**Technical Manual**

# ARM966E-S Microprocessor Core

**June 2001**

*Preliminary*

LSI LOGIC®

Document DB14-000111-00, First Edition (June 2001)
This document describes LSI Logic Corporation's ARM966E-S Microprocessor Core and will remain the official reference source for all revisions/releases of this product until rescinded by an update.

**To receive product literature, visit us at http://www.lsilogic.com.**

EH

# Preface

This book is the primary reference and Technical Manual for the ARM966E-S Microprocessor core. It contains a complete functional description of the core and describes the main features of the microarchitecture.

## Audience

This document assumes that you have some familiarity with microprocessors and related support devices. The people who benefit from this book are:

- Engineers and managers who are evaluating the processor for possible use in a system
- Engineers who are designing the processor into a system

## Organization

This document has the following chapters:

- Chapter 1, **Introduction**, provides an overview of the ARM966E-S Microprocessor core and the LSI Logic CoreWare® program.
- Chapter 2, **Signal Descriptions**, describes all of the external interface signals.
- Chapter 3, **Programmer's Model**, discusses the memory model and operating modes of the ARM966E-S, and describes its register set.
- Chapter 4, **Exception Processing**, describes the events that cause the ARM966E-S exceptions and discusses how the ARM966E-S handles them.
- Chapter 5, **AHB Interface Unit**, describes the operation of the Advanced High-performance Bus Interface Unit.

- Chapter 6, **Write Buffer**, describes the internal Write Buffer.
- Chapter 7, **System Controller**, describes the internal Coprocessor 15 block, which the CPU uses to control the Write Buffer and Instruction and Data RAMs.
- Chapter 8, **Tightly Coupled RAM**, describes the two built-in RAMs: the Instruction RAM and the Data RAM.
- Chapter 9, **External Coprocessor Interface**, describes the external coprocessor interface through which additional on-chip coprocessors connect.
- Chapter 10, **Debug**, describes the operation of the ARM966E-S debug interface, which is based on the IEEE Std. 1149.1-1990.
- Chapter 11, **Test Methodology**, discusses the available test methods.
- Appendix A, **ARM9E-S Enhanced Instructions**, describes the enhancements to the ARM9E-S instruction set.

## Related Publications

*ARM946E-S Microprocessor Core with Cache Technical Manual,* Document No. DB14-000104-00

*Standard Test Access Port and Boundary Scan Architecture*, IEEE Standard 1149.1-1990

*ARM940T Datasheet*, ARM Ltd., Document No. ARM DDI 0092A-04

*ARM9 Architecture Reference Manual*, ARM Ltd., Document No. ARM DDI 0100

*ARM9E-S Technical Reference Manual*, ARM Ltd., Document No. ARM DDI 0165

*ARM Hurricane Engineering Specification*

*AHB Specification (Rev. 2.0)*, ARM Ltd., Document No. ARM IHI 0011

## Conventions Used in This Manual

The first time a word or phrase is defined in this manual, it is *italicized.*

The word *assert* means to drive a signal true or active. The word *deassert* means to drive a signal false or inactive.

Hexadecimal numbers are indicated by the prefix “0x” —for example, 0x32CF. Binary numbers are indicated by the prefix “0b” —for example, 0b0011.0010.1100.1111.

# Contents

## Figures

## Tables

# Chapter 1 Introduction

This chapter introduces the ARM966E-S microprocessor core. This chapter contains the following sections:

- Section 1.1, “Overview”
- Section 1.2, “Block Diagram Description”
- Section 1.3, “Feature Summary”
- Section 1.4, “CoreWare Program”

## 1.1 Overview

The ARM966E-S microprocessor core is a synthesizable macrocell that integrates the ARM9E-S 32-bit processor, an instruction RAM, a data RAM, a write buffer, and an AHB bus interface.

The ARM966E-S implements the ARM architecture v5T, which supports both the 32-bit ARM and 16-bit Thumb instruction sets, allowing you to trade off between high performance and high code density. Additionally the ARM9E-S processor core provides an ARM9E-S instruction extension and an enhanced multiplier for increased DSP performance.

The AHB bus interface eases connection to cached and SRAM-based memory systems.

The ARM966E-S supports the ARM debug architecture and includes logic to assist in both hardware and software debug. It supports nonstopping hardware debug, which allows critical exception handlers to execute while debugging the system. The ARM966E-S provides real-time trace and supports external coprocessors.

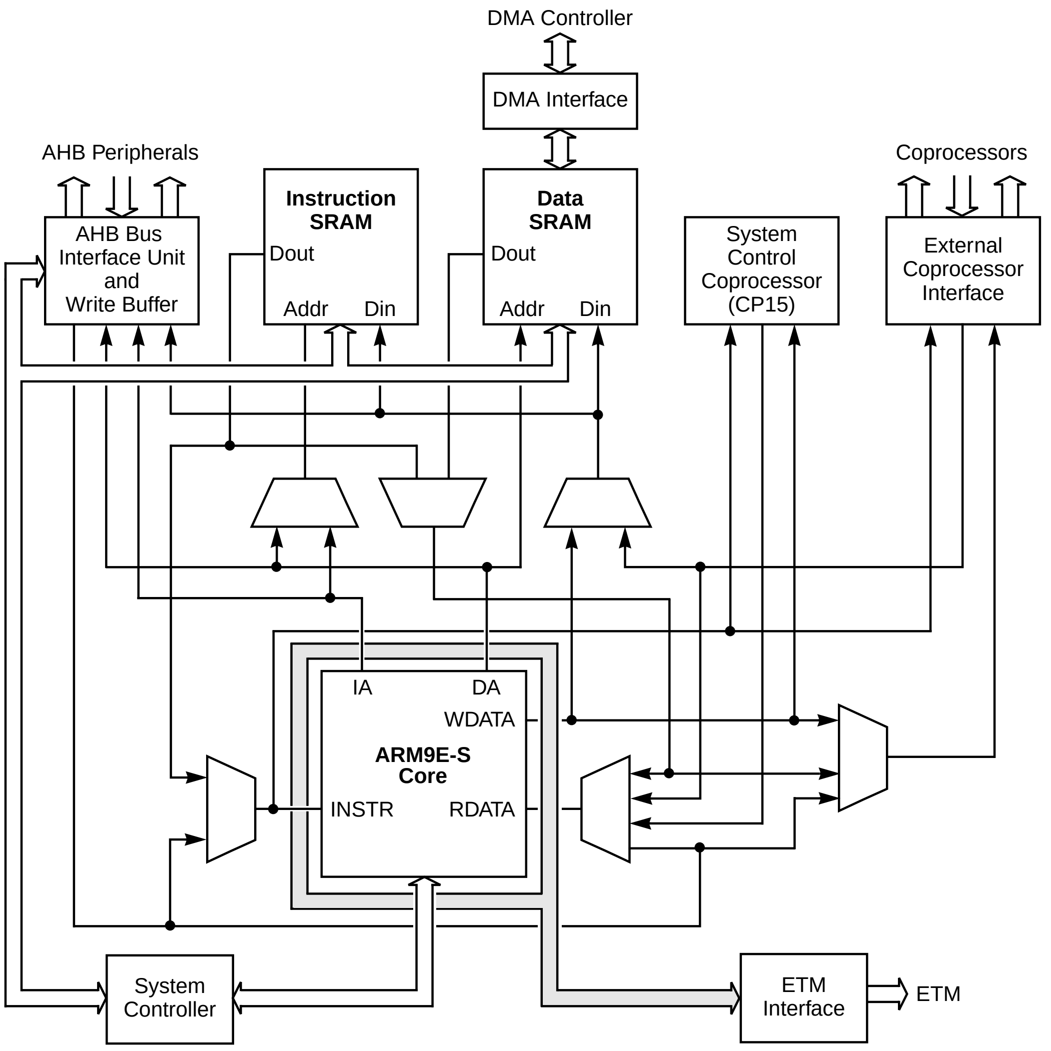
## 1.2 Block Diagram Description

Figure 1.1 shows a block diagram of the ARM966E-S. The main functional blocks are:

- ARM9E-S Processor Core
- System Controller
- CP15 System Control Coprocessor
- Instruction RAM
- Data RAM
- DMA Interface to Data RAM
- Address Decoders
- Write Buffer
- AHB Interface
- External Coprocessor Interface
- JTAG and Debug Port
- Embedded Trace Module Interface

These modules are briefly described following the block diagram. They are described in more detail in the subsequent chapters of this manual.

**Figure 1.1 ARM966E-S Block Diagram**



## 1.2.1 ARM9E-S Processor Core

The ARM9E-S processor core has a Harvard bus architecture with separate instruction and data interfaces. This design allows concurrent instruction and data accesses, and greatly reduces the cycles per instruction of the processor. For optimal performance, single-cycle memory accesses for both interfaces are required, although the core can be stalled for nonsequential accesses or slower memory systems.

The processor is implemented using a five-stage pipeline:

- Instruction Fetch (F)
- Instruction Decode (D)
- Execute (E)
- Data Memory Access (M)
- Register Write (W)

ARM implementations are fully interlocked, so that software functions identically across different implementations without concern for how the pipeline is affected.

### 1.2.2 System Controller

The System Controller oversees the interactions between the Instruction RAM, Data RAM, and the Bus Interface Unit. It controls internal arbitration between the blocks and stalls the appropriate blocks when required.

### 1.2.3 CP15 System Control Coprocessor

The processor core uses a set of registers in the CP15 Coprocessor to control the functionality of the RAMs and the Write Buffer. These registers are accessed using the coprocessor instructions MCR and MRC.

### 1.2.4 Address Decoders

The address decoders determine whether a memory request accesses the internal RAM or the Advanced High-Performance Bus (AHB) interface. The address decoders provide a hit/miss indication to the System Controller, which then either stalls the core if an AHB read or unbuffered write access is required or allows execution to continue if the access hits the RAM or is a buffered write.

### 1.2.5 Instruction and Data RAMs

The ARM966E-S incorporates internal instruction and data memories to allow high-speed operation without incurring the performance penalty of accessing the system bus or the die size penalty of a cached processor.

The Instruction and Data RAMs each consist of blocks of ASIC library compiled RAM. The RAM sizes can be of any size up to 64 Mbytes. The instruction and data memories can have unique sizes.

### 1.2.6 DMA Interface

The Direct Memory Access (DMA) interface allows an external device direct access to the ARM966E-S Data RAM. If a single-port Data RAM is used, then the DMA interface stalls the ARM966E-S microprocessor core during the DMA transfer. If a dual-port Data RAM is used, then the DMA interface does not stall the ARM966E-S during the DMA transfer.

### 1.2.7 AHB Interface Unit and Write Buffer

The AHB is a new generation of AMBA bus, which meets the requirements of high-performance synthesizable designs. The AHB Interface Unit arbitrates between the external bus transaction sources within the ARM966E-S. It stalls all other accesses until the current request has been completed. The AHB Interface Unit supports the following types of transactions: burst transfers, split transactions, single-cycle bus master handovers, single clock edge operations, and non-3-state implementations.

The Write Buffer is a 12-entry FIFO. It increases system performance.

### 1.2.8 External Coprocessor Interface

The ARM966E-S supports the connection of coprocessors through the external coprocessor interface. All types of ARM coprocessor instructions are supported. Coprocessors determine the instructions they need to execute using a pipeline follower in the coprocessor.

### 1.2.9 JTAG and Debug Port

The ARM966E-S debug interface is based on IEEE Std. 1149.1-1990. It can stop the processor core on a given instruction fetch (breakpoint), data access (watchpoint), or external debug request. The JTAG-style serial interface can serially insert instructions into the pipeline of the core without using the external data bus.

### 1.2.10 Embedded Trace Module Interface

This interface connects to an external Embedded Trace Module (ETM). The ETM provides a high-speed port for tracing of the processor core in real time.

## 1.3 Feature Summary

This section lists the key features of the ARM966E-S microprocessor core:

- ARM9E-S processor core
- Instruction and Data RAMs with independent sizes up to 64 Mbytes
- DMA Interface to Data RAM
- ARM Advanced High-Performance Bus (AHB) interface unit with write buffer
  - Write buffer depth: 16 words at up to four addresses
  - Burst transfers
  - Split transactions
- External coprocessor interface
- System controller arbitrates between instruction and data memories and AHB
- Embedded trace module provides a real-time trace capability

## 1.4 CoreWare Program

An LSI Logic core is a fully defined, optimized, and reusable block of logic. It supports industry-standard functions and has predefined timing and layout. The core is also an encrypted RTL simulation model for a wide range of VHDL and Verilog simulators.

The CoreWare library contains an extensive set of complex cores for the storage, communications, consumer, and computer markets. The library consists of high-speed interconnect functions such as the GigaBlaze® G10® Core, DSPs, MPEG-2 decoders, a PCI core, and many more.

The library also includes megafunctions or building blocks, which provide useful functions for developing a system on a chip. Through the CoreWare program, you can create a system on a chip uniquely suited to your applications.

Each core has an associated set of deliverables, including:

- Encrypted RTL or C simulation models for both Verilog and VHDL environments
- A System Verification Environment (SVE) for RTL-based simulation
- Netlists for full timing simulation
- Complete documentation
- LSI Logic FlexStream® design support

The LSI Logic FlexStream design solution provides seamless connectivity between products from leading Electronic Design Automation (EDA) vendors and the LSI Logic manufacturing environment. Standard interfaces for formats and languages such as VHDL, Verilog, Waveform Generation Language (WGL), Physical Design Exchange Format (PDEF), and Standard Delay Format (SDF) allow a wide range of tools to interoperate within the LSI Logic FlexStream design environment. In addition to design capabilities, full scan Automatic Test Pattern Generation (ATPG) tools and LSI Logic's specialized test solutions can be combined to provide high-fault coverage test programs that assure a fully functional design.

Because your design requirements are unique, LSI Logic is flexible in working with you to develop your system-on-a-chip CoreWare design. Three different work relationships are available:

- You provide LSI Logic with a detailed specification and LSI Logic performs all design work.
- You design some functions while LSI Logic provides you with the cores and megafunctions, and LSI Logic completes the integration.
- You perform the entire design and integration, and LSI Logic provides the core and associated deliverables.

Whatever the work relationship, LSI Logic’s advanced CoreWare methodology and ASIC process technologies consistently produce Right-First-Time™ silicon.

# Chapter 2
# Signal Descriptions

This chapter describes the external signals of the ARM966E-S microprocessor core. The descriptions are categorized according to the interface. The signal descriptions are listed alphabetically by mnemonic within each interface.

In the descriptions that follow, the verb *assert* means to drive TRUE or active. The verb *deassert* means to drive FALSE or inactive.

This chapter contains the following sections:

Figure 2.1 provides a signal summary for the ARM966E-S.

**Figure 2.1 ARM966E-S Signal Diagram**

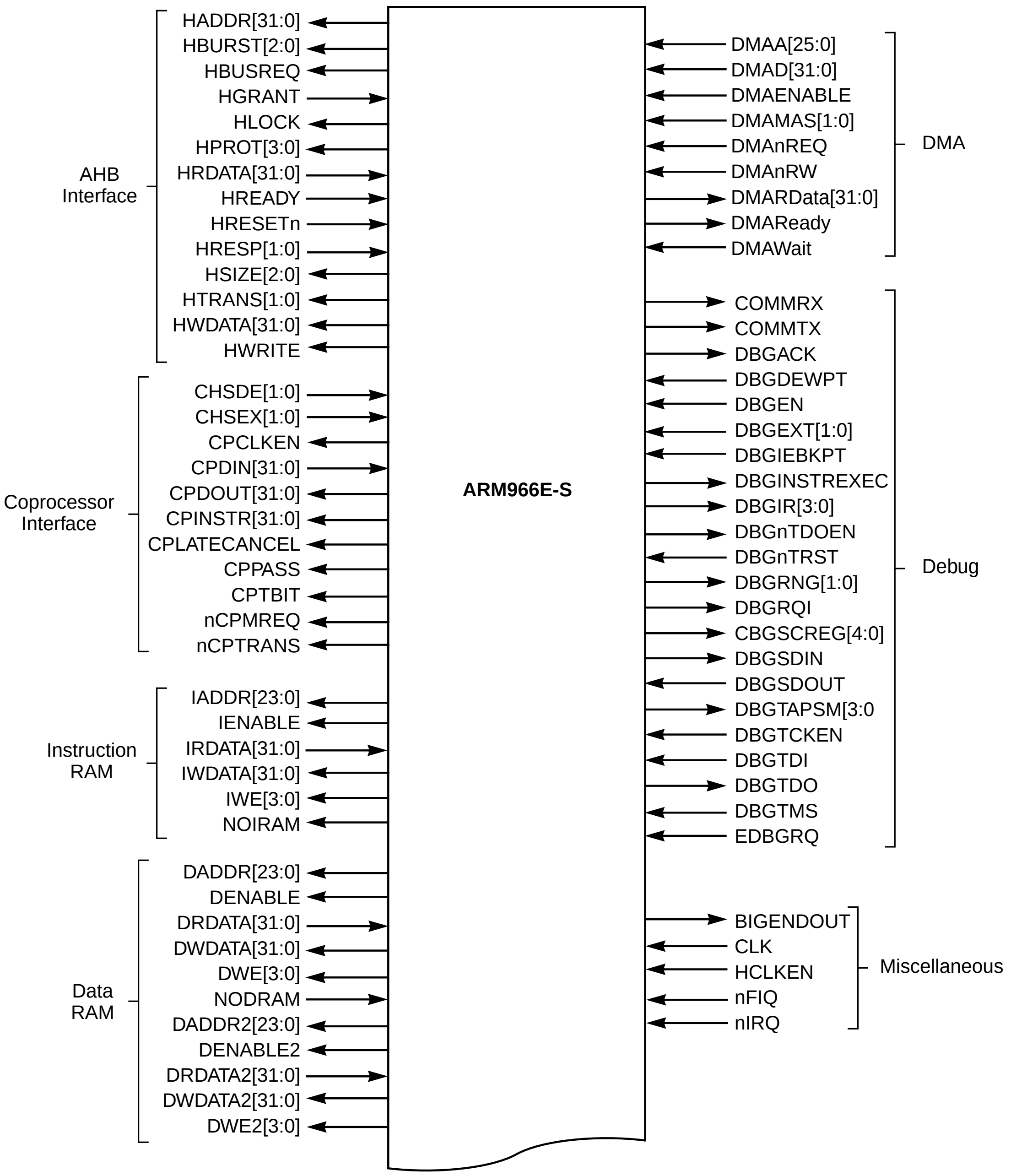
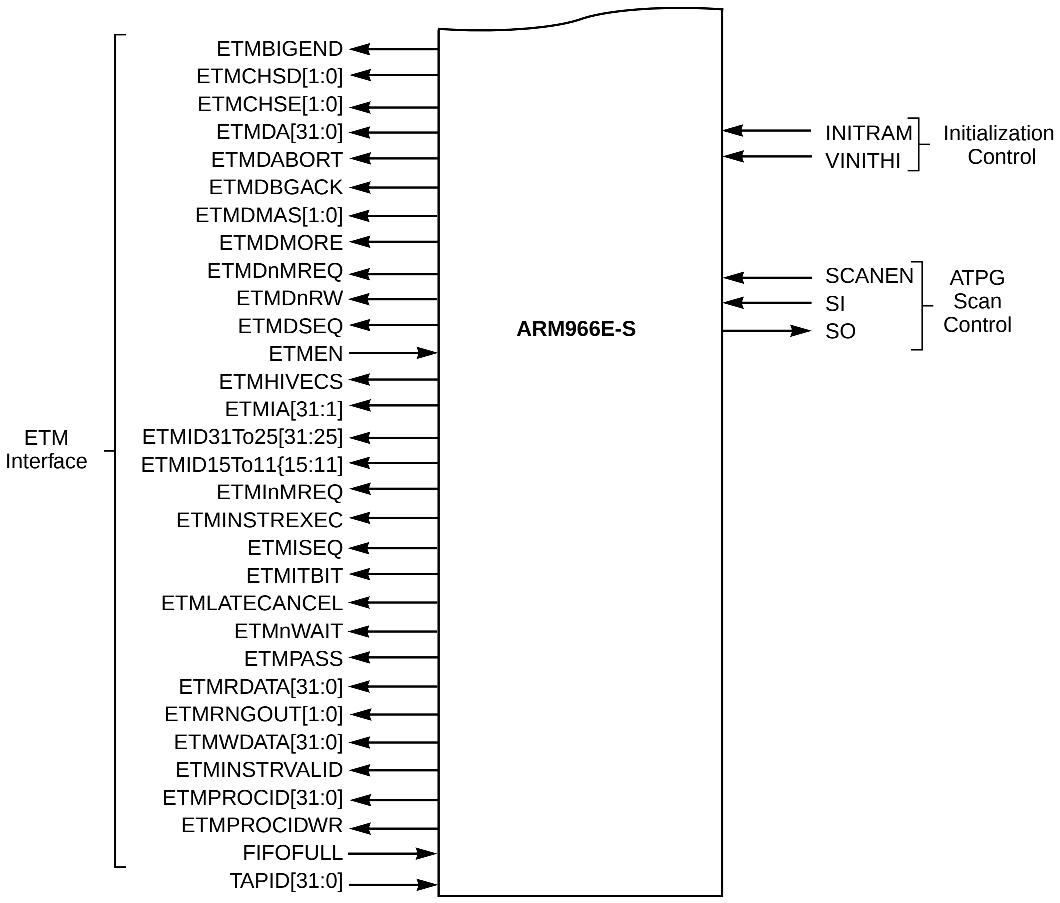


**Figure 2.1 ARM966E-S Signal Diagram (cont.)**



## 2.1 AHB Interface

**HADDR[31:0]** **Address Bus Out** **Output**

The ARM966E-S drives the AHB address on HADDR[31:0].

**HBURST[2:0]** **Burst Type** **Output**

This output indicates whether the transfer forms part of a burst. The ARM966E-S generates burst types of SINGLE and INCR only.

| HBURST[2:0] | Burst Type | Description |
|---|---|---|
| 000 | SINGLE | Single transfer |
| 001 | INCR | Incrementing burst of unspecified length |
| 010–111 | Reserved | Not supported |

**HBUSREQ** **Bus Request** **Output**

HBUSREQ is a signal from the ARM966E-S to the bus arbiter. A HIGH in this output indicates that the core requires the bus.

**HGRANT** **Bus Grant** **Input**

A HIGH on this signal indicates the ARM966E-S is currently the highest priority master. Ownership of the address/control signals changes at the end of a transfer when HREADY is HIGH. A master gets access to the bus when both HREADY and HGRANT are HIGH.

**HLOCK** **Locked Transfer** **Output**

When HLOCK is HIGH, it indicates that the ARM966E-S requires locked access to the bus, and that no other masters should be granted the bus until HLOCK is LOW. HLOCK is asserted when the ARM966E-S is executing the SWAP instruction.

**HPROT[3:0]** **Protection Control** **Output**

This output provides additional information about a bus access. HPROT[3:0] are primarily intended for use by any module that implements some level of protection.

The signals indicate whether the transfer is an opcode fetch or data access, and whether the transfer is a supervisor mode access or user mode access.

| **HPROT3 Cacheable** | **HPROT2 Bufferable** | **HPROT1 Supervisor** | **HPROT0 Data/Opcode** | **Description** |
|---|---|---|---|---|
| – | – | – | 0 | Opcode Fetch |
| – | – | – | 1 | Data Access |
| – | – | 0 | – | User Access |
| – | – | 1 | – | Supervisor Access |
| – | 0 | – | – | Not Bufferable |
| – | 1 | – | – | Bufferable |
| 0 | – | – | – | Not Cacheable |

Note that for the ARM966E-S, HPROT3 is forced LOW (noncacheable).

**HRDATA[31:0]** **Read Data Bus** **Input**
This bus transfers data from the bus slaves to the ARM966E-S during read operations. The ARM966E-S has a 32-bit wide data bus. The width can be easily extended outside the core to allow for higher bandwidth operation.

**HREADY** **Transfer Done** **Input**
When HIGH, the HREADY signal indicates the transfer on the bus has finished. Drive this signal LOW to extend a transfer.

Note: Slaves on the bus require HREADY to be both an input and an output.

**HRESETn** **Reset** **Input**
This input is the active-LOW system reset.

**HRESP[1:0]** **Transfer Response** **Input**
HRESP[1:0] provide additional information on the status of a transfer. When a slave must insert a number of wait

states prior to decoding what response to give, then it must drive the response to OKAY.

| HRESP[1:0] | Transfer Response | Description |
|---|---|---|
| 00 | OKAY | When HREADY is HIGH, the transfer has completed. |
| 01 | ERROR | This response shows an error has occurred. The error condition must be signaled to the bus master so that it is aware the transfer was unsuccessful.<br><br>A two-cycle response is required for an Error condition. |
| 10 | RETRY | The Retry response shows the transfer is not complete, so the bus master should retry the transfer. The master will continue to retry the transfer until it completes.<br><br>A two-cycle RETRY response is required. |
| 11 | SPLIT | The Split response indicates the transfer has not yet completed successfully. The bus master must retry the transfer when it is next granted access to the bus. The slave will request access to the bus on behalf of the master when the transfer can complete.<br><br>A two-cycle SPLIT response is required. |

**HSIZE[2:0]** **Transfer Size** **Output**

This output indicates the size of the transfer, which is typically byte (8 bits), halfword (16 bits), or word (32 bits).

| HSIZE[2:0] | Transfer Size | Description |
|---|---|---|
| 000 | 8 bits | Byte |
| 001 | 16 bits | Halfword |
| 010 | 32 bits | Word |
| 011–111 | Reserved | |

**HTRANS[1:0]** **Transfer Type Out** **Output**

This output indicates the current transfer type.

| HTRANS[1:0] | Transfer Type | Description |
|---|---|---|
| 00 | Idle | No data transfer required. |
| 01 | Busy | Used to insert an idle cycle in the middle of a burst of transfers. |
| 10 | Nonsequential | Indicates first transfer of a burst or single transfer. |
| 11 | Sequential | The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). For wrapping bursts, the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8, or 16). |

**HWDATA[31:0]**

**Write Data Bus** **Output**

This bus transfers data from the master to the bus slaves during write operations. The width can be extended external to the core to allow for higher bandwidth operation.

**HWRITE** **Transfer Direction Out** **Output**

When HWRITE is HIGH, the transfer is a write. When HWRITE is LOW, the transfer is a read.

## 2.2 Coprocessor Interface

**CHSDE[1:0]** **Coprocessor Handshake Decode** **Input**

These inputs are the handshake signals from the decode stage of the coprocessor's pipeline follower.

| CHSDE[1:0] | Encoding |
|---|---|
| 10 | ABSENT |
| 00 | WAIT |
| 01 | GO |
| 11 | LAST |

**CHSEX[1:0]** **Coprocessor Handshake Execute** **Input**

These inputs are the handshake signals from the execute stage of the coprocessor's pipeline follower.

| CHSEX[1:0] | Encoding |
|---|---|
| 10 | ABSENT |
| 00 | WAIT |
| 01 | GO |
| 11 | LAST |

**CPCLKEN** **Coprocessor Clock Enable** **Output**

This clock enable controls the timing of the coprocessor interface. It is used in conjunction with CLK to effectively run the coprocessor at a higher frequency than the data bus.

**CPDIN[31:0]** **Coprocessor Data In** **Input**

This 32-bit bus is the coprocessor data bus for transferring MRC and STC data from the coprocessor to the ARM966E-S.

**CPDOUT[31:0]**

**Coprocessor Data Out** **Output**

This 32-bit bus is the coprocessor data bus for transferring data to the coprocessor.

**CPINSTR[31:0]**

**Coprocessor Instruction** **Output**

This 32-bit bus is the coprocessor instruction data bus for transferring instructions to the pipeline follower in the coprocessor.

**CPLATECANCEL**

**Coprocessor Late Cancel** **Output**

When CPLATECANCEL is HIGH during the first memory cycle of a coprocessor instruction execution, the coprocessor instruction must be cancelled without updating any internal state.

This signal is asserted only in cycles where the previous instruction accessed memory and a data abort occurred.

**CPPASS** **Coprocessor Pass** **Output**

A HIGH on this signal indicates that there is a coprocessor instruction in the execute stage of the pipeline that should be executed.

**CPTBIT** **Coprocessor Interface in Thumb State** **Output**

When CPTBIT is HIGH, the coprocessor interface is in Thumb state (16-bit instructions); otherwise the interface supports 32-bit instruction execution.

**nCPMREQ** **Not Coprocessor Memory Request** **Output**

When nCPMREQ is LOW on a rising CLK edge and CPCLKEN is HIGH, the instruction on CPINSTR must enter the coprocessor pipeline follower's decode stage, and the instruction previously in the pipeline follower's decode stage should enter its execute stage.

**nCPTRANS** **Not Coprocessor Translate** **Output**

When nCPTRANS is LOW, the coprocessor interface is in a nonprivileged state. When nCPTRANS is HIGH, the coprocessor interface is in a privileged state. The coprocessor should sample this signal on every cycle when determining the coprocessor response. Refer to Section 3.4, "Processor Modes," for a description of the privileged and nonprivileged (User) modes.

## 2.3 Instruction RAM Signals

**IADDR[23:0]** **Instruction RAM Address** **Output**
This 24-bit bus contains the Instruction RAM address. Addressing is performed on word boundaries.

**IENABLE** **Word-Based Instruction Chip Enable** **Output**
The ARM966E-S asserts this output HIGH to indicate the Instruction RAM data bus is enabled.

