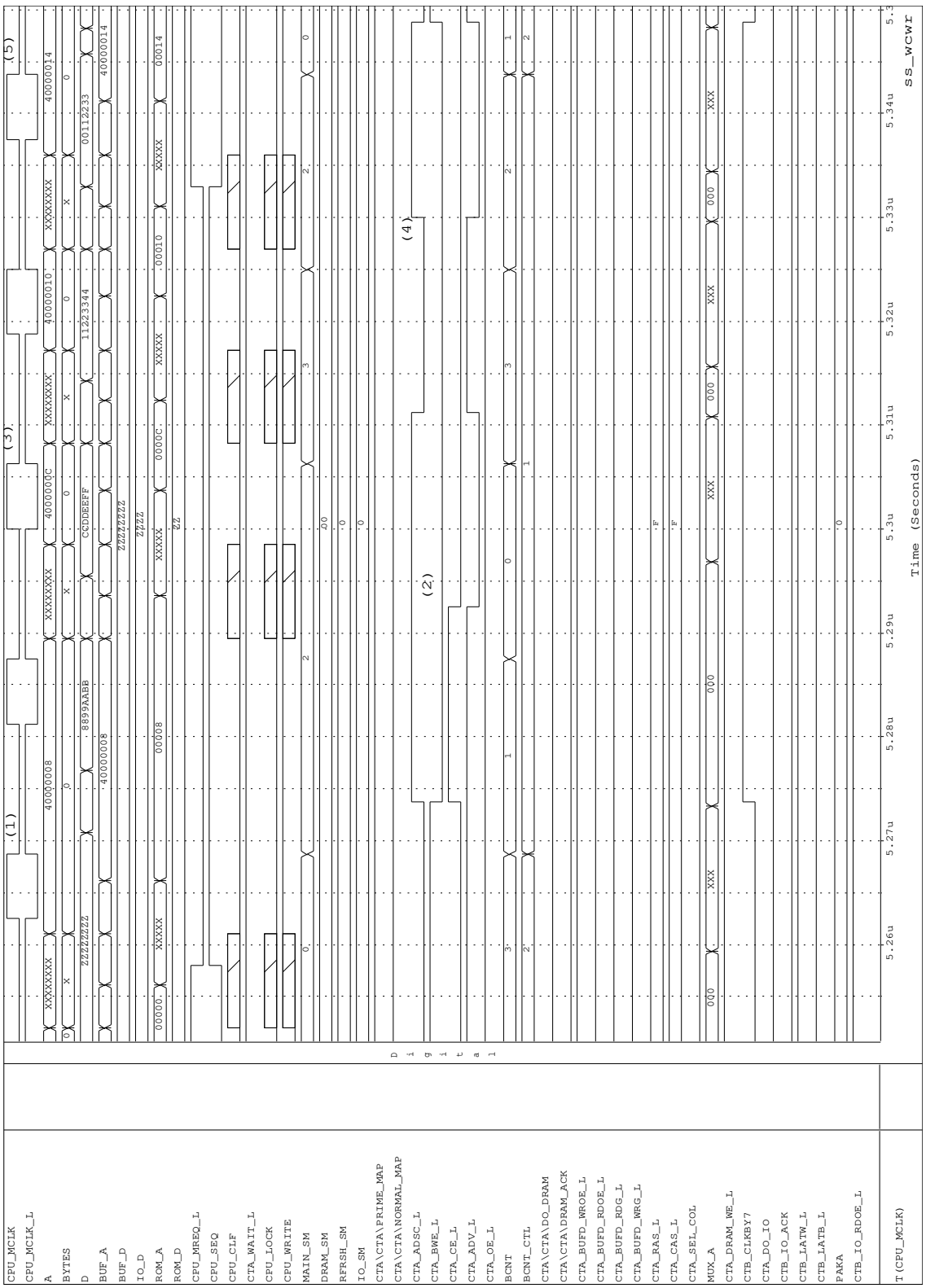


Figure 11–3 ss_wcwr



The waveform shows the sequence:

1. The first address is loaded into the SSRAM, and **bcnt** is loaded as before. At this stage, the cycle is identical to the read cycle, except that **cta_wait_l** is *not* asserted after the address is latched.
2. In the cycle after the address has been latched, the SSRAM read cycle is converted into a write cycle by the assertion of **cta_bwe_l** to the SSRAM. The first write data (0x8899.aabb) is latched.

   Notice that the address is valid at the time that the write data is latched into the SSRAM. The address itself is not required, but the byte masks (which have the same timing as the address) *are* required; they drive the SSRAMs directly (asynchronously; the synchronous write enable, **cta_bwe_l**, qualifies them).
3. In the next cycle, the second write data (0xccdd.eeff) is clocked into the SSRAM. At this time, **bcnt** has reached 0, so a new address must be loaded into the SSRAMs for the next beat of the write.
4. A new address (0x4000.0010) is loaded into the SSRAM, simultaneously with the third write data (0x1122.3344). This uses a different combination of control signals to the SSRAMs. (At the start of the cycle, the address was loaded with **cta_bwe_l** negated, and no write data supplied. This time, the address is loaded with **cta_bwe_l** asserted, and this allows write data to be supplied at the same time.)
5. The fourth write data (0x0011.2233) is loaded into the SSRAM.

Notice that the whole cycle occurs without the addition of any stall cycles (**cta_wait_l** is never asserted).

11.4 ss_rdwrap

This waveform, shown in Figure 11–4, was produced using the simulation script ‘do_ss_rdwrap.cmd’.

This simulation shows a CPU cache block fill from SSRAM. This is an 8-beat sequential read and shows the sub-block wrapping performed by the CPU.

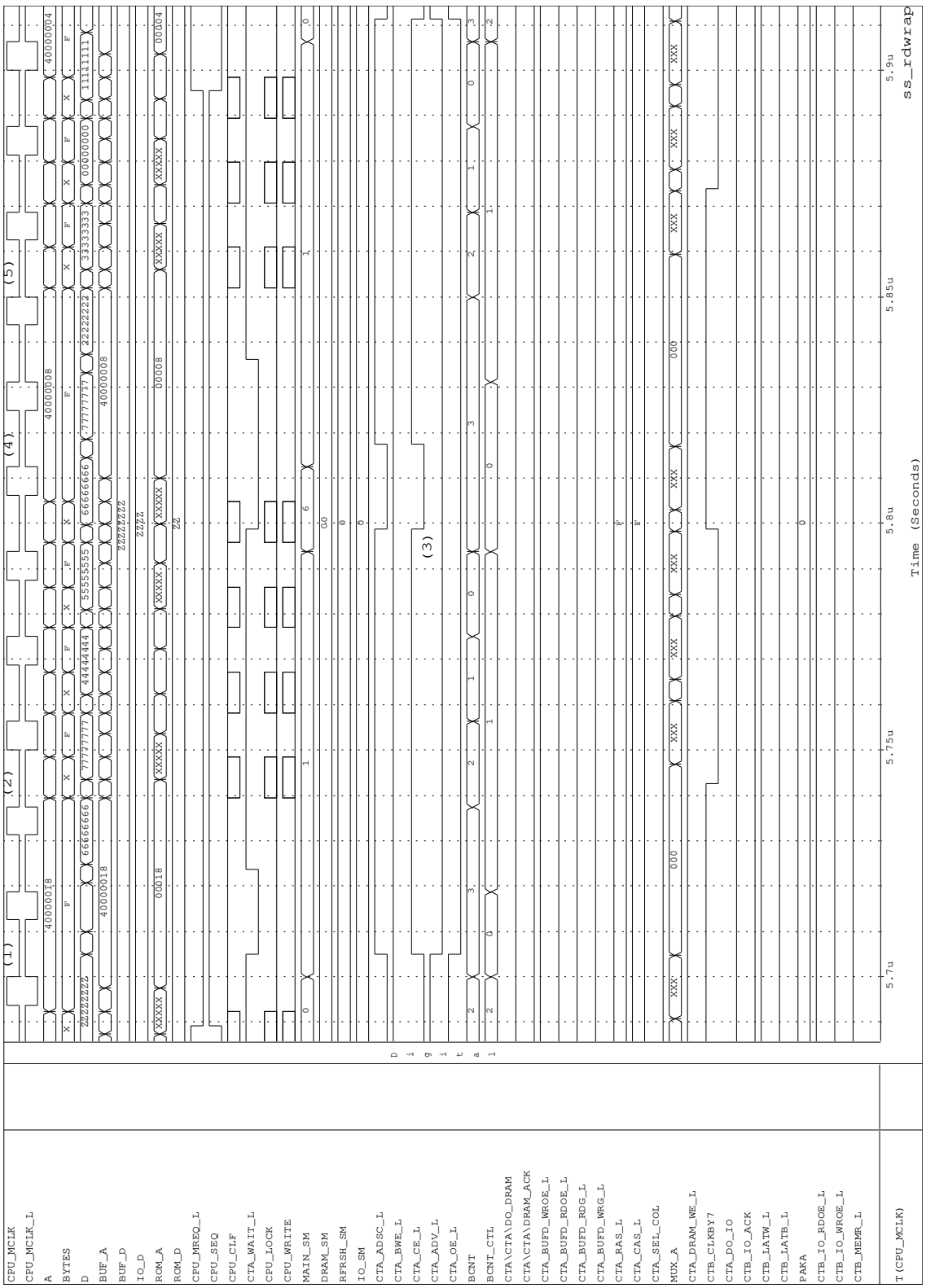
The CPU cache block size corresponds to eight 32-bit reads on the bus. Since burst memories (both SSRAMs and BEDO DRAMs) have a block size of 4, the CPU presents the addresses wrapped to a block size of 4. For example, if the first address was 1, the address sequence will be: 1,2,3,0,5,6,7,4. Notice that the second block starts with the same offset into a block as the first.

The CPU distinguishes a cache read (the only time that a sequential cycle has non-sequential addresses) by the assertion of **cpu_clf** (Cache Line Fill). The control logic uses this signal to force **bcnt** to load to 3, irrespective of the value of the low-order address lines.

The waveform shows the sequence:

1. The read starts with an SSRAM address load of 0x4000.0018, and a stall cycle is introduced, as before.

Figure 11–4 ss_rdwrap



2. All four locations in the block are read, with no further stall cycles being introduced. The addresses are 0x4000.001c, 0x4000.0010, 0x4000.0014. The read data is 0x6666.6666, 0x7777.7777, 0x4444.4444 and 0x5555.5555. Because of the magnification of the waveform, the address values cannot be seen, but it is clear from the data values that non-sequential locations are being accessed.
3. **bcnt** reaches 0 and is decremented to 3.
4. Two stall cycles are introduced; the first is needed to get the CPU address stable to meet the setup time into the SSRAM. The second accounts for the access latency of the SSRAM. The new address, 0x4000.0008, is loaded into the SSRAM.
5. The next four locations in the block are read, as before, with no further stall cycles being introduced. The addresses are 0x4000.000c, 0x4000.0000, 0x4000.0004. The read data is 0x2222.2222, 0x3333.3333, 0x0000.0000 and 0x1111.1111.

Notice that **cpu_clf** remains asserted throughout the cycle, and that a total of three stall cycles are introduced.

11.5 ss_rdall

This waveform, shown in Figure 11–5, was produced using the simulation script 'do_ss_rdall.cmd'.

This simulation shows a number of read sequences that exercise all the transitions in the read path of the Main state machine. In particular, it shows all the **bcnt** counter decrement/load transitions.

Although the scale of the waveform does not allow the individual CPU address and data sequences to be identified clearly, it does show all the Main state machine transitions and control of the **bcnt** counter.

The waveform shows the sequence:

1. 9-beat sequential read starting at the last address in a block. This causes three address loads to the SSRAM. One data beat is read at the first address, and four data beats are read at the second and third addresses.

   In the Main state machine, this causes the state transitions:

   ```
   hidle-hrd1-hrd1a-hrd2, hrd1-hrd2-hrd1-hidle
   ```

2. 2-beat sequential read starting at the last address in a block. This causes 2 address loads to the SSRAM. One data beat is read at each address.

   In the Main state machine, this causes the state transitions:

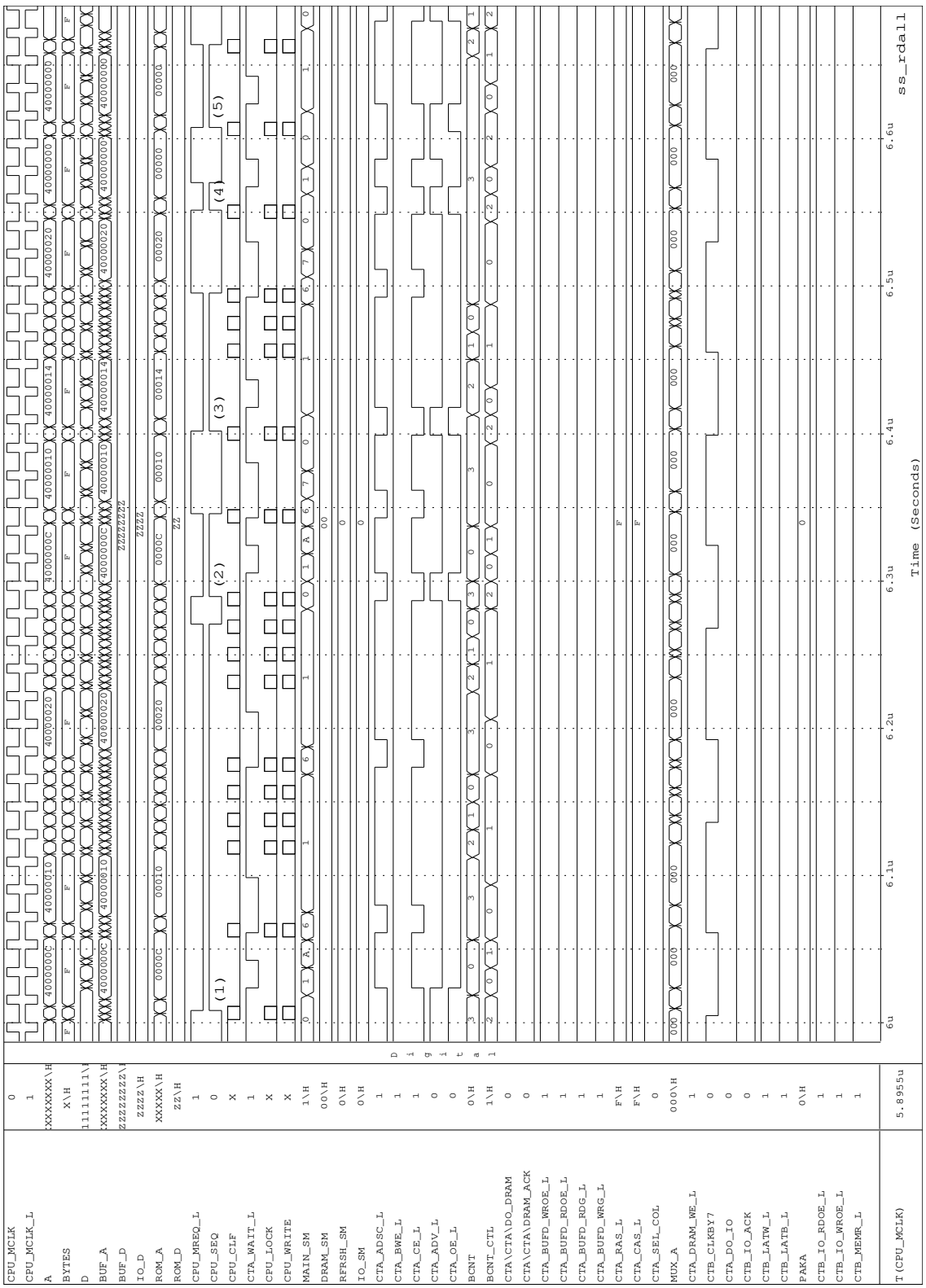
   ```
   hidle-hrd1-hrd1a-hrd2-hrd3-hidle
   ```

3. 4-beat sequential read starting at the second address within a block. This causes 2 address loads to the SSRAM. One data beat is read at the first address and three data beats are read at the second address.

   In the Main state machine, this causes the state transitions:

   ```
   hidle-hrd1-hrd2-hrd3-hidle
   ```

Figure 11–5 ss_rdall



4. 1-beat sequential read starting at the first address in a block. This causes a single address load.

   In the Main state machine, this causes the state transitions:

   ```
   hidle-hrd1-hidle
   ```

5. 2-beat sequential read at the first address in a block. This causes a single address load.

   In the Main state machine, this causes the state transitions:

   ```
   hidle-hrd1-hidle
   ```

11.6 ed_wcrd

This waveform, shown in Figure 11–6, was produced using the simulation script 'do_ed_wcrd.cmd'.

This simulation shows an EDO DRAM page-mode read sequence. This performs a 4-beat sequential read starting at address 0x0000.0008. Reads are performed from addresses 0x0000.0008, 0x0000.000c, 0x0000.0010 and 0x0000.0014. Since the EDO DRAM is not a burst device, each read cycle to the DRAM is performed individually. **ras_l** remains asserted throughout the access since all the addresses within the sequential read must be on the same DRAM page.

The waveform shows the sequence:

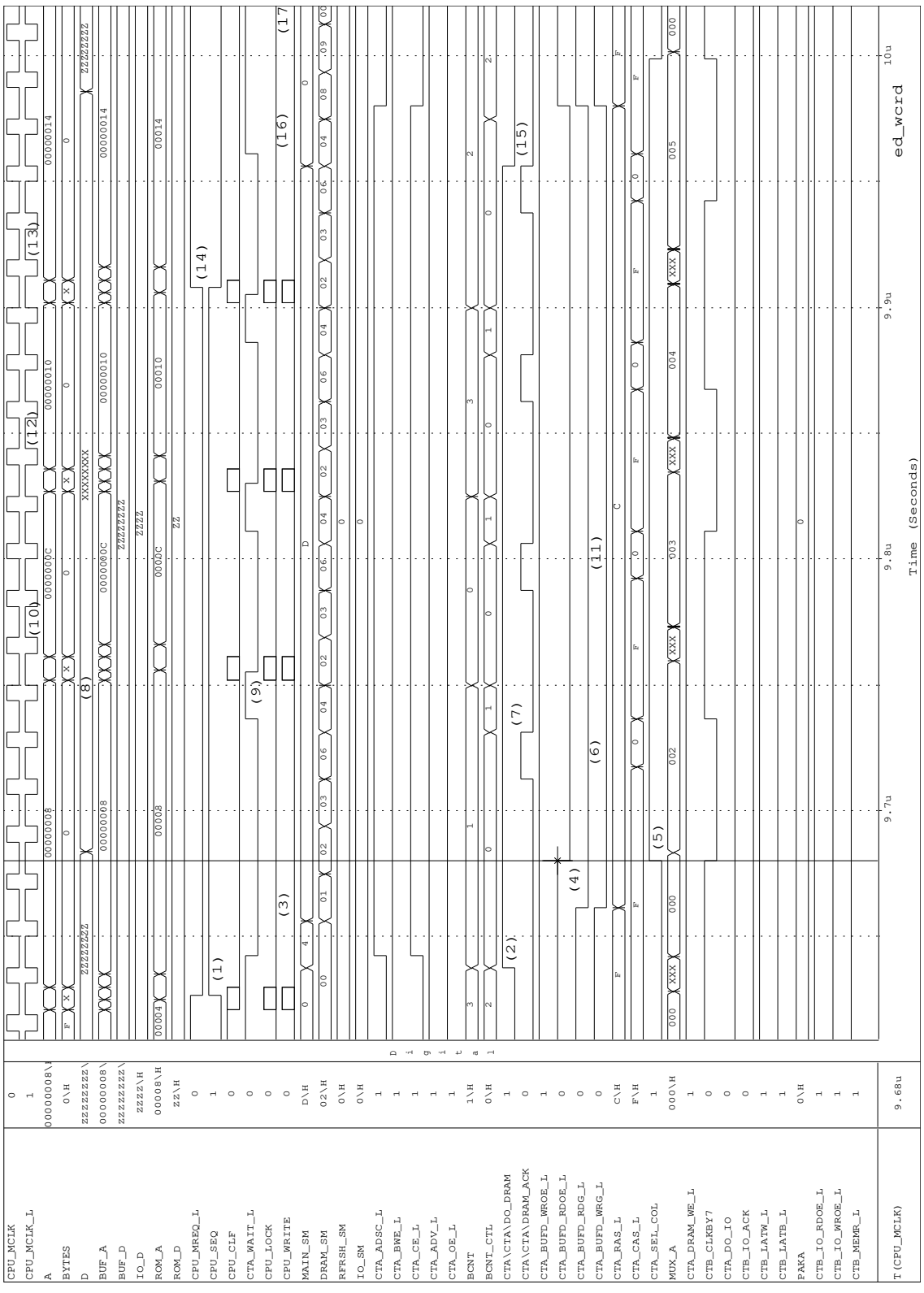
1. The CPU asserts **cpu_mreq_l** to start the cycle.
2. The Main state machine decodes a DRAM access and asserts **do_dram** to the DRAM state machine.
3. The DRAM state machine moves out of its idle state.
4. **cta_ras_l** asserts to the appropriate DRAM bank, latching the row address (shown as **mux_a** on the waveform).
5. **cta_sel_col** asserts to route the column address to the DRAM.
6. **cta_cas[3:0]_l** assert. For a read cycle, all four **cas_l** signals are always asserted. Since this is an EDO DRAM, **cas_** is only asserted for a single cycle, to allow the **cas_l** precharge to start as soon as possible.
7. The DRAM state machine generates **dram_ack** as an indication to the Main state machine that the access has completed. The Main state machine leaves **do_dram** asserted, because **cpu_mreq_l** is still asserted. The continued assertion of **do_dram** keeps the DRAM state machine in a loop, with **ras_l** asserted.
8. DRAM read data has propagated through the buffers to the CPU data bus.

