Master’s Thesis
University of Applied Sciences Augsburg
Department of Computer Science

Embedded Systems Trace Solutions

Analysis and Implementation of Embedded Systems Trace Solutions

Submitted by Dominic Rath, summer semester 2007
Examiner: Prof. Dr. Hubert Högl
Examiner: Prof. Burkhard Stork

Master’s Thesis
University of Applied Sciences Augsburg
Department of Computer Science

I affirm that the master’s thesis is my own work, and that it has never been submitted for examination purposes before. All sources and citations used have been quoted as such, and all utilized tools have been mentioned.

Dominic Rath

Embedded Systems Trace Solutions

Analysis of embedded systems trace solutions and prototype trace implementations for ARM7, ARM9 and XScale based microcontrollers

Dominic Rath

12 11 10 09 08 07   5 4 3 2 1 0

First edition: 27 September 2007

Contents

List of Figures

List of Tables

1 Preface

1.1 Aim

The aim of this master's thesis is to analyze the use of trace debugging techniques in the context of embedded systems development, with a focus on ARM based microcontrollers. Furthermore it is going to show prototype implementations of trace solutions for ARM7, ARM9 and XScale based devices utilizing both a low-bandwidth JTAG interface and a high-bandwidth ETM trace capture unit built using an FPGA.

1.2 Conventions

Throughout this document the following conventions will be used for better readability when referring to binary quantities (bits, bytes):

Table 1.1: prefixes

Prefix	Description	equivalent IEC prefix
K	kilobinary, $2^{10} = 1024$	kibi
M	megabinary, $2^{20} = 1024$	mibi
G	gigabinary, $2^{30} = 1024$	gibi

One gigabyte would be written as 1GB, and is the equivalent of 1024MB (megabyte), 1048576KB (kilobyte) or 1073741824B (byte). Bytes are expected to consist of eight bits (8b).

For all other units the SI symbols will be used, like 10MHz (ten megahertz, a frequency of $10 * 10^6$ Hertz) or 100ns (one hundred nanoseconds, or $100 * 10^{-9}$s= 0.0000001s).

Decimal numbers are written as is, hexadecimals are prefixed with '0x' and binary numbers are prefixed with a single lower case 'b': $57 = 0x39 = b00111001$.

Variable and function names will be typeset in a `typewriter` font, and register names and bitfields will be *emphasized*.

Footnotes[1] are sequentially numbered within a chapter, starting at one at the beginning of each chapter.

[1] This is a footnote.

1.3 Prerequisite Knowledge

This master's thesis builds upon knowledge gained while writing the diploma thesis "Open On-Chip Debugger - Design and Implementation of an On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family" [DR05].

The design of the OpenOCD+trace ETM (Embedded Trace Macrocell) TPA (Trace Protocol Analyzer) was implemented using VHDL, the VHSIC Hardware Description Language, and the chapter describing the OpenOCD+trace implementation requires basic knowledge about the language VHDL and synchronous design.

All software described in this document is written in C and familiarity with the language is assumed.

2 Debugging Techniques

Debugging becomes necessary when software or hardware does not behave the way it was designed to, and when it is not apparent why it is misbehaving. The deviation from the intended system behaviour is called a bug, and techniques for observing the circumstances that caused the system to fail is shall be described in this chapter.

2.1 Static Analysis

One way to avoid bugs at runtime is to eliminate them before code is run for the first time, by making use of static source code analysis.

Compiler Warnings

C compilers output only a limited number of warnings by default, making it easy to slip ambiguities into the code that later turn out to be bugs, because the code never described the intended system behaviour.

The following trivial code fragment for example compiles without warnings when using GCC version 4.1.2:

Listing 2.1: test.c

```
1  int max(int a, int b) {
2          if (a > b)
3                  return a;
4          if (b > a)
5                  return b;
6  }
```

However, after enabling additional warnings using the `-Wall` switch, it becomes apparent what potential defects this small piece of code contains:

```
gcc -c test.c -Wall
test.c: In function 'max':
test.c:6: warning: control reaches end of non-void function
```

The code is intended to return the greater of the two values, but misses the case when `a` equals `b`, in which the behaviour is undefined.

Static Analysis Tools

The Linux kernel is often used as a target for auditing purposes, for example because of its popularity, complexity, and source code availability. The Sparse (`http://www.kernel.org/pub/software/devel/sparse/`) project was specifically created to audit the Linux kernel. It consists of a compiler frontend that compiles ANSI C and some GCC specific enhancements, and a static analyzer part. Sparse can be used to convey additional semantic information about types and functions, allowing static verification of the code.