**IRDATA[31:0]** **Instruction RAM Read Data** **Input**
This 32-bit bus contains data read from the Instruction RAM.

**IWDATA[31:0]** **Instruction RAM Write Data** **Output**
This 32-bit bus contains write data for the Instruction RAM.

**IWE[3:0]** **Byte-Based Instruction Write Enable** **Output**
The ARM966E-S asserts these outputs HIGH to enable writes to the Instruction RAM.

| IWE Bit | Function |
|---|---|
| IWE3 | Write enable for IWDATA[31:24] |
| IWE2 | Write enable for IWDATA[23:16] |
| IWE1 | Write enable for IWDATA[15:8] |
| IWE0 | Write enable for IWDATA[7:0] |

**NOIRAM** **Instruction RAM Present** **Input**
The ARM966E-S asserts NOIRAM HIGH to indicate the Instruction RAM is not present and thus Instruction RAM decoding is disabled. NOIRAM asserted LOW indicates Instruction RAM is present, which enables Instruction RAM decoding.

## 2.4 Data RAM Signals

The Data RAM interface supports both single-port and dual-port RAMs. When single-port RAMs are used, the ARM966E-S stalls during DMA transfers. When dual-port RAMs are used, the ARM966E-S does not need to be stalled during DMA transfers.

The ARM966E-S uses the signals described below to access the Data RAM for both single-port and dual-port RAM implementations. The DMA Interface shares these signals with the ARM966E-S for single-port RAM implementations.

**DADDR[23:0]** **Data RAM Address** **Output**
This 24-bit bus contains the Data RAM address. Addressing is performed on word boundaries.

**DENABLE** **Word Based Data Chip Enable** **Output**
The ARM966E-S asserts this output HIGH to indicate the Data RAM data bus is enabled.

**DRDATA[31:0]** **Data RAM Read Data** **Input**
This 32-bit bus contains data read from the Data RAM.

**DWDATA[31:0]** **Data RAM Write Data** **Output**
This 32-bit bus provides write data to the Data RAM.

**DWE[3:0]** **Byte Based Data Write Enable** **Output**
The ARM966E-S asserts these outputs HIGH to enable writes to the Data RAM.

| DWE Bit | Function |
|---|---|
| DWE3 | Write enable for DWDATA[31:24] |
| DWE2 | Write enable for DWDATA[23:16] |
| DWE1 | Write enable for DWDATA[15:8] |
| DWE0 | Write enable for DWDATA[7:0] |

**NODRAM** **Data RAM Present** **Input**
The ARM966E-S asserts NODRAM HIGH to indicate Data RAM is not present and thus Data RAM decoding is disabled. NODRAM asserted LOW indicates Data RAM is present, which enables Data RAM decoding.

The DMA Interface uses the signals described below to access only the second port of a dual-port RAM.

**DADDR2[23:0]**
**Data RAM Address** **Output**
This 24-bit bus contains the Data RAM address. Addressing is performed on word boundaries.

**DENABLE2** **Word-Based Data Chip Enable** **Output**
The ARM966E-S asserts this output HIGH to indicate the Data RAM data bus is enabled.

**DRDATA2[31:0]**
**Data RAM Read Data** **Input**
This 32-bit bus contains data read from the Data RAM.

**DWDATA2[31:0]**
**Data RAM Write Data** **Output**
This 32-bit bus provides write data to the Data RAM.

**DWE2[3:0]** **Byte Based Data Write Enable** **Output**
The ARM966E-S asserts these outputs HIGH to enable writes to the Data RAM.

| DWE2 Bit | Function |
|---|---|
| DWE3 | Write enable for DWDATA[31:24] |
| DWE2 | Write enable for DWDATA[23:16] |
| DWE1 | Write enable for DWDATA[15:8] |
| DWE0 | Write enable for DWDATA[7:0] |

## 2.5 DMA Signals

**DMAA[25:0]** **DMA Address** **Input**
This 26-bit address contains the byte address for DMA transfers. Tie all unused address bits LOW.

**DMAD[31:0]** **DMA Write Data** **Input**
This 32-bit bus contains the DMA write data to the Data RAM.

**DMAENABLE** **DMA Port Enable** **Input**
DMAENABLE must be asserted HIGH for a DMA transfer to proceed. Asserting DMAENABLE LOW can be used to save power when the DMA interface is not being used. Tie DMAENABLE LOW if the DMA Interface is not used in the implementation.

**DMAMAS[1:0] DMA Memory Access Size** **Input**

DMAMAS[1:0] encodes the size of DMA writes. DMA reads are always one word wide.

| DMAMAS[1:0] | Memory Access Size |
|---|---|
| 00 | Byte |
| 01 | Halfword |
| 10 | Word |
| 11 | Reserved |

**DMAnREQ** **DMA Request** **Input**

DMAnREQ is an active-LOW DMA transfer request. Tie this input HIGH if the DMA interface is not used.

**DMAnRW** **DMA Write not Read** **Input**

DMAnRW is the DMA Read/Write signal.

| DMAnRW | Function |
|---|---|
| 0 | Read |
| 1 | Write |

**DMARData[31:0]**

**DMA Read Data** **Output**

This 32-bit bus contains DMA Data read from the Data RAM.

**DMAReady** **DMA Ready** **Output**

DMAReady is asserted HIGH when the ARM966E-S is stalled due to a DMA Wait request. DMAReady must be sampled HIGH before a DMA transfer to/from a single-port Data RAM can take place.

**DMAWait** **DMA Wait Request** **Input**

DMAWait is asserted HIGH to stall the ARM966E-S before proceeding with a DMA transfer to/from a single-port Data RAM implementation. A HIGH on DMAReady indicates when the ARM966E-S is stalled.

Only use this signal for single-port Data RAM implementations. Tie it LOW for dual-port Data RAM implementations.

## 2.6 Debug Signals

**COMMRX** **Communications Channel Receive** **Output**
When HIGH, this signal indicates that the Comms Channel Receive Buffer has data that the ARM9E-S processor core can read.

**COMMTX** **Communications Channel Transmit** **Output**
When HIGH, this signal indicates the Comms Channel Transmit Buffer is empty.

**DBGACK** **Debug Acknowledge** **Output**
When HIGH, DBGACK indicates that the ARM9E-S processor core is in debug mode.

**DBGDEWPT** **Debug Watchpoint** **Input**
This input can halt the processor for debug purposes. If HIGH at the end of a data memory request cycle, this input causes the ARM9E-S processor core to enter the debug state.

**DBGEN** **Debug Enable** **Input**
A LOW on this input disables the debug features of the ARM966E-S. Tie this input LOW when debugging is not required.

**DBGEXT[1:0]** **Breakpoint/Watchpoint External Condition** **Input**
These inputs to the EmbeddedICE logic make breakpoints/watchpoints dependent on external conditions.

**DBGIEBKPT** **Processor Execution Breakpoint** **Input**
When this input is asserted, processor execution is halted for debug purposes. If DBGIEBKPT is HIGH at the end of an instruction fetch, the ARM9E-S processor core enters the debug state if that instruction reaches the execute stage of the processor's pipeline.

**DBGINSTREXEC**
**Instruction Executed** **Output**
When this output is asserted HIGH, the instruction in the execute stage of the processor's pipeline was executed.

| | | |
|---|---|---|
| **DBGIR[3:0]** | **Tap Controller Instruction Register**<br>These outputs reflect the current instruction loaded into the TAP controller instruction register. They change when the TAP state machine is in the UPDATE_IR state on the rising edge of CLK when DBGTCKEN is asserted. | **Output** |
| **DBGnTDOEN** | **DBGTDO 3-State Enable**<br>When LOW, this signal indicates there is serial data on the DBGTDO output. DBGnTDOEN can be used as the output enable on a packaged part's DBGTDO pin. | **Output** |
| **DBGnTRST** | **Not Test Reset**<br>This active-LOW input is the internally synchronized reset signal for the EmbeddedICE internal state. | **Input** |
| **DBGRNG[1:0]** | **Watchpoint Register Match**<br>These outputs indicate that the corresponding EmbeddedICE Watchpoint register has matched the conditions currently present on the address, data, and control buses. These signals are independent of the state of the watchpoint's enable control bit. | **Output** |
| **DBGRQI** | **Internal Debug Request**<br>This signal is the debug request signal presented to the processor core's debug logic. It is the ANDing of EDBGRQ as presented to the ARM966E-S and bit 1 of the Debug Control Register. | **Output** |
| **DBGSCREG[4:0]** | **Scan Chain Register**<br>These outputs reflect the ID number of the scan chain currently selected by the TAP controller. They change when the TAP state machine is in the UPDATE_DP state on the rising edge of CLK when DBGTCKEN is asserted. | **Output** |
| **DBGSDIN** | **Boundary Scan Serial Input Data**<br>This output contains the serial data for an external scan chain. | **Output** |
| **DBGSDOUT** | **Boundary Scan Serial Output Data**<br>DBGSDOUT is the serial data input from an external scan chain. When an external scan chain is not implemented, tie this signal LOW. | **Input** |

| | | |
|---|---|---|
| **DBGTAPSM[3:0]** | **TAP Controller State Machine** | **Output** |
| | This bus reflects the current state of the TAP controller state machine. The TAP controller follows the IEEE 1149.1 Test Access Port protocol. | |
| **DBGTCKEN** | **Test Clock Enable** | **Input** |
| | This input is the synchronous enable for the test clock. | |
| **DBGTDI** | **Test Data In** | **Input** |
| | DBGTDI contains data input from the boundary scan logic. | |
| **DBGTDO** | **Test Data Out** | **Output** |
| | The ARM966E-S outputs test data on DBGTDO from its boundary scan logic. | |
| **DBGTMS** | **Test Mode Select** | **Input** |
| | DBGTMS is the JTAG test mode select signal. The test mode follows the IEEE 1149.1 Test Access Port protocol. | |
| **EDBGRQ** | **External Debug Request** | **Input** |
| | An external debugger asserts this signal to force the processor to enter the debug state. | |

## 2.7 ETM Interface Signals

These signals are part of the Trace module interface. All ETM outputs are registered from the corresponding core internal signals.

| | | |
|---|---|---|
| **ETMBIGEND** | **Endian Mode** | **Output** |
| | When this signal is HIGH, the endian mode is big endian; when ETMBIGEND is LOW, the mode is little endian. | |

**ETMCHSD[1:0]**

**ETM Coprocessor Handshake Decode** **Output**

These outputs are the handshake signals from the decode stage of the coprocessor's pipeline follower.

| ETMCHSD[1:0] | Encoding |
|---|---|
| 10 | ABSENT |
| 00 | WAIT |
| 01 | GO |
| 11 | LAST |

**ETMCHSE[1:0]**

**ETM Coprocessor Handshake Execute** **Output**

These outputs are the handshake signals from the execute stage of the coprocessor's pipeline follower.

| ETMCHSE[1:0] | Encoding |
|---|---|
| 10 | ABSENT |
| 00 | WAIT |
| 01 | GO |
| 11 | LAST |

**ETMDA[31:0]** **ETM Data Address** **Output**

This 32-bit bus contains the ETM data address.

**ETMDABORT** **ETM Data Abort** **Output**

The ARM966E-S asserts this signal to indicate a data abort to the ARM9E-S processor core.

**ETMDBGACK** **ETM Debug Mode Indication** **Output**

When HIGH, this signal indicates that the processor is in the debug state.

**ETMDMAS[1:0]ETM Data Size Indicator** **Output**

These signals indicate the data size of the ETM. They become valid in the same cycle as the data address bus:

| DMAS[1:0] | Transfer Size |
|---|---|
| 00 | Byte |
| 01 | Halfword |
| 10 | Word |
| 11 | Reserved |

| | | |
|---|---|---|
| **ETMDMORE** | **ETM Sequential Data Indication** | **Output** |
| | The ETMDMORE signal is active during load and store multiple instructions and only goes HIGH when ETMDnMREQ is LOW. This signal effectively gives the same information as ETMDSEQ, but a cycle ahead. This information is provided to allow external logic more time to decode sequential cycles. | |
| **ETMDnMREQ** | **ETM Data Memory Request** | **Output** |
| | This signal is asserted HIGH when the ARM966E-S is making a request to the ETM data memory. | |
| **ETMDnRW** | **ETM Data R/W** | **Output** |
| | If this signal is LOW at the end of the cycle, then any data memory access in the following cycle is a read. If this signal is HIGH, then the access is a write. | |
| **ETMDSEQ** | **ETM Sequential Data Indication** | **Output** |
| | If this signal is HIGH at the end of the cycle, then any data memory access in the following cycle is sequential from the last data memory access. | |
| **ETMEN** | **ETM Enable** | **Input** |
| | When this signal is HIGH, the ETM is enabled and the ARM966E-S interface signals are driven out of this module, pipelined by one clock stage. | |
| **ETMHIVECS** | **Exception Vector Location** | **Output** |
| | When this output is LOW, the ARM966E-S exception vectors start at address 0x0000.0000. When this signal is HIGH, the ARM966E-S exception vectors start at address 0xFFFF.0000. This output is a static configuration signal. | |
| **ETMIA[31:1]** | **ETM Instruction Address Bus** | **Output** |
| | This 31-bit bus contains the address for the ETM. | |
| **ETMID31To25[31:25]** | | |
| | **Bits [31:25] of Instruction Data** | **Output** |
| | These outputs reflect the status of bits [31:25] of the instruction data read by the ARM966E-S. | |
| **ETMID15To11[15:11]** | | |
| | **Bits [15:11] of Instruction Data** | **Output** |
| | These outputs reflect the status of bits [15:11] of the instruction data read by the ARM966E-S. | |

**ETMInMREQ** **ETM Instruction Memory Request** **Output**
The ARM966E-S drives this output LOW to indicate that an instruction fetch will take place.

**ETMINSTREXEC**
**ETM Instruction Execute Indicator** **Output**
The ARM966E-S asserts this output HIGH to indicate that the instruction in the execute stage of the processor pipeline has been executed.

**ETMISEQ** **ETM Sequential Instruction** **Output**
The ETMISEQ signal indicates whether the fetch is sequential (HIGH) or nonsequential (LOW) to the previous access.

**ETMITBIT** **ETM Thumb Indication** **Output**
When this signal is LOW, the processor is in ARM state and it fetches 32-bit instructions. When ETMITBIT is HIGH, the processor is in Thumb state and it fetches 16-bit instructions.

**ETMLATECANCEL**
**ETM Coprocessor Late Cancel Indicator** **Output**
If this output is HIGH during the first memory cycle of a coprocessor instruction, then the coprocessor should cancel the instruction without changing any internal state. This signal is only asserted in cycles where the previous instruction accessed memory and a data abort occurred.

**ETMnWAIT** **ETM Clock Stall** **Output**
Driving this output LOW stalls the ETM.

**ETMPASS** **ETM Coprocessor Instruction Execute Indicator**
**Output**
A HIGH on this signal indicates that there is a coprocessor instruction in the execute stage of the pipeline, which should be executed.

**ETMRDATA[31:0]**
**ETM Read Data** **Output**
This 32-bit bus contains ETM read data.

**ETMRNGOUT[1:0]**
**ETM Watchpoint Register Match** **Output**
This output indicates that corresponding EmbeddedICE Watchpoint register has matched the conditions currently

present on the address, data, and control buses. This signal is independent of the state of the watchpoint's enable control bit.

**ETMWDATA[31:0]**

**ETM Write Data** **Output**
This 32-bit bus contains ETM write data.

**ETMINSTRVALID**

**ETM Instruction Valid** **Output**
The ARM966E-S asserts this output HIGH to indicate the current instruction is valid for the ETM.

**ETMPROCID[31:0]**

**ETM Process ID** **Output**
This 32-bit output contains the Process ID for the ETM.

**ETMPROCIDWR]**

**ETM Process ID Write** **Output**
This output is asserted when ETMPROCID is written.

**FIFOFULL** **ETM FIFO FULL** **Input**
This input is asserted when the ETM FIFO is full. Tie this signal LOW if an ETM is not used.

**TAPID[31:0]** **Boundary Scan ID Code** **Input**
This bus specifies the ID Code value shifted out on DBGTDO when the IDCODE instruction enters the TAP Controller.

## 2.8 Miscellaneous Signals

**BIGENDOUT** **Big Endian** **Output**
When this output is HIGH, the ARM966E-S is in big-endian mode (byte 0 is the most-significant bit). When this output is LOW, the ARM966E-S is in little-endian mode.

This output is a static configuration signal. It must remain at one value from reset or be changed using a carefully constructed code sequence to avoid software problems.

**CLK** **System Clock** **Input**
CLK is the ARM966E-S system clock. CLK can be stretched in either state (held HIGH or LOW).

| | | |
|---|---|---|
| **HCLKEN** | **HCLK Enable** | **Input** |
| | HCLKEN is used in conjunction with CLK to effectively run the ARM966E-S at a higher frequency than the AHB system bus. HCLKEN is HIGH for a single CLK period and signifies the rising edge of the AHB clock, HCLK. All AHB outputs transition on the CLK rising edge in which HCLKEN is asserted. | |
| **nFIQ** | **Not Fast Interrupt** | **Input** |
| | This active-LOW input is the ARM Fast interrupt request. The ARM966E-S supports synchronous interrupts only. | |
| **nIRQ** | **Not Interrupt Request** | **Input** |
| | This active-LOW input is the ARM interrupt request. The ARM966E-S supports synchronous interrupts only. | |

## 2.9 Initialization Control Signals

| | | |
|---|---|---|
| **INITRAM** | **RAM Enable Configuration** | **Input** |
| | When INITRAM is HIGH, the Instruction and Data RAMs are enabled at the end of reset. When it is LOW, the Instruction and Data RAMs are disabled coming out of reset. | |
| | This input is a static configuration signal. Its value is sampled at reset only. | |
| **VINITHI** | **High Vectors Configuration** | **Input** |
| | When VINITHI is LOW at reset, the exception vectors start at address 0x0000.0000. When VINITHI is HIGH, the exception vectors start at 0xFFFF.0000. | |
| | This signal is a static configuration signal. Its value is sampled at reset only. | |

## 2.10 ATPG Scan Control Signals

| | | |
|---|---|---|
| **SCANEN** | **Scan Enable** | **Input** |
| | The ARM966E-S asserts this input HIGH to enable data scanning through the scan chain. | |

| | | |
|---|---|---|
| **SI** | **Scan Chain In** | **Input** |
| | SI is the input for the serial scan chain. There can be multiple SI/SO scan pairs. | |
| **SO** | **Scan Chain Out** | **Output** |
| | SO is the output for the serial scan chain. There can be multiple SI/SO scan pairs. | |

# Chapter 3
# Programmer's Model

This chapter describes the programmer's model of the ARM966E-S microprocessor core. This chapter contains the following sections:

## 3.1 About the Programmer's Model

The programmer's model for the ARM966E-S consists of the ARM9E-S programmer's model with some additions. The added features control both the operation of the ARM966E-S internal coprocessors and any coprocessor connected to the external coprocessor interface.

There are two internal coprocessors within the ARM966E-S:

- Coprocessor 14 (CP14) within the ARM9E-S core allows software access to the debug communications channel
- Coprocessor 15 (CP15) allows configuration of the tightly coupled SRAM and write buffer and other ARM966E-S system options, such as big- or little-endian operation.

The CP14 registers are accessible with MCR and MRC instructions. These registers are described in Section 10.9, "The Debug Communications Channel."

The CP15 registers are accessible with MCR and MRC instruction. These registers are defined in Section 3.7.2, “CP15 Registers.”

Any coprocessors and their registers that are attached to the external coprocessor interface are accessible with the appropriate coprocessor instructions.

## 3.2 Data Abort Model

The ARM966E-S implements the *base restored data abort model*, which differs from the *base updated data abort model* implemented by the ARM7TDMI. The difference in the Data Abort model affects only a very small section of operating system code, the Data Abort handler. It does not affect user code.

With the base restored data abort model, when a Data Abort exception occurs during the execution of a memory access instruction, the processor hardware always restores the base register to the value the register contained *before* the instruction was executed. This step removes the requirement for the Data Abort handler to *unwind* any base register update that might have been specified by the aborted instruction.

The base restored data abort model significantly simplifies the software Data Abort handler.

## 3.3 Data Types

The ARM966E-S supports the data types listed in Table 3.1.

**Table 3.1 Supported Data Types**

| **Data Type** | **Size** |
|---|---|
| Byte | 8 bits |
| Halfword | 16 bits (halfwords must be aligned to two-byte boundaries) |
| Word | 32 bits (words must be aligned to four-byte boundaries) |

ARM instructions are exactly one word (32 bits) and are aligned on a four-byte boundary. THUMB instructions are exactly one halfword (16 bits) and are aligned on a two-byte boundary.

All data operations (for example, ADD) are performed on word quantities. Load and store operations can transfer bytes, halfwords, and words to and from memory, automatically zero-extending or sign-extending bytes or halfwords as they are loaded. Signed operands are in two's complement format.

## 3.4 Processor Modes

The ARM966E-S supports seven processor modes. These modes are summarized in Table 3.2.

**Table 3.2 ARM9E-S Processor Modes**

| Processor Mode | Description |
|---|---|
| User (USR) | Normal program execution mode |
| FIQ (FIQ) | High-speed data transfer or channel process |
| IRQ (IRQ) | General-purpose interrupt handling |
| Supervisor (SVC) | Protected mode for the operating system |
| Abort (ABT) | Virtual memory and/or memory protection implementation |
| Undefined (UND) | Software emulation of hardware coprocessors |
| System (SYS) | Privileged operating system tasks (architecture version 4 only) |

Mode changes occur either through software control or as a result of external interrupts or exception processing. Most application programs execute in User mode. The other modes, known as *privileged modes*, are entered to service interrupts or exceptions or to access protected resources.

## 3.5 CP15 Instruction Format

System Control Coprocessor 15 (CP15) controls the operation and configuration of the Instruction RAM, Data RAM, and write buffer. This coprocessor is backward-compatible with the ARM7. All unused and reserved bits should be programmed to zeros.

To read and write the configuration registers, use the MRC and MCR instructions, respectively. These operations are only allowed in nonuser modes; an undefined instruction trap is taken if accesses are attempted in user mode.

Figure 3.1 shows the fields that make up the instruction format.

**Figure 3.1 Coprocessor Instruction Format**

| 31 28 | 27 24 | 23 21 | 20 | 19 16 | 15 12 | 11 8 | 7 5 | 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|
| Cond | 1110 | opcode_1 | n | CRn | Rd | 1111 | opcode_2 | 1 | CRm |

Table 3.3 defines the fields within the coprocessor instruction format.

**Table 3.3 Instruction Format Field Descriptions**

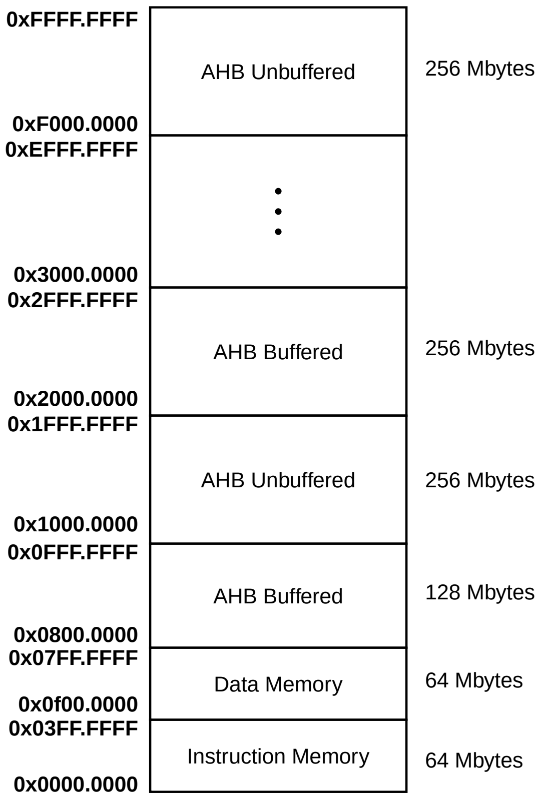
| Field Name | Function |
|---|---|
| Cond | ARM condition code |
| opcode_1 | ARM opcode 1. Opcode_1 is zero for all CP15 instructions. |
| n | Load/store bit<br>0 = MRC (CP15 register read)<br>1 = MCR (CP15 register write) |
| CRn | CP15 Source/Destination register. This field determines which configuration register is being accessed. |
| Rd | ARM CPU register |
| 1111 | Coprocessor number (p15) |
| opcode_2 | ARM opcode 2 |
| CRm | CP15 Operand register |

One example of an MRC register read instruction using the above format is `MRC p15, 0, Rd, c0, c0, 0`, where p15 = 1111, 0 = opcode_1, Rd = ARM CPU register, c0 = CRn, c0 = CRm, and 0 = opcode_2.

## 3.6 Memory Map

Figure 3.2 shows the memory map for the ARM966E-S.

**Figure 3.2 ARM966E-S Memory Map**



## 3.7 Registers

This section describes the CPU and CP15 register sets.

## 3.7.1 CPU Registers

The processor has a total of 37 registers:

- 30 32-bit-wide general-purpose registers
- 6 status registers
- 1 program counter

The registers are arranged in partially overlapping banks. Each of the seven processor modes has a different register bank. At any one time, 15 general-purpose registers (R0 through R14), one or two status registers, and the program counter are visible. The current processor mode determines which general-purpose registers and status registers are currently visible.

Figure 3.3 shows the register bank organization. The banked registers are shaded in the figure.

**Figure 3.3 CPU Register Organization**

| Mode | | | | | |
|---|---|---|---|---|---|
| **User/System** | **Supervisor** | **Abort** | **Undefined** | **Interrupt** | **Fast Interrupt** |
| R0 | R0 | R0 | R0 | R0 | R0 |
| R1 | R1 | R1 | R1 | R1 | R1 |
| R2 | R2 | R2 | R2 | R2 | R2 |
| R3 | R3 | R3 | R3 | R3 | R3 |
| R4 | R4 | R4 | R4 | R4 | R4 |
| R5 | R5 | R5 | R5 | R5 | R5 |
| R6 | R6 | R6 | R6 | R6 | R6 |
| R7 | R7 | R7 | R7 | R7 | R7 |
| R8 | R8 | R8 | R8 | R8 | R8_FIQ |
| R9 | R9 | R9 | R9 | R9 | R9_FIQ |
| R10 | R10 | R10 | R10 | R10 | R10_FIQ |
| R11 | R11 | R11 | R11 | R11 | R11_FIQ |
| R12 | R12 | R12 | R12 | R12 | R12_FIQ |
| R13 | R13_SVC | R13_ABORT | R13_UNDEF | R13_IRQ | R13_FIQ |
| R14 | R14_SVC | R14_ABORT | R14_UNDEF | R14_IRQ | R14_FIQ |
| PC | PC | PC | PC | PC | PC |

| | | | | | |
|---|---|---|---|---|---|
| CPSR | CPSR | CPSR | CPSR | CPSR | CPSR |
| | SPSR_SVC | SPSR_ABORT | SPSR_UNDEF | SPSR_IRQ | SPSR_FIQ |

#### 3.7.1.1 Stack Pointer (SP)

Register 13 is the stack pointer. It is banked across all modes to provide a private stack pointer for each mode (except for system mode which shares the user-mode R13).

#### 3.7.1.2 Link Register (LR)

Register 14 is the Link register. It holds the address of the next instruction after a Branch with Link (BL) instruction, which is the instruction used to make subroutine calls. All other times, R14 can be used as a general-purpose register.

#### 3.7.1.3 Program Counter (PC)

Register 15 is the program counter. It is used in most instructions as a pointer to the instruction that is two instructions after the instruction being executed. Because all ARM instructions are one word long and are always aligned on word boundaries, the bottom two bits of the PC are always zeros. When the PC is read, bits [1:0] are zeros and bits [31:2] contain the PC. When the PC is written, bits [1:0] are ignored and bits [31:2] are written to the PC. Depending on how the PC is used, its value is either the address of the instruction plus 8 or is unpredictable.

#### 3.7.1.4 Banked Registers in FIQ Mode

Registers R8 through R14 in FIQ mode are banked. They provide very fast interrupt processing without the need for preserving register contents by storing them to memory. Values are preserved across interrupt calls so that register contents do not need to be restored from memory.

#### 3.7.1.5 Current Program and Saved Program Status Registers (CPSR and SPSR)

The CPSR is accessible in all processor modes. It contains condition code flags, interrupt enable flags, and the current mode. Each privileged mode (except system mode) has an SPSR, which preserves the value of the CPSR when an exception occurs.

Figure 3.4 shows the format of the CPSR and SPSR.

**Figure 3.4 Program Status Registers (CPSR and SPSR) Format**

| 31 | 30 | 29 | 28 | 27 | 26 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| N | Z | C | V | S | Res | | I | F | T | M4 | M3 | M2 | M1 | M0 |

The N (negative), Z (zero), C (carry), V (overflow), and S (sticky) bits are the condition code flags. The condition code flags in the CPSR can be changed as a result of arithmetic and logical operations in the processor and can be tested by all instructions to determine whether the instruction is to be executed.

The S bit supports the ARM9E instruction extensions. It is set whenever either a saturation occurs during a QADD, QDADD, QSUB, or QDSUB instruction or when the result of an SMLAxx or SMLAWx instruction overflows 32 bits. You must use an MRS instruction to observe the contents of the S flag. The S flag is sticky. Once it is set, then no subsequent instruction will reset it apart from an MSR instruction writing to the CPSR. Refer to Appendix A for descriptions of the ARM9E instruction extensions.

Bits [26:8] are reserved. They read as zeros and may only be written with the same value read from the same field as the processor.