   Note

   These simulations do not include DRAM models, so the read data is never shown.

9. The Main state machine negates **cta_wait_l** for one cycle, so that the CPU will sample the read data.

Figure 11–6 ed_wcrd



10. The CPU generates the address for the next beat of the read cycle.
11. The DRAM state machine generates **cta_cas[3:0]_l** to latch the new address. The rest of the read access proceeds as before.
12. The CPU generates the address for the third beat of the read cycle.
13. The CPU generates the address for the fourth beat of the read cycle.
14. The CPU negates **cpu_mreq_l** because the current address is the final beat in the sequential access.
15. When the Main state machine samples **dram_ack** asserted, it negates **do_dram** because **cpu_mreq_l** is now negated.
16. Once the Main state machine has negated **cta_wait_l** for the final data beat, it goes back to its idle state.
17. The DRAM state machine cycles through an additional state sequence before returning to its idle state. This prevents it from responding to a new request from the Main state machine until it has satisfied the DRAM **ras_l** precharge requirement.

This sequence takes 356.25ns (the same as the time for the BEDO), timed from the clock edge on which **cpu_mreq_l** asserts to the clock edge on which the CPU detects the negation of the final **cta_wait_l**.

11.7 ed_wcwr

This waveform, shown in Figure 11–7, was produced using the simulation script 'do_ed_wcwr.cmd'.

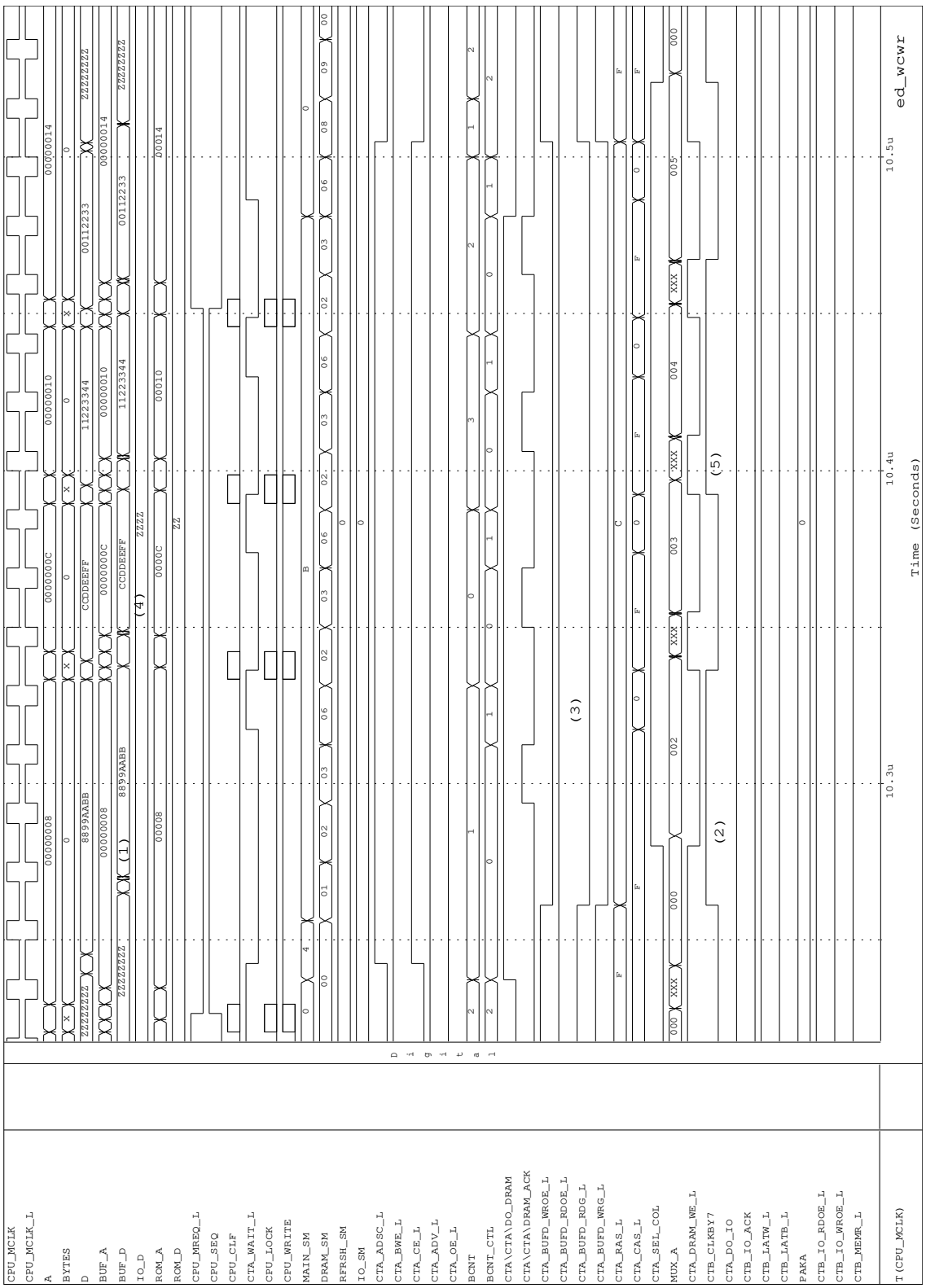
This simulation shows an EDO DRAM page-mode write sequence. This performs a 4-beat sequential write starting at address 0x0000.0008, and is the write equivalent of ed_wcrd.

The waveform is very similar to ed_wcrd. The notable differences are:

1. The write data for the first beat of the write becomes valid on the buffered data bus.
2. At the DRAM, a write is indicated by the assertion of **cta_dram_we_l**.
3. When **cas_l** asserts, the byte enables from the CPU determine which of **cta_cas[3:0]_l** assert. This provides byte-resolution on writes.
4. The first data beat is terminated, and the CPU drives write data for the second beat of the write.
5. During a sequential write, **cta_dram_we_l** negates for a single cycle as the address transitions across an INT16 boundary. This is a side-effect of the BEDO write caused by the BEDO write state machine flow. It does not affect EDO writes because **cta_dram_we_l** has always asserted again before the next assertion of **cas_l**.

Notice that the write cycle time is shorter than the read cycle time. The whole sequence takes 281.25ns, which is identical to the BEDO write timing, bw_wcwr.

Figure 11–7 ed_wcwr



11.8 ed_rdwrap

This waveform, shown in Figure 11–8, was produced using the simulation script 'do_ed_rdwrap.cmd'.

This simulation shows a CPU cache block fill from EDO DRAM. This is an 8-beat sequential read, like ss_rdwrap, and shows the sub-block wrapping performed by the CPU.

The sequence shown in the waveform is similar to ed_wcrd, except that a longer sequential access is performed.

This sequence takes 656.25ns (compared to 506.25ns for the BEDO).

11.9 bd_wcrd

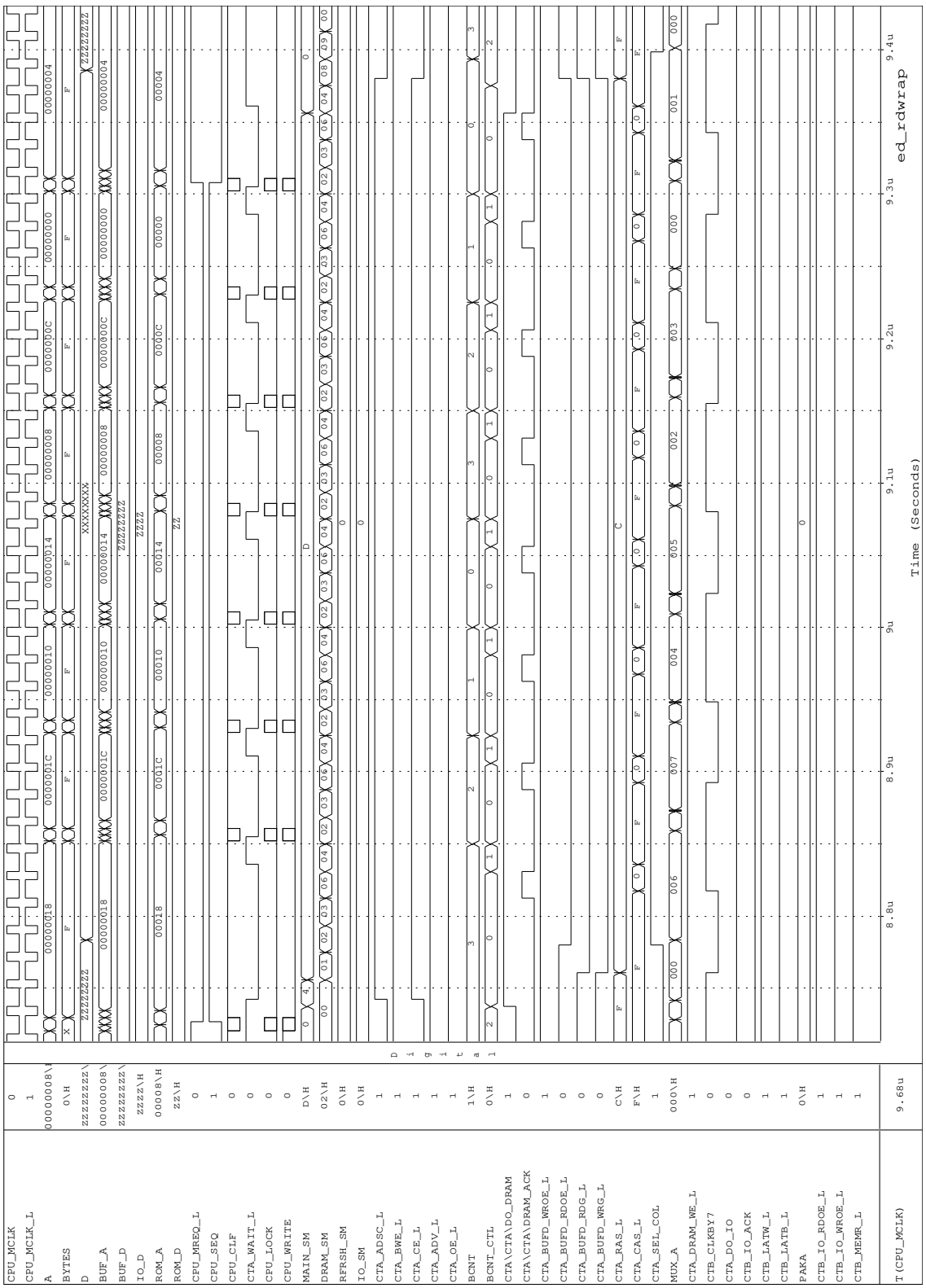
This waveform, shown in Figure 11–9, was produced using the simulation script 'do_bd_wcrd.cmd'.

This simulation shows a BEDO DRAM worst-case read sequence. This performs a 4-beat sequential read starting at address 0x0000.0008. Reads are performed from addresses 0x0000.0008, 0x0000.000c, 0x0000.0010 and 0x0000.0014. The first two locations are in one BEDO burst block and the second two locations are in the next BEDO burst block. Therefore, the DRAM state machine must cross a BEDO DRAM block boundary during the access.

The waveform shows the sequence:

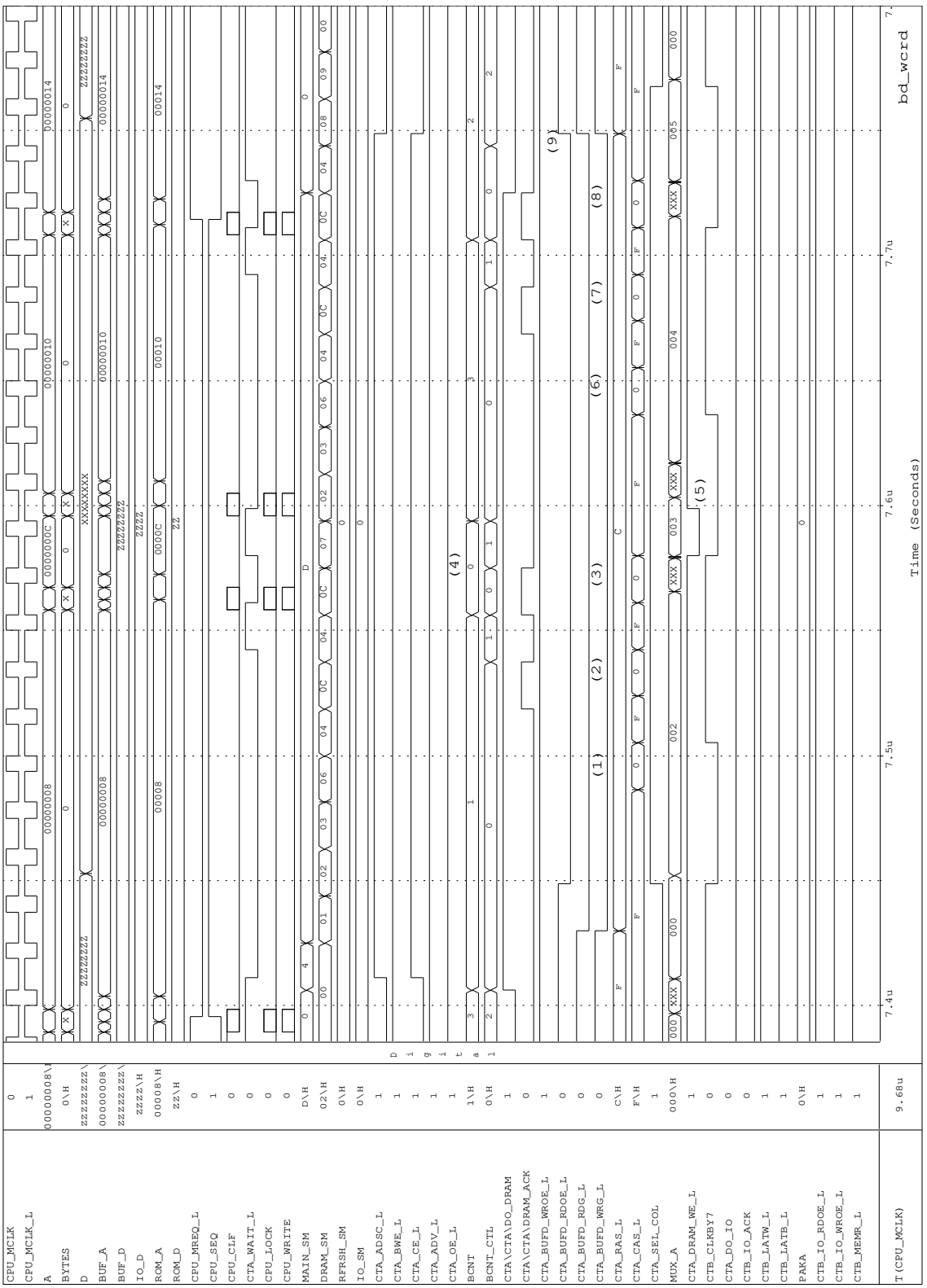
1. The sequence starts in the same way as an EDO DRAM read; up until the assertion of **cas_l**, the sequences are identical. For the BEDO DRAM, the first **cas_l** does not return read data; it only latches the column address. For this 4-beat sequential read, there are 6 **cas_l** pulses.
2. The second assertion of **cas_l** returns the first beat of read data. The column address is X (don't care) when **cas** asserts, since the read address is determined by the internal burst counter.
3. The third assertion of **cas_l** returns the second beat of data. Once again, the column address is X.
4. The burst counter reaches 0. This signals that, if another read were to be done from the BEDO DRAM, the BEDO DRAM's burst address counter would wrap (in this example, it would wrap to address 0x0000.0000).
5. Since this is *not* a wrapped access, it requires data from sequential addresses. Therefore, a burst count of 0 causes the DRAM state machine to terminate the current burst by toggling (asserting, in this case) **cta_dram_we_l**. Once the burst has been terminated, the next **cas_l** will latch a new column address.
6. The fourth assertion of **cas_l** latches a new column address. Stall cycles are introduced in the CPU cycle so that the address is stable for long enough to meet the BEDO DRAM setup time.
7. The fifth assertion of **cas_l** returns the third beat of data.
8. The sixth assertion of **cas_l** returns the final beat of data.

Figure 11–8 ed_rdwrap



Simulation Waveforms
11.9 bd_wcrd

Figure 11–9 bd_wcrd



9. **ras_l** negates and the DRAM precharge starts.

This sequence takes 356.25ns.

11.10 bd_wcwr

This waveform, shown in Figure 11–10, was produced using the simulation script 'do_bd_wcwr.cmd'.

This simulation shows a BEDO DRAM page-mode write sequence. This performs a 4-beat sequential write starting at address 0x0000.0008, and is the write equivalent of bd_wcrd.

Although this waveform is identical to the waveform for the ed_wcwr sequence, there are important differences in what is happening.