Coverity (`http://www.coverity.com`) is the company that offers the commercially available tool Prevent SQS (software quality system), a software that uses static source code analysis for automated identification of software defects. Since 2004 Coverity regularly checks the Linux source code for defects, and later expanded this service to over 150 open source projects. One of the first problems found by Coverity was the following bug in `arch/i386/boot/tools/build.c`[1], already fixed since Linux version 2.6.12:

Listing 2.2: arch/i386/boot/tools/build.c

```
74      unsigned int i, c, sz, setup_sectors;
```

Listing 2.3: arch/i386/boot/tools/build.c

```
125     for (i=0 ; (c=read(fd, buf, sizeof(buf)))>0 ; i+=c )
126         if (write(1, buf, c) != c)
127             die("Write call failed");
```

Variable `c` was declared as an unsigned int, but the `read()` system call could return `-1`. The comparison intended to catch this error condition wouldn't identify this, because the negative value would be interpreted as an unsigned value greater zero, causing a huge amount of data to be written on the disk in the following write call. The `read()` would presumably continue to fail, resulting in writes to the disk until it is full and the program exits with the "Write call failed" message.

2.2 Debug Output

One obvious way of gaining insight into the system's behaviour is the inclusion of debug output, often referred to as "printf-debugging". The developer includes `printf` statements (or the equivalent provided by the programming language or development environment) throughout the source code wherever information about the current system state is required. This approach has several drawbacks, especially in the context of embedded systems:

- It requires changes to the source code. Every change to the code could possibly affect the system behaviour, causing the original problem to disappear or new ones to show up.
- It is a short-sighted debug approach. Debug statements are usually included around the area where the defect is suspected and removed whenever the problem is believed to be fixed. For the next, similar problem, debug statements will be included again.

[1] A helper program that builds Linux disk images for use on x86 systems.

- Debug output could use up one of a limited number of communication channels like UARTs. If the final product requires the use of all available resources it wont be possible to output any debug information.
- Output of debug information negatively affects the system's realtime behaviour. Depending on the communication channel used to transfer the debug output every access could block the target for a non-deterministic time. For example when using the ARM7/9 debug communications channel (see 6) every access would have to block until the previous data was read from the transmit register by the debugging host.

2.3 Start/Stop Debugging

Start/stop debugging works by halting the target execution at some point to be able to examine the current system state. Using instruction breakpoints that halt execution once a particular address is reached and watchpoints that catch data accesses, optionally dependant on an address mask and a certain data pattern, the system behaviour is observed by gradually moving towards the original point of failure. This debugging technique requires either hardware support in the form of an in-circuit emulator or on-chip debug support, or extensive support within the software running on the target, in the form of a debug monitor or operating system debug facilities.

A disadvantage of this debug technique is the inability to debug problems that depend on realtime interaction of the microcontroller being debugged with other parts of the system, or problems where the root cause of the defect is too far away from the part that exhibits the malfunction. When a target is halted it is usually unable to react upon external events, which can lead to communication timeouts, missed interrupts, overruns in connected hardware, or other problems unrelated to the defect that is being debugged. It also requires knowledge about the approximate location of the problem in order to place a breakpoint or watchpoint that halts the target.

2.4 Trace

Tracing is an automated way of observing the system behaviour by outputting information about the current system state at regular intervals. It is similar to debugging via debug output, because it only allows the system to be observed, eventually up to the level of detail required to determine where execution deviated from the intended behaviour, but it doesn't allow execution to be influenced, like it can be done when using start/stop debugging.

Trace solutions usually allow the instruction flow to be reconstructed up to a certain depth, and optionally include information about the data accessed by instructions such as loads, stores or coprocessor transfers. The amount of preprocessing done before the data is captured and the analysis required to reconstruct the program flow greatly varies.

Often a trigger is used to specify a point at which trace collection is stopped. Using a programmable counter that starts counting down to zero once the trigger fires, the trace can be configured to capture data up to the trace point, around the trace point, or starting at the trace point until the maximum trace depth is reached.

3 Debugging Implementations

3.1 In-Circuit Emulators

An in-circuit emulator (ICE) replaces the target microcontroller with a special debug variant that includes hardware debugging facilities. The emulator is connected to a host computer which runs the debugger software. This allows both passive and active debugging, giving a non-intrusive view of the program flow, and allowing fine control over program execution, CPU state and memory contents. Read Only Memory (ROM) emulators substitute target non-volatile memory with dual-ported Random Access Memory (RAM) modules, that can be accessed from a debugger and the target at the same time. Where code has to be run from ROM this allows a debugger to replace instructions with hooks necessary for debug entry, like TRAP or Software Interrupt (SWI) instructions. Code testing is improved, as the memory chips don't have to be programmed with external tools.