Bits [7:0] are the control bits. They change when an exception arises and can be altered by software only when the processor is in a privileged mode. The I bit disables IRQ interrupts when it is set. When set, the F bit disables FIQ interrupts. When the T flag is zero, it indicates ARM execution; when it is set, it indicates THUMB execution.

The mode bits, M[4:0], determine the mode in which the processor operates. Table 3.4 shows the encoding of these bits. Values not shown in the table are invalid; their results are unpredictable.

**Table 3.4 Mode Bits**

| M[4:0] | Mode | Accessible Registers |
|---|---|---|
| 0b10000 | User | PC, R0 through R14, CPSR |
| 0b10001 | FIQ | PC, R0 through R7, R8_fiq through R14_fiq, CPSR, SPSR_fiq |
| 0b10010 | IRQ | PC, R0 through R12, R13_irq, R14_irq, CPSR, SPSR_irq |
| 0b10011 | SVC | PC, R0 through R12, R13_svc, R14_svc, CPSR, SPSR_svc |
| 0b10111 | Abort | PC, R0 through R12, R13_abt, R14_abt, CPSR, SPSR_abt |
| 0b11011 | Undef | PC, R0 through R12, R13_und, R14_und, CPSR, SPSR_und |
| 0b11111 | System | PC, R0 through R14, CPSR |

User mode and System mode do not have an SPSR, as these modes are not entered on any exception, so a register to preserve the CPSR is not required. In User or System mode, any reads to the SPSR read an unpredictable value, and any writes to the SPSR are ignored.

## 3.7.2 CP15 Registers

Use the MCR and MRC instructions to access the configuration registers; the processor must be in supervisor (privileged) mode.

Never access an invalid CRn register because neither the access nor an undefined instruction trap occurs. An access to a CRn register in user mode causes the undefined instruction trap to be taken.

**Table 3.5 CP15 Register Map**

| Register | Function |
|---|---|
| 0 | ID Code |
| 1 | Control |
| 2–6 | Reserved |
| 7 | Core Control |
| 8–12 | Reserved |

**Table 3.5 CP15 Register Map (Cont.)**

| Register | Function |
|---|---|
| 13 | Trace process identifier |
| 14 | Reserved (undefined instruction) |
| 15 | Test |

### 3.7.2.1 ID Code Register (0)

Register 0 is a read-only register that lists the core identifier. Accessing Register 0 with the `MRC p15, 0, Rd, c0, c0, 0` instruction returns the ID code.

**ID Code Register –** The ID Code Register is a read-only identity register that returns the LSI Logic code for this core. This code is made up of four fields.

| 31 24 | 23 16 | 15 4 | 3 0 |
|---|---|---|---|
| Implementor | Architecture Version | LSI Logic Core ID | Revision |

**Implementor** **[31:24]**
This field contains an eight-bit value unique to LSI Logic Corporation. Its value is 0x41.

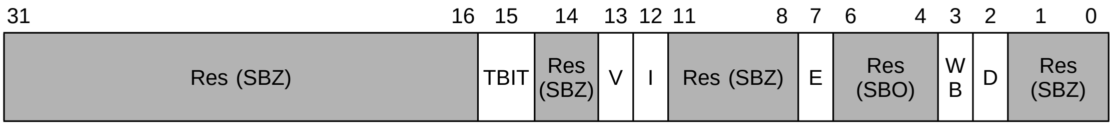
**Architecture Version** **[23:16]**
This field specifies the ARM9E-S architecture. Its value is 0x05 (Version 5T).

**LSI Logic Core ID** **[15:4]**
This field contains a 12-bit value that uniquely identifies this core. Its value is 0x966.

**Revision** **[3:0]**
This four-bit field indicates the revision number. Its value is 0x0.

### 3.7.2.2 Control Register (1)

The Control register is a read/write register that contains the control bits. The reserved bits denoted as SBZ should be written with zeros; the reserved bits denoted as SBO should be written with ones. All bits are cleared to zero on reset unless stated otherwise. The reserved bits have an unpredictable value when read.



**Res (SBZ)** **Reserved** **[31:16], 14, [11:8], [1:0]**
These bits are reserved and must be written as zeros.

**TBIT** **Configure Disable Loading TBIT** **15**
This bit controls the behavior of load PC instructions. When TBIT is cleared, a load to the PC uses bit 0 of the loaded data to control the entry into the THUMB state. When TBIT is set, this behavior is disabled. At reset, this bit is cleared.

**V** **Alternate Vector Select** **13**
This read-only bit controls the base address used for the exception vectors. When V is cleared, the base address for the exception vectors is 0x0000.0000. When V is set, the base address for the exception vectors is 0xFFFF.0000. At reset, this bit takes on the value of the VINITHI pin.

**I** **Instruction Memory Enable Bit** **12**
When this bit is set, all accesses to the Instruction memory space will access the Instruction RAM. When this bit is cleared, all accesses to the Instruction Memory space will access the AHB. At reset, this bit takes on the value of the INITRAM pin.

**E** **Big/Little Endian Bit** **7**
This bit determines the byte-ordering convention. It is cleared during reset.

| E | Endian |
|---|---|
| 0 | Little Endian |
| 1 | Big Endian |

**Res (SBO)** **Reserved** **[6:4]**
These bits are reserved and must be written as ones.

**WB** **Write Buffer Enable** **3**

This bit determines whether the write buffer is enabled or not. At reset, the write buffer is disabled.

| WB | Description |
|---|---|
| 0 | Write Buffer disabled |
| 1 | Write Buffer enabled |

**D** **Data Memory Enable Bit** **2**

This bit determines whether data accesses will access the Data RAM or the AHB. At reset, this bit takes on the value of the INITRAM pin.

| D | Description |
|---|---|
| 0 | Data RAM accessed |
| 1 | AHB accessed |

### 3.7.2.3 Core Control Register (7)

The Core Control register is a write-only register. Writes to this register can drain the write buffer, or cause a wait for interrupts. A read of this register returns an unspecified value. Table 3.6 shows the valid write instructions to this register with their corresponding functions.

**Table 3.6 Core Control Instructions**

| ARM Instruction | Data | Function |
|---|---|---|
| MCR p15, Rd, c7, c10, 4 | SBZ | Drain Write Buffer |
| MCR p15, Rd, c7, c0, 4 | SBZ | Wait For Interrupt |
| MCR p15, Rd, c15, c8, 2 | SBZ | Wait For Interrupt (StrongARM backward compatibility) |

**Drain Write Buffer –** Coprocessor 15 can stall instruction execution until the write buffer is emptied. This operation is useful in real-time applications where the processor needs to be sure that a write to a peripheral has completed before program execution continues. An example is where a peripheral in a bufferable region is the source of an interrupt. Once the interrupt has been serviced, the request must be removed before interrupts can be re-enabled. The processor can be sure of the removal if a drain write-buffer operation separates the store to the peripheral and the enable interrupt functions.

Invoking the Drain Write Buffer instruction stalls the processor core until any outstanding accesses in the write buffer have been completed (all data has been written to memory).

**Wait For Interrupt –** The Wait For Interrupt instructions allow the ARM966E-S to enter a low power standby mode. When the operation is invoked, the internal system clock enable signal (CLKEN) is negated until either an interrupt or a debug request occurs. This function is invoked by a write to Register 7 using the `MCR p15, Rd, c7, c0, 4` instruction. The `MCR p15, 0, Rd, c15, c8, 2` instruction is provided to support older software.

When the Wait For Interrupt instruction is executed, the processor is stalled until nFIQ, nIRQ, or EDBGRQ is asserted. Also if the debugger sets the debug request bit in the EmbeddedICE Control register, then the wait-for-interrupt condition terminates.

When either nFIQ or nIRQ is asserted, the processor “wakes up” regardless of whether the interrupts are enabled or disabled (independent of the I and F bits in the processor’s CPSR). The debug-related awakening only occurs if DBGEN is set (when debug is enabled).

If interrupts are enabled, the ARM966E-S is guaranteed to take the interrupt before executing the instruction after the Wait For Interrupt. If interrupts are disabled, the ARM966E-S restarts execution of the instruction following the Wait For Interrupt. If a debug request is used to wake up the system, then the processor enters the debug state before executing any further instructions.

Wait For Interrupt does not prevent the write buffer from emptying.

#### 3.7.2.4 Trace Process Identifier (13)

This register allows the real-time trace tools to identify the currently executing process in multitasking environments.

The contents of this register are replicated on the ETMPROCID[31:0] outputs of the ARM966E-S. The ETMPROCIDWR signal is asserted HIGH for a single clock cycle whenever a write to this register occurs. Table 3.7 shows the trace process identifier for reads and writes.

**Table 3.7 Register 13, Trace Process Identifier**

| Register | Read | Write |
|---|---|---|
| Trace Process Identifier | MRC p15, 0, Rd, c13, c1, 1 | MCR p15, 0, Rd, c13, c1, 1 |

### 3.7.2.5 CP15 Test (15)

LSI Logic does not implement the ARM test register set. Use the LSI Logic recommended tool to implement RAMBIST in your design.

Writes to the ARM test registers have no effect. Reads from the test registers result in default values of 0.

The CP15 RAMBIST Control Registers provide control of and access to the Instruction and Data RAMs so that memory failures can be isolated or forced. Table 3.8 shows the register map for CP15 Register 15.

**Table 3.8 CP15 RAMBIST Register Map**

| Register | Register Reads | Register Writes |
|---|---|---|
| 1 | BIST Control Register<br>`MRC p15, 1, Rd, c15, c0, 1` | BIST Control Register<br>`MCR p15, 1, Rd, c15, c0, 1` |
| 2 | Instruction BIST Fail Address Register<br>`MRC p15, 1, Rd, c15, c0, 2` | Instruction BIST Start/Pause Address Register<br>`MCR p15, 1, Rd, c15, c0, 2` |
| 3 | Instruction BIST Fail/Pause Read Data Register<br>`MRC p15, 1, Rd, c15, c0, 3` | Instruction BIST Test/Pause Write Data Register<br>`MCR p15, 1, Rd, c15, c0, 3` |
| 4–5 | Reserved (undefined instruction) | Reserved (undefined instruction) |
| 6 | Data BIST Fail Address Register<br>`MRC p15, 1, Rd, c15, c0, 6` | Data BIST Start/Pause Address Register<br>`MCR p15, 1, Rd, c15, c0, 6` |
| 7 | Data BIST Fail/Pause Data Register<br>`MRC p15, 1, Rd, c15, c0, 7` | Data BIST Test/Pause Data Register<br>`MCR p15, 1, Rd, c15, c0, 7` |

**BIST Control Register –** This register controls the operation of the memory BIST. When written, this register controls the Instruction and Data memory BIST wrappers.

| 31 21 | 20 | 19 | 18 | 17 | 16 | 15 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| IBISTSize | I Com | I Fail | I En | I Pause | I Run | DBISTSize | D Com | D Fail | D En | D Pause | D Run |

**IBISTSize** **Instruction BIST Size** **W [31:21]**
**Reserved** **R [31:21]**
This write-only field defines the size of the instruction memory as *n*. IBISTSize in bytes is 4 x $2^n$.

**ICom** **IBIST Complete** **R 20**
**Reserved** **W 20**
A one on this read-only bit indicates the Instruction BIST run has completed. This bit is hardcoded to 0x1, BIST complete.

**IFail** **IBIST Fail** **R 19**
**Reserved** **W 19**
A one on this read-only field indicates the Instruction BIST detected a memory failure. This bit is hardcoded to 0x0.

**IEn** **IBIST Enable** **R/W 18**
Setting this bit enables the IBIST controller to drive the instruction memory address, data, write enable, and chip select signals. This bit must be set before issuing an IBISTRun. It must remain set for the duration of the IBIST test.

**IPause** **IBIST Pause** **R/W 17**
Setting this bit pauses the instruction BIST operation. This facility is useful for inserting errors and isolating failures.

**IRun** **IBISTRunning** **R 16**
**IBISTRun** **W 16**
When reading this bit, a one indicates the instruction BIST is running. If IPause is set, then IBISTRunning is deasserted when BIST execution is paused. This bit is hardcoded to 0x0, BIST not running.

**DBISTSize** **Data BIST Size** **W [15:5]**
**Reserved** **R [15:5]**
This write-only field defines the size of the data memory as *n*. DBISTSize in bytes is 4 x $2^n$.

**DCom** **DBIST Complete** **R 4**
**Reserved** **W 4**
A one on this read-only bit indicates the data BIST run has completed. This bit is hardcoded to 0x1, BIST complete.

**DFail** **DBIST Fail** **R 3**
**Reserved** **W 3**
A one on this read-only field indicates the data BIST detected a memory failure. This bit is hardcoded to 0x0.

**DEn** **DBIST Enable** **R/W 2**
Setting this bit enables the DBIST controller to drive the data memory address, data, write enable, and chip select signals. This bit must be set before issuing a DBISTRun. It must remain set for the duration of the DBIST test.

**DPause** **DBIST Pause** **R/W 1**
Setting this bit pauses the data BIST operation. This facility is useful for inserting errors and isolating failures.

**DRun** **DBISTRunning** **R 0**
**DBISTRun** **W 0**
When reading this bit, a one indicates the data BIST is running. If DPause is set, then DBISTRunning is deasserted when BIST execution is paused. This bit is hardcoded to 0x0, BIST not running.

**Instruction BIST Start/Pause Address Register –** The function of this write-only register depends on the setting of the IPause bit in the CP15 BIST Control register.

| 31 26 | 25 0 |
|---|---|
| Reserved | IBIST Start Address/IBIST Pause Address |

When the Instruction BIST Pause bit is cleared, a write to this register determines the Instruction BIST Start Address. BIST testing is executed on instruction memory from the IBIST Start Address to an address that is a function of the IBIST size with respect to the Start Address. Normally this register is cleared to allow testing of the full instruction memory. However, it can be set to a nonzero value to limit testing to only a portion of instruction memory.

When the Instruction BIST Pause bit is set, a write to this register sets the Instruction Pause Address. When BIST testing is paused, the Pause address is held at the instruction memory RAM address input.

**CP15 Instruction BIST Test/Pause Write Data Register –** The function of this write-only register depends on the setting of the IPause bit in the CP15 BIST Control register.

| 31 0 |
|---|
| IBIST Test Data/IBIST Pause Write Data |

When the Instruction BIST Pause bit is cleared, a write to this register sets the Instruction BIST Data pattern (IBISTTestData). The BIST algorithm uses IBISTTestData and its inverse pattern for instruction memory BIST testing.

When the Instruction BIST Pause bit is set and BIST operation is paused, data written to this register is placed in the instruction memory at the IBIST Pause Address.

**CP15 Instruction BIST Fail Address Register –** This read-only register contains the address of the first detected instruction memory failure. This register is valid when the Instruction RAM BIST Fail bit is set in the CP15 BIST Control register.

Because ARM BIST was not implemented, this register is hardcoded to 0x00000000.

| 31 26 | 25 0 |
|---|---|
| Reserved | IBIST Fail Address |

**CP15 Instruction BIST Fail/Pause Read Data Register –** The function of this read-only register depends on the setting of the IPause bit in the CP15 BIST Control register.

Because ARM BIST was not implemented, this register has been hardcoded to 0x00000000.

| 31 … 0 |
|---|
| IBIST Fail Data/IBIST Pause Read Data |

**CP15 Data BIST Start/Pause Address Register –** The function of this write-only register depends on the setting of the DPause bit in the CP15 BIST Control register.

| 31 … 26 | 25 … 0 |
|---|---|
| Reserved | DBIST Start Address/DBIST Pause Address |

When the Data BIST Pause bit is cleared, a write to this register determines the Data BIST Start Address. BIST testing is executed on data memory from DBIST Start Address to a function of the DBIST size with respect to the Start Address. Normally, this register is cleared to allow testing of the full data memory. However, it can be set to a nonzero value to limit testing to only a portion of data memory.

When the Data BIST Pause bit is set, a write to this register sets the Data Pause Address. When BIST testing is paused, the Pause address is held at the data memory RAM address inputs.

**CP15 Data BIST Test/Pause Write Data Register –** The function of this write-only register depends on the setting of the DPause bit in the CP15 BIST Control register.

| 31 … 0 |
|---|
| DBIST Test Data/DBIST Pause Write Data |

When the Data BIST Pause bit is cleared, a write to this register sets the Data BIST Data pattern (DBISTTestData). The BIST algorithm uses DBISTTestData and its inverse pattern for data memory BIST testing.

When the Data BIST Pause bit is set and BIST operation is paused, data written to this register is placed in the data memory at the DBIST Pause Address.

**CP15 Data BIST Fail Address Register –** This read-only register contains the address of the first detected data memory failure. This register is valid when the Data RAM BIST Fail bit is set in the CP15 BIST Control register.

Because ARM BIST is not implemented, this register is hardcoded to 0x00000000.

| 31–26 | 25–0 |
|---|---|
| Reserved | DBIST Fail Address |

**CP15 Data BIST Fail/Pause Read Data Register –** The function of this read-only register depends on the setting of the DPause bit in the CP15 BIST Control register.

Because ARM BIST is not implemented, this register is hardcoded to 0x00000000.

| 31–0 |
|---|
| DBIST Fail Data/DBIST Pause Read Data |

# Chapter 4
# Exception Processing

This section describes all events that cause exceptions, defines the types of exceptions, and explains how they are resolved.

## 4.1 Overview

Internal and external sources generate exceptions, which cause the processor to handle an exceptional event, such as a hardware interrupt or an attempt to execute an undefined instruction. The processor state just before handling the exception must be preserved so that the original program can be resumed when the exception routine has completed. More than one exception may arise at the same time.

The ARM966E-S supports seven types of exceptions; each type of exception has a privileged processor mode. Table 4.1 shows the types of exceptions with their corresponding processor mode that processes that exception. When an exception occurs, execution is forced to an exception vector location.

**Table 4.1 Exception Processing Modes**

| Exception Type | Mode | Vector Address |
|---|---|---|
| Reset | SVC | 0x0000.0000 |
| Undefined Instructions | UNDEF | 0x0000.0004 |
| Software Interrupt (SWI) | SVC | 0x0000.0008 |
| Prefetch Abort (Instruction fetch memory abort) | ABORT | 0x0000.000C |
| Data Abort (Data Access memory abort) | ABORT | 0x0000.0010 |
| IRQ (Interrupt) | IRQ | 0x0000.0018 |
| FIQ (Fast Interrupt) | FIQ | 0x0000.001C |

Table 4.2 lists exceptions from highest to lowest priority. While more than one exception can occur for a single instruction, only the exception with the highest priority is reported.

**Table 4.2 Exception Priority Order**

| Exception | Priority |
|---|---|
| Reset | 1 (Highest) |
| Data Abort | 2 |
| FIQ | 3 |
| IRQ | 4 |
| Prefetch Abort | 5 |
| BKPT, Undefined Instruction, SWI | 6 (Lowest) |

## 4.2 Exception Flow

When an exception occurs, some of the standard registers are replaced with registers specific to the exception mode. All exceptions have banked registers for R14 and R13. One interrupt mode has more banked registers for fast interrupt processing.

After an exception, R14 holds the return address for exception processing. This address is used to return after the exception is processed and to address the instruction that caused the exception.

R13 is banked across exception modes to provide each exception handler with a private stack pointer (SP). The fast interrupt mode also banks registers 8 through 12, so that interrupt processing can begin without the need to save or restore these registers. The system mode does not have any banked registers; it uses the User-mode registers. System mode is used to run normal (nonexception) tasks that require a privileged processor mode.

All other processor states are held in the status registers: CPSR and SPSR. The current operating processor status is in the Current Program Status Register (CPSR). The CPSR holds four condition code flags (N, Z, C, and V), two interrupt disable bits (IRQ and FIQ), and five bits that encode the current processor mode.

All exception modes except for User and System have a Saved Program Status Register (SPSR), which holds the CPSR of the task immediately before the exception occurred. Both the CPSR and SPSR are accessed with special instructions.

When an exception occurs, the ARM9E-S processor core halts execution after the current instruction and begins execution at the fixed address in low memory, pointed at by the exception vectors. Each exception has a separate vector location. Memory aborts have two vector locations to distinguish between data and instruction accesses.

At initialization, the operating system installs a handler on every exception. Privileged operating system tasks are normally run in System mode to allow exceptions to occur within the operating system without state loss. Exceptions overwrite their R14 when an exception occurs, and System mode is the only privileged mode that cannot be entered by an exception.

## 4.3 Exception Descriptions

This section provides detailed descriptions of the seven exception types. It describes the cause of each exception and describes how each exception is processed.

### 4.3.1 Reset Exception

**Cause –** The Reset exception occurs when the processor reset signal is asserted.

**Processing –** The ARM9E-S processor core immediately stops execution of the current instruction. When the reset is deasserted, the following actions occur:

- R14_svc is loaded with an unpredictable value
- SPSR_svc is loaded with the contents of CPSR
- CPSR[5:0] are set to 0b010011 (supervisor mode)
- CPSR6 is set to one (fast interrupts are disabled)
- CPSR7 is set to one (normal interrupts are disabled)
- the program counter (PC) is loaded with the reset vector (0x0)

### 4.3.2 Undefined Instruction Execution

**Cause –** When the ARM9E-S executes a coprocessor instruction, it waits for any external coprocessor to acknowledge that it can execute the instruction. If no coprocessor responds, an undefined instruction exception occurs. If an attempt is made to execute an instruction that is undefined, an undefined instruction exception occurs.

This exception can be used for software emulation of a coprocessor in a system that does not have the physical coprocessor or for general-purpose instruction set extension by software emulation.

**Processing –** When an undefined instruction exception occurs, the following actions occur:

- R14_und is loaded with the address of the undefined instruction + 4
- SPSR_und is loaded with the contents of CPSR

- CPSR[5:0] are set to 0b011011 (undefined mode)
- CPSR6 remains unchanged (fast interrupt status is unchanged)
- CPSR7 is set to one (normal interrupts are disabled)
- The program counter (PC) is loaded with the undefined instruction exception vector (0x4)

To return after emulating the undefined instruction, use `MOVS PC, R14`. This instruction restores the PC (from R14_und) and CPSR (from SPSR_und) and returns to the instruction following the undefined instruction.

### 4.3.3 Software Interrupt Exception

**Cause –** The software interrupt instruction (SWI) causes a software interrupt exception. The ARM9E-S processor core enters Supervisor mode to request a particular supervisor (operating system) function.

**Processing –** When an SWI is executed, the following actions occur:

- R14_svc is loaded with the address of the SWI instruction + 4
- SPSR_svc is loaded with the contents of CPSR
- CPSR[5:0] are set to 0b010011 (supervisor mode)
- CPSR6 remains unchanged (fast interrupt status is unchanged)
- CPSR7 is set to one (normal interrupts are disabled)
- The program counter (PC) is loaded with the software interrupt exception vector (0x8)

To return after performing the SWI operation, use `MOVS PC, R14`. This instruction restores the PC (from (R14_svc) and CPSR (from SPSR_svc) and returns to the instruction following the SWI.

### 4.3.4 Prefetch Abort (Instruction Fetch Memory Abort)

**Cause –** The memory system indicates a memory abort. Activating an abort in response to an instruction fetch marks the fetched instruction as invalid. An abort takes place if the processor attempts to execute the invalid instruction. If the instruction is not executed (for example, as a result of a branch being taken while it is in the pipeline), no prefetch abort occurs.

**Processing –** When an attempt is made to execute an aborted instruction, the following actions occur:

- R14_abt is loaded with the address of the aborted instruction + 4
- SPSR_abt is loaded with the contents of CPSR
- CPSR[5:0] are set to 0b010111 (abort mode)
- CPSR6 remains unchanged (fast interrupt status is unchanged)
- CPSR7 is set to one (normal interrupts are disabled)
- The program counter (PC) is loaded with the prefetch abort exception vector (0xC)

To return after fixing the reason for the abort, use `SUBS PC, R14, #4`. This instruction restores both the PC (from R14_abt) and CPSR (from SPSR_abt) and returns to the aborted instruction.

### 4.3.5 Data Abort (Data Access Memory Abort)

**Cause –** The memory system indicates a memory abort. Activating an abort in response to a data access (load or store) marks the data as invalid. A data abort exception occurs before any subsequent instructions or exceptions have altered the state of the processor core.

**Processing –** The following actions occur:

- R14_abt is loaded with the address of the aborted instruction + 8
- SPSR_abt is loaded with the contents of CPSR
- CPSR[5:0] are set to 0b010111 (abort mode)
- CPSR6 remains unchanged (fast interrupt status is unchanged)
- CPSR7 is set to one (normal interrupts are disabled)
- The program counter (PC) is loaded with the data abort exception vector (0x10)

To return after fixing the reason for the abort, use `SUBS PC, R14, #8`. This instruction restores both the PC (from R14_abt) and CPSR (from SPSR_abt) and returns to re-execute the aborted instruction.

If the aborted instruction does not need to be re-executed, use `SUBS PC, R14, #4`. For memory access instructions that specify writebacks and generate data aborts (LDR, LDRH, LDRSH, LDRB, LDRSB, STR, STRH, STRB, LDM, STM, LDC, STC), the final value that these instructions leave in the base register is implementation-defined.

### 4.3.6 IRQ (Interrupt Request) Exception

**Cause –** The IRQ exception is generated when the nIRQ input is asserted. It has a lower priority than the FIQ exception and is masked out when an FIQ sequence is entered. Interrupts are disabled when the I bit in the CPSR is set (note that the I bit can only be changed from a privileged mode). If the I flag is cleared, the ARM9E-S processor core checks for an IRQ at instruction boundaries.

**Processing –** When an IRQ is detected, the following actions occur:

- R14_irq is loaded with the address of the next instruction to be executed + 4
- SPSR_irq is loaded with the contents of CPSR
- CPSR[5:0] are set to 0b010010 (interrupt mode)
- CPSR6 remains unchanged (fast interrupt status is unchanged)
- CPSR7 is set to one (normal interrupts are disabled)
- The program counter (PC) is loaded with the interrupt request exception vector (0x18)

To return after servicing the interrupt, use `SUBS PC, R14, #4`. This instruction restores both the PC (from R14_irq) and CPSR (from SPSR_irq) and resumes execution of the interrupted code.

### 4.3.7 FIQ (Fast Interrupt Request) Exception

**Cause –** The FIQ exception is generated when the nFIQ input is asserted. FIQ supports a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications (thus minimizing the overhead of context switching).

Fast interrupts are disabled when the F bit in the CPSR is set (note that the F bit can only be altered from a privileged mode). If the F flag is cleared, the ARM9E-S processor core checks for an FIQ at instruction boundaries.

**Processing –** When an FIQ is detected, the following actions occur:

- R14_fiq is loaded with the address of the next instruction to be executed + 4
- SPSR_fiq is loaded with the contents of CPSR
- CPSR[5:0] are set to 0b010001 (FIQ mode)
- CPSR6 is set to one (fast interrupt status is disabled)
- CPSR7 is set to one (normal interrupts are disabled)
- The program counter (PC) is loaded with the interrupt request exception vector (0x1C)

To return after servicing the interrupt, use `SUBS PC, R14, #4`. This instruction restores both the PC (from R14_fiq) and CPSR (from SPSR_fiq) and resumes execution of the interrupted code.

The FIQ vector is intentionally the last vector to allow the FIQ exception-handler software to be placed directly at address 0x1C and not require a branch instruction from the vector.

# Chapter 5
# AHB Interface Unit

This chapter describes the operation of the Advanced High-Performance Bus (AHB) Interface Unit. It contains the following sections:

## 5.1 Overview

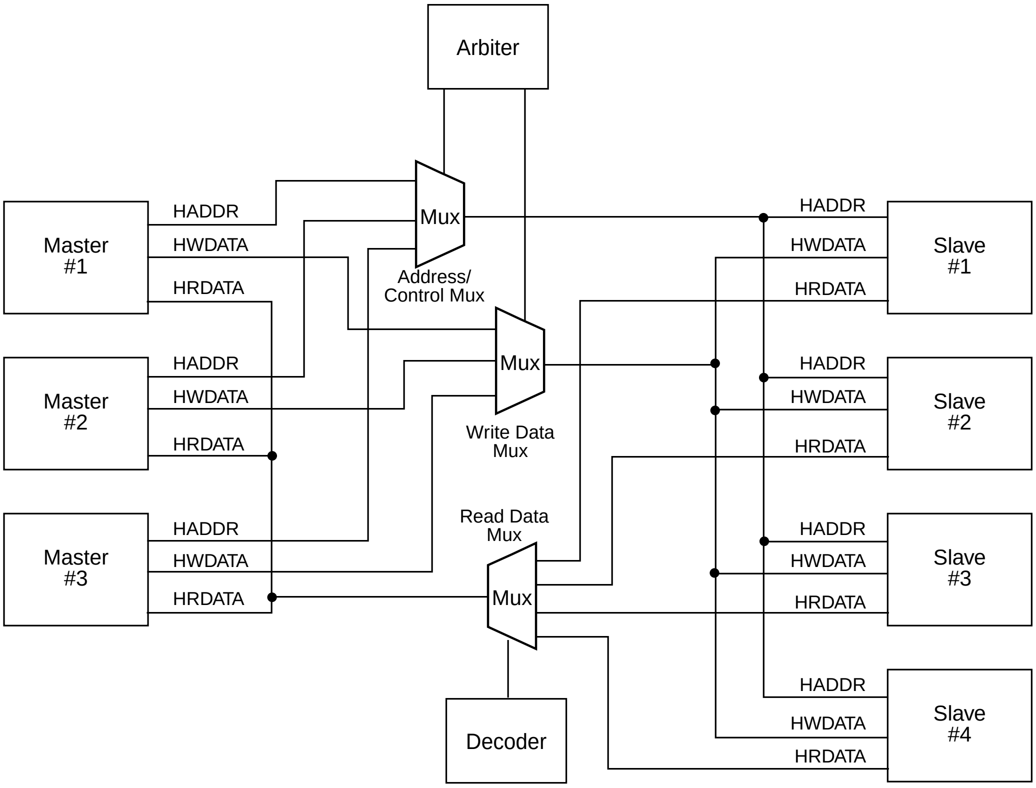
The AHB is a high-performance, burst-based bus protocol that complements the lower level AMBA Advanced System Bus (ASB) and APB protocols. The AHB bus supplies the needed requirements for high-performance/high clock frequency systems. This bus can handle the following:

- burst transfers
- single-cycle bus master handover
- single clock edge operation
- non-3-state implementation

The AHB bus protocol is used with a central multiplexer interconnection scheme. From this scheme, all bus masters drive out the address and control signals indicating the transfer they wish to perform. The arbiter determines which master has its address/control signals routed to all of the slaves. A central decoder is also required to control the read data and response signal multiplexer, which selects the signals from the slave that is involved in the transfer.