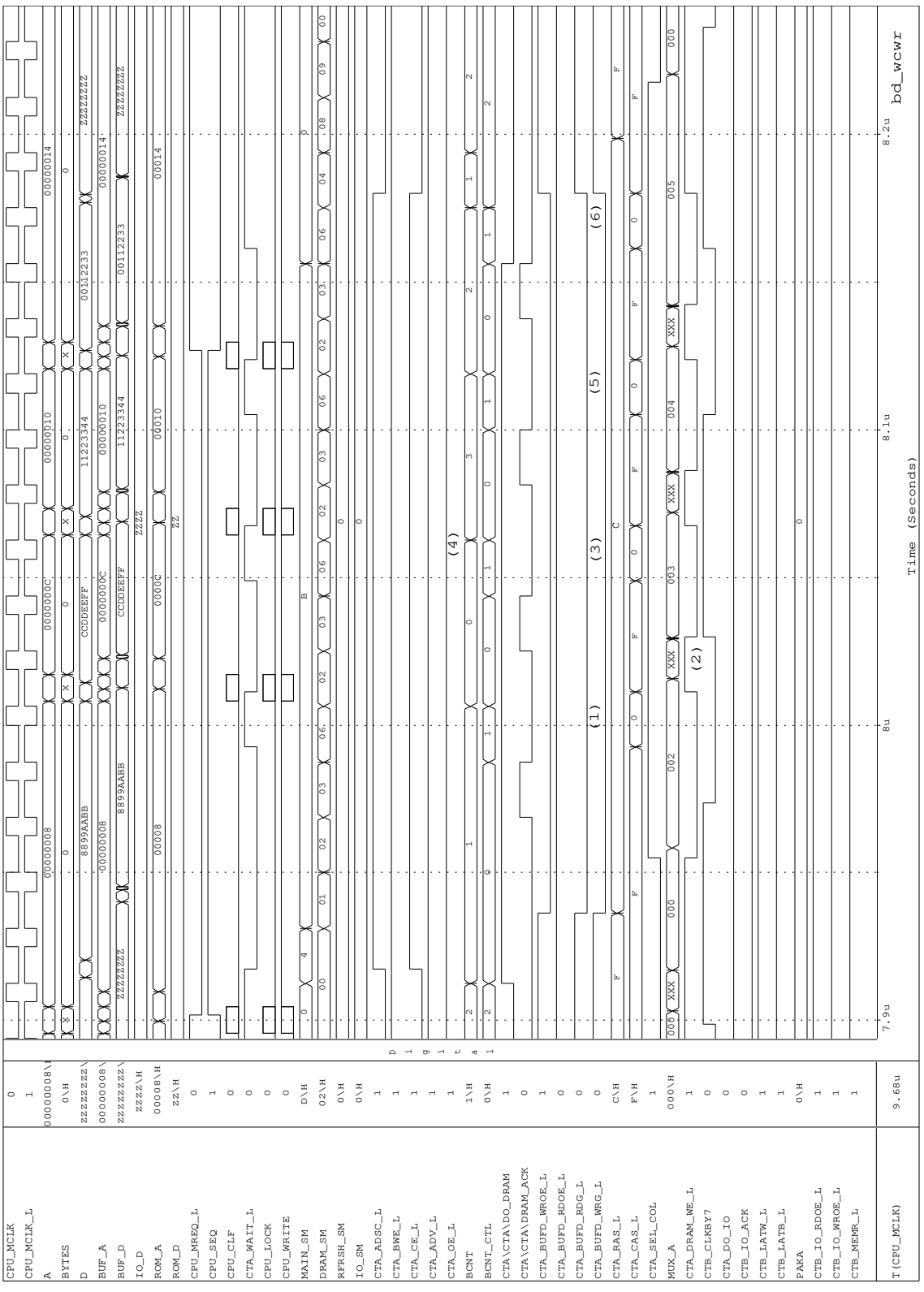
The waveform shows the sequence:

1. When **cas_l** asserts, the address for the burst access is latched into the BEDO DRAM. The byte enables from the CPU (which have the same timing as the address lines) are valid and meet the setup time into the DRAM state machine logic. These byte enables determine which of **cta_cas[3:0]_l** are asserted. The first beat of write data from the CPU is latched into the BEDO DRAM on this assertion of **cas_l**.
2. If the SA-110 has its write buffer enabled, random writes can merge in the write buffer. Therefore, the first beat of this sequential cycle could have had any combination of byte enables asserted (except none). For any byte lane that has **cas_l** asserted, the internal burst counter will increment to the next address in the block. However, for byte lanes that did not have **cas_l** asserted, the address counter will not increment. This would lead to data for subsequent beats being written to the wrong addresses. Therefore, the burst is unconditionally aborted at this point, by negating **cta_dram_we_l** †. An alternative would be to terminate the burst conditionally after any data beat in which one or more byte enables were negated. In some systems, this alternative would provide a performance advantage (by avoiding the stall cycles that aborting the burst necessitates). In the EBSA-110, this alternative method does not have any performance advantage because stalls must be inserted in order to acquire the byte enable information, as will be described below.
3. When **cas_l** asserts for the second time, the second beat of write data is latched into the BEDO DRAM. However, in order to determine which **cas_l** signals are to be asserted, the DRAM state machine requires the byte enables from the CPU to be valid. The byte enables have the same timing as the address, and 2 stall cycles are inserted so that the byte enables meet the setup time into the DRAM state machine.

† As an example of what would happen if the burst was not aborted, consider this sequence: Addresses 0x0-0xb contain 0. The CPU peforms a sequential write which is made up of a 32-bit store to address 0x0 (data 0xaaaa.aaaa), an 8-bit store to address 0x3 (data 0xbb) and a 32-bit store to address 0x8 (data 0xcccc.cccc). The final contents of the three longwords should be 0xaaaa.aaaa, 0x0000.00bb, 0xcccc.cccc, but is actually 0xaaaa.aaaa, 0xcccc.ccbb, 0x0000.00cc.

Simulation Waveforms
11.10 bd_wcwr

Figure 11–10 bd_wcwr



Even if we had been able to use the internal burst counter address at this point (by performing the optimization described above and not terminating the burst) we still would have had to insert stalls; getting valid byte masks requires as many stalls as getting a valid address would.

The disappointing end result is that writes proceed at the same rate as they do for EDO DRAM writes.

4. The burst counter reaches 0. If the burst was being terminated conditionally (which is not the case in the EBSA-110 design) the fact that the burst counter had reached 0 would be another factor that must cause the current burst to be aborted. In the EBSA-110 design, the state of the burst counter is irrelevant during these writes.
5. The third assertion of **cas_l** latches the third beat of write data, with its address.
6. The fourth assertion of **cas_l** latches the final beat of write data, with its address.

The end result is that writes cannot take advantage of the increased performance offered by BEDO DRAMs. The exception to this is the cache block evict sequence, shown in Section 11.12.

This sequence takes 281.25ns.

11.11 bd_rdwrap

This waveform, shown in Figure 11–11, was produced using the simulation script 'do_bd_rdwrap.cmd'.

This simulation is a CPU cache block fill from BEDO DRAM. It is an 8-beat sequential read that shows the sub-block wrapping performed by the CPU.

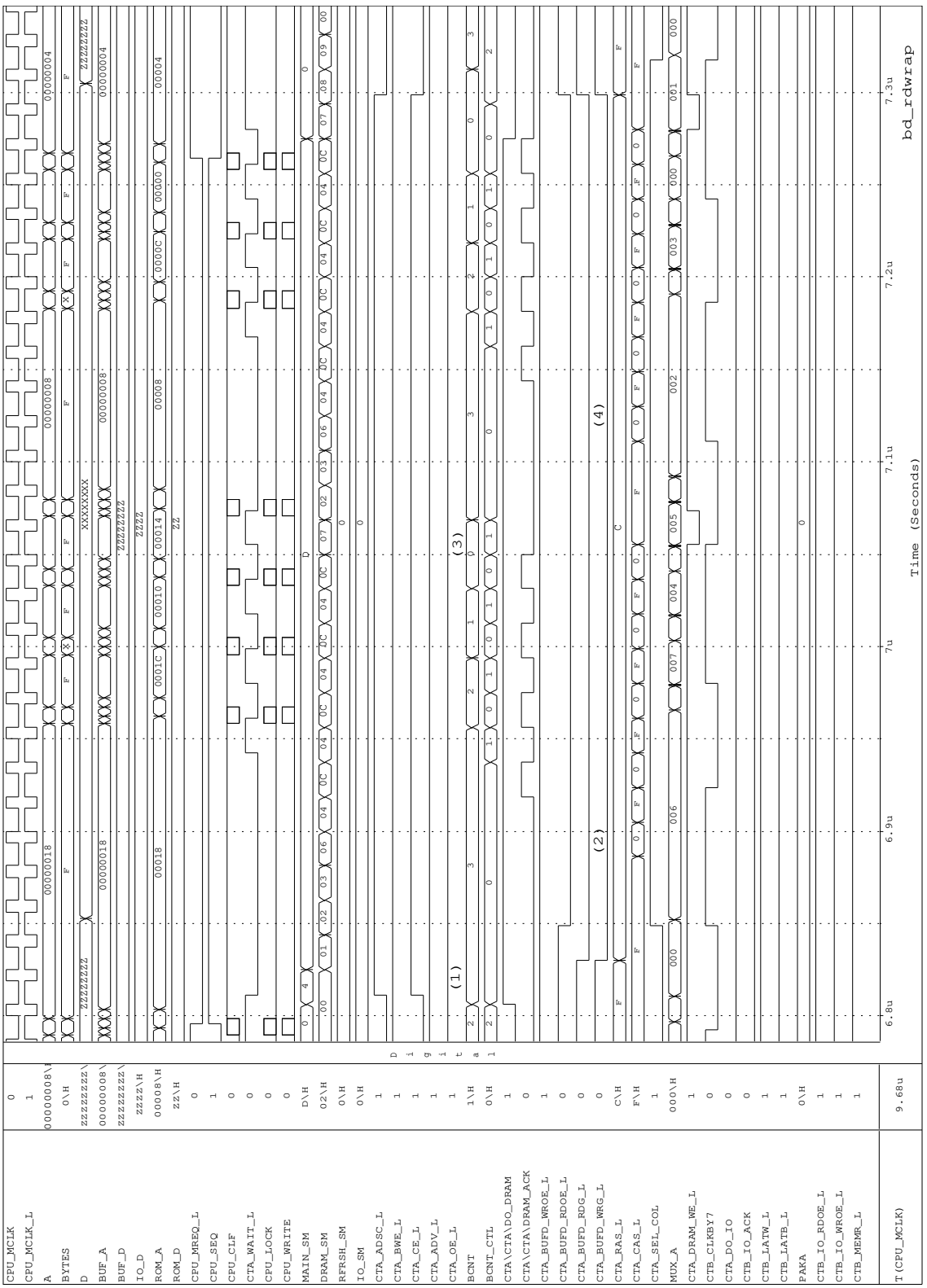
The waveform shows the sequence:

1. Since **cpu_clf** is asserted, **bcnt** is loaded with 3, regardless of the value of the address bus. This ensures that 4 data beats will be read, wrapping within the BEDO DRAM block if necessary.
2. The first assertion of **cas_l** latches the address for the burst read. The next four assertions of **cas_l** read four data beats.
3. The **bcnt** value reaches 0. Since this is a wrapped read, this indicates that 4 data beats have occurred rather than indicating that the burst address counter has wrapped. The result is the same, though; the DRAM state machine toggles **cta_dram_we_l** to terminate the burst.

   A BEDO read cycle that crosses from one block to another must always terminate the first block read with a burst abort sequence. The BEDO DRAMs allow a new address (for the next block) to be latched into the DRAM on the fifth **cas_l** (the **cas_l** that reads the fourth data beat from the DRAM). In other words, it will latch the address for read data 5 at the same time as it provides read data 4. However, the SA-110 will not generate the address for read data 5 until it has received read data 4. Therefore, the EBSA-110 cannot take advantage of this pipelining facility.
4. The fifth assertion of **cas_l** latches a new address. The next four assertions of **cas_l** read four more data beats.

Simulation Waveforms
11.11 bd_rdwrap

Figure 11–11 bd_rdwrap



This sequence takes 506.25ns.

11.12 bd_wrf

This waveform, shown in Figure 11–12, was produced using the simulation script 'do_dram6.cmd'.

This simulation shows a BEDO DRAM full write. This is a 5-beat † sequential write corresponding to a cache block castout or the write of a complete write buffer entry. For these cycles, the CPU asserts **cpu_clf** as a 'hint' that the external circuitry need not monitor the byte enable signals. This allows the DRAM state machine to assert all **cta_cas[3:0]_l** signals during all beats of the write, and therefore overcome the performance limitation described in Section 11.10.

CPU full write sequences have the additional characteristic that they always start on INT16 address boundaries.

The waveform shows the sequence:

1. The first **cas_l** assertion latches the column address and first beat of write data, as before. The three subsequent **cas_l** pulses latch the remaining data for the BEDO DRAM block.
2. The burst counter reaches 0. The DRAM state machine aborts the current burst ‡ by toggling **cta_dram_we_l**.
3. An additional stall cycle is introduced so that the CPU address (for the start of the new burst) will be valid at the BEDO DRAM.
4. The fifth **cas_l** assertion latches the column address and final beat of write data.

This sequence takes 281.25ns (a 5-beat non-clf sequence takes 318.75ns).

11.13 rfrsh

This waveform, shown in Figure 11–13, was produced using the simulation script 'do_rfrsh.cmd'.

This simulation shows a DRAM (EDO or BEDO) refresh sequence, sandwiched between two DRAM reads.

The waveform shows the sequence:

1. A non-sequential DRAM read starts. The DRAM state machine is idle, and so **do_dram** (asserted by the Main state machine) causes the DRAM state machine to transition out of its idle state and start a read access.

† In practice, CPU full write sequences will always be either 4 beats or 8 beats in length. This example is contrived so that the whole waveform can fit on the page and remain readable.

‡ Since the burst was aligned, it is not necessary to abort the burst at this point; the next **cas_l** would automatically latch the column address. However, the abort incurs no performance penalty in this design, and is a side-effect of the non-clf write sequence in the DRAM state machine.

Simulation Waveforms
11.13 rfrsh

Figure 11–12 bd_wrf

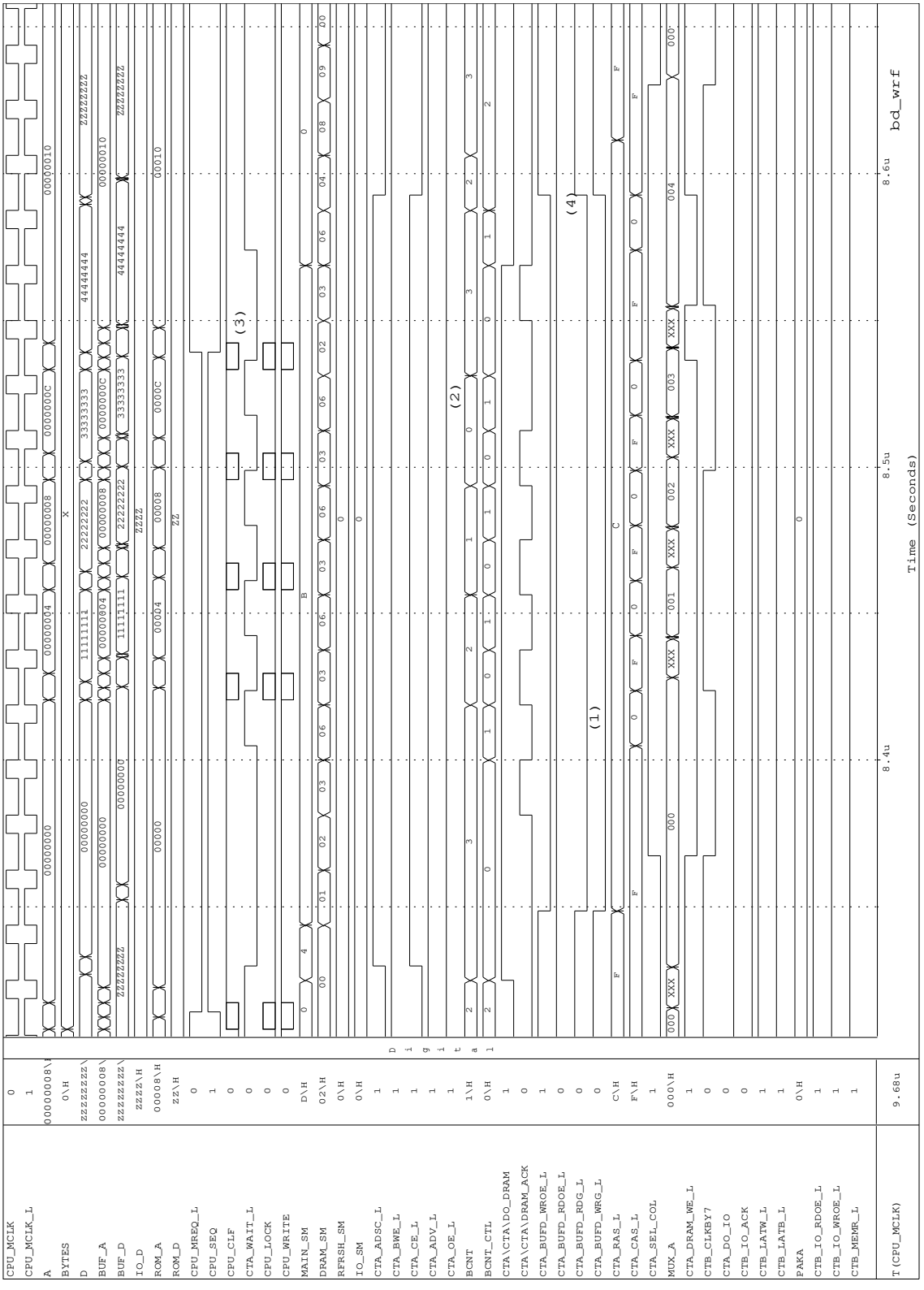
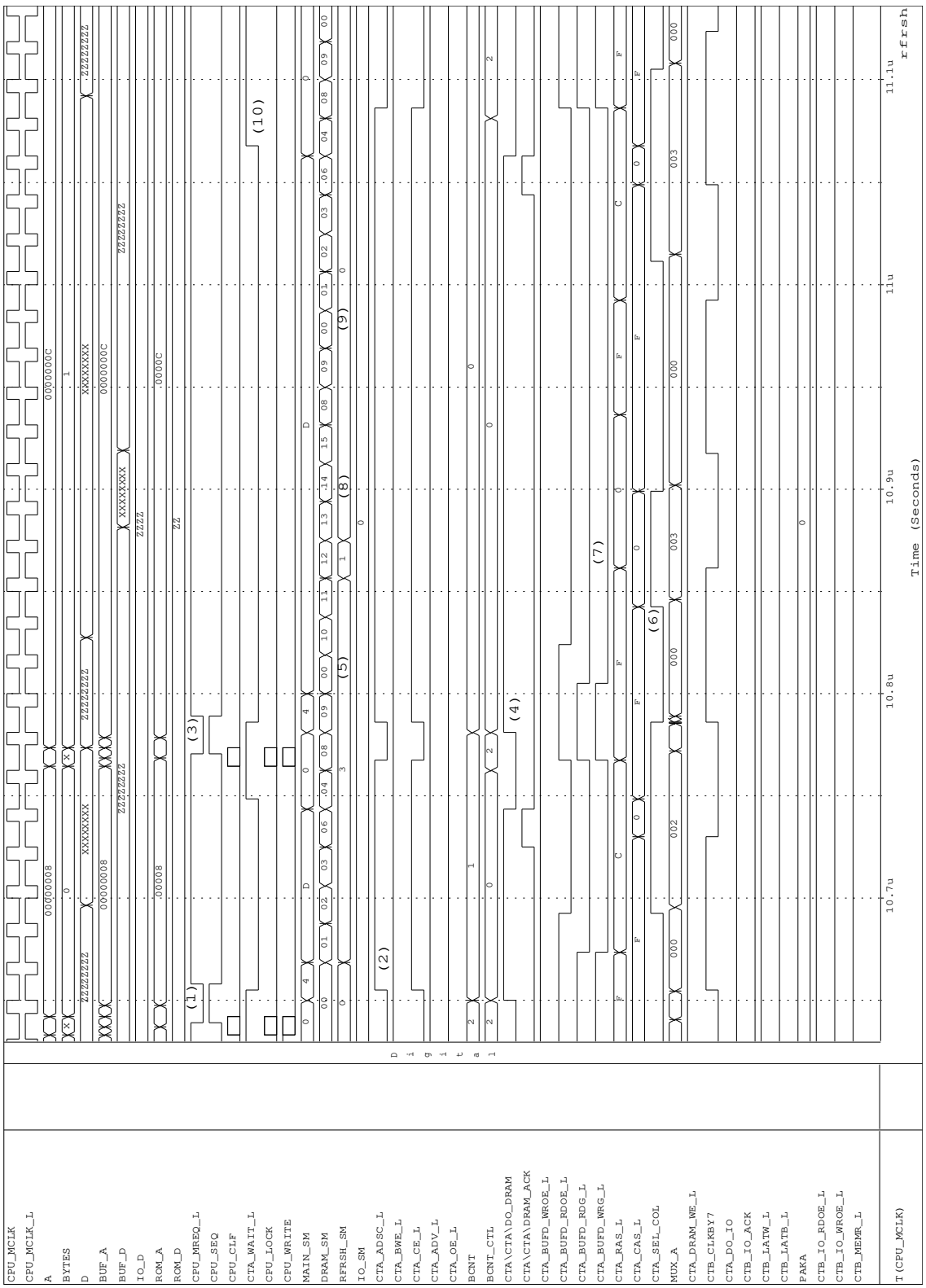


Figure 11–13 rfrsh



2. Simultaneously with the DRAM read sequence starting, the refresh counter times out and the REFRESH state machine generates a refresh request by transitioning out of its reset state. A refresh cannot start yet, because the DRAM state machine has committed to the read cycle.
3. A second non-sequential DRAM read starts.
4. **do_dram** asserts for the second read.
5. The DRAM state machine completes the RAS precharge for the first read and passes through its idle state. At this point, it samples both **do_rfrsh** and **do_dram** asserted. A refresh request always has a higher priority than a CPU cycle, and so the CPU continues to be stalled.
6. **cas_l** asserts with **ras_l** negated. This is the start of a CAS-before-RAS refresh cycle.
7. **ras_l** asserts, for the CAS-before-RAS refresh cycle.
8. The DRAM state machine goes back to idle as the result of a **rfrsh_ack** (not shown on these waveforms).
9. The refresh (and the RAS precharge) completes, and the DRAM state machine goes back to its idle state. At this point it samples **do_dram** asserted (**do_rfrsh** is now negated) and starts a DRAM read cycle.
10. The read cycle completes and the CPU cycle is terminated.

Note that a sequential read cycle will not be interrupted by a refresh; the DRAM state machine will complete the whole cycle. The refresh request occurs just after the first DRAM read has started but it is held off until the read has completed. Once the RAS precharge has been met, the (CAS-before-RAS) refresh sequence starts. At the same time, a further CPU read starts but is held off by the refresh in progress. Once the refresh has completed and the RAS precharge has been met, the CPU access proceeds.

11.14 cbr

This waveform, shown in Figure 11–14, was produced using the simulation script 'do_cbr.cmd'.

This simulation shows a CPU-initiated write-CAS-before-RAS (WCBR) cycle followed by a CPU-initiated CAS-before-RAS (CBR) cycle. These cycles are used to configure the BEDO DRAMs.

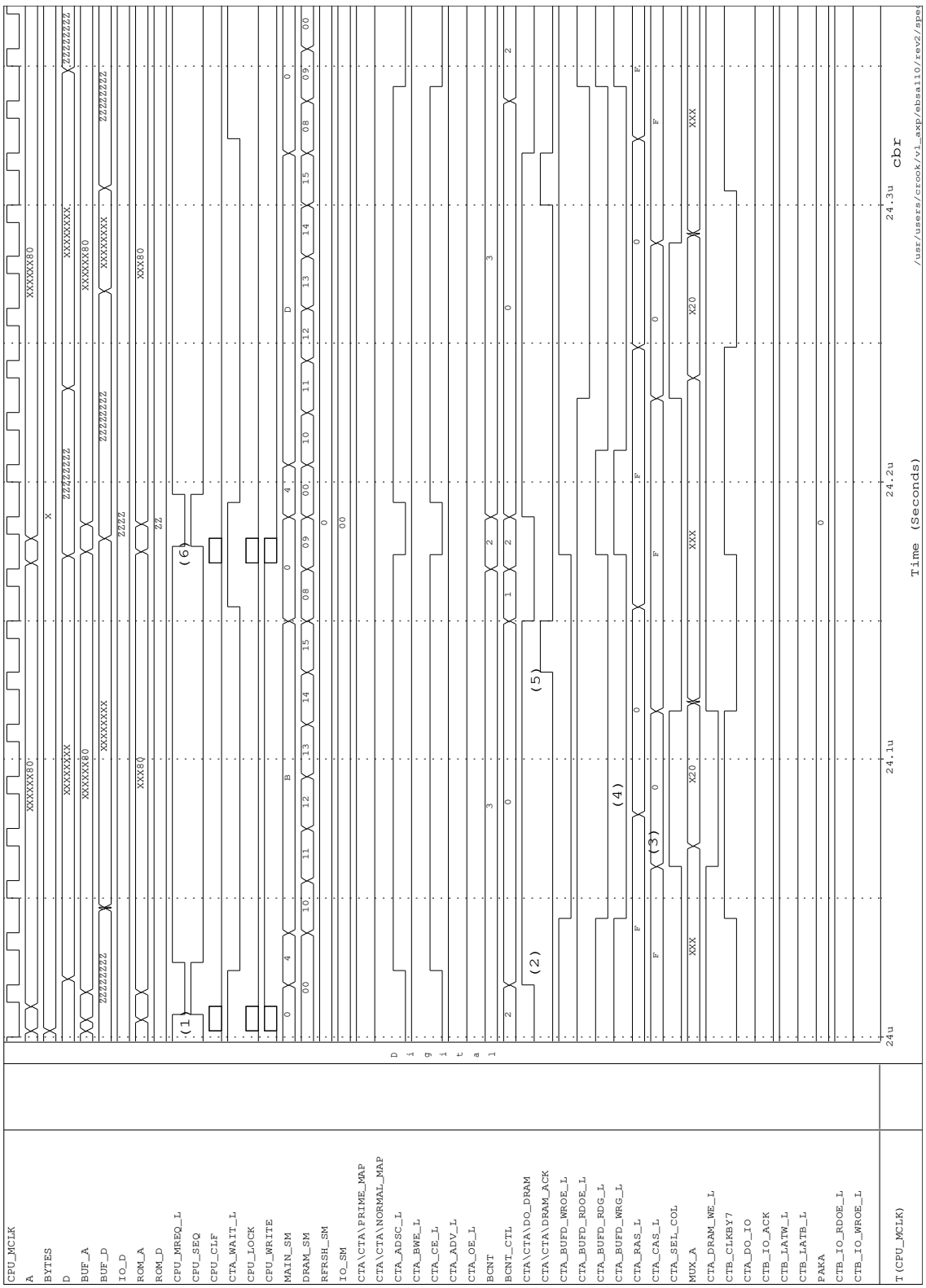
CBR and WCBR cycles are generated by CPU write and reads to the DRAM address space when the SOFT_DCBR bit is asserted.

The waveform shows:

1. A non-sequential write to address 0xXXXX.XX80 starts. (The upper-case X indicates that the address line has been set to the 'unknown' state. What you cannot see on the waveform † is that the two high-order address lines are set to select the DRAM space, and that some other high-order lines select which DRAM bank is decoded.)

† The waveform shows the value of each nibble. If any bit in the nibble is X, the whole nibble is shown as X.

Figure 11–14 cbr



2. **do_dram** asserts, taking the DRAM state machine out of its idle state. The DRAM state machine starts the state sequence 0x10, 0x11, which is the refresh sequence (refer to earlier timing diagrams to see that normal read and writes to DRAM cause a different state sequence).
3. **cta_cas_l** asserts (all the **cas_l** signals assert: the bus of four signals changes from 0xf to 0x0) whilst **cta_ras_l** is still negated. The **mux_a** value is 'unknown'.
4. One clock later, **cta_ras_l** asserts (some combination of **ras_l** signals assert, depending upon the address decode and the size of DRAM SIMMs fitted). The **mux_a** value is 0xX20. This value is a transformation of the address 0xXXXX.XX80 on the address bus.

   BEDO DRAMs latch configuration information on this falling edge of **ras_l**. The configuration information is latched from the low 8 bits of the address bus. The value 0x20 configures the DRAMs to operate with a linear (rather than interleaved) burst sequence.
5. Finally, the DRAM state machine asserts **dram_ack**, the Main state machine terminates the CPU cycle (by negating **cta_wait_l**) and the DRAMs perform their precharge sequence.
6. A non-sequential read from address 0xXXXX.XX80 starts. In this case, the address is only used to select the DRAM space, and the particular DRAM bank. This cycle is the same as the write cycle, except that **cta_dram_we_l** is negated when **ras_l** asserts.

11.15 romrd1

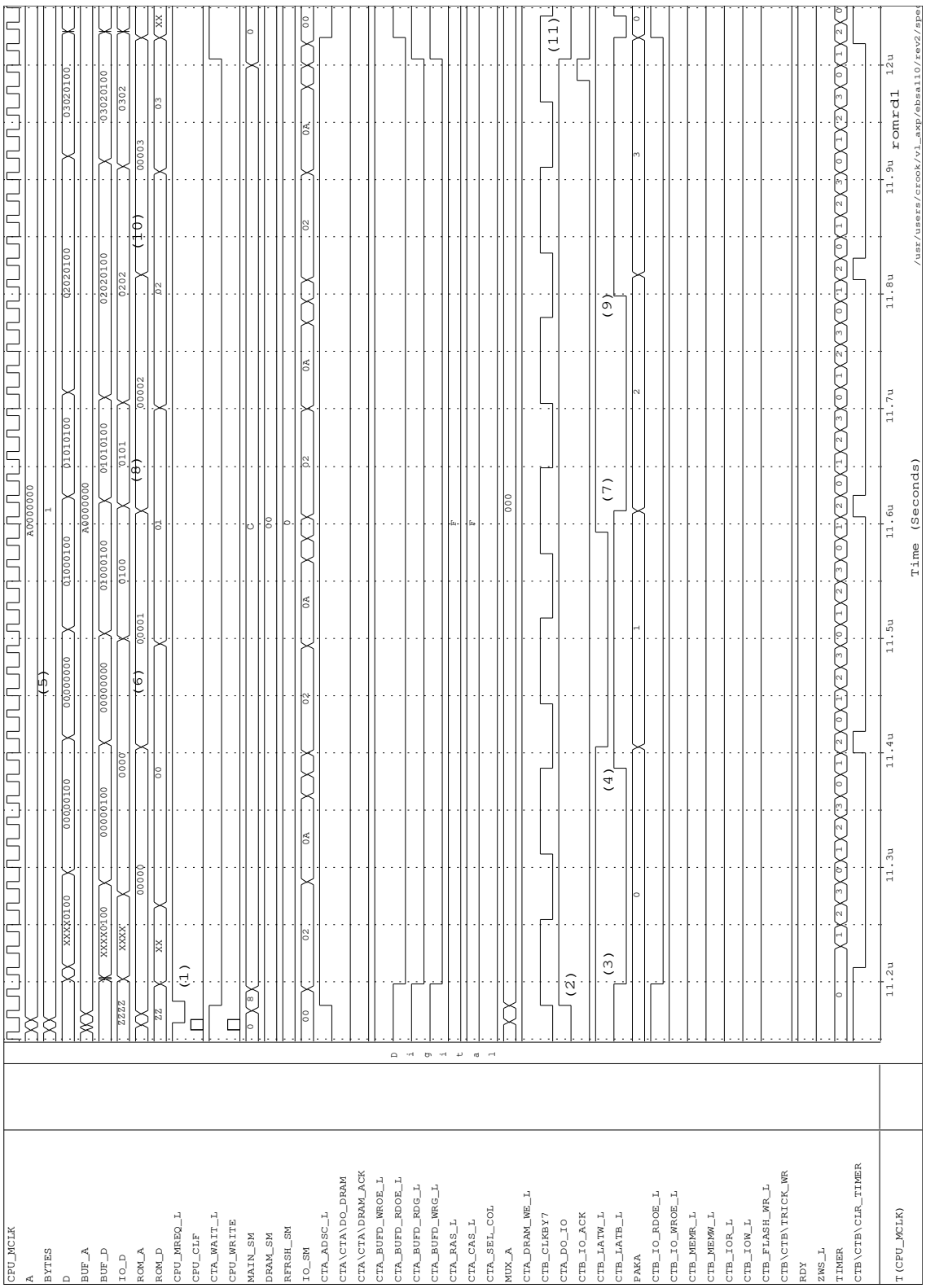
This waveform, shown in Figure 11–15, was produced using the simulation script 'do_flashrd4.cmd'.

This simulation shows a 1-beat (non-sequential) read from EPROM. Since the EPROM is an 8-bit device, four 8-bit values are packed to supply a 32-bit value to the CPU. The two low-order address lines to the EPROM are supplied by a 2-bit counter, **pak_a**.

The waveform shows the sequence:

1. The CPU starts a read cycle from the EPROM.
2. This causes **do_io** to assert; the IO state machine moves out of its idle state.
3. The low-order EPROM address lines are provided by the **paka** counter. This is reset to 0 at the start of the cycle, and counts through to 3 during the cycle.
4. When the read access time of the EPROM has been satisfied, the first byte of data from the EPROM is latched in the ROM data buffer by negating **ctb_latb_l** (latch byte).
5. The read data propagates through to the CPU data bus, but the CPU cycle remains stalled. The CPU data bus shows the value 0x0000.0000.
6. The EPROM address is incremented to 0x0000.0001. The second EPROM read commences.

Figure 11–15 romrd1



7. When the read access time of the EPROM has been satisfied, the first and second bytes of data are latched in the IO_D data buffer by the negation of **ctb_latw_l** (latch word). The first byte of data is provided (on **io_d[7:0]**) by the latch in the ROM_D data buffer, the second byte is flow-through (onto **io_d[15:8]**) through the ROM data buffer.

   At this point, the CPU data bus shows the value 0x0101.0100.

   The waveform shows the progress of read data from **rom_d** to **io_d** to **buf_d** and finally to **d**, the CPU data bus.

8. The EPROM address is incremented to 0x0000.0002. The third EPROM read commences.

9. When the read access time of the EPROM has been satisfied, the third byte of data from the EPROM is latched in the ROM data buffer in the same way as the first byte was; by the negation of **ctb_latb_l**.

10. The EPROM address is incremented to 0x0000.0003. The fourth and final EPROM read commences.

    When the read access time of the EPROM has been satisfied, all 32 bits of data are available on the CPU data bus. The IO state machine asserts **ctb_io_ack** to show that its cycle has completed, and the Main state machine negates **cta_wait_l** to terminate the cycle.

11. When the Main state machine samples **io_ack** asserted, it determines that **cpu_mreq_l** is negated, and so it negates **cta_do_io** to indicate that no further data beats are required.

    When the 32 bits of data are driven on the CPU **d** bus, the four bytes are sourced like this:

    - Bits **d[31:16]** are driven from the latch in the IO_D data buffer, and flow through the BUF_D data buffer.
    - Bits **d[15:8]** are driven from the latch in the ROM_D data buffer, and flow-through the low half of the IO_D data buffer and the BUF_D data buffer.
    - Bits **d[7:0]** are driven from the EPROM and flow-through the ROM_D data buffer, the IO_D data buffer and the BUF_D data buffer.

11.16 romrd2

This waveform, shown in Figure 11–16, was produced using the simulation script ‘do_flashrd4.cmd’.

This simulation shows a 2-beat sequential read from EPROM. It shows how the IO state machine handshakes with the Main state machine.

The waveform shows the sequence:

1. The read cycle starts.

2. The Main state machine asserts **cta_do_io** and the IO state machine performs a packing sequence as before.

3. The IO state machine asserts **ctb_io_ack** for one clock cycle.

Figure 11–16 romrd2

4. The Main state machine responds by negating **cta_wait_l** for one cycle so that the CPU can sample the read data. However, **cpu_mreq_l** remains asserted (indicating that further data beats are required) and so the Main state machine keeps **cta_do_io** asserted.
5. The EPROM address counter wraps around to 0 and a new packing sequence begins.
6. The packing sequence completes and the IO state machine asserts **ctb_io_ack** for one cycle.
7. The Main state machine responds by negating **cta_wait_l**, as before. Since **cpu_mreq_l** has now negated, **cta_do_io** is now negated.
8. The cycle has completed so the IO state machine goes back to its idle state.

11.17 flashwr

This waveform, shown in Figure 11–17, was produced using the simulation script 'do_flashwr1.cmd'.

This simulation shows write accesses to 5 sequential addresses in Flash. These must be performed as separate, non-sequential writes.

The waveform shows the sequence:

1. The first write access is to address 0x8000.0000. The Flash address bus (shown as **rom_a**) is showing address 0x0000.0000. The two low-order address lines are supplied by **paka[1:0]**.
2. The Flash write cycle is performed by the IO state machine, and the actual write is performed by the assertion of **ctb_flash_wr_l**.

   During Flash writes, data is always provided on the low-order byte lane.
3. The second write access is to address 0x8040.0000. High-order address lines (**buf_a[23:22]** are used to jam-load the **paka** counter. The Flash address bus drives the value 0x0000.0001.
4. Subsequent write accesses are to addresses 0x8080.0000, 0x80C0.0000 and 0x80C0.0004. The values on **rom_a** shows that the control logic converts these into an incrementing set of addresses.

11.18 io

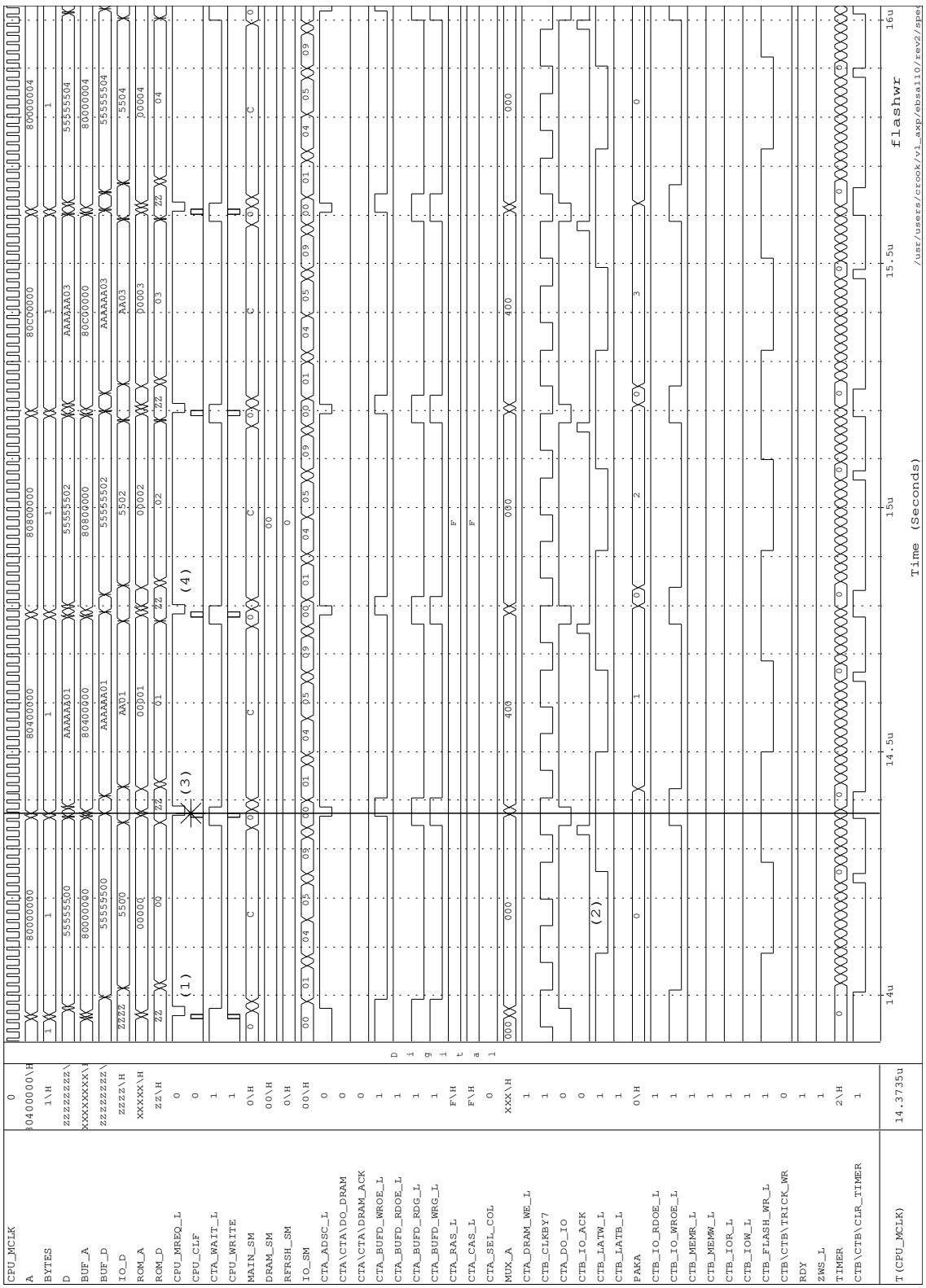
This waveform, shown in Figure 11–18, was produced using the simulation script 'do_io.cmd'.

This simulation shows a read-write-read sequence to ISAIO space. Each of the three cycles is a non-sequential access, but the cycles are performed back-to-back. This allows the data bus turn-on/turn-off times to be seen, showing that there is no tristate overlap on the data buses.