Figure 5.1 shows the structure that is required to implement an AHB design with three masters and four slaves.

**Figure 5.1 Multiplexer Interconnection**



## 5.2 AHB Interface Signals

This section describes the signals that determine the transfer type and burst type of each AHB transfer. It also describes the control signals that provide additional information on the transfers.

### 5.2.1 Transfer Types

Every transfer is classified into one of four different types, as determined by the HTRANS[1:0] signals.

**Table 5.1 Transfer Type Encoding**

| HTRANS[1:0] | Transfer Type | Description |
|---|---|---|
| 00 | IDLE | Idle transfer is used when a bus master is granted the bus, but does not transfer data. |
| 01 | BUSY | Busy transfer is used to insert an idle cycle in the middle of a burst of transfers. |
| 10 | NONSEQ | Nonsequential transfer indicates the first transfer of a burst or a single transfer. |
| 11 | SEQ | Sequential transfer is used for subsequent transfers in a burst. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the size (in bytes). For wrapping bursts, the address of the transfer wraps at the address boundary equal to the size (in bytes) multiplied by the number of beats in the transfer (either 4, 8, or 16). |

The first transfer of any burst is nonsequential; the address is unrelated to the previous transfer. The remaining transfers in a burst are sequential; the address is related to the previous transfer. Single transfers on the bus are treated as bursts of one, and thus the transfer type is nonsequential.

The IDLE transfer type is used when a bus master is granted the bus, but does not wish to perform a data transfer. Slaves must provide a zero wait state OKAY response to IDLE transfers.

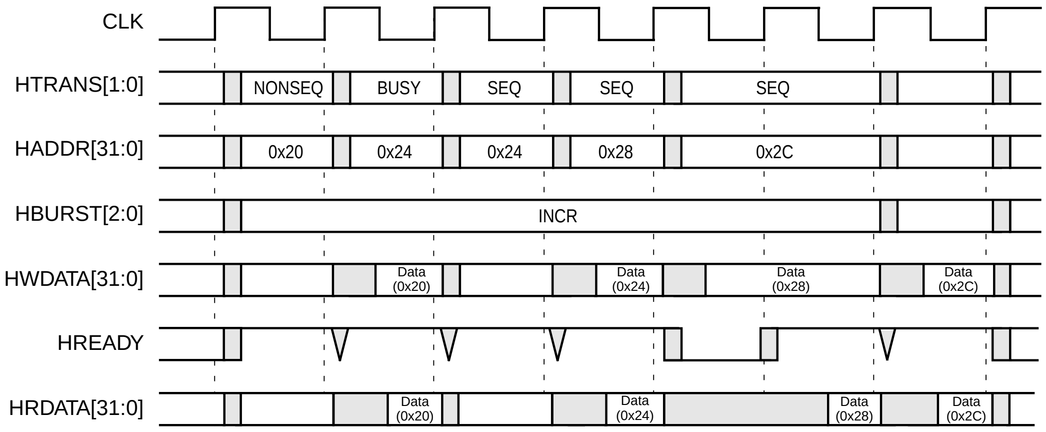
The BUSY transfer type allows bus masters to insert idle cycles in the middle of a burst of transfers. This transfer type indicates that the bus master is continuing with a burst, but the next transfer cannot take place immediately.

When a master uses the BUSY transfer type, the address and control signals must reflect the next transfer in the burst. The master must always perform this transfer and cannot cancel it at a later point in time. Slaves must provide a zero wait state OKAY response in the same way that they respond to IDLE transfers.

Figure 5.2 shows usage of three transfer types. The first transfer is the start of a burst; it is nonsequential. The master is unable to perform the second transfer of the burst immediately. It uses a BUSY transfer to delay the start of the next transfer. In this example, the master only requires one cycle before it is ready to start the next transfer in the burst, which completes with no wait states.

The master performs the third transfer of the burst immediately, but this time the slave is unable to complete and uses HREADY to insert a single wait state. The final transfer of the burst completes with zero wait states.

**Figure 5.2 Transfer Type Examples**



## 5.2.2 Burst Types

The AHB protocol defines four, eight, and 16-beat bursts as well as undefined length bursts and single transfers. The protocol also supports incrementing and wrapping bursts.

All ARM966E-S bursts are defined as incrementing bursts of undefined length (INCR).

HBURST[2:0] determine the burst type as indicated in Table 5.2. The ARM966E-S supports SINGLE and INCR types only (HBURST[2:0] = 0b000 and 0b001).

**Table 5.2 Burst Signal Encoding**

| HBURST[2:0] | Burst Type | Description |
|---|---|---|
| 000 | SINGLE | Single transfer |
| 001 | INCR | Incrementing burst of unspecified length |
| 010–111 | | Not supported |

Incrementing bursts access sequential locations; the address of each transfer in the burst is just an increment of the previous address.

Bursts must not cross a 1 Kbyte address boundary. An incrementing burst can be of any length, but the upper limit is set by the fact that the address must not cross the 1 Kbyte boundary.

Single transfers can be done using an unspecified length incrementing burst that has a burst length of one.

All transfers within a burst must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (HADDR[1:0] = 00), and halfword transfers must be aligned to halfword address boundaries (HADDR0 = 0).

### 5.2.3 Control Signals

The control signals provide additional information regarding the transfer. These signals include HWRITE, HSIZE[2:0], HPROT[3:0], HREADY, and HRESP[1:0].

#### 5.2.3.1 HWRITE

HWRITE determines the transfer direction. When HIGH, HWRITE indicates a write transfer; the master broadcasts data on the write data bus, HWDATA[31:0]. When HWRITE is LOW, a read transfer is performed. The slave places the data on the read data bus, HRDATA[31:0].

#### 5.2.3.2 HSIZE[2:0]

HSIZE[2:0] indicate the transfer size. The size is used in conjunction with the HBURST[2:0] signals to determine the address boundary for wrapping bursts. The ARM966E-S only supports transfer sizes of byte, halfword, and word (HSIZE[2:0] = 0b000, 0b001, and 0b010).

**Table 5.3 Size Encoding**

| HSIZE[2:0] | Transfer Size | Description |
|---|---|---|
| 000 | 8 bits | Byte |
| 001 | 16 bits | Halfword |
| 010 | 32 bits | Word |
| 011–111 | | Not supported |

#### 5.2.3.3 HPROT[3:0]

HPROT[3:0] are the protection control signals. They provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection.

HPROT0 indicates whether the instruction is an opcode fetch or data access. HPROT1 indicates whether the transfer is a supervisor mode access or user mode access. For bus masters with an MMU, HPROT2 indicates whether the current access is bufferable. HPROT3 is tied to 0, indicating noncacheable.

**Table 5.4 Protection Signal Encoding**

| HPROT3 Cacheable | HPROT2 Bufferable | HPROT1 Supervisor | HPROT0 Data/Opcode | Description |
|---|---|---|---|---|
| – | – | – | 0 | Opcode Fetch |
| – | – | – | 1 | Data Access |
| – | – | 0 | – | User Access |
| – | – | 1 | – | Supervisor Access |
| – | 0 | – | – | Not Bufferable |
| – | 1 | – | – | Bufferable |
| 0 | – | – | – | Not Cacheable |

Not all bus masters can generate accurate protection information because they do not contain sufficient information about the nature of the transfer. Therefore slaves should not use HPROT[3:0] unless absolutely necessary.

#### 5.2.3.4 HREADY

HREADY is used to extend the data portion of an AHB transfer. When LOW, HREADY indicates the transfer is to be extended; when HIGH, HREADY indicates the transfer can complete. A typical slave uses HREADY to insert the appropriate number of wait states into the transfer.

To ensure accurate bus access latency, every slave must have a predetermined number of wait states it inserts before backing off the bus. It is recommended that slaves do not insert more than 16 wait states to prevent any single access locking the bus for a large number of clock cycles.

#### 5.2.3.5 HRESP[1:0]

The HRESP[1:0] signals indicate the transfer response: OKAY, ERROR, SPLIT, or RETRY.

When the transfer completes with an OKAY response on HRESP[1:0], the transfer was successfully completed. The OKAY response is also used for any additional cycles that are inserted with HREADY LOW, prior to giving one of the other three responses.

When it is necessary for a slave to insert a number of wait states prior to deciding what response to give, then it must drive the response to OKAY.

A slave uses the ERROR response to indicate to the bus master that some form of error condition with the associated transfer has occurred. Typically, this response is used for a protection error, such as an attempt to write to a read-only memory location. A two-cycle response is required for an Error condition.

The SPLIT and RETRY response combinations allow slaves to delay the completion of a transfer, but free up the bus for use by other masters. These response combinations are usually only required by slaves that have a high access latency and can make use of these response codes to ensure that other masters are not prevented from accessing the bus for long periods of time.

The RETRY response indicates the transfer has not yet completed, so the bus master should retry the transfer. The master should continue to retry the transfer until it completes. A two-cycle RETRY response is required.

The SPLIT response indicates the transfer has not yet completed. The bus master must retry the transfer when it is next granted the bus. The slave will request access to the bus on behalf of the master when the transfer can be completed. A two-cycle SPLIT response is required.

### 5.2.4 Data Buses

Because the AHB interface does not implement 3-state drivers, it contains separate read and write data buses.

#### 5.2.4.1 HWDATA[31:0]

The bus master drives the write data bus during write transfers. If the transfer is extended, then the bus master must hold the data valid until the transfer completes, as indicated by HREADY HIGH.

All transfers must be aligned to the address boundary equal to the size of the transfer. For example, word transfers must be aligned to word address boundaries (HADDR[1:0] = 00), halfword transfers must be aligned to halfword address boundaries (HADDR0 = 0).

For transfers that are narrower than the width of the bus, for example, a 16-bit transfer on a 32-bit bus, then the bus master only has to drive the appropriate byte lanes. The slave is responsible for selecting the write data from the correct byte lanes. Table 5.5 and Table 5.6 show which byte lanes are active for little-endian and big-endian systems, respectively. Burst transfers that have a transfer size less than the width of the data bus have different active byte lanes for each beat of the transfer.

**Table 5.5 Active Byte Lanes for a 32-Bit Little-Endian Data Bus**

| Transfer Size | Address Offset | DATA[31:24] | DATA[23:16] | DATA[15:8] | DATA[7:0] |
|---|---|---|---|---|---|
| Word | 0 | x | x | x | x |
| Halfword | 0 | | | x | x |
| Halfword | 2 | x | x | | |
| Byte | 0 | | | | x |
| Byte | 1 | | | x | |
| Byte | 2 | | x | | |
| Byte | 3 | x | | | |

Table 5.6 **Active Byte Lanes for a 32-Bit Big-Endian Data Bus**

| Transfer Size | Address Offset | DATA[31:24] | DATA[23:16] | DATA[15:8] | DATA[7:0] |
|---|---|---|---|---|---|
| Word | 0 | x | x | x | x |
| Halfword | 0 | x | x | | |
| Halfword | 2 | | | x | x |
| Byte | 0 | x | | | x |
| Byte | 1 | | x | | |
| Byte | 2 | | | x | |
| Byte | 3 | | | | x |

The active byte lane also depends on the endianess of the system, but the AHB does not specify the required endianess. Therefore all master and slaves must have the same endianess.

#### 5.2.4.2 HRDATA[31:0]

The appropriate slave drives the read data bus during read transfers. If the slave extends the read transfer by holding HREADY LOW, then the slave only needs to provide valid data at the end of the final cycle of the transfer, as indicated by HREADY HIGH.

For transfers that are narrower than the width of the bus, the slave only needs to provide valid data on the active byte lanes, as indicated in Table 5.5 and Table 5.6. The bus master is responsible for selecting the data from the correct byte lanes.

A slave only has to provide valid data when a transfer completes with an OKAY response. ERROR, RETRY, and SPLIT responses do not require valid read data.

### 5.2.5 Endianess

For the system to function correctly, all modules must use the same endianess. The same is true of any data routing or bridges. Dynamic endianess is not supported.

Only modules that are used in a wide variety of applications should support both big- and little-endian ordering. Either a configuration pin or internal control bit can select between the two endian settings.

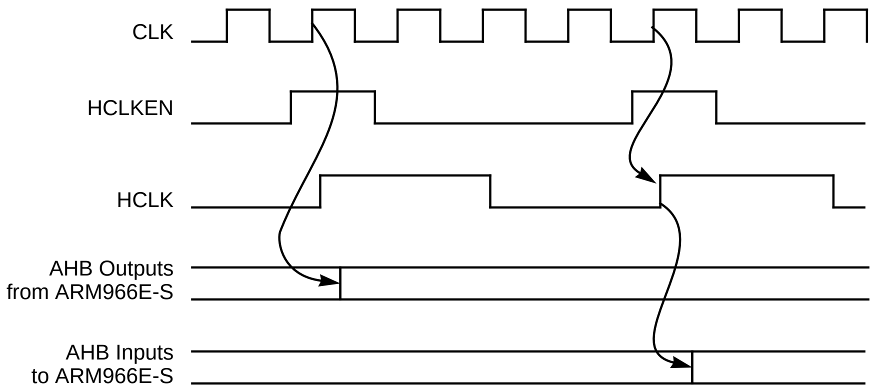
## 5.3 AHB Clocking

The ARM966E-S uses a single clock, CLK. In many systems, it might be desirable for the ARM966E-S to run at a higher frequency than the AHB system bus. To support this option, the ARM966E-S provides a clock enable, HCLKEN.

HCLKEN is HIGH for a single CLK period and signifies the rising edge of the AHB clock HCLK. CLK and HCLK must be synchronous. The skew between CLK and HCLK must be minimized. Note that HCLK is not used by the core and might not be used in all implementations.

Figure 5.3 shows the relationship between CLK, HCLKEN, and HCLK.

**Figure 5.3 AHB Clock Relationships**



## 5.4 AHB Operation

Before an AHB transfer can begin, the bus master must be granted access to the bus. This process is started by the master asserting a request signal to the arbiter. Then the arbiter indicates when the master is granted use of the bus.

When a master is granted the bus, its address and control signals are driven to all slaves. These signals provide information on the address, direction, and width of the transfer, as well as an indication when the transfer forms part of an incrementing burst. Incrementing bursts do not wrap at address boundaries.

A write data bus moves data from the master to a slave, while a read data bus moves data from a slave to the master.

Every transfer consists of an address/control cycle followed by one or more cycles for the data. The address cannot be extended and therefore all slaves must sample the address during this time. The data, however, can be extended using the HREADY signal. When LOW, this signal causes wait states to be inserted into the transfer and allows extra time for the slave to provide or sample data.

During a transfer, the slave shows the status using the response signals, HRESP[1:0]. The OKAY response indicates that the transfer is progressing normally, and, when HREADY goes HIGH, shows the transfer has completed successfully.

The other possible transfer responses are ERROR, RETRY, and SPLIT. The ERROR response indicates that a transfer error occurred and the transfer was unsuccessful. Both SPLIT and RETRY transfer responses indicate that the transfer cannot complete immediately, but the bus master should continue to attempt the transfer.

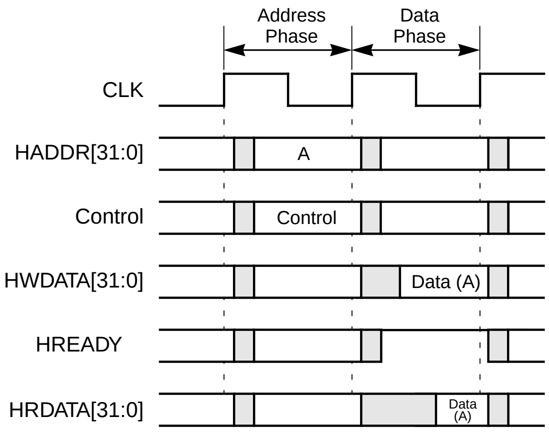
In normal operation, a master is allowed to complete all the transfers in a particular burst before the arbiter grants another master access to the bus. However, in order to avoid excessive arbitration latencies, it is possible for the arbiter to break up a burst and in such cases, the master must rearbitrate for the bus in order to complete the remaining transfers in the burst.

## 5.5 Basic Transfers

An AHB transfer consists of two distinct sections, the address and the data. The address lasts only a single cycle, while the data might require several cycles, using the HREADY signal.

Figure 5.4 shows the simplest transfer with no wait states. The master drives the address and control signals onto the bus after the rising edge of CLK. The slave then samples the address and control information on the next rising edge of the clock. After the slave has sampled the address/control, it can start to drive the appropriate response. The bus master samples the response on the third rising edge of the clock.

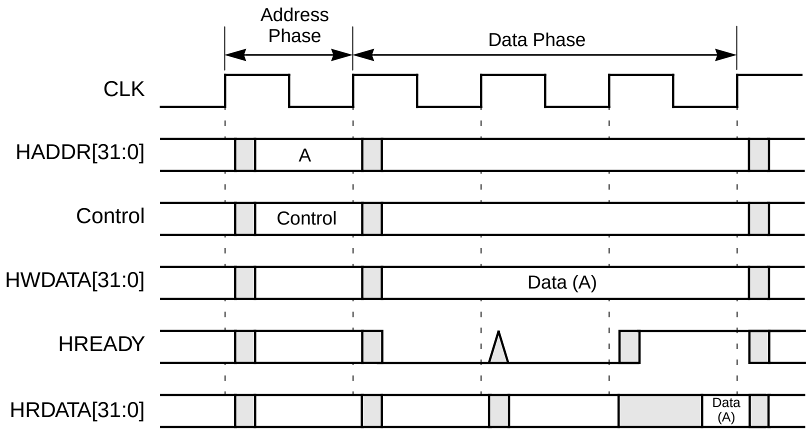
**Figure 5.4 Simple Transfer**



This example shows how the address and data phases of the transfer occur during different clock periods. In fact, the address phase of any transfer occurs during the data phase of the previous transfer. This overlapping of address and data is fundamental to the pipelined nature of the bus. It allows for high-performance operation while still providing adequate time for a slave to provide the response to a transfer.

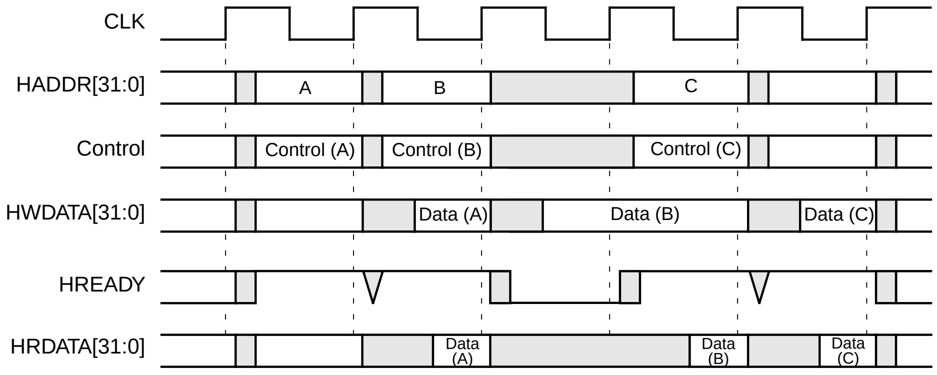
Figure 5.5 shows a transfer with wait states. A slave can insert wait states into any transfer, allowing additional time for completion. For write operations, the bus master holds the data stable throughout the extended cycles. For read operations, the slave does not have to provide valid data until the transfer is about to complete.

**Figure 5.5 Transfer with Wait States**



When a transfer is extended with wait states, it ends up extending the address phase of the next transfer. Figure 5.6 shows three transfers to unrelated addresses: A, B, and C. The transfers to addresses A and C both have zero wait states, but the transfer to address B has one wait state. Extending the data phase of the transfer to address B extends the address phase of the transfer to address C.

**Figure 5.6 Multiple Transfers**



## 5.6 Burst Operations

This section provides examples of burst operations the ARM966E-S supports.

### 5.6.1 Early Burst Termination

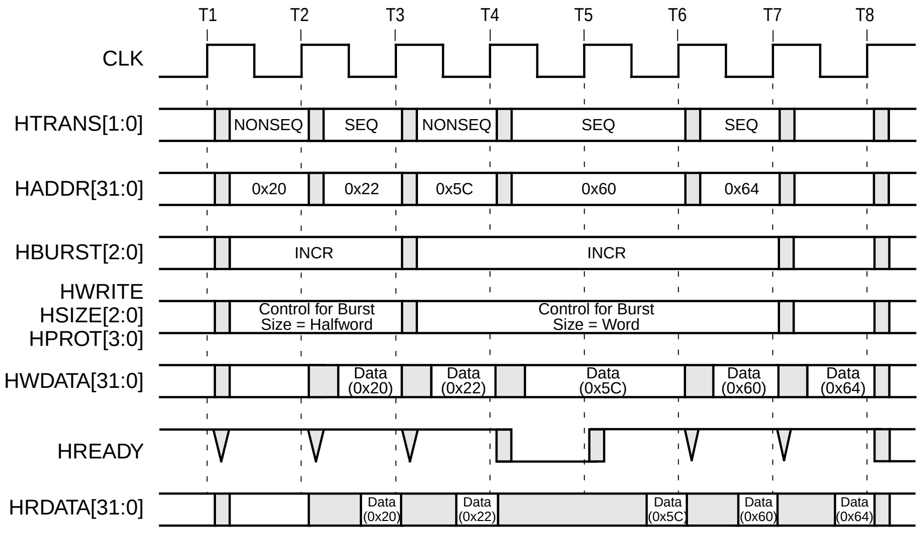
Sometimes a burst is not allowed to complete. To determine when a burst has terminated early, the slave should monitor the HTRANS signals and ensure that after the start of the burst, every transfer is sequential or busy. Occurrence of a nonsequential or idle transfer indicates a new burst has started, and thus the previous burst must have been terminated.

If a bus master cannot complete a burst because it loses ownership of the bus, then it must rebuild the burst appropriately when it next gains access to the bus. For example, if a master has only completed one beat of a four-beat burst, then it must use an undefined length burst to perform the remaining three transfers.

### 5.6.2 Burst Operation Example

Figure 5.7 shows incrementing bursts of undefined length. The figure shows two bursts. The first burst is two transfers starting at address 0x20; the second burst is three transfers starting at address 0x5C. The first burst consists of halfword transfers, so the addresses increment by two. The second burst consists of word transfers, so the addresses increment by four.

**Figure 5.7 Incrementing Bursts with Undefined Lengths**



## 5.7 Slave Transfer Responses

After a master has started a transfer, the slave then determines how the transfer should progress. No provision is made within the AHB specification for a bus master to cancel a transfer once it has started.

Whenever a slave is accessed, it must provide a response that indicates the status of the transfer. HREADY is used to extend the transfer. It works in combination with the response signals, HRESP[1:0], which provide the status of the transfer.

The options that the slave has on how it can complete the transfer are:

- Complete the transfer immediately
- Insert one or more wait states to allow time to complete the transfer
- Signal an error to indicate that the transfer has failed
- Delay the completion of the transfer, but allow the master and slave to back off the bus, leaving it available for other transfers

## 5.7.1 Two-Cycle Response

Only an OKAY response can be given in a single cycle. The ERROR, RETRY, and SPLIT responses require at least two cycles.

To complete with any of these three responses, in the cycle previous to the last cycle, the slave drives HRESP[1:0] to indicate ERROR, RETRY, or SPLIT while driving HREADY LOW to extend the transfer for an extra cycle. In the final cycle, the slave drives HREADY HIGH to end the transfer, while continuing to drive HRESP[1:0].

If the slave needs more than two cycles to provide the ERROR, RETRY, or SPLIT response, then additional wait states can be inserted at the start of the transfer. During this time, the HREADY signal is LOW and the response must be set to OKAY.

The two-cycle response is required because of the pipelined nature of the bus. By the time a slave starts to issue an ERROR, RETRY, or SPLIT response, then the address for the following transfer has already been broadcast onto the bus. The two-cycle response allows sufficient time for the ARM966E-S to cancel this address and drive HTRANS[1:0] to IDLE before the start of the next transfer.

For the RETRY or SPLIT responses, the following transfer must be cancelled because it must not take place before the current transfer has completed. However, for the ERROR response, where the current transfer is not repeated, it is optional whether or not the next transfer is allowed to complete.

Figure 5.8 shows an example of a retry operation. The ARM966E-S starts with a transfer to address A. Before the response is received for this transfer, the ARM966E-S moves the address to A + 4. However, the slave at address A is unable to complete the transfer immediately; therefore it issues a RETRY or SPLIT response.

This response indicates to the ARM966E-S that the transfer at address A is unable to complete. So the transfer at address A + 4 is cancelled and replaced with an IDLE transfer.

**Figure 5.8 Transfer with Retry Response**

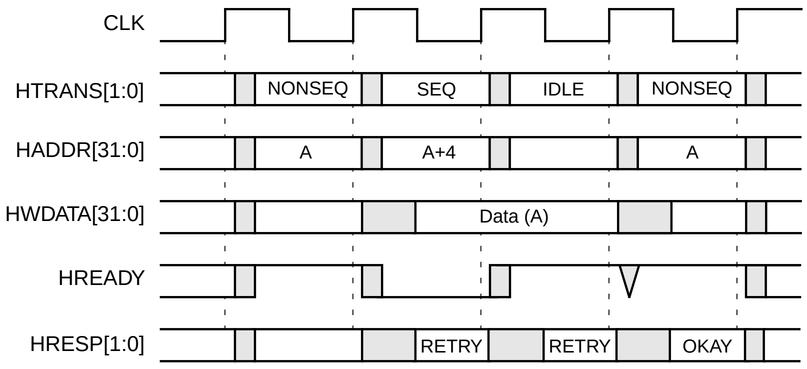
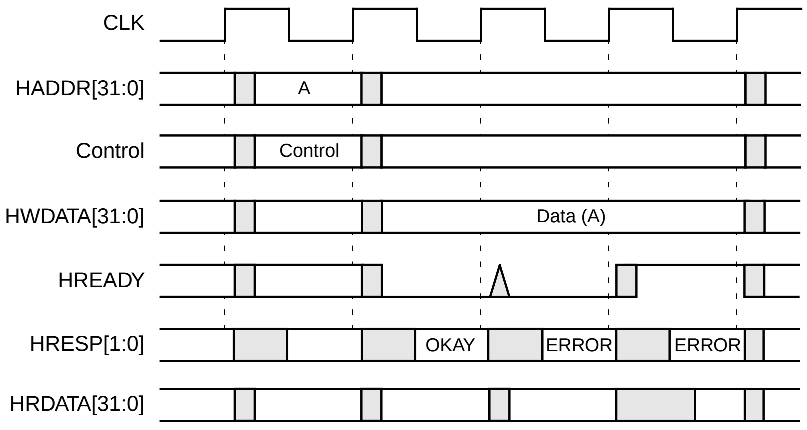


Figure 5.9 shows a transfer where the slave requires one cycle to decide on the response it is going to give (during which time HRESP indicates OKAY). Then the slave ends the transfer with a two-cycle ERROR response.

**Figure 5.9 Error Response**



## 5.7.2 Error Response

If a slave provides an ERROR response, then the ARM966E-S can choose to cancel the remaining transfers in the burst. However, this requirement is not strict; it is also acceptable for the ARM966E-S to continue the remaining transfers in the burst.

## 5.7.3 Retry Responses

The RETRY response provides a mechanism for slaves to release the bus when they are unable to supply data for a transfer immediately. This mechanism allows the transfer to finish on the bus. Therefore a higher priority ARM966E-S can get access to the bus.

After a RETRY has occurred, the arbiter continues to use the normal priority scheme. Only masters that have a higher priority gain access to the bus.

The ARM966E-S continues to request the bus and attempts the transfer until it either has completed successfully or has been terminated with an error response.

## 5.7.4 Split Responses

The SPLIT response provides a mechanism for slaves to release the bus when they are unable to immediately supply data for a transfer. This mechanism allows the transfer to finish on the bus. Therefore a higher priority ARM966E-S can get access to the bus.

After a SPLIT has occurred, the arbiter adjusts the priority scheme so that any other master requesting the bus gets access, even if it has a lower priority. The slave requests access to the bus on behalf of the master when the transfer can complete.

The ARM966E-S will continue to request the bus and attempt the transfer until it is either completed successfully or terminated with an error response.

# Chapter 6
# Write Buffer

This chapter describes the ARM966E-S Write Buffer.