The waveform shows the sequence:

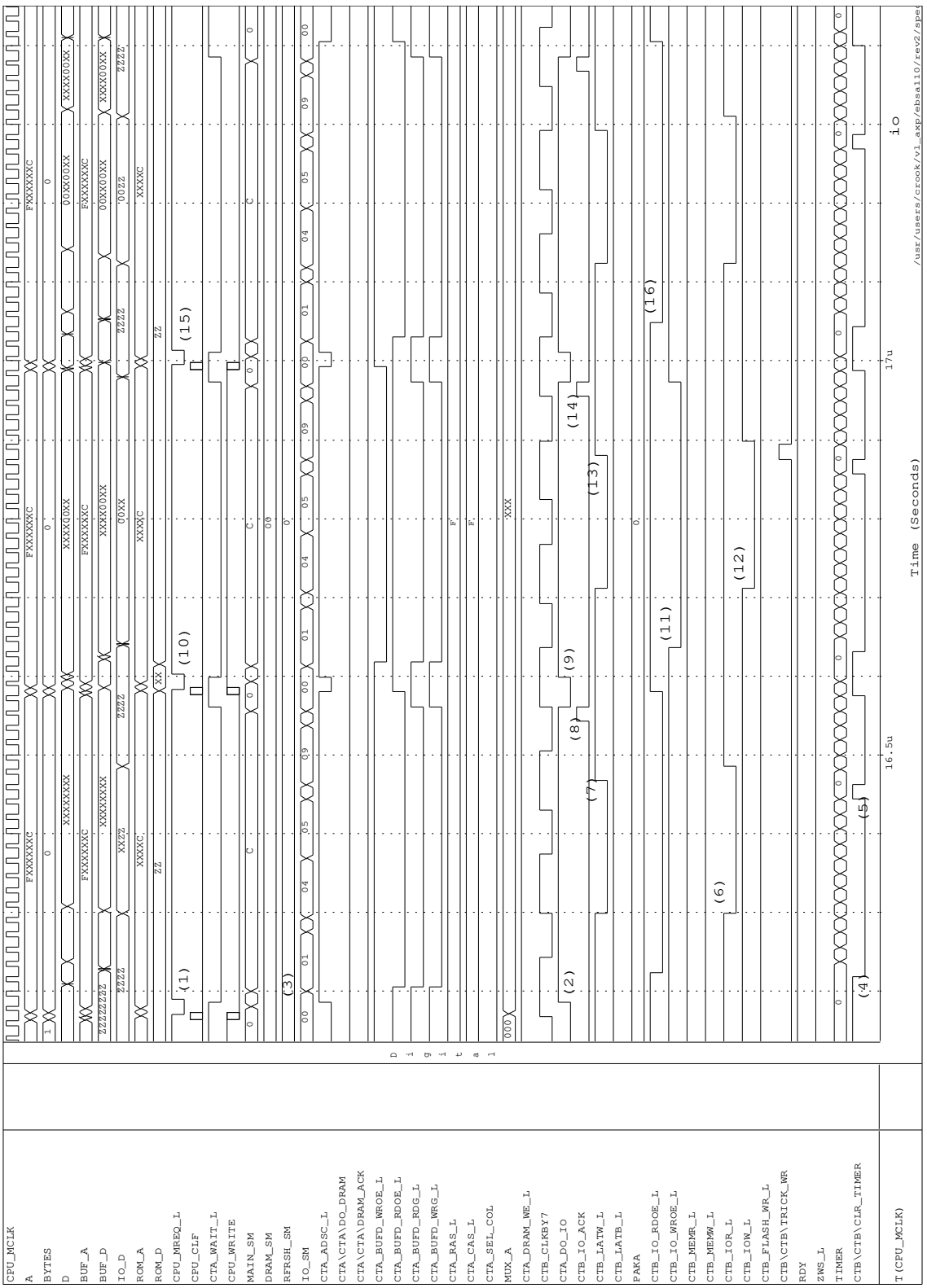
1. The CPU starts a read cycle, and is stalled by the Main state machine.

Figure 11–17 flashwr



Simulation Waveforms
11.18 io

Figure 11–18 io



2. The Main state machine decodes an I/O access and asserts **cta_do_io** to the IO state machine.
3. The IO state machine moves out of its idle state to start the read cycle.
4. A 2-bit timer (**timer**) is used as a resource by several state sequences within the IO state machine. This avoids adding a large number of states to the state machine (which would have the effect of increasing dependencies in the next-state and output-decode logic).
5. The IO state machine can detect the count value of the timer. The timer will wrap around from 3 to 0, or can be explicitly reset by the IO state machine using **clr_timer**.
6. After a delay (imposed to provide an address setup for the I/O device), the IO state machine asserts **ctb_ior_l**.
7. After a delay (imposed to provide the read access time for the slowest I/O device on the board), the IO state machine negates **ctb_latw_l** to latch the read data in the IO_D latching buffer. **ctb_ior_l** negates one cycle later, so that there is a positive data hold time at the latch.

   This read cycle is a read of the PIT, which drives data on **io_d[15:8]**. The register is uninitialized, so the value read back is 0xXX.
8. After a delay (imposed to provide an address hold time for the I/O device), the IO state machine asserts **ctb_io_ack** to the Main state machine to terminate the cycle.
9. **ctb_do_io** negates, and the IO state machine returns to idle.
10. The CPU starts a write cycle, and starts up the IO state machine as before. This causes **ctb_do_io** to assert again, so that it was only negated for a single cycle. This shows that **ctb_do_io** is guaranteed to negate, even when non-sequential back-to-back CPU cycles are performed.
11. The IO_D bus is driven with the CPU's write data when **ctb_io_wroe_l** asserts. In this back-to-back read-write pair, 3 **mclk** cycles have elapsed since the data bus was turned off for the previous read. 8 **mclk** cycles have elapsed since **ctb_ior_l** was negated for the previous read. This ensures that there is no tristate overlap (bus contention) on the IO_D bus.
12. After the address setup delay, the IO state machine asserts **ctb_iow_l** for long enough to meet the data-in requirement of the slowest I/O device.
13. **ctb_latw_l** asserts for write cycles, but this is simply a side-effect of the state machine implementation and it serves no useful purpose. None of the data bus latches are used to latch data during a write.
14. After the address hold delay, the IO state machine asserts **ctb_io_ack**, causing the Main state machine to terminate the cycle.
15. The CPU starts a read cycle, which proceeds as before.

    This time, the PIT register is initialized and the value 0x00 is read back. Since the PIT drives **io_d[15:8]**, the **io_d** bus shows the value 0x00ZZ. The CPU data bus eventually shows the value 0x00XX00XX.
16. The sequencing of **ctb_io_wroe_l** and **ctb_io_rdoe_l** ensures that no tristate overlap occurs.

ISAMEM cycles behave identically to ISAIO cycles, except that **ctb_memr_l** and **ctb_memw_l** are asserted rather than **ctb_ior_l** and **ctb_iow_l**.

11.19 iordy

This waveform, shown in Figure 11–19, was produced using the simulation script 'do_iordy.cmd'.

This simulation shows the same read-write-read sequence as io (Section 11.18). It shows how **rdy** can be used to extend the cycle length; the assertion time of either **ctb_ior_l** or **ctb_iow_l**.

rdy is a open-collector signal with an associated pull-up resistor. Any I/O device can negate **rdy** during an ISAIO cycle. If **rdy** is never driven during an access, the access completes with a fixed cycle time. If **rdy** is negated during a cycle, the assertion of **ctb_ior_l** or **ctb_iow_l** will be extended until **rdy** asserts. When **rdy** asserts, the cycle will terminate with exactly the same sequence as an unextended cycle.

rdy can transition asynchronously; CTB contains synchronizing logic for this signal.

ISAMEM cycles can be extended, using **rdy**, in exactly the same way.

11.20 iozws

This waveform, shown in Figure 11–20, was produced using the simulation script 'do_iozws.cmd'.

This simulation shows the same read-write-read sequence as io (Section 11.18). It shows how **zws_l** can be used to truncate the cycle length; the assertion time of either **ctb_ior_l** or **ctb_iow_l**.

zws_l is a open-collector signal with an associated pull-up resistor. Any I/O device can assert **zws_l** during an ISAIO or ISAMEM cycle. If **zws_l** is never driven during an access, the access completes with a fixed cycle time. If **zws_l** is asserted during a cycle, **ctb_ior_l** or **ctb_iow_l** will be negated as soon as possible. Once asserted, **zws_l** should remain asserted until the **ctb_ior_l** or **ctb_iow_l** strobe has negated.

zws_l can transition asynchronously; CTB contains synchronizing logic for this signal. The synchronizer limits the maximum time between **zws_l** asserting and a cycle terminating.

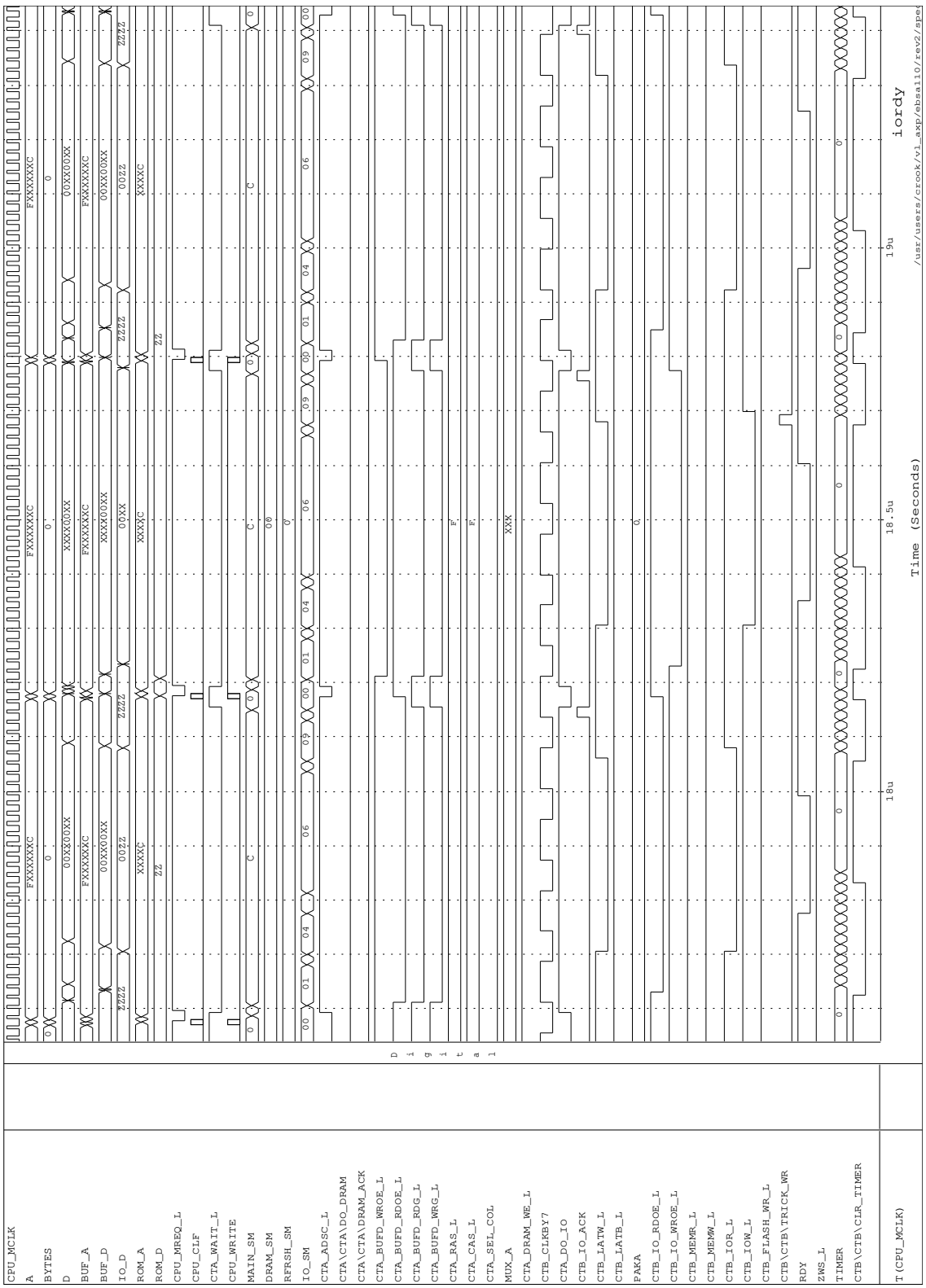
Figure 11–20 shows that the minimum ISAIO cycle time can be achieved by asserting **zws_l** at the same time as **ctb_ior_l** or **ctb_iow_l**. In practice, an I/O device must decode its address before asserting **zws_l**.

ISAMEM cycles can be truncated, using **zws_l**, in exactly the same way.

11.21 iorfrdy

This waveform, shown in Figure 11–21, was produced using the simulation script 'do_iorfrdy.cmd'.

Figure 11–19 iordy



Simulation Waveforms
11.21 iorfrdy

Figure 11–20 iozws

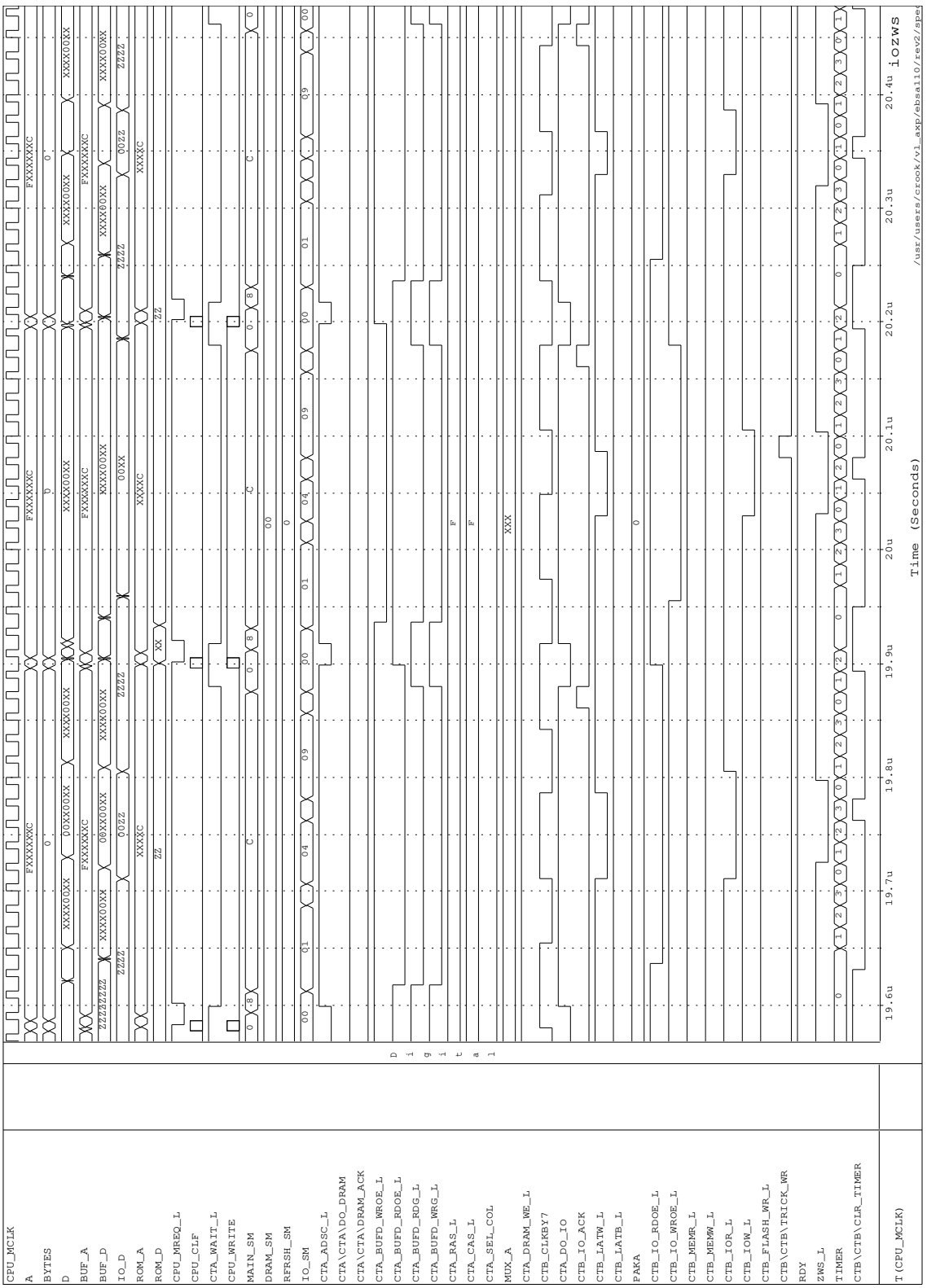
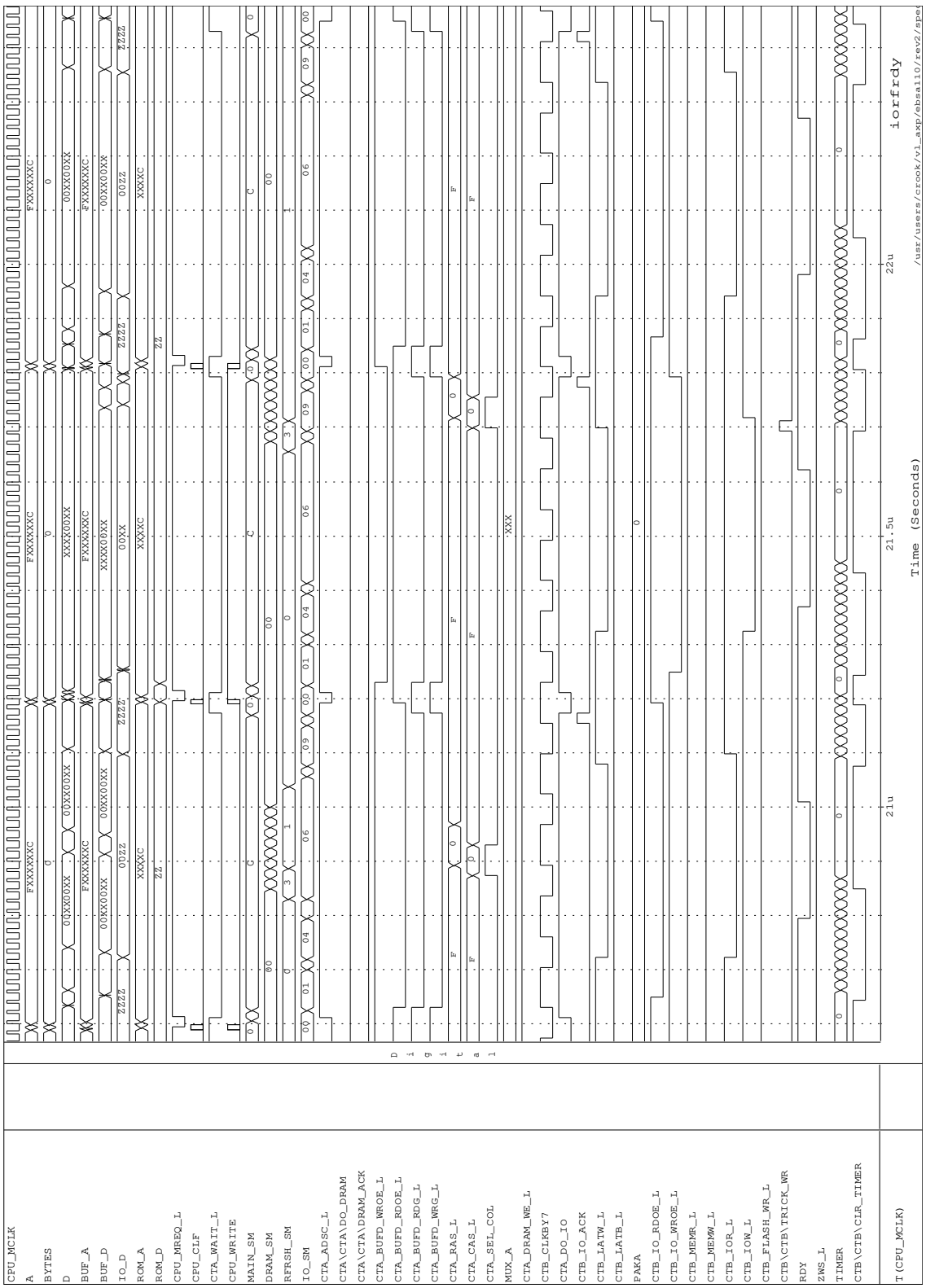


Figure 11–21 iorfrdy



This simulation shows the same extended read-write-read sequence as iordy (Section 11.19). It shows refresh cycles happening in parallel. This demonstrates that DRAM refresh activity is not held off during (potentially long) I/O cycles.

Notice that DRAM refresh cycles do not cause any activity on **do_dram** or **dram_ack**.

11.22 iotrick

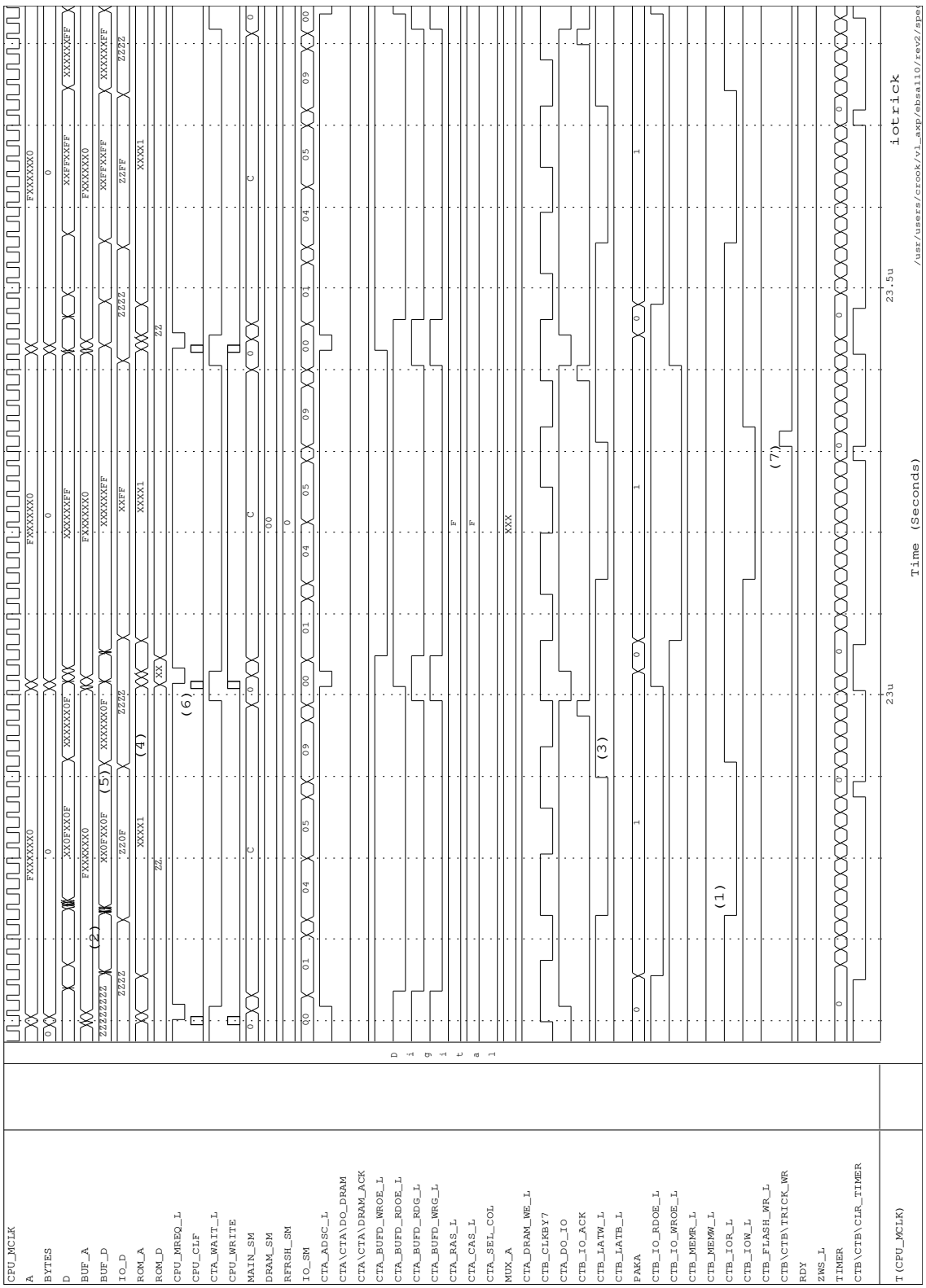
This waveform, shown in Figure 11–22, was produced using the simulation script 'do_io2.cmd'.

This simulation shows a read-write-read sequence like io (Section 11.18). It shows read and write cycles to internal registers in the CTB control logic. These registers (the trickbox registers) are decoded in the ISAIO space, so **ctb_ior_l** and **ctb_iow_l** strobes occur. A decode within the CTB logic generates the **trick_w** pulse which performs the register write.

The waveform shows the sequence:

1. The CPU read starts and causes **ctb_ior_l** to assert.
2. The I/O device in CTB drives a data byte of 0x0f on **io_d[7:0]**. The IO_D buffer duplicates the data from **io_d[15:0]** onto both **buf_d[31:16]** and **buf_d[15:0]**. Since **io_d[15:8]** is floating (tristate), unknown data (X) is driven on the associated bytes of **io_d**. Therefore, **buf_d[31:0]** drives the value 0xXX0fXX0f. This value is driven from **buf_d[31:0]** onto the CPU data bus, **d[31:0]**.
3. The IO state machine negates **ctb_latw_l** to latch the read data in the IO_D data buffer latch. On the next clock cycle it negates **ctb_ior_l**, causing the I/O device in CTB to tristate its data bus.
4. **io_d[7:0]** goes tristate, and therefore the whole of **io_d[15:0]** is now tristate.
5. **ctb_latw_l** only latches data in the low half of the IO_D buffers, the half that drives **buf_d[15:0]**. The half that drives **buf_d[15:0]** onto **buf_d[31:16]** remains transparent, and so it drives unknown data. Therefore, the value on **buf_d[31:0]** (and therefore, **d[31:0]**) changes from 0xXX0fXX0f to 0xXXXXXX0f. When the cycle completes, this is the value that the CPU reads.
6. The CPU write starts.
7. The IO state machine generates a pulse on **trick_wr** during the final cycle of **ctb_iow_l**. **trick_wr** is only used within the CTB control logic, and is used to latch write data for the Trickbox registers. (Data is actually latched into the Trickbox register two clock cycles later.)

Figure 11–22 iotrick



A

Configuration Guide

This appendix describes:

- The default configuration of the board
- The settings for all links and jumpers
- The pinout of all connectors
- The meaning of all LEDs
- The cables required for connection to the board
- How to upgrade the DRAM SIMMs

A.1 Default Configuration

Use this section to set your board back to the factory default settings.

The default hardware configuration of the EBSA-110 is for a 161.9 MHz CPU running at a core voltage of 1.5V, with a 53.9 MHz external bus. To set this configuration, remove any jumper on J1 and fit jumpers to these pins on J4: pin 1-2, pin 3-4, pin 7-8, pin 13-14.

The default software configuration of the EBSA-110 is to boot the ARM remote debugger from Flash. To set this configuration, remove any jumpers from J2 pins 1-2, 9-10, 11-12 and 13-14.

A.2 Description of All Jumpers

2-pin jumpers are used to configure behavior that you may wish to change. J1 is a 2-pin jumper to configure the CPU core voltage, J4 is a 16-pin jumper block (accommodating 8 jumpers) that configures clock frequencies. J2 is a 16-pin jumper block that configures software boot options. J2 is also used to connect to the speaker and external switches. Figure A–1 shows the settings for these jumpers, and Table A–1 describes the function of each jumper.

Note

The EBSA-110 supports a range of core frequencies for the SA-110 microprocessor. Ensure that you do not try to run your board with a core frequency that exceeds the specification of the SA-110 that is fitted.

Figure A–1 EBSA-110 Configuration Links

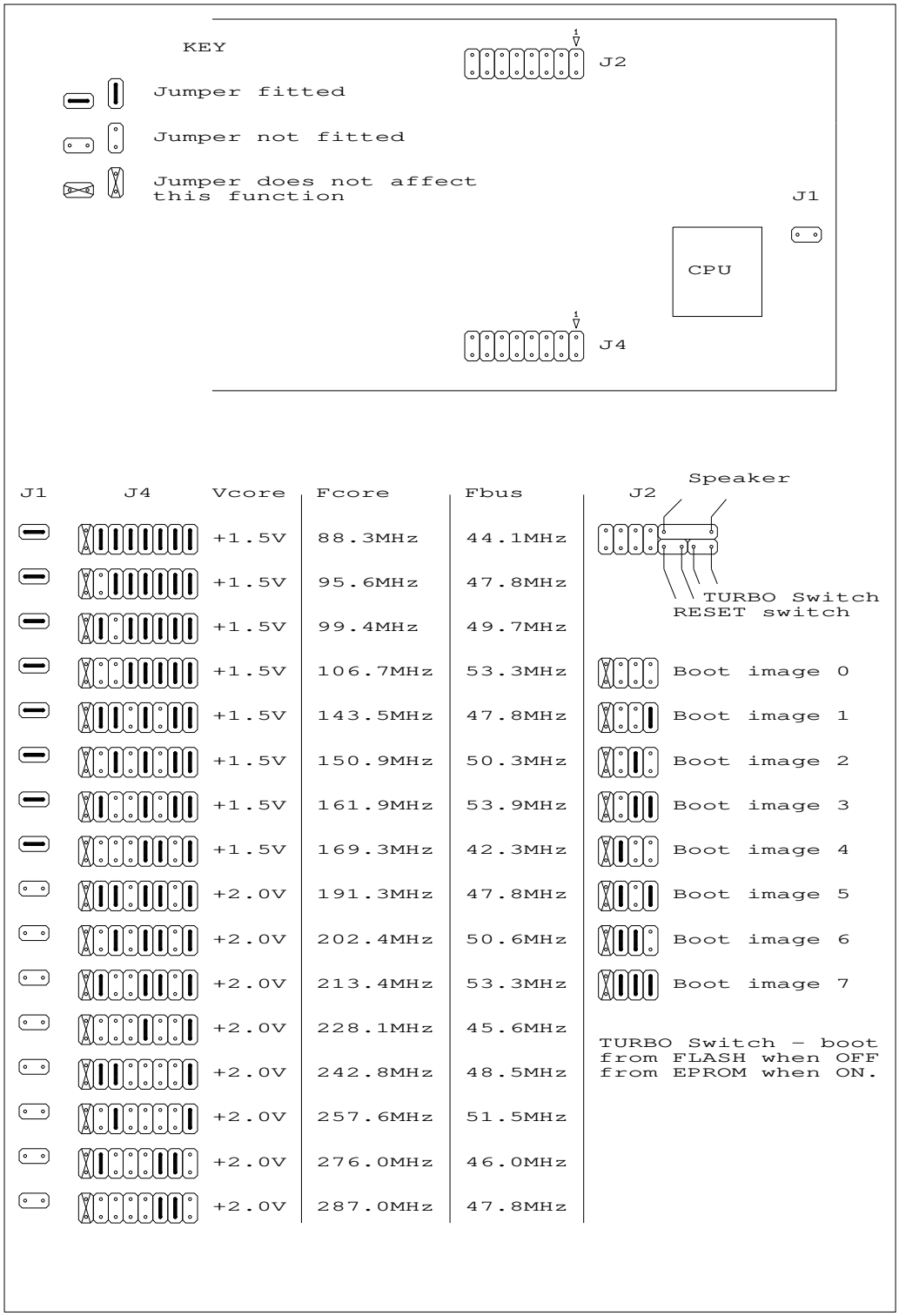


Table A–1 Jumpers

Reference	Pin	Function
J1	1-2	CPU Core voltage. When removed, the CPU core voltage is configured to +2V. When fitted, the CPU core voltage is reduced to +1.5V.
J4	1-2	**cpu_mccfg[2]** is tied LOW when this jumper is fitted.
J4	3-4	**cpu_mccfg[1]** is tied LOW when this jumper is fitted.
J4	5-6	**cpu_mccfg[0]** is tied LOW when this jumper is fitted. **cpu_mccfg[2:0]** set the CPU bus clock as a sub-multiple of the CPU core clock.

1-2	3-4	5-6	mccfg	Divisor
-	-	-	111	core clock/9
-	-	fit	110	core clock/8
-	fit	-	101	core clock/7
-	fit	fit	100	core clock/6
fit	-	-	011	core clock/5
fit	-	fit	010	core clock/4
fit	fit	-	001	core clock/3
fit	fit	fit	000	core clock/2

J4	7-8	**cpu_cccfg[3]** is tied LOW when this jumper is fitted.
J4	9-10	**cpu_cccfg[2]** is tied LOW when this jumper is fitted.
J4	11-12	**cpu_cccfg[1]** is tied LOW when this jumper is fitted.

(continued on next page)

Table A–1 (Cont.) Jumpers

Reference	Pin	Function
J4	13-14	**cpu_cccfg[0]** is tied LOW when this jumper is fitted. **cpu_ cccfg[3:0]** set the CPU core clock.

7-8	9-10	11-12	13-14	cccfg	Frequency MHz
fit	fit	fit	fit	0000	88.3
fit	fit	fit	-	0001	95.6
fit	fit	-	fit	0010	99.4
fit	fit	-	-	0011	106.7
fit	-	fit	fit	0100	143.5
fit	-	fit	-	0101	150.9
fit	-	-	fit	0110	161.9
fit	-	-	-	0111	169.3
-	fit	fit	fit	1000	191.3
-	fit	fit	-	1001	202.4
-	fit	-	fit	1010	213.4
-	fit	-	-	1011	228.1
-	-	fit	fit	1100	242.8
-	-	fit	-	1101	257.6
-	-	-	fit	1110	276.0
-	-	-	-	1111	287.0

J2	1-7	Loudspeaker connection.
J2	2-4	Reset. When these pins are connected, the system will be held in reset. Normally, this is wired to a reset switch.
J2	6-8	EPROM_BOOT. When this jumper is removed (default), the initial bootstrap code is read from the Flash ROM. When this jumper is fitted, the initial bootstrap code is read from the EPROM. This could be wired to a switch so that this selection can be made from the front panel, without removing the system's cover.
J2	9-10	SOFTI3: Select boot image.
J2	11-12	SOFTI2: Select boot image.

(continued on next page)

Table A–1 (Cont.) Jumpers

Reference	Pin	Function			
J2	13-14	SOFTI1: Select boot image.			
		J2:13-14	**J2:11-12**	**J2:9-10**	**Action**
		-	-	-	Boot image 0
		-	-	fit	Boot image 1
		-	fit	-	Boot image 2
		-	fit	fit	Boot image 3
		fit	-	-	Boot image 4
		fit	-	fit	Boot image 5
		fit	fit	-	Boot image 6
		fit	fit	fit	Boot image 7
		One or more pairs of these pins could be wired to switches so that different images could be selected easily.			
J2	15-16	SOFTI0: The function of this link is unassigned; it may be freely used by application software.			

A.2.1 Supported Clock Configurations

Figure A–1 shows which combinations of core and bus frequency are supported by the EBSA-110. The clock configuration is set by configuring jumpers on J4 and J1.

It is possible to run with higher MCLK divisors (slower MCLKs) than those shown but the DRAM timing will be sub-optimal and the DRAM refresh interval will need to be reprogrammed to avoid data corruption.

A.3 Description of All Links

Configuration links are pieces of copper etch on the PCB that have been layed out in such a way as to make them easy to cut (with a scalpel) and reconfigure (to an adjacent pad). They are provided to allow the experienced hardware engineer to experiment with different modes of operation of the EBSA-110. The links are shown in Table A–2. Links with a reference designator greater that 100 are sited on side 2 of the board.

Table A–2 Links

Reference	Schematic Sheet	Description
EL1	18	By default, channel 1 of the Programmable Interval Timer is clocked at 1/7th of the bus frequency. Cutting this link isolates the channel's clock so that an alternative frequency can be driven in.

(continued on next page)

Table A–2 (Cont.) Links

Reference	Schematic Sheet	Description
EL2	18	By default, channel 2 of the Programmable Interval Timer is clocked at 1/7th of the bus frequency. Cutting this link isolates the channel's clock so that an alternative frequency can be driven in.
EL101	1	PWRSLP_L input to CPU. Default is HIGH, alternative is LOW.
EL104	1	TCK_BYP input to CPU. Default is LOW, alternative is HIGH.
EL105	1	APE input to CPU. Default is LOW, alternative is HIGH.
EL106	1	SNA input to CPU. Default is HIGH, alternative is LOW.
EL107	1	Default is to route nMCLK from the CPU to on-board logic. Cutting this track would allow the on-board logic to be driven from an alternative clock source.
EL108	1	If SNA (see above) is reconfigured to run the CPU bus interface asynchronously to the CPU core clock, these pads allow the bus clock to be driven into the CPU via a coaxial cable from an external signal generator.
EL109	1	DBE input to CPU. Default is HIGH, alternative is LOW.
EL110	1	ABE input to CPU. Default is HIGH, alternative is LOW.
EL111	5	SSRAM pin 14 input. Default is LOW, alternative is HIGH. This wiring is intended to accommodate next-generation SSRAMs. Some SSRAMs have +3V power on pin 14, others have 0V. The default is correct for Micron C4 and D7 parts.
EL112	1	CONFIG input to CPU. Default is HIGH, alternative is LOW.
EL113	1	SPDF input to CPU. Default is HIGH, alternative is LOW.
EL114	1	MSE input to CPU. Default is HIGH, alternative is LOW.
EL115	1	TESTCLK input to CPU. Default is LOW, alternative routes TESTCLK to output of on-board oscillator. This arrangement has been designed to allow the CPU to be driven from a 1x clock either by an on-board oscillator or via a coaxial cable from an off-board signal generator.

A.4 Connectors

Refer to the circuit schematics for the pinout of connectors. The connectors are shown in Table A–3.

Table A–3 Connectors

Reference	Schematic Sheet	Description
J21	12	COM1: 10-way male IDC for connection to RS232 serial port.
J23	12	COM2: 10-way male IDC for connection to RS232 serial port.
J24	12	LPT1: 26-way male IDC for connection to parallel printer port.

(continued on next page)

Table A–3 (Cont.) Connectors

Reference	Schematic Sheet	Description
J3	18	JTAG: 14-way male IDC for connection to ARM debug box.
J22	17	PCMCIA: dual-socket PCMCIA connector. Socket A is the lower socket (closer to the board). Socket B is the upper socket.
J20	21	Power: 12-way male connector for power.
J25	15	Ethernet: shielded female RJ45 (modular jack) for connection to 10Mbps 10-BaseT Ethernet.

A.5 Debug Connectors

The EBSA-110 is layed out with a number of debug connectors suitable for connection to a Tektronix DAS logic analyzer. These connectors may not be fitted as standard. Refer to the circuit schematics for the pinouts of individual connectors. On all connectors, all odd-numbered pins are connected to 0V. The debug connectors are shown in Table A–4.

An alternative use for these connectors is that it allows an expansion board to be attached to the EBSA-110, as described in Section 10.4.

Table A–4 Debug Connectors

Reference	Schematic Sheet	Description
J7	4	Buffered data: **buf_d31:24]**
J9	4	Buffered data: **buf_d[23:16]**
J8	4	Buffered data: **buf_d[15:8]**
J10	4	Buffered data: **buf_d[7:0]**
J18	4	Buffered address: **buf_a[29:22]**
J14	4	Buffered address: **buf_a[21:14]**
J15	4	Buffered address: **buf_a[13:6]**
J16	4	Buffered address: **buf_a[5:2]**, **buf_be[3:0]_l**
J11	4	I/O data: **io_d[15:8]**
J12	4	I/O data: **io_d[7:0]**
J19	4	Miscellaneous control signals
J17	4	Miscellaneous control signals and CTA observability
J13	4	Miscellaneous control signals and CTB observability
J26	4	2x8 header wired to **gnd** as logic analyzer reference
J27	4	2-pin header;**cpu_mclk** and **gnd**
J28	4	2-pin header;**cpu_mclk_l** and **gnd**
J29	4	2x8 header wired to **gnd** as logic analyzer reference

A.6 Debug Pick-up Points

Debug pick-up points are etch positions for individual test pins. These were used for high-speed signals where it was undesirable to increase the etch length by routing the signal to a central connector. The debug pick-up points are shown in Table A–5.