This chapter contains the following sections:

## 6.1 Introduction

The ARM966E-S Write Buffer enhances system performance. It decouples the processor from a slower memory interface during processor writes. The Write Buffer allows the core to perform writes without waiting for access to the AHB bus.

The Write Buffer consists of a 12-entry FIFO. Each entry can be either address or data, where the use of each entry is specified by an address/data flag. Each address entry is tagged with the size of the transfer as indicated by the ARM9E-S processor core (byte, halfword, or word).

The Write Buffer is flushed at reset. Any entries still in the buffer when reset is asserted are lost. To enable the Write Buffer, write to CP15 Register 1. To drain the Write Buffer, write to CP15 Register 7.

## 6.2 Normal Operation

The Write Buffer only buffers writes to areas of memory that are identified as being bufferable (see the memory map on page 3-5). Provided that the Write Buffer is enabled and the processor performs a write to a bufferable area, the data is placed in the Write Buffer. The Write Buffer is then marked as *not empty*. Once data is in the Write Buffer, it is always written, even if it is subsequently disabled, unless a reset occurs.

A Write Buffer Flush condition is defined as one of the following:

- Reset
- The Write Buffer is full and the processor attempts another write
- An unbuffered write occurs
- A Read-Lock-Write instruction occurs
- A read to address was previously posted, but is not yet written to memory

Note: The ARM966E-S does not support reordering and merging of writes. In cases, for example, where there are two nonsequential writes to two adjacent but not incremental addresses (such as 0x84 followed by 0x80), they could be reordered and merged into a single two-word burst transfer, but software might have intended the write to 0x84 to occur before the write to 0x80. This feature also increases the complexity of the Write Buffer.

If the ARM966E-S enters Wait-for-Interrupt mode, then the Write Buffer continues to perform accesses until it is empty.

Read-modify-write sequences to buffered AHB regions are treated as unbuffered accesses.

## 6.3 Full Write Buffer

When full, the Write Buffer must provide a response to the System Controller to prevent lost data.

When the Write Buffer is full and the processor attempts a buffered write, the first buffer entry must be written to memory to make space for the new write entry. If the first Write Buffer entry was from a store multiple, then all of the write cycles needed to complete the store multiple are performed before the new write cycle is placed in the buffer.

## 6.4 Unbuffered Writes

An unbuffered write to the AHB bus causes the Write Buffer to drain before the write takes place. Flushing ensures the writes are in the order the software intended. The order is important, for example, when setting up some parameters before triggering an external event. During this time, the core must be stalled to prevent execution advancing.

## 6.5 Read-Lock-Write

The Read-Lock-Write sequence is atomic. It asserts the HLOCK signal during the transfer to prevent any other bus masters from breaking the sequence. The write phase of the sequence is treated as an unbuffered write, which means that the Write Buffer needs to be flushed before the write can complete. The Write Buffer is flushed before starting the read phase of the sequence. Therefore only the memory bus is locked for the minimum number of cycles (the single read followed by single write cycles).

## 6.6 Read to Write-Posted Address

If a read occurs to an address contained in the Write Buffer, then the buffer must be flushed before allowing the read to propagate to the bus. During this time, the core must stall to prevent execution from advancing.

## 6.7 Write Buffer Nonrecoverable Error and Abort Conditions

Buffered writes cannot be successfully aborted. When an ERROR response to a transfer on the AHB bus occurs, the Write Buffer drives the DABORT signal to the processor. However, the data abort handler cannot correlate the processor’s present state to the bus cycle that caused the ERROR. This event is considered to be a nonrecoverable error condition.

# Chapter 7
# System Controller

This chapter describes the ARM966E-S System Controller, which arbitrates between the Instruction RAM, Data RAM, and the AHB Bus Interface Unit. This chapter contains the following sections:

- Section 7.1, “Operation”
- Section 7.2, “Clock Control”

## 7.1 Operation

The System Controller oversees the interactions between the Instruction/Data RAMs and the AHB Bus Interface Unit. The System Controller prevents the processor core from advancing if its memory requests cannot be satisfied simultaneously (for example, accessing AHB memories while executing code from the tightly coupled RAM). The System Controller prevents these occurrences from happening by controlling the internal system clock enable signal (CLKEN) to the ARM9E-S core.

When the core is accessing the on-chip memory, stall cycles are required when:

- there are simultaneous data accesses and instruction fetches to the Instruction Memory
- a write to the RAM is followed by a read of the same memory

If the Write Buffer is full, when the ARM966E-S writes to a bufferable region of memory (see memory map on page 3-5), the System Controller must stall it to prevent data loss. When the address decoders indicate an ARM966E-S read misses the on-chip memory, the System Controller must stall the core until the Write Buffer is empty and maintain the stall until the access to the AHB is complete.

When the core performs a nonbufferable write that misses the on-chip memory as indicated by the address decoders, the System Controller must stall the core until the Write Buffer is empty and maintain the stall until the access to the AHB is complete.

## 7.2 Clock Control

When accessing the AHB bus for accesses that do not use the Write Buffer, the internal clock enable CLKEN must include the effects of HCLKEN, HREADY, and HGRANT to ensure that the core is synchronized to the AHB bus. The AHB BIU must inform the System Controller when each transfer has completed. The System Controller determines when an external access is being performed and qualifies the internal clock enables with HCLKEN from the AHB interface. Using HCLKEN effectively slows down the operation of the ARM966E-S to the AHB bus frequency.

# Chapter 8
# Tightly Coupled RAM

This chapter describes the Instruction and Data RAMs within the ARM966E-S design. It contains the following sections:

- Section 8.1, “Tightly Coupled Memory (TCM) Overview”
- Section 8.2, “ARM966E-S SRAM Requirements”
- Section 8.3, “Enabling the SRAM”
- Section 8.4, “ARM966E-S SRAM Wrapper”
- Section 8.5, “Example SRAM Interfaces”

## 8.1 Tightly Coupled Memory (TCM) Overview

The ARM966E-S supports synchronous SRAM for the tightly coupled RAM. The TCM interfaces are zero wait state, and data must be returned in a single clock cycle. The Instruction and Data RAMs can be of any size up to 512 Kbytes. The Instruction RAM and Data RAM are independent and thus can be different sizes.

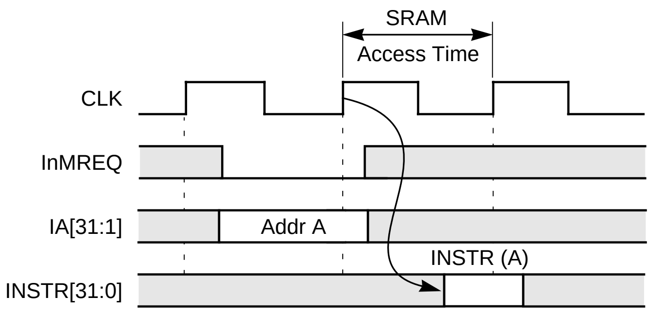
## 8.2 ARM966E-S SRAM Requirements

The tightly-coupled SRAM is built from blocks of ASIC library compiled SRAM. The Instruction SRAM (I-SRAM) and Data SRAM (D-SRAM) can each be any size from 0 bytes to 512 KBytes; the size must be an integer power of two. The I-SRAM and D-SRAM can have different sizes.

To allow the I-SRAM to be initialized and for access to literal tables during execution, the data interface of the ARM966E-S must be able to access the I-SRAM. Thus the instruction and data addresses must be multiplexed before entering the I-SRAM, and the instruction data is routed both to the instruction and data interfaces of the core. See Figure 1.1 on page 1-3 for details of this data and address multiplexing.

The ARM966E-S supports the use of synchronous SRAM. The SRAM control has been implemented in a way that expects the compiled SRAM memory cells to return read data to the ARM966E-S in a single cycle. This requirement applies to both the I-SRAM and D-SRAM. See Figure 8.1 for a typical read cycle (I-SRAM shown).

**Figure 8.1 SRAM Read Cycle**



During normal program execution, the instruction and data interfaces of the ARM966E-S can be active simultaneously. In this case, both SRAMs can be simultaneously accessed allowing the core to continue execution without any stall cycles. There are cases, however, where stall cycles are encountered when accessing the SRAM.

## 8.3 Enabling the SRAM

There are two mechanisms for controlling the SRAM enable:

- both I-SRAM and D-SRAM can be enabled or disabled during reset by the input INITRAM
- the I-SRAM and D-SRAM can be individually enabled or disabled through software MCR instructions to CP15.

### 8.3.1 Using INITRAM to Enable SRAM

Two resets are described in the following subsections, depending on the state of the INITRAM input.

#### 8.3.1.1 Reset with INITRAM LOW

INITRAM allows the ARM966E-S to boot with both SRAM blocks either enabled or disabled. If INITRAM is held LOW during reset, the ARM966E-S comes out of reset with both SRAMs disabled. All accesses to I-SRAM and D-SRAM space go to the AHB. The I-SRAM and D-SRAM can then be individually or jointly enabled by writing to the CP15 Control Register (register 1).

#### 8.3.1.2 Reset with INITRAM HIGH

If INITRAM is held HIGH during reset, both SRAM blocks are enabled when the ARM966E-S comes out of reset. This case is normally used for a warm reset where the SRAM has already been programmed before nRESET is asserted to the ARM966E-S. In this case, the SRAM contents are preserved and the ARM966E-S can run directly from the tightly coupled SRAM following reset. Either one or both SRAMs can be further disabled or enabled by writing to the CP15 control register.

Important: If INITRAM is held HIGH during a cold reset (the SRAM has not previously been initialized), VINITHI must be held HIGH to ensure that the ARM966E-S boots from 0xFFFF.0000, which is in AHB address space and is outside the SRAM address space. This boot location is necessary because if VINITHI is LOW, the ARM966E-S will attempt to boot from 0x0000.0000, which selects the uninitialized I-SRAM.

## 8.3.2 Using CP15 Control Register to Enable SRAM

When out of reset, the state of the CP15 Control register controls the behavior of the tightly coupled SRAM. See Section 3.7.2, “CP15 Registers,” page 3-9 for details on how to read and write the CP15 Control Register.

#### 8.3.2.1 Enabling the I-SRAM

To enable the I-SRAM, set bit 12 (the I bit) of the CP15 Control Register. This register must be accessed in a read-modify-write fashion to preserve the contents of the bits not being modified. When the I-SRAM has been enabled, all future ARM9E-S instruction fetches and data accesses to the I-SRAM address space access the I-SRAM. Figure 3.2 on page 3-5 shows the I-SRAM address space.

Enabling the I-SRAM greatly increases the performance of the ARM966E-S because most accesses to it have no stall cycles, whereas accessing the ARM966E-S through the AHB can cause several stall cycles for *each* access.

You must ensure that the I-SRAM is appropriately initialized before it is enabled and used to supply instructions to the ARM966E-S. If the core tries to execute instructions from uninitialized I-SRAM, the behavior is unpredictable.

#### 8.3.2.2 Disabling the I-SRAM

To disable the I-SRAM, clear bit 12 of the CP15 Control Register. When the I-SRAM has been disabled, all further ARM966E-S instruction fetches access the AHB. If the ARM966E-S performs a data access to the I-SRAM address space, an AHB access is performed. Figure 3.2 on page 3-5 shows the I-SRAM address space.

The contents of the I-SRAM are preserved when it is disabled. If it is re-enabled, accesses to previously initialized I-SRAM locations returns the preserved data.

#### 8.3.2.3 Enabling the D-SRAM

To enable the D-SRAM, set bit 2 (the D bit) of the CP15 Control Register. When the D-SRAM has been enabled, all future read and write accesses to the D-SRAM address space cause the D-SRAM to be accessed. Figure 3.2 on page 3-5 shows the D-SRAM address space.

#### 8.3.2.4 Disabling the D-SRAM

To disable the D-SRAM, clear bit 2 of the CP15 Control Register. When the D-SRAM is disabled, all further reads and writes to the D-SRAM address space access the AHB. Figure 3.2 on page 3-5 shows the D-SRAM address space. Read and write accesses to D-SRAM address space use the D-SRAM (if enabled) or access the AHB.

## 8.4 ARM966E-S SRAM Wrapper

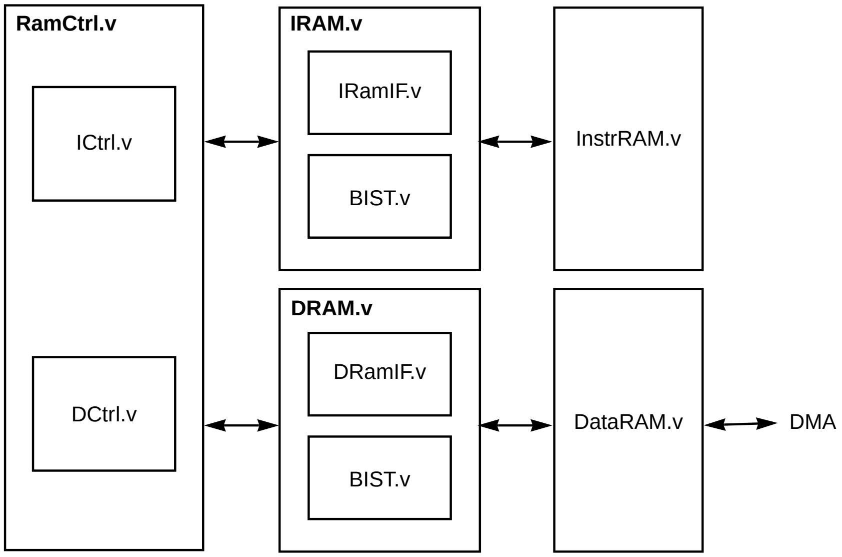
The ARM966E-S allows you to have control over the size of the I-SRAM and D-SRAM (up to a maximum of 512 KBytes each). It is not possible to have a single generic interface between the ARM966E-S and the SRAM, due to the large number of differing compiled SRAM that can be integrated into an ARM966E-S system, potentially each with a unique interface.

To ease the task of integrating differing SRAM into the ARM966E-S, an interface *wrapper* block has been developed to ensure that when wrapped, the SRAM provides a standard interface to the ARM966E-S SRAM control. Section 8.5, "Example SRAM Interfaces," provides an example SRAM wrapper with three example interfaces. Study these examples and decide which is most appropriate for the type of SRAM available. A script is provided that automates any required changes.

The RAM interface RTL allows you to trade off speed against power performance so that you can tailor the ARM966E-S to suit a particular requirement.

There are five SRAM modules instantiated at the top-level of the ARM966E-S. Figure 8.2 shows the structure of these three modules.

**Figure 8.2 ARM966E-S SRAM Hierarchy**



`RamCtrl.v` contains the RAM control logic that is partner-independent. This logic is fixed.

`IRamIF.v` and `DRamIF.v` generate the SRAM-specific ChipSelect, WriteEnable, and ByteWrite signals. Your own library RAMs are instantiated inside `InstrRAM.v` and `DataRAM.v`.

## 8.5 Example SRAM Interfaces

The example wrapper contains three RAM interface examples. All of the interface modifications are done in the `IRamIF.v` and the `DRamIF.v` blocks for the I-SRAM and D-SRAM, respectively. The example SRAM interfaces are:

- Section 8.5.1, "ONESEGX32"
- Section 8.5.2, "FOURSEGX32"
- Section 8.5.3, "FOURSEGX8"

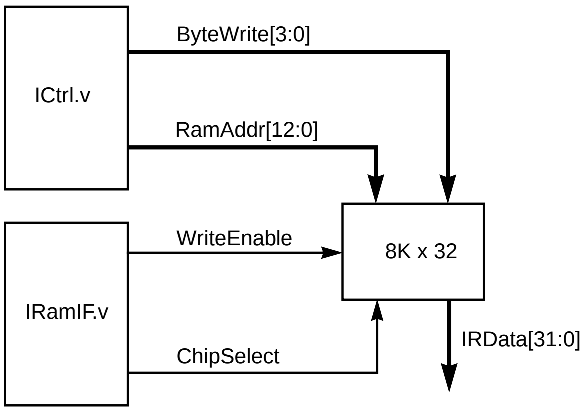
Note: The examples shown here are for 32-Kbyte I-SRAM (8 Kwords x 4 bytes). The interface for D-SRAM is identical.

### 8.5.1 ONESEGX32

Figure 8.3 shows the simplest I-SRAM interface. To use this example, the SRAM must consist of a single word-wide RAM that has byte-write control.

Only single ChipSelect and WriteEnable signals are required.

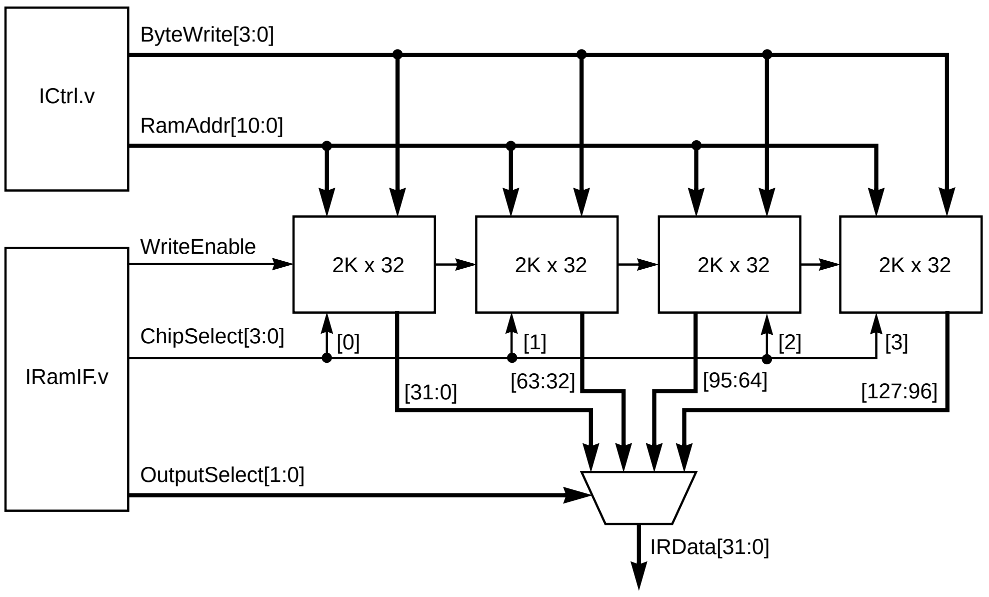
**Figure 8.3 ONESEGX32 Interface**



### 8.5.2 FOURSEGX32

Use the example shown in Figure 8.4 when it is not possible to construct the SRAM from a single physical block due to either layout constraints or generator constraints, or because a single SRAM segment does not meet timing constraints.

**Figure 8.4 FOURSEGX32 Interface**



Separate chip select signals are required for each SRAM block.

- The generation of separate chip select signals for each SRAM block ensures good power performance, because only the segment being accessed is enabled.
- The SRAM address is 11 bits in this example (compared with the 13 bit address in Section 8.5.1, "ONESEGX32"). RamAddr[12:11] are used to generate separate chip selects for each segment.

If it is not possible to have separate chip select signals for each block of RAM, for example, if the RAM is asynchronous, then separate write enable signals are required for each segment. The use of asynchronous RAMs is not recommended due to the increased power consumption of this solution.

Note: The wrapper RTL does not support asynchronous RAMs.

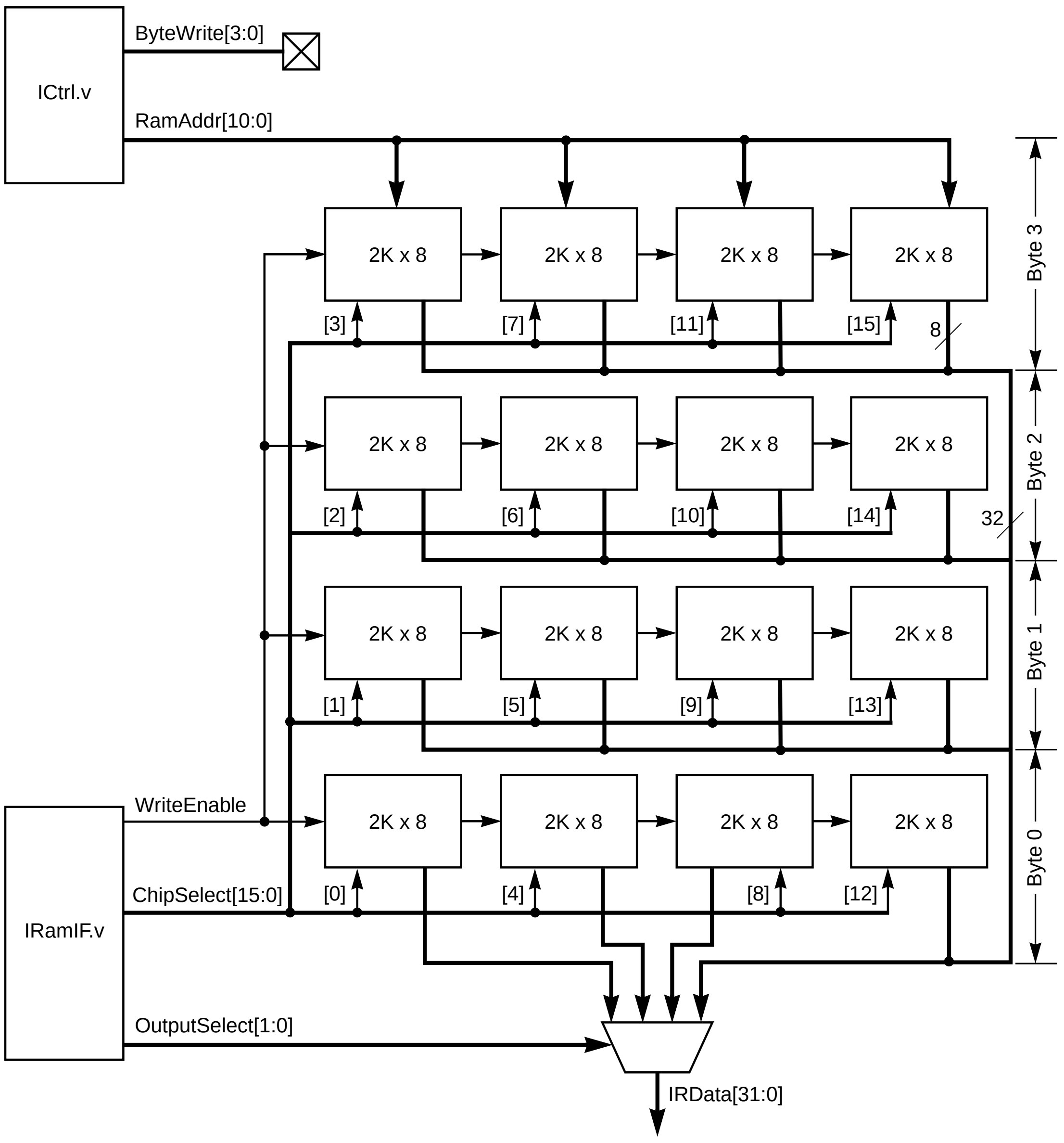
### 8.5.3 FOURSEGX8

Figure 8.5 shows that the SRAM needs to be split into four-byte wide segments where an SRAM does not support byte writes. In order to give an example of the most complex interface possible, Figure 8.5 assumes that each byte-wide SRAM needs to be split into four blocks (see word-wide SRAM in Section 8.5.2, “FOURSEGX32,”).

In Section 8.5.2, “FOURSEGX32,” the SRAM Address is 11 bits. Bits [12:11] of the address are used to decode which of the four word-wide RAMs is selected.

In Figure 8.5, ByteWrite[3:0] is used (inside `IRamIF.v`) to decode each word-wide chip select into four separate chip select signals, one for each byte of the word.

**Figure 8.5 FOURSEGX8 Interface**



# Chapter 9 External Coprocessor Interface

This chapter describes the ARM966E-S external coprocessor interface. The ARM966E-S connects to on-chip coprocessors through this interface. This interface supports all types of coprocessor instructions.

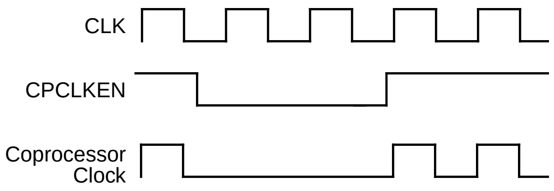
This chapter contains the following sections:

## 9.1 Overview

Coprocessors determine which instructions they need to execute by a *pipeline follower* in the coprocessor. As each instruction arrives from memory, it enters both the ARM9E-S pipeline and the coprocessor’s pipeline. To avoid a critical path for the instruction being latched by the coprocessor, the coprocessor’s pipeline operates one clock cycle later than the ARM9E-S core pipeline. However, the ARM966E-S also includes a mechanism that stalls the pipeline so the processor can catch up with the external coprocessor pipeline. So, in effect, both pipelines are synchronized. The ARM9E-S core informs the coprocessor when instructions move from decode into execute, and whether the instruction needs to be executed.

To enable coprocessors to continue execution of coprocessor data operations while the ARM9E-S core pipeline is stalled, the coprocessor receives the CLK clock, and a clock enable signal CPCLKEN. If CPCLKEN is LOW on the rising edge of CPCLK, then the ARM9E-S core pipeline is stalled and the coprocessor pipeline should not advance. Figure 9.1 indicates the timing for these signals and when the coprocessor pipeline should advance its state. Coprocessor Clock shows the result of ANDing CLK with CPCLKEN; this is one technique for generating a clock that reflects the ARM9E-S core pipeline advancing.

**Figure 9.1 CLK and CPCLKEN Timing**



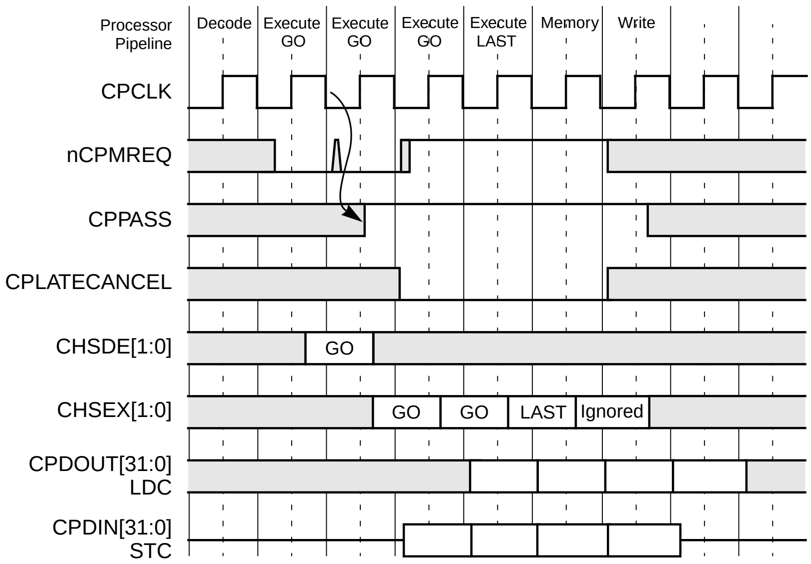
## 9.2 Coprocessor Instruction Execution

There are three classes of coprocessor instructions: LDC/STC, MCR/MRC, and CDP. The following subsections give examples of how a coprocessor should execute these instruction classes.

### 9.2.1 LDC/STC Instructions

The cycle timing for this operation is shown in Figure 9.2 In this example four words of data are transferred. The number of words transferred is determined by how the coprocessor drives the CHSDE[1:0] and CHSEX[1:0] buses.

**Figure 9.2    LDC/STC Cycle Timing**



As with all other instructions, the ARM9E-S core performs the main decode off the rising edge of the clock during the decode stage. From this, the core commits to executing the instruction and so performs an instruction fetch. The coprocessor's instruction pipeline keeps in step with the ARM9E-S core by monitoring nCPMREQ, which is a latched copy of the ARM9E-S core instruction memory request signal InMREQ. Whenever nCPMREQ is LOW, an instruction fetch is occurring and CPINSTR is updated with the fetched instruction in the next cycle. Thus the instruction currently on CPINSTR should enter the decode stage of the coprocessor pipeline, and the instruction in the decode stage of the coprocessor's pipeline should enter its execute stage.

During the execute stage, the condition codes are combined with the flags to determine whether the instruction really executes or not. The output CPPASS is asserted HIGH if the instruction in the execute stage of the coprocessor pipeline is a coprocessor instruction and has passed its condition codes. If a coprocessor instruction stalls, then CPPASS is asserted on every cycle until the coprocessor instruction is executed. If an interrupt occurs during stalling then CPPASS is driven LOW and the coprocessor should stop the coprocessor instruction execution.

The CPLATECANCEL output is used to cancel a coprocessor instruction when the instruction preceding it causes a data abort. This output is valid on the rising edge of CLK on the cycle after the coprocessor instruction's first execute cycle.

On the rising edge of the clock, the ARM9E-S processor core examines the coprocessor handshake signals CHSDE[1:0] and CHSEX[1:0]. If a new instruction is entering the execute stage in the next cycle, then the core examines CHSDE[1:0]. If the coprocessor instruction currently in execution requires another execute cycle, then the core examines CHSEX[1:0]. The handshake signals encode one of four states:

- ABSENT

  If there is no coprocessor attached that can execute the coprocessor instruction, then the handshake signals indicate the ABSENT state, and the ARM9E-S core takes the undefined instruction trap.