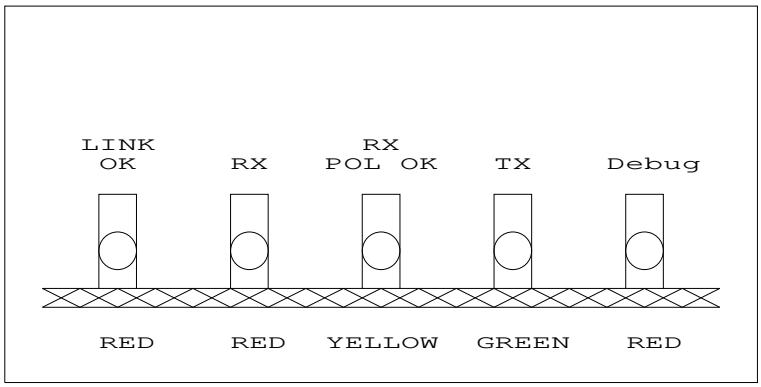
Table A–5 Pick-up point

Reference	schematic sheet	Description
TP1	22	+1.5V/+2.0V test/sense point, close to CPU
TP2	22	0V **gnd** reference, close to CPU
TP3	4	**cpu_lock**
TP4	20	+1.5V/+2.0V test point, close to regulator
TP5	20	+3.3V test point, close to regulator
TP6	20	0V **gnd** test point
TP7	4	**cpu_seq**
TP8	4	**cpu_clf**
TP9	4	**cpu_write**
TP10	4	**cpu_mreq_l**
TP11	4	**cta_wait_l**
TP12	4	**cpu_a[30]**
TP13	4	**cpu_a[31]**
TP16	16	**cia_iocs16_l**
TP17	16	**cia_memcs16_l**

A.7 LEDs

The EBSA-110 has 5 LEDs for displaying status information. The LEDs are positioned on the back edge of the board. When the board is cased, the LEDs are only visible from the rear of the unit. The order of the LEDs is shown in Figure A–2. The LEDs provide this information:

Figure A–2 Position of LEDs



- DEBUG (red) - this LED is used by the on-board diagnostics and by the ARM remote debug stub. When the on-board diagnostics are enabled, it flashes quickly 8 times when the board is reset or power-cycled. When the ARM remote debug stub is enabled it comes on for approximately 0.5s when the board is reset or power-cycled and is then extinguished.
- TX (red) - this LED is illuminated to indicate transmit activity on the Ethernet port.
- RX_POL (yellow) - this LED is illuminated to indicate correct receive polarity on the 10-BaseT link. If this LED is extinguished, the link will still operate correctly, since the Ethernet controller can automatically reverse the receive polarity.
- RX (green) - this LED is illuminated to indicate receive activity on the Ethernet port.
- LINK_OK (red) - this LED is illuminated when 10-BaseT link status is good.

A.8 Cables Within the Enclosure

The EBSA-110 uses flying leads to connect from the board to the power supply and to the connectors mounted onto the chassis.

A.8.1 Power Supply

The EBSA-110 power connector, J20, is on one edge of the board.

PC-style power supplies usually have a pair of power connectors. These connectors are polarized. When viewing the EBSA-110 so that the CPU is in the bottom right-hand corner of the board, the power connectors are positioned correctly when there are 3 red cables on the left and 4 black cables in the centre. The power supply cables may be labelled ‘P9’ on the left-hand connector and P8 on the right-hand connector.

The silk screen around J20 is marked to show the VDD (red), GND (black) and +12V (yellow) connections.

A.8.2 Serial Ports

The COM ports on the board are connected to 9-way male D-type connectors on the bulkhead of the enclosure. The cable is 10-way ribbon cable with a 9-way male D-type IDC on one end and a 10-way female header IDC on the other end. The connectors are aligned so that pin 1 on the D-type is connected to pin 1 on the header through pin 1 of the ribbon cable.

A.8.3 Parallel Port

The LPT1 port on the board is connected to a 25-way female D-type connector on the bulkhead of the enclosure. The cable is 26-way ribbon cable with a 25-way female D-type IDC on one end and a 26-way female header IDC on the other end. The connectors are aligned so that pin 1 on the D-type is connected to pin 1 on the header through pin 1 of the ribbon cable.

A.8.4 Reset Switch

The reset pins on J2 are connected to a front-panel reset switch by a 2-pin header and cable.

A.8.5 Turbo Switch

Many PC Cabinets have a 2-position 'TURBO' switch on the front panel. This can be connected to pin 2-4 on J2 to allow this switch to select between EPROM boot or Flash boot. Alternatively, it can be connected to pins 9-10 on J2 to allow this switch to select between Flash images 0 and 1 after reset.

A.8.6 Loudspeaker

The speaker pins on J2 are connected to the chassis' speaker by a 2-core cable terminated in a 4-pin connector. The speaker is wired to pins 1 and 4 of the connector and can be connected to J2 with either polarity.

A.9 Cables for External Connection

The EBSA-110 uses standard, readily available cables to attach to external equipment.

A.9.1 Serial Ports

The COM1 and COM2 serial ports on the EBSA-110 are wired as they would be on a PC, therefore they are wired as DTE (data terminal equipment) ports. Use a null-modem (twist) cable to connect a terminal or host system to this port.

A suitable cable will have a female 9-way D-type connector on one end and a female 9-way or 25-way (depending upon the connector on the other system) on the other end. Table A–6 shows the wiring of a suitable cable.

Table A–6 Null-MoDem Cable

(9-way) Connector A Pin	(25-way) Connector B Pin	(9-way) Connector B Pin	Description
5	7	5	Gnd - Gnd
3	3	2	TxD - RxD
7	5	8	RTS - CTS

(continued on next page)

Table A–6 (Cont.) Null-MoDem Cable

(9-way) Connector A Pin	(25-way) Connector B Pin	(9-way) Connector B Pin	Description
6,1	20	4	DSR, DCD - DTR
2	2	3	RxD - TxD
8	4	7	CTS - RTS
4	6,8	6,1	DTR - DSR, DCD

A.9.1.1 Serial Cable for SUN Workstation

ARM suggest a cable with the wiring shown in Table A–7 for connection to the 25-way connector on a SUN workstation.

Table A–7 SUN Null-MoDem Cable

(9-way) Connector A Pin	(25-way) Connector B Pin	Description
2	2	RxD - TxD
3	3	TxD - RxD
5	7	Gnd - Gnd
7-8	-	RTS - CTS
-	4-5	RTS - CTS
4-6-1	-	DTR-DSR-DCD
-	20-6-8	DTR-DSR-DCD

A.9.2 Parallel Port

The LPT1 parallel printer port on the EBSA-110 is wired as it would be on a PC. With suitable software, it can be used to drive a parallel printer or for bidirectional data exchange.

To use the port for bidirectional data exchange with a PC, use a bidirectional parallel cable. Such cables are readily available and are often described as *laplink* or *interlnk* cables. A suitable cable will have a male 25-way D-type connector on each end. Table A–8 shows the wiring of a suitable cable.

Table A–8 Bidirectional Parallel Cable

(25-way) Connector A Pin	(25-way) Connector B Pin	Description
2	15	Data 0
3	13	Data 1
4	12	Data 2
5	10	Data 3
6	11	Data 4
15	2	Error

(continued on next page)

Table A–8 (Cont.) Bidirectional Parallel Cable

(25-way) Connector A Pin	(25-way) Connector B Pin	Description
13	3	Online
12	4	Paper End
10	5	ACK
11	6	BUSY
25	25	GND

A.9.3 Parallel Port Loopback

The EBSA-110 diagnostics allow the optional use of a loopback connector on the parallel printer port. This loopback connector can only allow a partial test of the port. A suitable loopback can be made by wiring a male 25-way D-connector in accordance with Table A–9.

Table A–9 Parallel Port Loopback Connector

Connect Pins	Description
2-15	Data 0 to Error
3-13	Data 1 to Online
4-12	Data 2 to Paper Out
5-10	Data 3 to ACK

A.9.4 Ethernet Port

The connection to the Ethernet port is via a standard unshielded twisted pair Ethernet cable, terminated at each end with an RJ45 modular jack plug.

A.9.5 JTAG Port

The wiring of the JTAG port is shown in Table A–10. The JTAG port operates at 5V TTL levels.

Table A–10 JTAG Cable

Pin	Type	Description
1	-	Pulled up to +5V through a 33R resistor
2	-	GND
3	Input to board	TRST_L
4	-	Not connected
5	Input to board	TDI
6	-	GND
7	Input to board	TMS
8	-	GND
9	Input to board	TCK

(continued on next page)

Table A–10 (Cont.) JTAG Cable

Pin	Type	Description
10	-	GND
11	Output from board	TDO
12	Input to board	SRST_L - asserting this signal resets the board
13	-	Connected to pin 1
14	-	GND

A.10 Upgrading the DRAM SIMMs

The EBSA-110 can accommodate 2 DRAM SIMMs. If only one is fitted, it must be fitted in position J6. If two are fitted, they must be of the same type and density.

The EBSA-110 diagnostics report:

- The DRAM size
- The type of DRAM fitted (EDO or BEDO)
- The number of DRAM SIMMs fitted

Before handling DRAM SIMMs, ensure that you are observing the handling precautions described in Section 1.2.

To upgrade the DRAM SIMMs, power-down the system, remove the cover and click the new part(s) into the SIMM sockets, J5 and J6. The sockets are polarized so the SIMMs can only be inserted in the correct orientation.

The DRAM memory size and type is automatically detected and configured by a combination of on-board hardware and software.

After upgrading the DRAM SIMMs, run the EBSA-110 diagnostics to verify that the new configuration has been recognized and is working correctly.

The EBSA-110 can support 60ns 72-pin 5V 32-bit DRAM SIMMs in three densities: 1Mx32, 2Mx32, 4Mx32. Some suitable DRAM SIMMs are listed in Table A–11.

Table A–11 Suitable DRAM SIMMs

Part	Type	Description
Generic	1Mx32 EDO	2 RAS, driven simultaneously
Generic	2Mx32 EDO	2 RAS, decoded from high-order address
Generic	1Mx32 BEDO	2 RAS, driven simultaneously
Generic	2Mx32 BEDO	2 RAS, driven simultaneously
Generic	4Mx32 BEDO	2 RAS, driven simultaneously
MT16D232M-6 X	2Mx32 EDO	Micron
MT4D232M-6 B ES	2Mx32 BEDO	Micron
MT8D432M-6 B ES	4Mx32 BEDO	Micron

B

Debugging a Broken Board

At some time in the life of your EBSA-110, you may find that it suddenly ceases to function, due to mis-configuration or hardware failure. This section provides some hints on tracking down the fault.

B.1 Basic Checks

- Check that your power supply fan is running (a clue that the power supply is working).
- Check that the jumpers correctly set the core voltage and clock frequencies (see Figure A–1).
- Check that jumper J2, 2-4 (EPROM_BOOT) is not fitted.
- Check that the chassis or chassis mountings are not shorting against the board.
- Follow the procedure in Section 1.4, if you have not already done so. Ensure that any jumpers have been removed from J2 pins 9-10, 11-12, 13-14.

If the LED does not flash at all and you have an EPROM fitted in U22, fit jumper J2, 2-4 (EPROM_BOOT) and reset the board. If the LED flashes, this indicates that the Flash PBL has become corrupt; follow the procedure in Section 8.3. If the LED still does not flash, attempt to run the on-board diagnostics using the procedure described in Section 8.4.

If the diagnostics will not start, refer to Section B.2. If they start but one of the tests fails, refer to Section B.3.

B.2 Checking the Board

If your board is still showing no signs of life, perform these checks:

- Check that all socketed components are properly seated in their sockets.
- Check that all cables are correctly polarized and aligned.
- Check that there are no stray cables within the enclosure that could be shorting out.

To proceed any further, you will need access to an oscilloscope and a set of EBSA-110 schematics. Check the following:

- Power: +5V, +12V (from the external power supply) +3.3V, +2V (from on-board regulators).
- Clocks: **osc3** (3.68 MHz), **cpu_mclk_l** (49 MHz-55 MHz), **osc24** (20 MHz), **ctb_clkby7** (approximately 7 MHz).
- Reset: check that **rst_reset_l** toggles when the reset switch is pressed, and that **buf_reset_l** toggles as a result.

- CPU activity: check for transitions on **cpu_mreq_l** when the reset switch is released. If **cpu_mreq_l** asserts and stays asserted, see whether **cta_wait_l** is permanently negated.
- State machine activity: check for transitions on **cta_do_io** when the reset switch is released. If **cta_do_io** asserts and stays asserted then see whether **ctb_io_ack** is permanently negated.
- Packer activity: check for transitions on the Flash (or EPROM) chip selects and output enables. Check for transitions on the ROM_D data buffer control signals.

If none of these things help you to find the fault, you probably need to use a logic analyzer to track down the problem. Read Chapter 10 to get an understanding of the operation of the board. Use the debug connectors and pick-up points to attach your logic analyzer.

B.3 Diagnostic Failure

If your board is showing signs of life, but is failing diagnostics, then the first failure message from the diagnostics should provide some clue to the fault. If multiple error messages cause the relevant message to scroll off the screen, try using a virtual terminal with scroll bars (for example, the terminal emulator in Microsoft Windows™).

C
The Design Database

The EBSA-110 Hardware Developers Kit (HDK) includes a number of FAT-format 3.5" floppy disks which provide a (hardware) design database and a firmware database.

These databases are intended to help designers use the EBSA-110 design as a starting point for their own designs.

The hardware design database includes:

- A VIEWlogic® hardware design database
- Simulation test scripts
- A VHDL bus transactor model of the SA-110
- A VHDL model of the synchronous SRAM used in the design
- ABEL source files for the programmable logic
- Layout drawings
- Timing Designer waveforms and libraries
- Simulation waveforms

The firmware database includes source tree and executables for:

- The PBL
- The bootp utility
- The FMU
- The ARM remote debugger stub

Refer to the README files on the floppy disks for more information.

D

SA-110 Bus Transactor Model User's Guide

The SA-110 bus transactor model (BTM) was intended to help hardware designers by providing an easy-to-use tool for generating bus cycles in a simulation environment.

This appendix describes what the BTM is, how to instantiate the BTM into your simulation environment, and how to use it.

Consider this example:

```
set_addr 00400000\h
do_wr 12345678\h
do_idle 1\h
do_rd 12345678\h
set_addr 00400004\h
do_rd aaaa5555\h
do_idle 1\h
sim 1000ns
```

This set of commands can be typed at the simulator command line. When the 'sim' command is given, the BTM performs 2 bus cycles. The first bus cycle is a non-sequential write to address 0x400000. This is followed by a single idle cycle (that is, the next bus cycle follows back-to-back). The second bus cycle is a sequential read of 2-beats, starting at address 0x400000. Because there is no 'do_idle' command between the two 'do_rd' commands, they are performed as a sequential cycle. The BTM mimics the SA-110 worst-case timing and monitors the WAIT input during bus cycles, in order to control the cycle length. The parameters to the 'do_rd' commands tell the BTM to check the data returned during read cycles and report an error if there is a mismatch.

The BTM implements this functionality:

- Performs sequential and non-sequential read and write cycles
- Performs lock cycles
- Allows arbitrary amounts of idle time between cycles
- Provides data checking on reads
- Reports transitions on **fiq**, **irq**, **abort**
- Checks setup and hold times of **abort**
- Reports configuration (bus mode, clock speeds, clock mode) after reset
- Supports standard and enhanced bus modes
- Supports synchronous and asynchronous clocking
- Supports both APE modes (normal and FASTBUS)

D.1 Instantiating the Model

This section assumes that you wish to use the BTM in your own design within a VIEWlogic simulation environment.

The normal way of instantiating the BTM is simply to add the SA-110 symbol to your schematic, and then wire it up.

In order to access the BTM from within your design, you must add an entry to your viewdraw.ini that points to the subdirectory containing the BTM and its associated files.

There are 2 versions of the SA-110 symbol (body). sarm.1 is automatically generated from the VHDL source. It has bussed pins, and no PCB information. To use the command files with this symbol you must attach the label 'sarm\sarm' to the symbol when you instantiate it. sa110.1 is a hand-generated symbol with scalar pins; it includes PCB information like power/ground pins and pin numbers. This symbol is hierarchical and instantiates the sarm.1 symbol via the schematic page sa110.1. To use the command files with this symbol, you must attach the label 'sarm' to the symbol when you instantiate it.

To use the BTM, you need copies of the command files in your project area. Change directory to your project area and then execute the script get_sarm_ command_files (which is in the same subdirectory as the BTM); this will make copies of the command files in your project area.

In order to use the BTM, your simulation ticksize must be set to 10ps, because some of the delays are specified to the resolution of 0.25ns. The simplest way to do this is to create a file in your project area called viewsim.ini and containing the line 'ticksize 10ps'.

The BTM does not model the SA-110 PLL. It expects a clock at the core (multiplied) frequency to be supplied on the CLK (and/or TESTCLK) input. The value of CCCFG is reported after reset, but not used in any other way. MCCFG is modelled, selecting the divisor. TCK_BYP controls whether CLK or TESTCLK generates the MCLK. When SnA is sampled asserted, MCLK and nMCLK are generated and driven as outputs. For odd divisors, MCLK is modelled as low for the longer period, and nMCLK is its Boolean complement. When SnA is sampled negated, MCLK is tristated so that it can be driven *in* to the CPU. nMCLK is driven to X in this mode.

Once you have followed these steps, you should be able to start a simulation. When the BTM comes out of reset it should produce a start-up banner that looks something like this:

```
SARM\SARM\BFM: Resetting all state..
SARM\SARM\BFM:
SARM\SARM\BFM: ------------------------------------------------------------------------
SARM\SARM\BFM:
SARM\SARM\BFM: StrongArm Bus Transactor Model V0.14
SARM\SARM\BFM: Copyright (c) 1996 by Digital Equipment Corporation, Maynard, Ma. USA
SARM\SARM\BFM:
SARM\SARM\BFM: APE is selecting FASTBUS address timing
SARM\SARM\BFM: CONFIG is selecting ENHANCED bus mode (masks/wraps/merges)
SARM\SARM\BFM: SnA is selecting SYNCHRONOUS MCLK (MCLK output)
SARM\SARM\BFM: TCK_BYP is selecting the PLL as the core clock
SARM\SARM\BFM: MCCFG[2:0] is selecting an MCLK divisor of 3
SARM\SARM\BFM: CCCFG[3:0] is selecting a core frequency of 161.9 MHz
SARM\SARM\BFM: SPDF is GROUNDED
SARM\SARM\BFM: ------------------------------------------------------------------------
SARM\SARM\BFM:
```

If you have problems instantiating the BTM, use the EBSA-110 design database as an example of a working environment.

D.2 Command Reference

Ten commands are used to control the bus cycles produced by the BTM.

D.2.1 set_addr {address}

Set a (32-bit) address. The address applies to all subsequent bus operations until another set_addr is performed (but see set_page below, which acts independently).

D.2.2 set_page {offset}

Set a (32-bit) page. The page applies to all subsequent bus operations until another set_page is performed (but see set_addr, which acts independently).

The actual address used for the bus operation is the Boolean OR of the addr and page values. The page will normally be set to 0. There are two situations where it is useful to change the page:

- General-purpose tests: if you write a command script that tests memory, you can write the test to use memory at address 0, using set_addr, and never use set_page within the test. You can then use set_page to set the start address of the test. For example, on the EBSA-110 design, you would use set_page 0 and run the test once (to test the SSRAM) and then use set_page 40000000 and run the test again (to test the DRAM).
- To put don't-care states into addresses: if you are writing a test to verify address decodes, it is often useful to use X (don't-care) states. If you expect that A29:A28 are not required for a decode, you could use the command set_page 00XX00000000000000000000000000\b and then use a number of set_addr commands. This avoids the messiness of having to express every address in binary.

The reason that a Boolean OR is used rather than an ADD, is that an ADD operation prevents you from being able to use X (don't care) states with the same flexibility that an OR provides.

D.2.3 set_bytes {byte masks}, set_size {size}

These commands are mutually exclusive. They are used to set the size mask for read and write cycles.

When the SA-110 is configured with **config** negated (standard bus mode), the set_size command is used, to control the state of the **mas[1:0]** outputs. set_size takes a 2-bit argument, corresponding to the values required on **mas[1:0]** during the access.

When the SA-110 is configured with **config** asserted (enhanced bus mode), the set_bytes command is used to control the state of the **a[1:0]** and **mas[1:0]** outputs. These four signals behave as asserted-low byte enables. set_bytes takes a 4-bit argument, corresponding to the state required on the byte enables during the access. For example, a value of 0xc would correspond to the byte lanes associated with data bits 15:0 being active (since the byte enable pins are active-low).

D.2.4 do_rd {expected read data}

This command is used to enqueue a read cycle at the current address and page, and with the size mask. If the previous enqueued command was a read, then this command will continue from the previous read as a sequential cycle.

When the read cycle completes, the data returned from the system will be checked against the 32-bit parameter {expected read data} and an error message will be generated if there is a mismatch. {expected read data} may include Z (tristate) and X (don't-care) bits.

The BTM always checks all 32 bits of read data, irrespective of the size masks.

D.2.5 do_crd {expected read data}

This command behaves like do_rd, except that the CLF signal is asserted, indicating to the system that the read cycle is a cache line fill.

D.2.6 do_wr {write data}

This command is used to enqueue a write cycle at the current address and page, and with the size mask. If the previous enqueued command was a write, then this command will continue from the previous write as a sequential cycle.