- WAIT

  If there is a coprocessor attached that can handle the instruction, but not immediately, then the coprocessor handshake signals are driven to indicate that the ARM9E-S core should stall until the coprocessor can catch up. In this case, the ARM9E-S core loops in an idle state waiting for CHSEX[1:0] to be driven to another state, or for an interrupt to occur. If CHSEX[1:0] changes to ABSENT then the undefined instruction trap is taken. If CHSEX[1:0] changes to GO or LAST, then the instruction proceeds as described below. If an interrupt occurs, then the ARM9E-S core is forced out of the stalled state. This condition is indicated to the coprocessor by a LOW transition on CPPASS. The instruction is restarted at a later time. The coprocessor should not commit to the instruction (change any of the coprocessor's states) until it has seen CPPASS HIGH when the handshake signals indicate the GO or LAST condition.

- GO

  The GO state indicates that the coprocessor can execute the instruction immediately, and that it requires another cycle of execution. Both the ARM9E-S core and the coprocessor must also consider the state of the CPPASS signal before actually committing to the instruction. For an LDC or STC instruction, the coprocessor instruction drives the handshake signals with GO when two or more words still need to be transferred. When only one more word is required, the coprocessor drives the handshake signals with LAST.

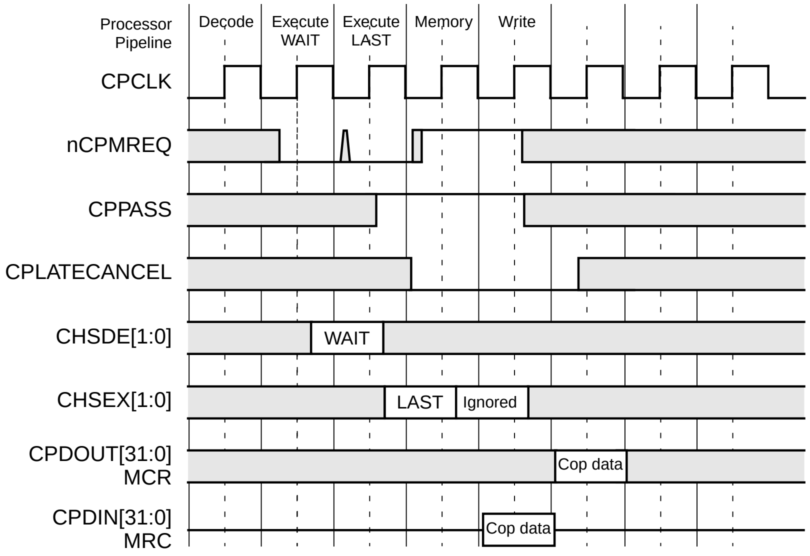
- LAST

  If an LDC or STC instruction is for more than one item of data, then after stalling, the coprocessor might drive the coprocessor handshake signals with a number of GO states, and in the cycle LAST, where LAST indicates that the next transfer is the final one. If there is only one transfer then the sequence would be: [WAIT,[WAIT,...]],LAST.

### 9.2.2 MCR/MRC Instructions

These cycles look very similar to STC/LDC cycles. Figure 9.3 shows an example with a stall (WAIT) state. First nCPMREQ is driven LOW to denote that the instruction on CPINSTR is entering the decode stage of the pipeline. This low state causes the coprocessor to decode the new instruction and drive CHSDE[1:0] as required. In the next cycle nCPMREQ is driven LOW to denote that the instruction has now been issued to the execute stage. If the condition codes pass (thus the instruction is to be executed), then CPPASS is driven HIGH and the CHSDE[1:0] handshake bus is examined (it is ignored in all other cases). For any successive execute cycles, the CHSEX[1:0] handshake bus is examined. When the LAST condition is observed, the instruction is committed. In the case of an MCR instruction, the CPDOUT[31:0] bus is driven with the register data during the coprocessor write stage. In the case of an MRC instruction, CPDIN[31:0] is sampled at the end of the ARM9E-S memory stage and is written to the destination register during the next cycle.

**Figure 9.3 MCR/MRC Cycle Timing**

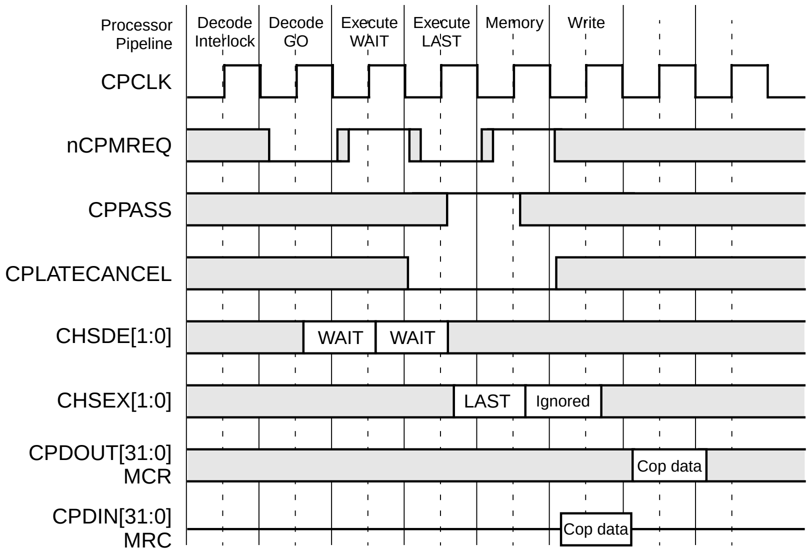


## 9.2.3 Interlocked MCRs

If the data for an MCR operation is not available inside the ARM9E-S core pipeline during its first decode cycle, then the ARM9E-S core pipeline interlocks for one or more cycles until the data is available. An example of this is where the register being transferred is the destination from a preceding LDR instruction.

In this situation, the MCR instruction enters the decode stage of the coprocessor pipeline, and then remains there for a number of cycles before entering the execute stage. Figure 9.4 is an example of an interlocked MCR.

**Figure 9.4 Interlocked MCR**



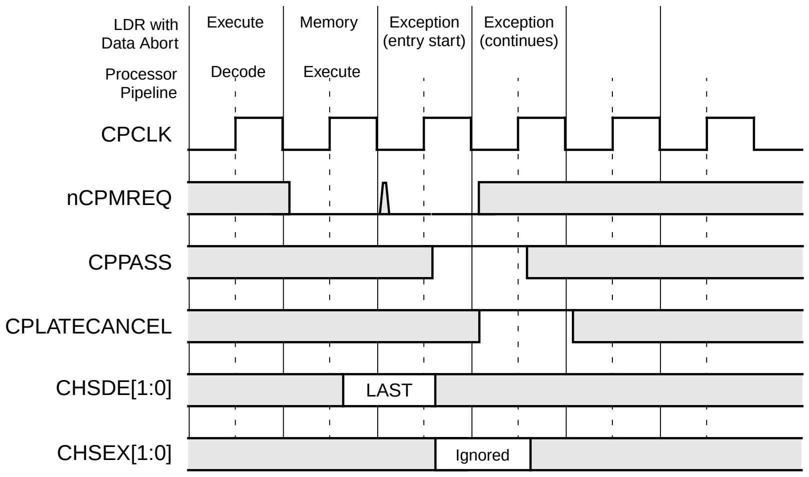
## 9.2.4 CDP Instructions

CDP instructions normally execute in a single cycle. Like all the previous cycles, nCPMREQ is driven LOW to signal when an instruction is entering the decode and then the execute stage of the pipeline. If the instruction is to be executed then CPPASS is driven HIGH during the execute. If the coprocessor can execute the instruction immediately, it drives CHSDE[1:0] with LAST. If the instruction requires a wait cycle, then the coprocessor drives CHSDE[1:0] with WAIT and CHSEX[1:0] with LAST.

Figure 9.5 shows a CDP that is cancelled due to the previous instruction causing a data abort. The CDP instruction enters the execute stage of the pipeline and is signaled to execute when CPPASS is HIGH. In the following phase, CPLATECANCEL is asserted, which causes the coprocessor to terminate execution of the CDP instruction and to cause no state changes to the coprocessor.

Note in Figure 9.5 that CPLATECANCEL can be asserted during the memory cycle as well as during the execution cycle. The coprocessor should be able to handle instruction aborts during these two stages.

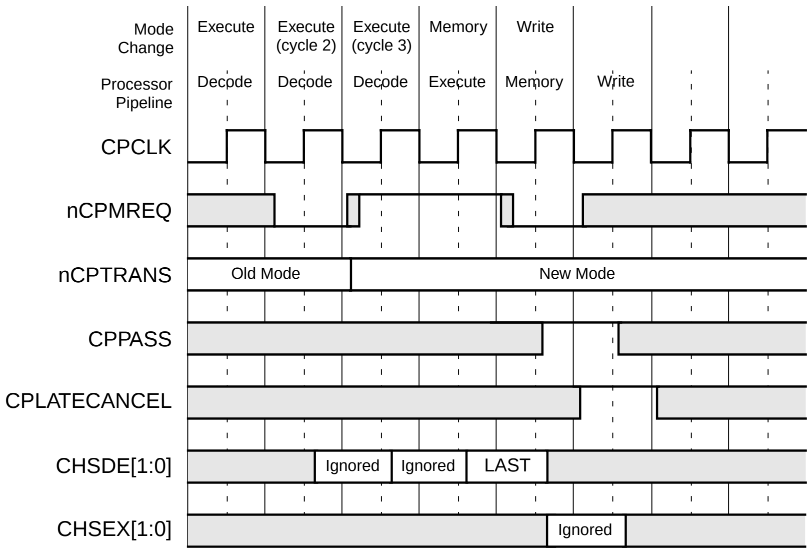
**Figure 9.5 Late Cancelled CDP**



## 9.3 Privileged Instructions

The coprocessor can restrict certain instructions for use in privileged modes only. To do this, the coprocessor has to track the nCPTRANS output. Figure 9.6 shows how nCPTRANS changes after a mode change.

**Figure 9.6 Privileged Instructions**

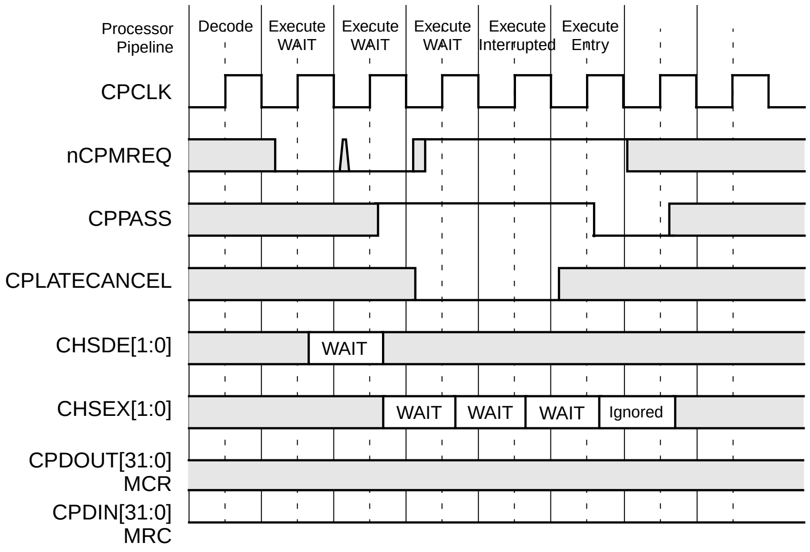


## 9.4 Stalling and Interrupts

Figure 9.7 shows a stalled coprocessor instruction being abandoned due to an interrupt. The coprocessor can stall the processor during the execution of a coprocessor instruction if, for example, it is still busy with an earlier coprocessor instruction. To do so, the coprocessor associated with the decode stage instruction drives WAIT on CHSDE[1:0]. When the instruction concerned enters the execute stage of the pipeline, the coprocessor can drive WAIT onto CHSEX[1:0] for as many cycles as it wants in order to keep the instruction in the busy-wait loop.

For interrupt latency reasons, the coprocessor can be interrupted while stalling, thus causing the instruction to be abandoned. Abandoning execution is done through CPPASS. The coprocessor must monitor the state of CPPASS during every stall cycle. If it is HIGH, the instruction should still be executed. If it is LOW, the instruction should be abandoned. Note in Figure 9.7 that CPLATECANCEL is also asserted as a result of the execute interruption.

## Figure 9.7 Stalling and Interrupts



# Chapter 10
# Debug

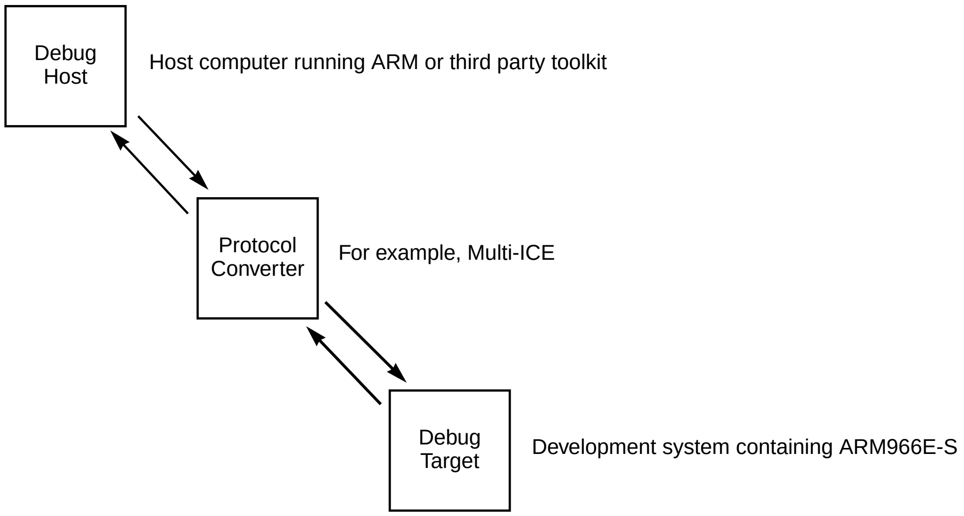
This chapter describes the operation of the ARM966E-S debug interface. The debug interface is based on IEEE Std 1149.1-1990. This chapter contains the following sections:

A more detailed description of the ARM9E-S debug features and JTAG interface is provided in Appendix D, Debug in Depth, of the *ARM9E-S Technical Reference Manual.*

## 10.1 Debug Systems

The ARM966E-S forms one component of a debug system that interfaces from the high-level debugging you perform to the low-level interface the ARM966E-S supports. Figure 10.1 shows a typical debug system.

**Figure 10.1 Typical Debug System**


Debug Host
Host computer running ARM or third party toolkit
Protocol Converter
For example, Multi-ICE
Debug Target
Development system containing ARM966E-S

A debug system typically has three parts:

- The Debug Host
- The Protocol Converter
- Debug Target

The debug host and the protocol converter are system-dependent.

## 10.1.1 The Debug Host

The debug host is a computer that is running a software debugger, such as *armsd*. The debug host allows you to issue high-level commands, such as setting breakpoints or examining the contents of memory.

## 10.1.2 The Protocol Converter

An interface, such as a parallel port, connects the debug host to the ARM966E-S development system. The messages broadcast over this connection must be converted to the interface signals of the ARM966E-S. The protocol converter performs this conversion.

## 10.1.3 Debug Target

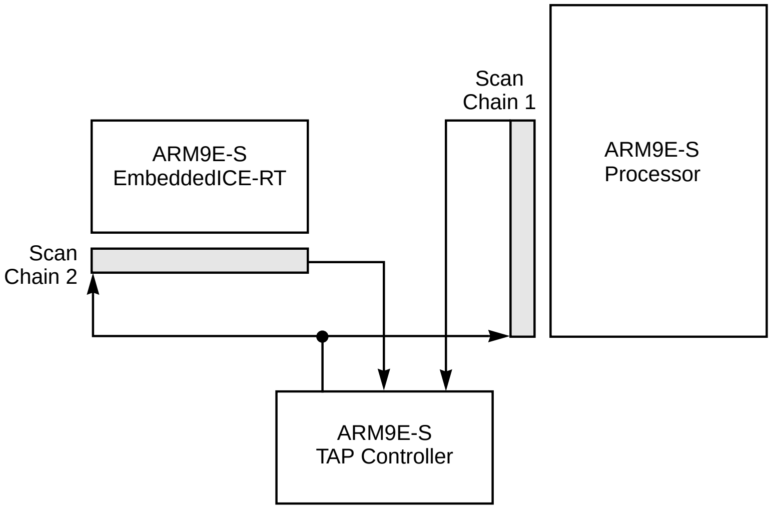
The ARM9E-S processor within the ARM966E-S has hardware extensions that ease debugging at the lowest level. The debug extensions allow you to:

- stall the core from program execution
- examine the core internal state
- examine the state of the memory system
- resume program execution.

The major blocks of the ARM9E-S debug model are described below and are shown in Figure 10.2:

| | |
|---|---|
| **ARM9E-S Processor Core** | Includes hardware support for debug. |
| **EmbeddedICE-RT Logic** | This set of registers and comparators is used to generate debug exceptions (such as breakpoints). This unit is described in Section 10.7, "About the EmbeddedICE-RT Logic," page 10-14. |
| **TAP Controller** | Controls the action of the scan chains using a JTAG serial interface. |

**Figure 10.2 ARM9E-S Processor and Debug Logic**



The ARM9E-S debug model is extended within the ARM966E-S with the addition of scan chain 15. This scan chain is used for debug access to the CP15 register bank when BIST is implemented. It allows the system state within the ARM966E-S to be configured while in the debug state, for instance, to enable or disable the SRAM before performing a debug load or store.

The rest of this chapter describes the hardware debug extensions.

## 10.2 About the Debug Interface

The ARM966E-S debug interface is based on IEEE Std. 1149.1-1990, *Standard Test Access Port and Boundary-Scan Architecture*. Refer to this standard for an explanation of the terms used in this chapter and for a description of the Test Access Port (TAP) controller states.

The ARM9E-S processor core contains hardware extensions for advanced debugging features, which make it easier to develop the hardware, application software, and operating systems.

The debug extensions allow you to force the core into the *debug state*. In the debug state, the ARM9E-S processor and ARM966E-S memory system are effectively stopped and isolated from the rest of the system in *halt mode*. From this mode, you can examine the internal state of the ARM9E-S processor, the ARM966E-S system, and the external state of the AHB while all other system activity continues as normal. When debug is complete, the ARM9E-S processor restores the core and system state, and resumes program execution.

In addition, the ARM9E-S supports a real-time debug mode called *monitor mode*, where instead of generating a breakpoint or watchpoint, an internal Instruction Abort or Data Abort is generated. When monitor mode is used in conjunction with a debug monitor program that is activated by the abort exception entry, you can debug the ARM966E-S while allowing the execution of critical interrupt service routines. The debug monitor program typically communicates with the debug host over the ARM966E-S debug communication channel. Monitor mode debug is described in Section 10.10, “Real-Time Debug,” page 10-20.

### 10.2.1 Stages of Debug

A request on one of the external debug interface signals or on an internal functional unit known as the *EmbeddedICE-RT logic* forces the ARM9E-S processor into the debug state. The interrupts that activate debug are:

- a breakpoint (a given instruction fetch)
- a watchpoint (a data access)
- an external debug request

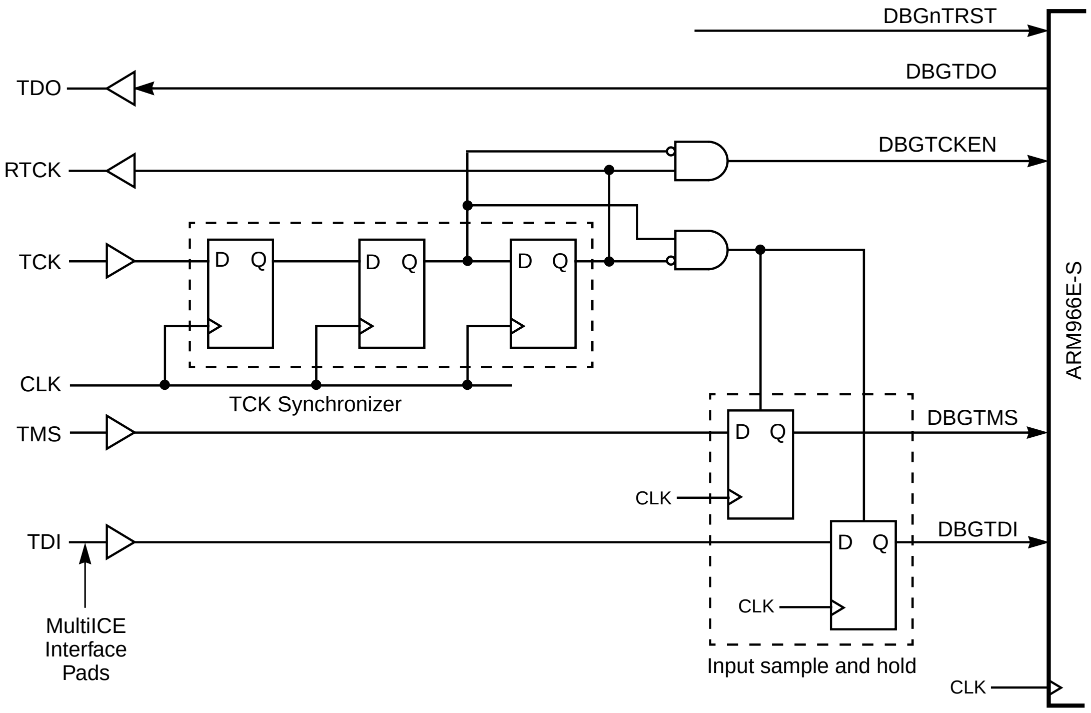
The internal state of the ARM9E-S processor is examined using a JTAG-style serial interface, allowing instructions to be serially inserted into the pipeline without using the external data bus. For example, when in the debug state, a Store Multiple instruction (STM) can be inserted into the instruction pipeline, which exports the contents of the ARM966E-S registers. This data can be serially shifted out without affecting the rest of the system.

### 10.2.2 Clocks

The system and test clocks must be synchronized externally to the ARM966E-S macrocell. The ARM Multi-ICE debug agent directly supports one or more cores within an ASIC design. To synchronize off-chip debug clocking with the ARM966E-S macrocell requires a three-stage synchronizer. The off-chip device (for example, Multi-ICE) issues a TCK signal, and then waits for the RTCK (Returned TCK) signal to come back. Synchronization is maintained because the off-chip device does not progress to the next TCK until after RTCK is received.

Figure 10.3 shows this synchronization logic.

**Figure 10.3 Clock Synchronization Logic**



## 10.3 Scan Chain 15

Scan chain 15 (SC15) provides debug access to the CP15 register bank, allowing the system state within the ARM966E-S to be configured while in the debug state.

Table 10.1 shows the order of SC15 from the DBGTDI input to the DBGTDO output.

**Table 10.1 Scan Chain 15 Addressing Mode Bit Order**

| Bits | Contents |
|---|---|
| 38 | Read = 0, write = 1 |
| 37:32 | CP15 register address |
| 31:0 | CP15 register value |

Table 10.2 shows the CP15 register address field of SC15, which provides debug access to the CP15 registers.

**Table 10.2 Mapping of Scan Chain 15 Address Field to CP15 Registers**

| Bit 38 | Bits[37:32] | Bits[31:30] | CP15 Reg Number | Meaning |
|---|---|---|---|---|
| 0 | 0 0000 0 | xx | C0 | Read ID Register |
| 0 | 0 0001 0 | xx | C1 | Read Control Register |
| 1 | 0 0001 0 | xx | C1 | Write Control Register |
| 0 | 1 1111 1 | 00 | C15 | Read BIST Control Register |
| 1 | 1 1111 1 | 00 | C15 | Write BIST Control Register |
| 0 | 1 1111 0 | 01 | C15 | Read IBIST Address |
| 1 | 1 1111 0 | 01 | C15 | Write IBIST Address |
| 0 | 1 1111 1 | 01 | C15 | Read IBIST General |
| 1 | 1 1111 1 | 01 | C15 | Write IBIST General |
| 0 | 1 1111 0 | 11 | C15 | Read DBIST Address |
| 1 | 1 1111 0 | 11 | C15 | Write DBIST Address |
| 0 | 1 1111 1 | 11 | C15 | Read DBIST General |
| 1 | 1 1111 1 | 11 | C15 | Write DBIST General |

The scan address decode supersedes the existing functional decode logic that is used to access the CP15 registers during MCR and MRC instructions (see Section 3.7.2, “CP15 Registers,” page 3-9).

The decode overload is performed as the follows:

| | |
|---|---|
| **Bit 37** | Corresponds to Opcode 1 of an MCR or MRC instruction. |
| **Bits [36:33]** | Correspond to the CRn field of an MCR or MRC instruction. |
| **Bit 32** | Corresponds to bit 0 of the Opcode 2 field of an MCR or MRC instruction. |
| **Bits [2:1]** | Bits [2:1] of opcode 2 are tied to 00 during the debug state. |

SC15 only allows access to bit 0 of the OpCode2 field by default. To allow access to the Address and General BIST registers within CP15

Register 15, bits [31:30] of SC15 are overloaded as shown in Table 10.2. There are certain restrictions with the overloading. When writing to the BIST General registers (writing a new seed), bits[31:30] of the seed are restricted to those values shown in Table 10.2. These bits are not used in the BIST Address registers; so there are no debug restrictions when accessing these registers.

The ability to control the ARM966E-S system state through scan chain 15 provides extra debug visibility. For example, if the debugger wishes to compare the contents of an address that maps to the I-SRAM or D-SRAM with the same address in external memory, the debugger can:

1. Load from the address with the SRAM enabled to return the SRAM data.
2. Disable the SRAM.
3. Perform the load again. The second load now accesses the AHB because the SRAM is disabled, returning the value from AHB memory.

## 10.4 Breakpoints, Watchpoints, and External Debug Requests

The ARM966E-S enters the debug state when a breakpoint, watchpoint, or external debug request occurs. There are four primary external signals associated with the debug interface:

- DBGIEBKPT, DBGDEWPT, and EDBGRQ are system requests for the ARM966E-S core to enter the debug state.
- The DBGACK signal informs the system that the ARM966E-S is in the debug state.

Several figures in the following subsections illustrate ARM966E-S waveforms in the debug state. Table 10.3 defines the notations used in these figures.

**Table 10.3 Debug State Figure Notations**

| Term | Definition |
|---|---|
| Fx, Dx, Ex, Mx, Wx | Fetch, Decode, Execute, Memory, and Writeback stages for instruction x on INSTR[31:0] |
| Ddebug | Decode debug entry |
| Edebug1, Edebug2 | Execute debug entry |
| ldr | Load register from memory instruction |
| Dp | Data processing instruction |
| B | Branch instruction |
| T | Trigger instruction |

## 10.4.1 Entry into Debug State on Breakpoint

Any instruction being fetched from memory is sampled at the end of a cycle. To apply a breakpoint to that instruction, the breakpoint signal must be asserted by the end of the same cycle as shown in Figure 10.4.

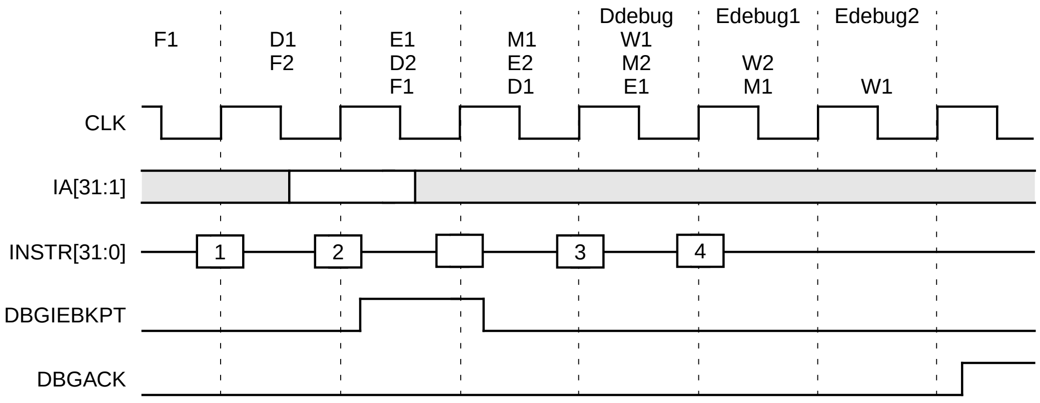
You can build external logic, such as additional breakpoint comparators, to extend the breakpoint functionality of the EmbeddedICE-RT logic. These outputs must be applied to the DBGIEBKPT input. This signal is ORed with the internally generated breakpoint signal before being applied to the ARM9E-S core control logic. The timing of the input makes it unlikely that data-dependent external breakpoints are possible.

A breakpointed instruction is allowed to enter the Execute stage of the pipeline, but any state change as a result of the instruction is prevented. All writes from previous instructions complete as usual.

The Decode cycle of the debug entry sequence occurs during the Execute cycle of the breakpointed instruction. The latched breakpoint signal forces the processor to start the debug sequence.

Figure 10.4 shows the breakpoint timing.

**Figure 10.4 Breakpoint Timing**



## 10.4.2 Breakpoints and Exceptions

A breakpointed instruction can have a Prefetch Abort associated with it. If so, the Prefetch Abort takes priority and the breakpoint is ignored. (If there is a prefetch abort, instruction data might be invalid, the breakpoint might be data-dependent, and if the data is incorrect, the breakpoint could have triggered incorrectly.)

SWI and undefined instructions are treated the same as any other instruction with a breakpoint set on it. Therefore the breakpoint takes priority over the SWI or undefined instruction.

On an instruction boundary, if there is a breakpointed instruction and an interrupt (nIRQ or nFIQ), the interrupt is taken and the breakpointed instruction is discarded. When the interrupt is being serviced, the execution flow is returned to the original program. Thus the instruction that was previously breakpointed is fetched again, and if the breakpoint is still set, the processor enters the debug state when it reaches the Execute stage of the pipeline.

When the processor enters the halt-mode debug state, it is important that further interrupts do not affect the instructions executed. For this reason, as soon as the processor enters the stop-mode debug state, interrupts are disabled, although the state of the I and F bits in the Program Status Register (PSR) are not affected.

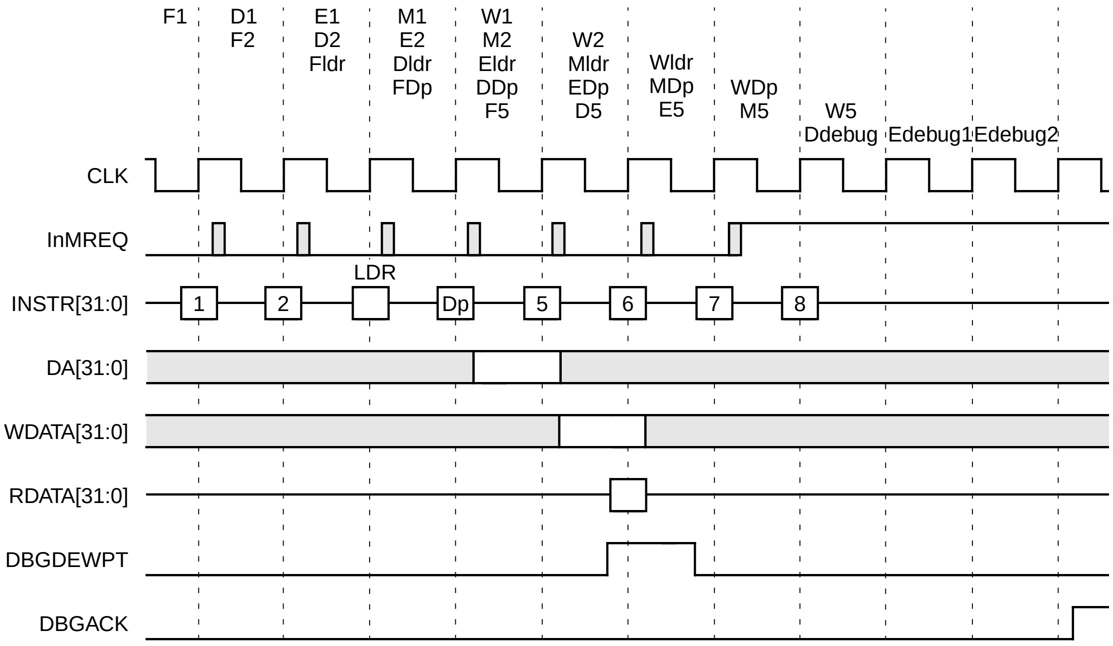
## 10.4.3 Watchpoints

Entry into the debug state following a watchpointed memory access is imprecise because of the nature of the pipeline.

External logic, such as external watchpoint comparators, can be built to extend the functionality of the EmbeddedICE-RT logic. Their outputs must be applied to the DBGDEWPT input. This signal is simply ORed with the internally generated Watchpoint signal before being applied to the ARM9E-S core control logic. The timing of the input makes it unlikely that data-dependent external watchpoints are possible.

After a watchpointed access, the next instruction in the processor pipeline is always allowed to complete execution. Where this instruction is a single-cycle data-processing instruction, entry into the debug state is delayed for one cycle while the instruction completes. The timing of debug entry following a watchpointed load in this case is shown in Figure 10.5.

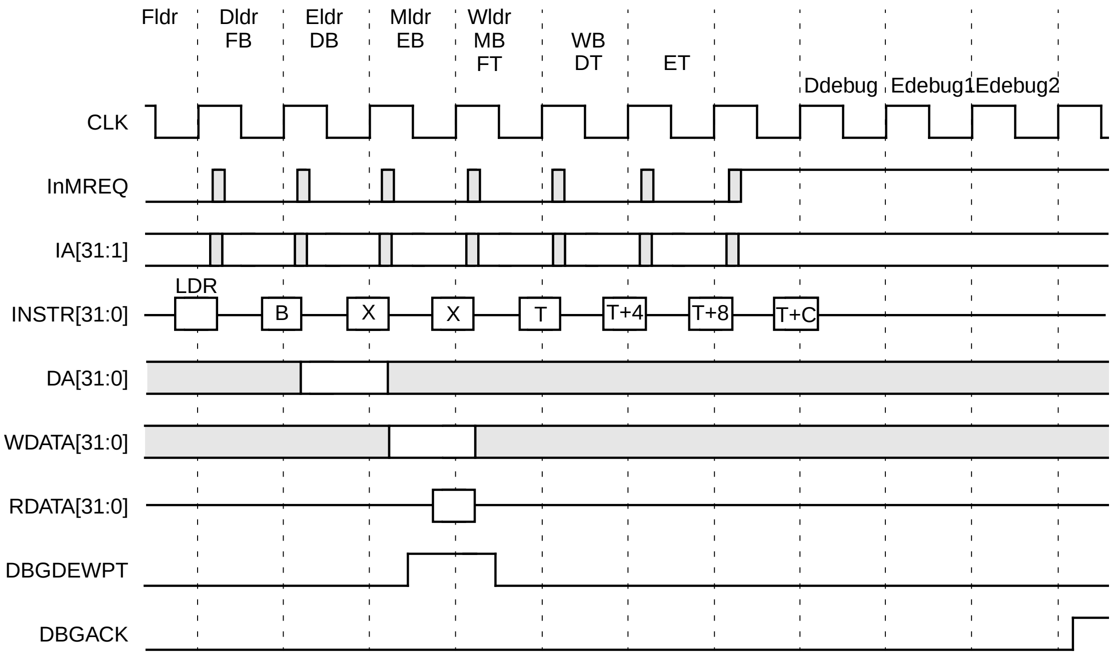
**Figure 10.5 Watchpoint Entry with Data Processing Instruction**



Although instruction 5 enters the Execute stage, it is not executed, and there is no state update as a result of this instruction. When the debugging session is complete, normal continuation involves a return to instruction 5, the next instruction in the code sequence to be executed.

The instruction following the instruction that generated the watchpoint might have modified the Program Counter (PC). So it is not possible to determine the instruction that caused the watchpoint. A timing diagram in Figure 10.6 shows debug entry after a watchpoint where the next instruction is a branch. However, it is always possible to restart the processor.

**Figure 10.6 Watchpoint Entry with Branch**



When the processor enters the debug state, the ARM9E-S core is interrogated to determine its state. In the case of a watchpoint, the PC contains a value that is five instructions after the address of the next instruction to be executed. Therefore if on entry to the debug state in ARM state, the instruction `SUB PC, PC, #20` is scanned in and the processor restarted, execution flow returns to the next instruction in the code sequence.

### 10.4.4 Watchpoints and Exceptions

If there is an abort with the data access as well as a watchpoint, the watchpoint condition is latched, the exception entry sequence performed, and then the processor enters the debug state. If there is an interrupt pending, the ARM9E-S core allows the exception entry sequence to occur and then enters the debug state.

### 10.4.5 Debug Request

A debug request can take place through the EmbeddedICE-RT logic or through assertion of the EDBGRQ signal. The request is synchronized and passed to the processor. A debug request takes priority over any pending interrupt. Following synchronization, the core enters the debug state when the instruction at the Execute stage of the pipeline is completed (when Memory and Write stages of the pipeline have completed). While waiting for the instruction to finish executing, no more instructions are issued to the Execute stage of the pipeline.

Caution: Asserting EDBGRQ in monitor mode results in unpredictable behavior.

### 10.4.6 Actions of the ARM9E-S Core in Debug State

When the ARM9E-S core is in the debug state, both memory interfaces indicate internal cycles, which ensures that both the tightly coupled SRAM within the ARM966E-S and the AHB interface are quiescent. The rest of the AHB system can thus ignore the ARM9E-S processor core and function as normal. Because the rest of the system continues operation, the ARM9E-S processor core ignores aborts and interrupts.

The nRESET signal must be held stable during debug. If the system applies a reset to the ARM966E-S (nRESET is driven LOW), the ARM9E-S processor changes state without the knowledge of the debugger.

## 10.5 ARM9E-S Clock Domains

The ARM9E-S clock, CLK, is qualified by two clock enables:

- SYSCLKEN controls access to the memory system
- DBGTCKEN controls debug operations

During normal operation, SYSCLKEN conditions CLK to clock the core. When the ARM966E-S is in the debug state, DBGTCKEN conditions CLK to clock the core.

## 10.6 Determining the Core and System States

When the ARM966E-S is in the debug state, you can examine the core and system state by forcing the load and store multiples into the instruction pipeline.

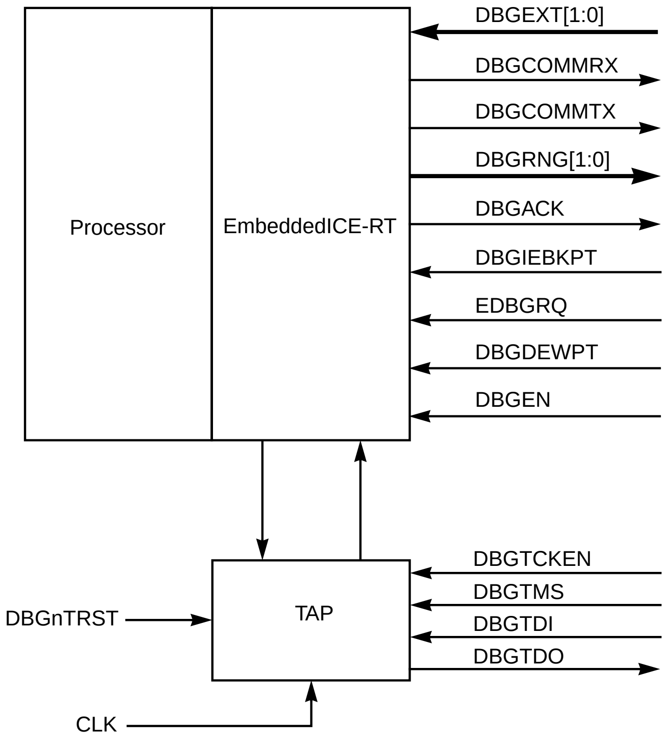
Before you can examine the core and system state, the debugger must determine whether the processor entered debug from Thumb state or ARM state, by examining bit 4 of the EmbeddedICE-RT Debug Status Register. When bit 4 is HIGH, the core enters debug from Thumb state.

## 10.7 About the EmbeddedICE-RT Logic

The ARM9E-S EmbeddedICE-RT logic provides integrated on-chip debug support for the ARM9E-S processor core.

EmbeddedICE-RT is programmed serially using the ARM9E-S TAP controller. Figure 10.7 illustrates the relationship between the core, EmbeddedICE-RT, and the TAP controller, showing only the signals that are pertinent to EmbeddedICE-RT.

**Figure 10.7 The ARM9E-S, TAP Controller, and EmbeddedICE-RT**



The EmbeddedICE-RT logic comprises:

- two real-time watchpoint units
- two independent registers: the debug control register and the debug status register
- debug communications channel.

The Debug Control Register and the Debug Status Register provide overall control of EmbeddedICE-RT operation.

You can program one or both watchpoint units to halt the execution of instructions by the core. Execution halts when the values programmed into the EmbeddedICE-RT match the values currently appearing on the address bus, data bus, and various control signals.

Any bit can be masked so that its value does not affect the comparison.

Each watchpoint unit can be configured to be either a watchpoint (monitoring data accesses) or a breakpoint (monitoring instruction fetches). Watchpoints and breakpoints can be data-dependent.

## 10.8 Disabling the EmbeddedICE-RT Logic

To disable the EmbeddedICE-RT logic, force the DBGEN input LOW. Hardwiring DBGEN LOW *permanently* disables debug access.

When DBGEN is LOW, it inhibits DBGDEWPT, DBGIEBKPT, and EDBGRQ to the ARM9E-S, and DBGACK from the ARM966E-S is always LOW.

## 10.9 The Debug Communications Channel

The EmbeddedICE-RT logic contains a communications channel for passing information between the target and the host debugger. This channel is implemented as coprocessor 14 (CP14).

The communications channel comprises:

- a 32-bit communications data read register
- a 32-bit communications data write register
- a 6-bit communications control register for synchronized handshaking between the processor and the asynchronous debugger.

These registers are located in fixed locations in the EmbeddedICE-RT logic register map and are accessed from the processor using MCR and MRC instructions to CP14.

In addition to the communications channel registers, the processor can access a one-bit Debug Status Register for use in the real-time debug configuration.

## 10.9.1 Debug Communication Channel Registers

CP14 contains four registers. Table 10.4 shows the register allocations in CP14.

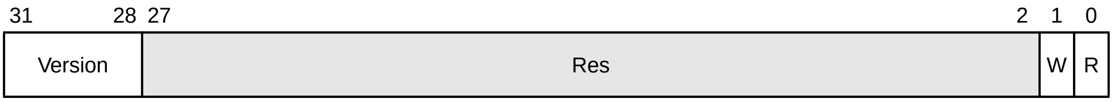
**Table 10.4 CP14 Register Map**

| Register Name | Register Number | Notes |
|---|---|---|
| Communications Channel Status | C0 | Read only |
| Communications Channel Data Read | C1 | For reads |
| Communications Channel Data Write | C1 | For writes |
| Communications Channel Monitor Mode Debug Status | C2 | Read/write |

## 10.9.2 Debug Communications Channel Status Register

The Debug Communications Channel Status Register is read-only. It controls synchronized handshaking between the processor and the debugger. The Debug Communications Channel Status Register is shown in Figure 10.8.

**Figure 10.8 Debug Communications Channel Status Register**



**Version** **Version** **[31:28]**
This field contains a fixed pattern that denotes the EmbeddedICE-RT version number (in this case 0b0011).

**Res** **Reserved** **[27:2]**
These bits are reserved and read as zeros.

**W** **Write Available** **1**
This bit indicates whether the Communications Data Write Register is available (from the viewpoint of the processor). If, from the viewpoint of the processor, the Communications Data Write Register is free (W = 0), new data can be written. If the register is not free (W = 1), the processor must poll until W = 0. From the viewpoint of the debugger, when W = 1, new data is written that can be scanned out.

| | | |
|---|---|---|
| **R** | **Read Available**<br>This bit indicates whether there is new data in the Communications Data Read Register. From the viewpoint of the processor, if R = 1, there is new data that can be read using an MRC instruction. From the viewpoint of the debugger, if R = 0, the Communications Data Read Register is free, and new data can be placed there through the scan chain. If R = 1, data previously placed there through the scan chain is not collected by the processor, and so the debugger must wait. | **0** |

From the viewpoint of the debugger, the registers are accessed using the scan chain in the usual way. From the viewpoint of the processor, these registers are accessed using coprocessor register transfer instructions.

You must use the following instructions:

| | |
|---|---|
| **MRC p14, 0, Rd, c0, c0** | Returns the Debug Communications Control Register into Rd. |
| **MCR p14, 0, Rn, c1, c0** | Writes the value in Rn to the Communications Data Write Register. |
| **MRC p14, 0, Rd, c1, c0** | Returns the Debug Data Read Register into Rd. |

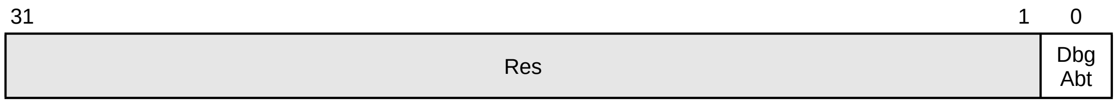
Because the Thumb instruction set does not contain coprocessor instructions, you are advised to access this data using SWI instructions when in Thumb state.

### 10.9.3 Communications Channel Monitor Mode Debug Status Register

The CP14 Debug Status Register is provided for a debug monitor when the ARM9E-S processor is configured into monitor mode.

The CP14 Debug Status Register is a one-bit wide read/write register as shown in Figure 10.9.

**Figure 10.9 Coprocessor 14 Debug Status Register Format**



Bit 0, the DbgAbt bit, indicates whether the processor took a Prefetch (DbgAbt = 1) or Data Abort (DbgAbt = 0) in the past because of a breakpoint or watchpoint. If the ARM966E-S takes a Prefetch Abort as a result of a breakpoint or watchpoint, then the bit is set. If on a particular instruction or data fetch, both the debug abort and external abort signals are asserted, the external abort takes priority and the DbgAbt bit is not set. You can read or write the DbgAbt bit by means of MRC or MCR instructions.

This bit can be used by a real-time debug aware abort handler. This handler examines the DbgAbt bit to determine whether the abort is externally or internally generated. If the DbgAbt bit is set, the abort handler initiates communication with the debugger over the communications channel.

## 10.9.4 Using the Communications Channel

Messages can be sent and received using the communications channel.

### 10.9.4.1 Sending a Message to the Debugger

When the processor wishes to send a message to the debugger, it must check to see if the Communications Data Write Register is free for use. The processor reads the Debug Communications Control Register to check the status of the W bit.

- If W bit is cleared, the Communications Data Write Register is free for use.
- If the W bit is set, previously written data is not read by the debugger. The processor must continue to poll the Debug Communications Control Register until the W bit is cleared.

When the W bit is cleared, a message is written by a register transfer to coprocessor 14. Because the data transfer occurs from the processor to the Communications Data Write Register, the W bit is set in the Debug Communications Control Register.

The debugger sees both the R and W bits when it polls the Debug Communications Control Register through the JTAG interface. When the debugger sees that the W bit is set, it can read the Communications Data Write Register, and scan the data out. The action of reading this data register clears the W bit in the Debug Communications Control Register. At this point, the communications process can begin again.

### 10.9.4.2 Receiving a Message from the Debugger

Transferring a message from the debugger to the processor is similar to sending a message to the debugger. In this case, the debugger polls the R bit of the Debug Communications Control Register:

- if the R bit is cleared, the Communications Data Read Register is free for use, and data can be placed there for the processor to read
- if the R bit is set, previously deposited data is not yet collected, so the debugger must wait.

When the communications data read register is free, data is written there using the JTAG interface. The action of this write sets the R bit in the Debug Communications Control Register.

The processor polls the Debug Communications Control Register. If the R bit is set, there is data that can be read using an MRC instruction to CP14. The action of this load clears the R bit in the Debug Communications Control Register. When the debugger polls this register and sees that the R bit is cleared, the data is taken, and the process can be repeated.

## 10.10 Real-Time Debug

The ARM9E-S processor within the ARM966E-S contains logic that allows the debugging of a system without stopping the core entirely. Thus critical interrupt routines can still be serviced while the core is being interrogated by the debugger. Setting bit 4 of the Debug Control Register enables the real-time debug features of the ARM9E-S. When bit 4 is set, the EmbeddedICE-RT logic is configured so that a breakpoint or watchpoint causes the ARM966E-S to enter abort mode, taking the Prefetch Abort or Data Abort vectors, respectively. When the ARM966E-S is configured for real-time debugging, you must be aware of the following restrictions:

- Breakpoints or watchpoints might not be data-dependent. No support is provided for use of the range and chain functionalities. Breakpoints or watchpoints can only be based on:
  - instruction or data addresses
  - external watchpoint conditioner (DBGEXTERN)

  - user or privileged mode access (DnTRANS and InTRANS)
  - read or write access (watchpoints)
  - access size (breakpoints, ITBIT, and watchpoints, DMAS[1:0]).
- The single-step hardware is not enabled.
- External breakpoints and watchpoints are not supported.
- The vector-catching hardware can be used but must not be configured to catch the Prefetch or Data Abort exceptions.

Caution: No support is provided to mix halt mode and monitor mode debug functionalities. When the core is configured into the monitor mode, asserting the external EDBGRQ signal results in unpredictable behavior. Setting the internal EDBGRQ bit results in unpredictable behavior.

When an abort is generated by the monitor mode it is recorded in the Debug Status Register in CP14 (see Section 10.9.3, “Communications Channel Monitor Mode Debug Status Register”).

Because the monitor mode debug does not put the ARM9E-S into the debug state, it is necessary to change the contents of the Watchpoint registers while external memory accesses are taking place rather than changing them when in the debug state. If the Watchpoint registers are written to during an access, all matches from the affected watchpoint unit using the register being updated are disabled for the cycle of the update.

If there is a possibility of false matches occurring during changes to the Watchpoint registers, caused by old data in some registers and new data in others, then you must:

1. Disable that watchpoint unit using the control register for that watchpoint unit.
2. Change the other registers.
3. Re-enable the watchpoint unit by rewriting the control register.

# Chapter 11
# Test Methodology

This chapter describes the ARM966E-S test methodology. Full-scan ATPG and RAMBIST are provided to achieve >99% stuck at faults and RAM data sensitivity faults. This chapter includes the following sections:

- Section 11.1, “Scan Insertion”
- Section 11.2, “RAMBIST”

For additional information, please contact your LSI Logic applications engineer.

## 11.1 Scan Insertion

Scan insertion ensures a high level of fault coverage using Automatic Test Pattern Generation (ATPG) tools and compatible synthesis library cells. It has an impact on the area and performance of a design. It also imposes constraints on the use of clock gating within the HDL code.

## 11.2 RAMBIST

LSI Logic does not implement ARM’s ARM966E-S RAMBIST controller. Check with your LSI Logic applications engineer on how to add RAMBIST to your design.

# Appendix A
# ARM9E-S Enhanced Instructions

This appendix describes the instruction enhancements made to the ARM9E-S instruction set. The extensions have been developed to improve the ARM architecture's performance in signal processing algorithms.

Many real-time applications require both the benefits of a microcontroller and a DSP. Microcontrollers bring high-level language support with solid development tools, low cost memory systems, low interrupt latency, and fast context switching time; DSPs tend to have features for fast math performance in real-time control tasks.

Real-time applications are typically those involving devices that move, including hard disk drives, printers, engine controllers, and general-purpose servos such as those found in automotive steering control. However, applications such as voice processing and modems also benefit greatly from a controller with the mix of both microcontroller and DSP functionalities. It is for these reasons that recent microcontroller architectures have featured both controller and DSP functionality.

The ARM9E-S core features an enhanced 32 x 16 hardware multiplier, which increases the ARM's performance in hard real-time signal processing applications. The extensions consist of:

- New multiply instructions that allow the efficient use of data bandwidth so that best use can be made of the new multiplier.
- New saturation extensions to existing math instructions for use in the design of stable control loops and bit-exact algorithms.

In each of the instruction definitions, the following abbreviations are used:

| | |
|---|---|
| `{cond}` | Two character conditional execution mnemonic. Refer to the ARM Architecture Reference Manual for a complete listing of condition codes. |
| `Rd,Rs,Rn,Rm` | Denote ARM register numbers. |
| `x/y` | Denotes use of either the high or low half of an ARM register as a source operand. ‘T’ is used to denote the top (most significant), and ‘B’ denotes the bottom (least significant) halfword. |
| `SBZ` | A field in the instruction opcode that should be written as zero. Nonzero values produce unpredictable results. |
| `SignExtend(a)` | Sign-extends (propagates the sign bit) its argument to 32 bits. |
| `SAT(f)` | Saturates the result of *f* to 32 bits. If *f* overflows 32 bits and is negative, then SAT(*f*) returns the maximum negative value that can be represented in 32 bits (0x80000000). If *f* overflows 32 bits and is positive, SAT(*f*) returns 0x7FFFFFFF. |
| `DoesSat(f)` | Returns 1 if *f* would saturate when truncated to 32 bits, otherwise returns 0. |

Note also that in the instruction definitions, the term UNPREDICTABLE means the result of an instruction cannot be relied upon. Unpredictable instructions or results must not represent security holes. UNPREDICTABLE instructions must not halt or hang the processor, or any parts of the system.

## CLZ Count Leading Zeros

| 31 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 | 6 | 5 | 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | SBZ | Rd | SBZ | 0 | 0 | 0 | 1 | Rm |

**Format** `CLZ{cond} Rd, Rm`

**Description** This operation returns the number of leading zeros in the register Rm into the destination register Rd. This operation was originally defined as part of the ARMv5T instruction set extensions. It is mentioned here because CLZ is supported in an ARM9E-compatible processor regardless of whether that processor also supports the full ARMv5 instruction set. Used for speeding up the normalization at the start of a Newton-Raphson based division and normalization of fixed-point numbers.

**Operation**

```
if (Rm == 0)
                  Rd = 32
else
                  for (i = 31; i >= 0; i--) {
                         if (Rm[i] == 1) then
                                Rd = 31 – i
                                break
                  }
```

**Exceptions** None

**Qualifiers** Condition Code

**Notes** **Writing to R15:** Specifying R15 (PC) for register Rd or Rm is UNPREDICTABLE.

## QADD Saturating Add

| 31–28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19–16 | 15–12 | 11–8 | 7 | 6 | 5 | 4 | 3–0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Rn | Rd | SBZ | 0 | 1 | 0 | 1 | Rm |

**Format**

```
QADD{cond} Rd, Rm, Rn
```

**Description**

This instruction saturates the sum of Rm and Rn and stores the result in Rd. There are no immediate or shifted variants of this instruction. This operation affects the sticky-overflow bit ‘S’ due to overflow in addition.

**Operation**

```
Rd[31:0] = SAT(Rm[31:0] + Rn[31:0])
if (DoesSat(Rm[31:0] + Rn[31:0]))
                S Flag = 1
```

**Exceptions**

None

**Qualifiers**

Condition Code

**Notes**

**Writing to R15:** Specifying R15 (PC) for register Rd, Rn, or Rm is UNPREDICTABLE.

**Assembler Mnemonic:** This mnemonic has reversed operands to be consistent with the QDSUB mnemonic.

## QDADD — Saturated Double Rn and Saturated Add

| 31 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 | 6 | 5 | 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | Rn | Rd | SBZ | 0 | 1 | 0 | 1 | Rm |

**Format**

`QDADD{cond} Rd, Rm, Rn`

**Description**

Double Rn and saturate, then add to Rm and saturate. The result is stored in Rd. There are no immediate or shifted variants of this instruction. This operation affects the sticky-overflow bit ‘S’ due to overflow in either the double operation or the addition.

**Operation**

```
Rd[31:0] = SAT(Rm[31:0] + SAT(Rn[31:0] *2))
if (DoesSat(Rn[31:0]*2) || DoesSat(Rm[31:0] +
SAT(Rn[31:0]*2)))
                S Flag = 1
```

**Exceptions**

None

**Qualifiers**

Condition Code

**Notes**

**Writing to R15:** Specifying R15 (PC) for register Rd, Rm, or Rn is UNPREDICTABLE.

**Assembler Mnemonic:** This mnemonic has reversed operands to be consistent with the QDSUB mnemonic.

## QDSUB — Saturated Double Rn and Saturated Subtract

| 31 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 | 6 | 5 | 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | Rn | Rd | SBZ | 0 | 1 | 0 | 1 | Rm |

**Format**

```
QDSUB{cond} Rd, Rm, Rn
```

**Description**

Double Rn and saturate, then subtract from Rm and saturate. The result is stored in Rd. There are no immediate or shifted variants of this instruction. This operation affects the sticky-overflow bit ‘S’ due to overflow in either the double operation or the subtraction.

**Operation**

```
Rd[31:0] = SAT(Rm[31:0] – SAT(Rn[31:0]*2))
if (DoesSat(Rn[31:0]*2) || DoesSat(Rm[31:0] –
SAT(Rn[31:0]*2)))
                S Flag = 1
```

**Exceptions**

None

**Qualifiers**

Condition Code

**Notes**

**Writing to R15:** Specifying R15 (PC) for register Rd, Rm or Rn is UNPREDICTABLE.

**Assembler Mnemonic:** This operation performs a reverse subtract (when compared to the normal ARM SUB operation). The operands in the assembler mnemonic have been reversed so that the mnemonic can be QDSUB instead of QDRSB.

## QSUB Saturating Subtract

| 31–28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19–16 | 15–12 | 11–8 | 7 | 6 | 5 | 4 | 3–0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Rn | Rd | SBZ | 0 | 1 | 0 | 1 | Rm |

**Format** `QSUB{cond} Rd, Rm, Rn`

**Description** Subtract Rn from Rm and saturate. The result is stored in Rd. There are no immediate or shifted variants of this instruction. This operation affects the sticky-overflow bit 'S' due to overflow in the subtraction.

**Operation**

```
Rd[31:0] = SAT(Rm[31:0] - Rn[31:0])
if (DoesSat(Rm[31:0] – Rn[31:0]))
                    S Flag = 1
```

**Exceptions** None

**Qualifiers** Condition Code

**Notes** **Writing to R15:** Specifying R15 (PC) for register Rd, Rm or Rn is UNPREDICTABLE.

**Assembler Mnemonic:** This mnemonic has reversed operands to be consistent with the QDSUB mnemonic.

**SMLAxy** **Signed Integer Multiply-Accumulate**

| 31 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 | 6 | 5 | 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | Rd | Rn | Rs | 1 | y | x | 0 | Rm |

**Format** `SMLAxy {cond} Rd, Rm, Rs, Rn`

**Description** The signed integer multiply-accumulate operation performs the multiply on two 16-bit source operands from half of register Rm and half of Rs, producing a 32-bit product and then a 32-bit accumulation with Rn. This operation affects the sticky-overflow bit ‘S’ due to overflow in the accumulation.