When the write cycle starts, the BTM will drive the 32-bit value of {write data} on the bus. {write data} may include Z (tristate) and X (don't-care) bits.

D.2.7 do_fwr {write data}

This command behaves like do_wr, except that the CLF signal is asserted, indicating to the system that the write cycle is a full write.

D.2.8 do_idle {number of cycles}

This command is used to enqueue an idle bus cycle. The parameter specifies the number of idle cycles to be inserted. The minimum legal value is 1. If a parameter of 0 is specified, the command will be ignored. The main use for do_idle is to prevent two bus cycles from being merged into a sequential cycle.

D.2.9 do_swap {expected read data} {write data}

This command is used to enqueue a swap cycle. A swap cycle is a read cycle followed by a write cycle; the CPU asserts **lock** during the whole sequence.

D.3 How It Works

Each time you type a command to the BTM, you are actually executing a command file; when you type set_addr you are executing the command file set_addr.cmd.

There are command files for each of the 10 commands recognized by the BTM. The command files are divided into two groups:

- Those prefixed with do_ advance the simulation clock (by a very small amount).
- Those prefixed with set_ do not advance the simulation clock.

For example, the set_addr.cmd and do_crd.cmd files are shown below:

```
|| set_addr.cmd
|| usage example: set_addr 12345678\h
a sarm\sarm\cur_mem_a %1
|| end

|| do_crd.cmd
|| usage example: do_crd 12345678\h
|| does read in which clf is asserted. It is up to the user
|| to ensure that a legal combination of these commands is
|| strung together
a sarm\sarm\cur_mem_d %1
a sarm\sarm\cur_mem_op 00\b
a sarm\sarm\cur_mem_clf 1\b
sim 10ps
a sarm\sarm\write_mem_queue 1
defaults -time
sim 10ps
a sarm\sarm\write_mem_queue 0
a sarm\sarm\cur_mem_clf 0\b
defaults time
|| end
```

Within the BTM model there is a circular queue of commands. Each buffer entry holds information for a single clock cycle (for example, do a read, from a certain address, and expect certain data). All of the commands that are prefixed with set_ simply deposit an argument onto ports within the model. The commands that are prefixed with do_ toggle a port within the model that causes a new entry (with appropriate values) to be enqueued. As the simulator clock is advanced, commands are taken off the queue and used to produce bus transactions. When the queue is empty, the BTM becomes idle.

The source code for the BTM is heavily commented and it should be possible to understand it even of you are not familiar with VHDL.

D.4 It Is Not Idiot-Proof!

The BTM allows arbitrary transactions to be built up out of transaction 'atoms'. As such, it allows you to model bus transactions that could not actually occur. In particular, no address checks are done during sequential transactions. Some illegal sequences are detected, and result in an error message and the forced insertion of a stall cycle.

Here are some examples of illegal operations:

Example 1

```
set_addr 0
do_rd 0
set_addr 10000\h
do_rd 12345678\h
do_idle 1\h
```

Here, the address sequence could never occur in a real sequential cycle. The BTM will not detect or report the illegal sequence.

Example 2

```
set_addr 0
do_rd 0
set_addr 4
do_crd 12345678\h
do_idle 1\h
```

Here, the sequential access started as a normal read, with CLF negated, but continued as a cache line fill with CLF asserted. The BTM will not detect or report the illegal sequence.

Example 3

```
set_addr 0
do_crd 0
set_addr 4
do_crd 12345678\h
do_idle 1\h
```

Here, the sequential access was supposed to be a cache line fill, but it was only 2 beats long - a real cache-line fill would be 8 beats long. The BTM will not detect or report the illegal sequence.

Example 4

```
set_addr 0
do_rd 0
set_addr 4
do_wr 12345678\h
do_idle 1\h
```

Here, an attempt has been made to perform a sequential access that is a write for the first beat and a read for the second beat. This is the only type of illegal sequence that the BTM will detect. In these circumstances, the BTM will report an error and will force an idle cycle between the write and the read.

D.4.1 Completeness, Known Bugs and Model Support

The following signals are *not* modelled in the current version: **pwrslp_l**, **tdi**, **tdo**, **tms**, **ntrst**, **spdf**, **cccfg** (but the value of **cccfg** is reported).

Refer to the header of the sarm.vhp file for an up-to-date list of implemented functionality and known bugs.

If you discover a bug or omission in the BTM, please use the contact information in the header of the sarm.vhp file to report it.

D.4.2 Porting, Modifying and Rebuilding

The BTM can be rebuilt by setting default to the behv/ subdirectory of the sarm_ model project area and typing 'make'.

The main source file for the BTM is called sarm.vhp. During the build process, sarm.vhp is processed by the C preprocessor to generate sarm.vhd, which is used as the input to the VHDL compiler. The BTM uses a number of routines from pgk.vhp, which is preprocessed to generate pgk.vhd.

The BTM is written in VIEWlogic's VHDL, but should be portable to other vendors' VHDL environment. For help with porting, refer to the contact information in the header of the sarm.vhp file.

E
ABEL Tutorial

ABEL is a highly structured, industry standard PLD synthesis language. It provides a powerful yet straightforward way to merge state machines with random glue logic. There are tools available to take the high-level description and split the functions into PLDs.

- The AND function is represented by the ampersand (&), and the OR function is represented by the pound sign (#).
- There are two ways to assign an equation to an output. Using just an equal sign (=) assigns a combinatorial output; using a colon followed by an equal sign (:=) assigns a registered output. All registered outputs in this design are edge triggered, clocked on the rising edge of **cpu_mclk_l**.
- A comparison is done by using a double equal sign (==) for true and an exclamation point and an equal sign (!=) for false. So, for example, the equation (DEL_CREQ == READ_BLOCK) compares the signal set DEL_CREQ with the constant value READ_BLOCK, and is considered true if they are the same.
- An equation or signal can be asserted low by adding an exclamation point before it (!). An example of using the inversion operator:

  ```
  DOE_L = SIG1 # (SIG2 & !(SIG3 & SIG4));
  ```

  The equation above inverts the (SIG3 & SIG4) equation before using it in the rest of the right-hand side. The output signal DOE_L is just a name. The fact that it ends with an "_L" does *not* invert it. That signal is still asserted high. In order to cause the output to be asserted low, the inversion operator (!) must precede the output signal, as follows:

  ```
  !DOE_L = SIG1 # (SIG2 & !(SIG3 & SIG4));
  ```

  If the signal is an output pin for the ABEL partition, it can be inverted there as well. Consider this code segment:

  ```
  DECLARATIONS

   !DOE_L               PIN;
   !ACK                 PIN;

  EQUATIONS

   DOE_L = SIG1 # (SIG2 & !(SIG3 & SIG4));
   ACK = 1;
  ```

  The signals DOE_L, ACK are both used as asserted high signals inside the ABEL file, and only inverted when sent out of the partition.
- The state machine CASE dispatch method follows the general form of a decision equation and a dispatch point, separated by a colon (:). There are usually other signals associated with that dispatch, surrounded by WITH/ENDWITH keywords. The WITH/ENDWITH signals are treated

slightly differently, depending upon whether they are edge triggered or combinatorial.

- An edge triggered signal (OUT := IN) surrounded by WITH/ENDWITH and associated with a state dispatch asserts the signal on the next clock edge, effectively changing with the state that is being dispatched to.
- A combinatorial signal (OUT = IN) surrounded by WITH/ENDWITH asserts during the current state. The signal is thus (hopefully) stable by the next clock, to be *used* at the start of the next state.

As an example, consider this ABEL code segment:

```
STATE IDLE:
 CASE
  (DEL_CREQ == READ_BLOCK) : RDMEM_1
   WITH
     DOE_DIS := 1;
     MEM_RAS = 1;
   ENDWITH;
 ENDCASE;
```

The delayed cReq lines are compared to the READ_BLOCK value, and if they are the same the RDMEM_1 state is dispatched to. The signal DOE_DIS is asserted coincidentally with the start of RDMEM_1, and will not become true until then. The signal MEM_RAS, on the other hand, will assert during the IDLE cycle.

F

Getting Started with an Uncased Board

If your EBSA-110 has been supplied as an uncased board, use this section to help you identify a suitable power supply and enclosure.

Before removing the board from its antistatic bag, read the handling precautions in Section 1.2.

Use Appendix A to help you attach the appropriate cables to the board, and Section A.1 to help you to set the jumpers correctly.

Once you have fitted the EBSA-110 into an enclosure and wired it up, use the checklist and power-on sequences described in Section 1.3.1 and Section 1.4.

F.0.1 Choosing a Power Supply

The EBSA-110 requires +5V and +12V power (the power requirements are described in Section 2.4).

Power is supplied to the EBSA-110 board by a 12-way connector which is configured to be suitable for an industry-standard PC power supply. The other signals provided by a PC power supply (DC_OK, -12V, -5V) are not used by the EBSA-110 and are not connected on the board.

Refer to Section A.8.1 for information on how to identify the power connector and align the power supply cable correctly.

Because of the very low current draw on the +12V power rail (only 30mA if no PCMCIA card is fitted) ensure that the power supply has adequate low-load regulation on the +12V power rail. Because PC-style power supplies are designed to provide a high current on +12V, they may have poor low-load regulation. Use a multimeter to verify that the off-load voltage of the +12V supply does not exceed 12.6V. If the low-load regulation is poor, add a dummy resistive load to the +12V power rail. (An automobile light bulb provides a convenient load. Two bulbs in parallel will provide redundancy in case one fails.)

F.0.2 Choosing an Enclosure

The EBSA-110 may be mounted in a standard PC enclosure. Any enclosure that can accommodate a baby-AT format board will probably be suitable. If you intend to use the PCMCIA sockets, you will probably need to modify the enclosure to provide access. Choose an enclosure that makes modification easy.

The board has mounting holes which should align with mounting posts in most enclosures. The mounting hole close to the Ethernet connector serves the secondary purpose of providing a connection from the board 0V to chassis ground. Therefore, a conductive mounting bolt should be used in this position. All other positions can use a plastic mounting clip. The conductive path is required for proper operation of the Ethernet and should minimize electrical and magnetic radiation. This connection does not affect the safety of the board (since the power supply 0V leads are also connected to chassis).

Getting Started with an Uncased Board

If the EBSA-110 is used without an enclosure, it is advisable to put stand-offs in the mounting holes. This will ensure that the bottom of the board is free from the conductive detritus that inevitably accummulates on a laboratory bench. You may also need to fit a wire link in position L1. The bottom of the board contains large copper planes for distributing +3.3V and +2V and you should be careful to make sure that these do not short against anything.

G

Technical Support and Ordering Information

Obtaining Technical Support

If you need technical support or help deciding which literature best meets your needs, call the Digital Semiconductor Information Line:

United States and Canada	**1–800–332–2717**
Outside North America	**+1–508–628–4760**

Ordering Digital Semiconductor Products

To order the following Digital Semiconductor products, contact your local distributor.

To obtain a Digital Semiconductor Product Catalog, contact the Digital Semiconductor Information Line.

Product	Order Number
SA-110 StrongARM microprocessor (100 MHz)	21281-BA
SA-110 StrongARM microprocessor (160 MHz)	21281-AA
SA-110 StrongARM microprocessor (200 MHz)	21281-CA
EBSA-110 Hardware Developer's Kit	QR-21A81-11
ARM Software Developer's Kit - End User License	QR-21B81-01

Note

The EBSA-110 is not available for sale; it is only available under a loan agreement to qualified customers. Contact your Digital Semiconductor distributor for further information.

Ordering Digital Semiconductor Literature

The following table lists some of the available Digital Semiconductor literature. For a complete list, contact the Digital Semiconductor Information Line.

Title	Order Number
EBSA-110 Hardware Developer's Kit Read-Me-First	EC-QU9ZA-TE
Digital Semiconductor SA-110 Microprocessor Tools Brochure	EC-QPWJB-TE
Digital Semiconductor SA-110 Microprocessor Product Brief	EC-QPWKC-TE
Digital Semiconductor SA-110 Microprocessor Technical Reference Manual	EC-QPWLB-TE

Index

A

ABEL tutorial, E–1
Address decoding
  See Memory map
AIF, 8–2
Antistatic precautions, 1–1
Architectural compliance
  overview, 2–6
  test facilities, 3–7, 4–4
Architectural verification
  test facilities, 3–7
Associated literature, G–1

B

bcnt, 10–15
Big-endian, 2–7
Binary notation, xii
8-bit accesses to odd addresses
  Ethernet addresses, 3–11
  hardware implementation, 10–18
  PCMCIA addresses, 3–12
  software techniques, 3–13
bootp utility, 9–4
Buffering, 10–7
Burst counter, 10–15

C

Cables
  external to the box, A–10
  within the box, A–9
Cabling, 1–3
CLKBY state machine, 10–13
Clocks, 10–4
  CLKBY state machine, 10–13
  ctb_clkby7, 3–9, 10–5, 10–11, 10–12, 10–13
  mclk, 10–5
  osc24, 10–5
  overview, 2–1
  SA-110 PLL, 10–4
  setting the frequency, A–5
Configuration, A–1
  clock frequency, A–5
  default, A–1
  links, A–5
Configuration (cont'd)
  of jumpers, A–1
Configuration of memory, 5–1
Configuration of memory space, 5–1
Configuration of VLSI devices, 5–1
Connection to host system, 1–4
Connectors
  description, A–6
Control logic, 10–12
  overview, 2–7
CPU
  See SA-110
CTA, 10–12
  See also control logic
CTB, 10–12
  See also control logic
CTB_ARCH, 4–4
  See also control logic
  distinguishing from CTB_OS, 4–2
  FIQ_CNT, 4–6
  FIQ_MASK, 4–4
  IRQ_CNT, 4–5
  IRQ_MASK, 4–5
CTB_OS, 4–2
  See also control logic
  distinguishing from CTB_ARCH, 4–2
  FIQ_MASK, 4–2
  IRQ_MASK, 4–3
  IRQ_MCLR, 4–3
  IRQ_MSET, 4–3
  IRQ_MSKD, 4–4
  IRQ_RAW, 4–4

D

Debug, B–1
  LED, 3–12
Debug support
  connector pinouts, A–7
  connectors, 10–6
  LED, 1–4
  overview, 2–7
  pickup points, A–8
Decoupling, 10–3
Design database, C–1

Design improvements, 10–22
Diagnostics, 8–4
description of tests, 8–4
example output, 8–7
getting ready to run them, 8–4
Documentation, G–1
do_crd, D–4
do_fwr, D–4
do_idle, D–4
do_rd, D–4
do_swap, D–4
do_wr, D–4
DRAM
CBR (simulation), 11–24
configuration after reset, 5–1
configuration cycles, 10–19
configuration space decodes, 3–5
decodes, 3–4
disabling refresh, 5–2
enabling refresh, 5–3
interface, 10–7
overview, 2–5
refresh (simulation), 11–21
setting wrapping mode, 5–2
telling EDO from BEDO, 5–3
upgrading, A–13
waking up, 5–2
WCBR (simulation), 11–24
DRAM, BEDO
full write(simulation), 11–21
worst-case read (simulation), 11–14
worst-case write (simulation), 11–17
wrapped read (simulation), 11–19
DRAM, EDO
4-beat read (simulation), 11–10
4-beat write (simulation), 11–12
wrapped read (simulation), 11–14
DRAM state machine, 10–12

E

EBUFMEM space, 3–8
Eggs, 2–7
Enclosure
choosing a suitable, F–1
Endian, 2–7
EPROM
overview, 2–4
See also ROM, 2–4
startup, 8–3
Etch links, 10–6
Ethernet
buffer memory access, 3–8
buffer memory bandwidth, 6–4
cable, A–12
configuration, 5–7
control registers, 3–11
interface, 10–10
Ethernet (cont'd)
layout, 10–10
loading images, 9–4
Expansion
hints, 10–21
overview, 2–7

F

Flash
image header format, 8–2
organization, 8–2
overview, 2–4
programming: see FMU, 9–1
programming image into new system, 8–3
See also ROM, 2–4
sequential writes, 3–6
write (simulation), 11–30
FMU, 9–1
block-number parameter, 9–3
NoBoot option, 9–4

G

Gulliver's Travels, 2–7

H

Handling, 1–1
Hardware Developers Kit, C–1
HDK, C–1
Hexadecimal notation, xii
How it works, 10–1

I

I/O
read/write (simulation), 11–30
read/write with refresh (simulation), 11–34
stalled read/write (simulation), 11–34
trickbox read/write (simulation), 11–38
truncated read/write (simulation), 11–34
I/O Space
decodes, 3–6
I/O sub-system
overview, 2–5
Image selection, 8–1
Interrupts, 4–1
overview, 2–7
INTnn notation, xii
IO state machine, 10–13
ISAIO space, 3–8
PIT port-to-address conversion, 3–10
self-decoding, 3–9
ISAMEM Space, 3–7

J

JTAG
  cable, A–12
  overview, 2–7
  port, 10–11
Jumpers, 10–6

L

LEDs
  description, A–8
  overview, 2–8
Literature, G–1
Little-endian, 2–7
Loadable debuggable images, 7–1

M

Main state machine, 10–12
Manufacturers' data sheets, xiii
Manufacturing bring-up process, 8–3
Memory
  See also Performance
  relative speeds, 3–4
  See also DRAM, 2–4
  See also EPROM, 2–4
  See also Flash, 2–4
  See also ROM, 2–4
  See also SSRAM, 2–4
Memory map, 3–1
  after reset, 3–3
  decodes, 3–4, 10–19
  switching, 3–3, 10–18
  switching (simulation), 11–1
  switching overview, 2–5

O

Ordering products, G–1

P

Packer address counter, 10–17
paka, 10–17
Parallel port
  loopback connector, A–12
Parallel port cable, A–11
Parts
  ordering, G–1
PBL, 8–1
PCMCIA
  initialization, 5–4
  interface, 10–11
  memory access, 3–8
  programming voltage, 5–5
  setting a memory window, 5–5
PCMCIAMEM space, 3–8
PCMCIA registers, 3–12
Performance
  impact of DRAM refresh, 6–3
  of BEDO DRAM accesses, 6–2
  of EDO DRAM accesses, 6–1
  of I/O accesses, 6–4
  of overlapping cycles, 6–4
  of SSRAM accesses, 6–1
  overview, 6–1
  ROM accesses, 6–3
Physical description, 1–1
PIT
  configuration, 5–2, 5–3, 5–9
  interface, 10–12
PIT registers, 3–9
Power, 10–3
Power sequencing
  overview, 2–4
Power supply
  choosing a suitable, F–1
  connecting to board, A–9
  overview, 2–3
  +12V regulation, F–1
Principal buses, 10–2
Printed circuit board, 10–22
Programmers' guide, 3–1

R

Related documentation, G–1
Religious persecution, 2–7
Reset, 10–5
  memory map after, 3–3
  overview, 2–3
  state after, 3–13
RFRSH state machine, 10–12
ROM
  decodes, 3–6
  interface, 10–8
  multiple read (simulation), 11–28
  overview, 2–4
  single read (simulation), 11–26
RW_ABORT space, 3–7
R_ABORT space, 3–7

S

SA-110
  bus cycles, 10–14
  configuration options, 10–6
  in socket, 1–3
  overview, 2–1
  types of bus cycle, 10–15
Semiconductor Information Line, G–1
Serial port cable, A–10
  for SUN workstation, A–11

set_addr, D–3
set_masks, D–3
set_page, D–3
set_size, D–3
Simulation
  bus transactor model, D–1
  waveforms, 11–1
Soft register, 3–12
Software
  overview, 2–8
Software development environment, 7–1
Speaker
  connection, A–1
  output, 3–12
SSRAM
  decodes, 3–5
  interface, 10–7
  overview, 2–4
  read sequences(simulation), 11–8
  read wrapping (simulation), 11–6
  worst-case read (simulation), 11–3
  worst-case write (simulation), 11–3
Stall cycles, 6–1
Standalone Flash images, 7–2
Startup EPROM, 8–3
Sub-block wrapping, 10–15
SuperI/O
  configuration, 5–9
  interface, 10–9
SuperI/O registers, 3–10
Support services, G–1
Swift, J. L., 2–7

T

Technical support, G–1
Test points, 10–6
Tour of the schematics, 10–1

V

Visual Inspection, 1–1
Voltage domains
  overview, 2–3
Voltage levels, 10–4

W

Wait states
  See Stall cycles
Word
  use of the term, xii
Wrapping, 10–15