**Operation**

```
if (bit[5] == 0)
   <operand1> = SignExtend(Rm[15:0])
else
   <operand1> = SignExtend(Rm[31:16])

if (bit[6] == 0)
   <operand2> = SignExtend(Rs[15:0])
else
   <operand2> = SignExtend(Rs[31:16])

Rd[31:0] = Rn[31:0] + (<operand1> * <operand2>)
if (DoesSat(Rn[31:0] + (<operand1> * <operand2>)))
                S Flag = 1
```

**Exceptions** None

**Qualifiers** Condition Code

**Notes** **Writing to R15:** Specifying R15 (PC) for register Rd, Rm, Rs or Rn is UNPREDICTABLE.

## SMLALxy Signed Multiply-Accumulate

| 31 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 | 6 | 5 | 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | RdHi | RdLo | Rs | 1 | y | x | 0 | Rm |

**Format** `SMLALxy{cond} RdLo, RdHi, Rm, Rs`

**Description** The signed multiply-accumulate operation performs a multiply on two 16-bit source operands from half of register Rm and half of Rs. Then a 64-bit accumulate is done with RdLo and RdHi.

**Operation**

```
if (bit[5] == 0)
   <operand1> = SignExtend(Rm[15:0])
else
   <operand1> = SignExtend(Rm[31:16])

if (bit[6] == 0)
   <operand2> = SignExtend(Rs[15:0])
else
   <operand2> = SignExtend(Rs[31:16])

RdHi[31:0].RdLo[31:0] = RdHi[31:0].RdLo[31:0] + (<operand1>
* <operand2>)
```

**Exceptions** None

**Qualifiers** Condition Code

**Notes** **Writing to R15:** Specifying R15 (PC) for register RdHi, RdLo, Rm, or Rs is UNPREDICTABLE.

## SMLAWy Signed Integer Multiply-Accumulate

| 31 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 | 6 | 5 | 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Rd | Rn | Rs | 1 | y | 0 | 0 | Rm |

**Format** `SMLAWy{cond} Rd, Rm, Rs, Rn`

**Description** The signed integer multiply-accumulate operation performs a 32 x 16 bit multiply on the 32-bit operand in Rm and the 16-bit source operand from half of register Rs. A 32-bit accumulate of the upper 32 bits of the 48-bit product is done with Rn. This operation affects the sticky-overflow bit ‘S’ due to overflow in the accumulation.

**Operation**

```
if (bit[6] == 0)
   <operand2> = SignExtend(Rs[15:0])
else
   <operand2> = SignExtend(Rs[31:16])

Rd[31:0] = Rn[31:0] + (Rm[31:0]*<operand2>)[47:16]
if (DoesSat(Rn[31:0] + (Rm[31:0]*<operand2>)[47:16]))
                S Flag = 1
```

**Exceptions** None

**Qualifiers** Condition Code

**Notes** **Writing to R15:** Specifying R15 (PC) for register Rd, Rm, Rs, or Rn is UNPREDICTABLE.

## SMULxy Signed Integer Multiply

| 31 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 16 | 15 12 | 11 8 | 7 | 6 | 5 | 4 | 3 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | Rd | SBZ | Rs | 1 | y | x | 0 | Rm |

**Format**

```
SMULxy{cond} Rd, Rm, Rs
```

**Description**

The signed integer multiply operation performs the multiply on two 16-bit source operands from half of register Rm and half of Rs, producing a 32-bit result in Rd.

**Operation**

```
if (bit[5] == 0)
   <operand1> = SignExtend(Rm[15:0])
else
   <operand1> = SignExtend(Rm[31:16])

if (bit[6] == 0)
   <operand2> = SignExtend(Rs[15:0])
else
   <operand2> = SignExtend(Rs[31:16])

Rd[31:0] = (<operand1> * <operand2>)
```

**Exceptions**

None

**Qualifiers**

Condition Code

**Notes**

**Writing to R15:** Specifying R15 (PC) for register Rd, Rm, or Rs is UNPREDICTABLE.

## SMULWy — Signed Integer Multiply

| 31–28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19–16 | 15–12 | 11–8 | 7 | 6 | 5 | 4 | 3–0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| cond | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | Rd | SBZ | Rs | 1 | y | 1 | 0 | Rm |

**Format**

`SMULWy{cond} Rd, Rm, Rs`

**Description**

The signed integer multiply operation performs a 32 x 16 bit multiply on the 32-bit operand in Rm and the 16-bit source operand from half of register Rs, taking the upper 32 bits of the 48-bit product.

**Operation**

```
if (bit[6] == 0)
   <operand2> = SignExtend(Rs[15:0])
else
   <operand2> = SignExtend(Rs[31:16])

Rd[31:0] = (Rm[31:0]*<operand2>)[47:16]
```

**Exceptions**

None

**Qualifiers**

Condition Code

**Notes**

**Writing to R15:** Specifying R15 (PC) for register Rd, Rm, or Rs is UNPREDICTABLE.

# Index

# Customer Feedback

We would appreciate your feedback on this document. Please copy the following page, add your comments, and fax it to us at the number shown.

If appropriate, please also fax copies of any marked-up pages from this document.

Important: Please include your name, phone number, fax number, and company address so that we may contact you directly for clarification or additional information.

Thank you for your help in improving the quality of our documents.

## Reader's Comments

| | |
|---|---|
| Fax your comments to: | LSI Logic Corporation<br>Technical Publications<br>M/S E-198<br>Fax: 408.433.4333 |

Please tell us how you rate this document: *ARM966E-S Microprocessor Core Technical Manual.* Place a check mark in the appropriate blank for each category.

| | **Excellent** | **Good** | **Average** | **Fair** | **Poor** |
|---|---|---|---|---|---|
| Completeness of information | ____ | ____ | ____ | ____ | ____ |
| Clarity of information | ____ | ____ | ____ | ____ | ____ |
| Ease of finding information | ____ | ____ | ____ | ____ | ____ |
| Technical content | ____ | ____ | ____ | ____ | ____ |
| Usefulness of examples and illustrations | ____ | ____ | ____ | ____ | ____ |
| Overall manual | ____ | ____ | ____ | ____ | ____ |

What could we do to improve this document?

If you found errors in this document, please specify the error and page number. If appropriate, please fax a marked-up copy of the page(s).

Please complete the information below so that we may contact you directly for clarification or additional information.

Name ______________________ Date ____________

Telephone ________________ Fax ________________

Title ______________________________________

Department ________________ Mail Stop ________

Company Name ______________________________

Street _____________________________________

City, State, Zip ____________________________

# U.S. Distributors by State

| | |
|---|---|
| A. E. | Avnet Electronics |
| http://www.hh.avnet.com | |
| B. M. | Bell Microproducts, Inc. (for HAB's) |
| http://www.bellmicro.com | |
| I. E. | Insight Electronics |
| http://www.insight-electronics.com | |
| W. E. | Wyle Electronics |
| http://www.wyle.com | |

**Alabama**
Daphne
I. E. Tel: 334.626.6190
Huntsville
A. E. Tel: 256.837.8700
B. M. Tel: 256.705.3559
I. E. Tel: 256.830.1222
W. E. Tel: 800.964.9953

**Alaska**
A. E. Tel: 800.332.8638

**Arizona**
Phoenix
A. E. Tel: 480.736.7000
B. M. Tel: 602.267.9551
W. E. Tel: 800.528.4040
Tempe
I. E. Tel: 480.829.1800
Tucson
A. E. Tel: 520.742.0515

**Arkansas**
W. E. Tel: 972.235.9953

**California**
Agoura Hills
B. M. Tel: 818.865.0266
Granite Bay
B. M. Tel: 916.523.7047
Irvine
A. E. Tel: 949.789.4100
B. M. Tel: 949.470.2900
I. E. Tel: 949.727.3291
W. E. Tel: 800.626.9953
Los Angeles
A. E. Tel: 818.594.0404
W. E. Tel: 800.288.9953
Sacramento
A. E. Tel: 916.632.4500
W. E. Tel: 800.627.9953
San Diego
A. E. Tel: 858.385.7500
B. M. Tel: 858.597.3010
I. E. Tel: 800.677.6011
W. E. Tel: 800.829.9953
San Jose
A. E. Tel: 408.435.3500
B. M. Tel: 408.436.0881
I. E. Tel: 408.952.7000
Santa Clara
W. E. Tel: 800.866.9953
Woodland Hills
A. E. Tel: 818.594.0404
Westlake Village
I. E. Tel: 818.707.2101

**Colorado**
Denver
A. E. Tel: 303.790.1662
B. M. Tel: 303.846.3065
W. E. Tel: 800.933.9953
Englewood
I. E. Tel: 303.649.1800
Idaho Springs
B. M. Tel: 303.567.0703

**Connecticut**
Cheshire
A. E. Tel: 203.271.5700
I. E. Tel: 203.272.5843
Wallingford
W. E. Tel: 800.605.9953

**Delaware**
North/South
A. E. Tel: 800.526.4812
Tel: 800.638.5988
B. M. Tel: 302.328.8968
W. E. Tel: 856.439.9110

**Florida**
Altamonte Springs
B. M. Tel: 407.682.1199
I. E. Tel: 407.834.6310
Boca Raton
I. E. Tel: 561.997.2540
Bonita Springs
B. M. Tel: 941.498.6011
Clearwater
I. E. Tel: 727.524.8850
Fort Lauderdale
A. E. Tel: 954.484.5482
W. E. Tel: 800.568.9953
Miami
B. M. Tel: 305.477.6406
Orlando
A. E. Tel: 407.657.3300
W. E. Tel: 407.740.7450
Tampa
W. E. Tel: 800.395.9953
St. Petersburg
A. E. Tel: 727.507.5000

**Georgia**
Atlanta
A. E. Tel: 770.623.4400
B. M. Tel: 770.980.4922
W. E. Tel: 800.876.9953
Duluth
I. E. Tel: 678.584.0812

**Hawaii**
A. E. Tel: 800.851.2282

**Idaho**
A. E. Tel: 801.365.3800
W. E. Tel: 801.974.9953

**Illinois**
North/South
A. E. Tel: 847.797.7300
Tel: 314.291.5350
Chicago
B. M. Tel: 847.413.8530
W. E. Tel: 800.853.9953
Schaumburg
I. E. Tel: 847.885.9700

**Indiana**
Fort Wayne
I. E. Tel: 219.436.4250
W. E. Tel: 888.358.9953
Indianapolis
A. E. Tel: 317.575.3500

**Iowa**
W. E. Tel: 612.853.2280
Cedar Rapids
A. E. Tel: 319.393.0033

**Kansas**
W. E. Tel: 303.457.9953
Kansas City
A. E. Tel: 913.663.7900
Lenexa
I. E. Tel: 913.492.0408

**Kentucky**
W. E. Tel: 937.436.9953
Central/Northern/ Western
A. E. Tel: 800.984.9503
Tel: 800.767.0329
Tel: 800.829.0146

**Louisiana**
W. E. Tel: 713.854.9953
North/South
A. E. Tel: 800.231.0253
Tel: 800.231.5775

**Maine**
A. E. Tel: 800.272.9255
W. E. Tel: 781.271.9953

**Maryland**
Baltimore
A. E. Tel: 410.720.3400
W. E. Tel: 800.863.9953
Columbia
B. M. Tel: 800.673.7461
I. E. Tel: 410.381.3131

**Massachusetts**
Boston
A. E. Tel: 978.532.9808
W. E. Tel: 800.444.9953
Burlington
I. E. Tel: 781.270.9400
Marlborough
B. M. Tel: 800.673.7459
Woburn
B. M. Tel: 800.552.4305

**Michigan**
Brighton
I. E. Tel: 810.229.7710
Detroit
A. E. Tel: 734.416.5800
W. E. Tel: 888.318.9953
Clarkston
B. M. Tel: 877.922.9363

**Minnesota**
Champlin
B. M. Tel: 800.557.2566
Eden Prairie
B. M. Tel: 800.255.1469
Minneapolis
A. E. Tel: 612.346.3000
W. E. Tel: 800.860.9953
St. Louis Park
I. E. Tel: 612.525.9999

**Mississippi**
A. E. Tel: 800.633.2918
W. E. Tel: 256.830.1119

**Missouri**
W. E. Tel: 630.620.0969
St. Louis
A. E. Tel: 314.291.5350
I. E. Tel: 314.872.2182

**Montana**
A. E. Tel: 800.526.1741
W. E. Tel: 801.974.9953

**Nebraska**
A. E. Tel: 800.332.4375
W. E. Tel: 303.457.9953

**Nevada**
Las Vegas
A. E. Tel: 800.528.8471
W. E. Tel: 702.765.7117

**New Hampshire**
A. E. Tel: 800.272.9255
W. E. Tel: 781.271.9953

**New Jersey**
North/South
A. E. Tel: 201.515.1641
Tel: 609.222.6400
Mt. Laurel
I. E. Tel: 856.222.9566
Pine Brook
B. M. Tel: 973.244.9668
W. E. Tel: 800.862.9953
Parsippany
I. E. Tel: 973.299.4425
Wayne
W. E. Tel: 973.237.9010

**New Mexico**
W. E. Tel: 480.804.7000
Albuquerque
A. E. Tel: 505.293.5119

## U.S. Distributors by State (Continued)

**New York**
Hauppauge
I. E. Tel: 516.761.0960
Long Island
A. E. Tel: 516.434.7400
W. E. Tel: 800.861.9953
Rochester
A. E. Tel: 716.475.9130
I. E. Tel: 716.242.7790
W. E. Tel: 800.319.9953
Smithtown
B. M. Tel: 800.543.2008
Syracuse
A. E. Tel: 315.449.4927

**North Carolina**
Raleigh
A. E. Tel: 919.859.9159
I. E. Tel: 919.873.9922
W. E. Tel: 800.560.9953

**North Dakota**
A. E. Tel: 800.829.0116
W. E. Tel: 612.853.2280

**Ohio**
Cleveland
A. E. Tel: 216.498.1100
W. E. Tel: 800.763.9953
Dayton
A. E. Tel: 614.888.3313
I. E. Tel: 937.253.7501
W. E. Tel: 800.575.9953
Strongsville
B. M. Tel: 440.238.0404
Valley View
I. E. Tel: 216.520.4333

**Oklahoma**
W. E. Tel: 972.235.9953
Tulsa
A. E. Tel: 918.459.6000
I. E. Tel: 918.665.4664

**Oregon**
Beaverton
B. M. Tel: 503.524.1075
I. E. Tel: 503.644.3300
Portland
A. E. Tel: 503.526.6200
W. E. Tel: 800.879.9953

**Pennsylvania**
Mercer
I. E. Tel: 412.662.2707
Philadelphia
A. E. Tel: 800.526.4812
B. M. Tel: 877.351.2355
W. E. Tel: 800.871.9953
Pittsburgh
A. E. Tel: 412.281.4150
W. E. Tel: 440.248.9996

**Rhode Island**
A. E. 800.272.9255
W. E. Tel: 781.271.9953

**South Carolina**
A. E. Tel: 919.872.0712
W. E. Tel: 919.469.1502

**South Dakota**
A. E. Tel: 800.829.0116
W. E. Tel: 612.853.2280

**Tennessee**
W. E. Tel: 256.830.1119
East/West
A. E. Tel: 800.241.8182
Tel: 800.633.2918

**Texas**
Arlington
B. M. Tel: 817.417.5993
Austin
A. E. Tel: 512.219.3700
B. M. Tel: 512.258.0725
I. E. Tel: 512.719.3090
W. E. Tel: 800.365.9953
Dallas
A. E. Tel: 214.553.4300
B. M. Tel: 972.783.4191
W. E. Tel: 800.955.9953
El Paso
A. E. Tel: 800.526.9238
Houston
A. E. Tel: 713.781.6100
B. M. Tel: 713.917.0663
W. E. Tel: 800.888.9953
Richardson
I. E. Tel: 972.783.0800
Rio Grande Valley
A. E. Tel: 210.412.2047
Stafford
I. E. Tel: 281.277.8200

**Utah**
Centerville
B. M. Tel: 801.295.3900
Murray
I. E. Tel: 801.288.9001
Salt Lake City
A. E. Tel: 801.365.3800
W. E. Tel: 800.477.9953

**Vermont**
A. E. Tel: 800.272.9255
W. E. Tel: 716.334.5970

**Virginia**
A. E. Tel: 800.638.5988
W. E. Tel: 301.604.8488
Haymarket
B. M. Tel: 703.754.3399
Springfield
B. M. Tel: 703.644.9045

**Washington**
Kirkland
I. E. Tel: 425.820.8100
Maple Valley
B. M. Tel: 206.223.0080
Seattle
A. E. Tel: 425.882.7000
W. E. Tel: 800.248.9953

**West Virginia**
A. E. Tel: 800.638.5988

**Wisconsin**
Milwaukee
A. E. Tel: 414.513.1500
W. E. Tel: 800.867.9953
Wauwatosa
I. E. Tel: 414.258.5338

**Wyoming**
A. E. Tel: 800.332.9326
W. E. Tel: 801.974.9953

# Sales Offices and Design Resource Centers

**LSI Logic Corporation**
**Corporate Headquarters**
**1551 McCarthy Blvd**
**Milpitas CA 95035**
**Tel: 408.433.8000**
**Fax: 408.433.8989**

**NORTH AMERICA**

**California**
Irvine
18301 Von Karman Ave
Suite 900
Irvine, CA 92612
♦ Tel: 949.809.4600
Fax: 949.809.4444

Pleasanton Design Center
5050 Hopyard Road, 3rd Floor
Suite 300
Pleasanton, CA 94588
Tel: 925.730.8800
Fax: 925.730.8700

San Diego
7585 Ronson Road
Suite 100
San Diego, CA 92111
Tel: 858.467.6981
Fax: 858.496.0548

Silicon Valley
1551 McCarthy Blvd
Sales Office
M/S C-500
Milpitas, CA 95035
♦ Tel: 408.433.8000
Fax: 408.954.3353
Design Center
M/S C-410
Tel: 408.433.8000
Fax: 408.433.7695

Wireless Design Center
11452 El Camino Real
Suite 210
San Diego, CA 92130
Tel: 858.350.5560
Fax: 858.350.0171

**Colorado**
Boulder
4940 Pearl East Circle
Suite 201
Boulder, CO 80301
♦ Tel: 303.447.3800
Fax: 303.541.0641

Colorado Springs
4420 Arrowswest Drive
Colorado Springs, CO 80907
Tel: 719.533.7000
Fax: 719.533.7020

Fort Collins
2001 Danfield Court
Fort Collins, CO 80525
Tel: 970.223.5100
Fax: 970.206.5549

**Florida**
Boca Raton
2255 Glades Road
Suite 324A
Boca Raton, FL 33431
Tel: 561.989.3236
Fax: 561.989.3237

**Georgia**
Alpharetta
2475 North Winds Parkway
Suite 200
Alpharetta, GA 30004
Tel: 770.753.6146
Fax: 770.753.6147

**Illinois**
Oakbrook Terrace
Two Mid American Plaza
Suite 800
Oakbrook Terrace, IL 60181
Tel: 630.954.2234
Fax: 630.954.2235

**Kentucky**
Bowling Green
1262 Chestnut Street
Bowling Green, KY 42101
Tel: 270.793.0010
Fax: 270.793.0040

**Maryland**
Bethesda
6903 Rockledge Drive
Suite 230
Bethesda, MD 20817
Tel: 301.897.5800
Fax: 301.897.8389

**Massachusetts**
Waltham
200 West Street
Waltham, MA 02451
♦ Tel: 781.890.0180
Fax: 781.890.6158

Burlington - Mint Technology
77 South Bedford Street
Burlington, MA 01803
Tel: 781.685.3800
Fax: 781.685.3801

**Minnesota**
Minneapolis
8300 Norman Center Drive
Suite 730
Minneapolis, MN 55437
♦ Tel: 612.921.8300
Fax: 612.921.8399

**New Jersey**
Red Bank
125 Half Mile Road
Suite 200
Red Bank, NJ 07701
Tel: 732.933.2656
Fax: 732.933.2643

Cherry Hill - Mint Technology
215 Longstone Drive
Cherry Hill, NJ 08003
Tel: 856.489.5530
Fax: 856.489.5531

**New York**
Fairport
550 Willowbrook Office Park
Fairport, NY 14450
Tel: 716.218.0020
Fax: 716.218.9010

**North Carolina**
Raleigh
Phase II
4601 Six Forks Road
Suite 528
Raleigh, NC 27609
Tel: 919.785.4520
Fax: 919.783.8909

**Oregon**
Beaverton
15455 NW Greenbrier Parkway
Suite 235
Beaverton, OR 97006
Tel: 503.645.0589
Fax: 503.645.6612

**Texas**
Austin
9020 Capital of TX Highway North
Building 1
Suite 150
Austin, TX 78759
Tel: 512.388.7294
Fax: 512.388.4171

Plano
500 North Central Expressway
Suite 440
Plano, TX 75074
♦ Tel: 972.244.5000
Fax: 972.244.5001

Houston
20405 State Highway 249
Suite 450
Houston, TX 77070
Tel: 281.379.7800
Fax: 281.379.7818

**Canada**
**Ontario**
Ottawa
260 Hearst Way
Suite 400
Kanata, ON K2L 3H1
♦ Tel: 613.592.1263
Fax: 613.592.3253

**INTERNATIONAL**

**France**
Paris
**LSI Logic S.A.**
**Immeuble Europa**
53 bis Avenue de l'Europe
B.P. 139
78148 Velizy-Villacoublay
Cedex, Paris
♦ Tel: 33.1.34.63.13.13
Fax: 33.1.34.63.13.19

**Germany**
Munich
**LSI Logic GmbH**
Orleansstrasse 4
81669 Munich
♦ Tel: 49.89.4.58.33.0
Fax: 49.89.4.58.33.108

Stuttgart
Mittlerer Pfad 4
D-70499 Stuttgart
♦ Tel: 49.711.13.96.90
Fax: 49.711.86.61.428

**Italy**
Milan
**LSI Logic S.P.A.**
Centro Direzionale Colleoni Palazzo
Orione Ingresso 1
20041 Agrate Brianza, Milano
♦ Tel: 39.039.687371
Fax: 39.039.6057867

**Japan**
Tokyo
**LSI Logic K.K.**
Rivage-Shinagawa Bldg. 14F
4-1-8 Kounan
Minato-ku, Tokyo 108-0075
♦ Tel: 81.3.5463.7821
Fax: 81.3.5463.7820

Osaka
Crystal Tower 14F
1-2-27 Shiromi
Chuo-ku, Osaka 540-6014
♦ Tel: 81.6.947.5281
Fax: 81.6.947.5287

## Sales Offices and Design Resource Centers (Continued)

**Korea**
Seoul
**LSI Logic Corporation of Korea Ltd**
10th Fl., Haesung 1 Bldg.
942, Daechi-dong,
Kangnam-ku, Seoul, 135-283
Tel: 82.2.528.3400
Fax: 82.2.528.2250

**The Netherlands**
Eindhoven
**LSI Logic Europe Ltd**
World Trade Center Eindhoven
Building 'Rijder'
Bogert 26
5612 LZ Eindhoven
Tel: 31.40.265.3580
Fax: 31.40.296.2109

**Singapore**
Singapore
**LSI Logic Pte Ltd**
7 Temasek Boulevard
#28-02 Suntec Tower One
Singapore 038987
Tel: 65.334.9061
Fax: 65.334.4749

**Sweden**
Stockholm
**LSI Logic AB**
Finlandsgatan 14
164 74 Kista
♦ Tel: 46.8.444.15.00
Fax: 46.8.750.66.47

**Taiwan**
Taipei
**LSI Logic Asia, Inc.**
**Taiwan Branch**
10/F 156 Min Sheng E. Road
Section 3
Taipei, Taiwan R.O.C.
Tel: 886.2.2718.7828
Fax: 886.2.2718.8869

**United Kingdom**
Bracknell
**LSI Logic Europe Ltd**
Greenwood House
London Road
Bracknell, Berkshire RG12 2UB
♦ Tel: 44.1344.426544
Fax: 44.1344.481039

♦ Sales Offices with
Design Resource Centers

## International Distributors

**Australia**
New South Wales
**Reptechnic Pty Ltd**
3/36 Bydown Street
Neutral Bay, NSW 2089
♦ Tel: 612.9953.9844
Fax: 612.9953.9683

**Belgium**
**Acal nv/sa**
Lozenberg 4
1932 Zaventem
Tel: 32.2.7205983
Fax: 32.2.7251014

**China**
Beijing
**LSI Logic International Services Inc.**
**Beijing Representative Office**
Room 708
Canway Building
66 Nan Li Shi Lu
Xicheng District
Beijing 100045, China
Tel: 86.10.6804.2534 to 38
Fax: 86.10.6804.2521

**France**
Rungis Cedex
**Azzurri Technology France**
22 Rue Saarinen
Sillic 274
94578 Rungis Cedex
Tel: 33.1.41806310
Fax: 33.1.41730340

**Germany**
Haar
**EBV Elektronik**
Hans-Pinsel Str. 4
D-85540 Haar
Tel: 49.89.4600980
Fax: 49.89.46009840

Munich
**Avnet Emg GmbH**
Stahlgruberring 12
81829 Munich
Tel: 49.89.45110102
Fax: 49.89.42.27.75

Wuennenberg-Haaren
**Peacock AG**
Graf-Zepplin-Str 14
D-33181 Wuennenberg-Haaren
Tel: 49.2957.79.1692
Fax: 49.2957.79.9341

**Hong Kong**
Hong Kong
**AVT Industrial Ltd**
Unit 608 Tower 1
Cheung Sha Wan Plaza
833 Cheung Sha Wan Road
Kowloon, Hong Kong
Tel: 852.2428.0008
Fax: 852.2401.2105

**Serial System (HK) Ltd**
2301 Nanyang Plaza
57 Hung To Road, Kwun Tong
Kowloon, Hong Kong
Tel: 852.2995.7538
Fax: 852.2950.0386

**India**
Bangalore
**Spike Technologies India Private Ltd**
951, Vijayalakshmi Complex,
2nd Floor, 24th Main,
J P Nagar II Phase,
Bangalore, India 560078
♦ Tel: 91.80.664.5530
Fax: 91.80.664.9748

**Israel**
Tel Aviv
**Eastronics Ltd**
11 Rozanis Street
P.O. Box 39300
Tel Aviv 61392
Tel: 972.3.6458777
Fax: 972.3.6458666

**Japan**
Tokyo
**Daito Electron**
Sogo Kojimachi No.3 Bldg
1-6 Kojimachi
Chiyoda-ku, Tokyo 102-8730
Tel: 81.3.3264.0326
Fax: 81.3.3261.3984

**Global Electronics Corporation**
Nichibei Time24 Bldg. 35 Tansu-cho
Shinjuku-ku, Tokyo 162-0833
Tel: 81.3.3260.1411
Fax: 81.3.3260.7100
Technical Center
Tel: 81.471.43.8200

**Marubeni Solutions**
1-26-20 Higashi
Shibuya-ku, Tokyo 150-0001
Tel: 81.3.5778.8662
Fax: 81.3.5778.8669

**Shinki Electronics**
Myuru Daikanyama 3F
3-7-3 Ebisu Minami
Shibuya-ku, Tokyo 150-0022
Tel: 81.3.3760.3110
Fax: 81.3.3760.3101

Yokohama-City
**Innotech**
2-15-10 Shin Yokohama
Kohoku-ku
Yokohama-City, 222-8580
Tel: 81.45.474.9037
Fax: 81.45.474.9065

**Macnica Corporation**
Hakusan High-Tech Park
1-22-2 Hadusan, Midori-Ku,
Yokohama-City, 226-8505
Tel: 81.45.939.6140
Fax: 81.45.939.6141

**The Netherlands**
Eindhoven
**Acal Nederland b.v.**
Beatrix de Rijkweg 8
5657 EG Eindhoven
Tel: 31.40.2.502602
Fax: 31.40.2.510255

**Switzerland**
Brugg
**LSI Logic Sulzer AG**
Mattenstrasse 6a
CH 2555 Brugg
Tel: 41.32.3743232
Fax: 41.32.3743233

**Taiwan**
Taipei
**Avnet-Mercuries Corporation, Ltd**
14F, No. 145,
Sec. 2, Chien Kuo N. Road
Taipei, Taiwan, R.O.C.
Tel: 886.2.2516.7303
Fax: 886.2.2505.7391

**Lumax International Corporation, Ltd**
7th Fl., 52, Sec. 3
Nan-Kang Road
Taipei, Taiwan, R.O.C.
Tel: 886.2.2788.3656
Fax: 886.2.2788.3568

**Prospect Technology Corporation, Ltd**
4Fl., No. 34, Chu Luen Street
Taipei, Taiwan, R.O.C.
Tel: 886.2.2721.9533
Fax: 886.2.2773.3756

**Wintech Microeletronics Co., Ltd**
7F., No. 34, Sec. 3, Pateh Road
Taipei, Taiwan, R.O.C.
Tel: 886.2.2579.5858
Fax: 886.2.2570.3123

**United Kingdom**
Maidenhead
**Azzurri Technology Ltd**
16 Grove Park Business Estate
Waltham Road
White Waltham
Maidenhead, Berkshire SL6 3LW
Tel: 44.1628.826826
Fax: 44.1628.829730

Milton Keynes
**Ingram Micro (UK) Ltd**
Garamonde Drive
Wymbush
Milton Keynes
Buckinghamshire MK8 8DF
Tel: 44.1908.260422

Swindon
**EBV Elektronik**
12 Interface Business Park
Bincknoll Lane
Wootton Bassett,
Swindon, Wiltshire SN4 8SY
Tel: 44.1793.849933
Fax: 44.1793.859555

♦ Sales Offices with Design Resource Centers