An ICE might support hardware breakpoints, where address comparators constantly monitor the address bus, and force the system into debug state when an address matches during an instruction fetch. This allows breakpoints to be set on code contained in ROM without using a ROM emulator. If the ICE further provides overlay memory, it is possible to load code into the target, replacing instructions contained in ROM regions. The ICE watches the accessed memory space, and switches to its included RAM when an access to overlaid memory occurs.

In-circuit emulators have lost significance because modern chip families like the NXP LPC2000, Atmel AT91SAM7, or ST Microelectronics STR7 come in a wide variety of different devices, making it difficult to provide drop-in replacements with debug functionality for each of them.

3.2 Debug Monitors

Debug monitors, also called debug stubs, are pieces of software running on the target that communicate with the debug host using a communications channel and some kind of debug protocol. The debug protocol allows the host to halt normal execution of the target, to inspect and modify registers and memory content, and to resume execution again. Breakpoints that halt execution once a certain address has been reached are usually implemented as well, and if the hardware allows, watchpoints that halt upon specific memory accesses may be available.

Examples of debug monitors used in embedded systems are the RealMonitor in conjunction with the EmbeddedICE-RT[1] macrocell found on some versions of the ARM7TDMI-S core, the Angel debug

[1]The EmbeddedICE-RT macrocell is an enhanced EmbeddedICE macrocell with support for monitor mode debugging.

monitor, or the gdbserver.

RealMonitor provides a lightweight implementation of a debug monitor that establishes communication between a target and the debug host using the ARM7/9 debug communications channel. It makes use of the monitor mode debug feature available on EmbeddedICE-RT implementations to implement breakpoint and watchpoint functionality, and allows IRQ and FIQ exceptions to be serviced during a debug session, reducing the impact debugging has on realtime behaviour.

The Angel debug monitor is a target resident debug monitor that communicates with a debug host over a variety of communication channels like the ARM7/9 DCC, a serial link or an TCP/IP connection.

The gdbserver is a standalone application running on supported operating systems that launches the target application on behalf of the debugger and allows the debugger to control target execution remotely. Communication can be established over a serial line or using a TCP/IP network connection.

3.3 On-Chip Debugging

Debugging using on-chip debug facilities that allow target execution to be completely halted is the prevalent debugging technique used with highly integrated targets that embed a microprocessor core, caches, on-chip memories and a wide variety of peripherals within a single package. The interface used to control the target hardware is often Joint Test Access Group (JTAG) or some other serial interface like the background debug mode (BDM) or common on-chip processor (COP) interfaces found on 68k and Power systems.

While the target is stopped, the debugger has full control over all system resources[2]. The debugger may then examine registers and the target's memory content, modify system state to analyze the system's reaction on those changed circumstances, single step through the code, and eventually resume execution.

Depending on the support provided by the debug facilities, breakpoints may be placed to halt execution at certain points of interest, and watchpoints could allow specific memory accesses to trigger a target halt, just like when using debug monitors.

The main advantage start/stop debugging using on-chip debug facilities has over debug monitors is a lower level of intrusiveness while the target is running. On-chip debugging usually works over a dedicated communication channel and places no restrictions on the software running on the target. Debug monitors on the other hand require one of a limited number of communication channels and also need to be integrated with the target application or operating system. Because the debug monitor requires code running on the target it also adds to the memory footprint of the target where memory is often a sparse resource.

Examples for on-chip debug support are the ARM7/9 families with their EmbeddedICE macrocell or the XScale family with its mini I-Cache that can hold a debug handler.

[2]Some implementations provide advanced security features that could potentially limit this access, but during development it is often possible to gain access to all available resources.

3.4 Software Trace

The simplest kind of trace requires no support from the hardware. It uses an area of memory on the target system to store trace information generated by the software itself. An alternative approach transfers the trace information using some kind of communication channel like the ARM7/9 DCC or a serial connection and stores it on the debug host.

Storing trace data on the target has the obvious drawback of requiring a reserved area of memory. Also, depending on the nature of the software defect, the trace data itself could get corrupted, rendering the captured information useless. Transferring trace data to the debug host suffers from similar disadvantages as the debug output approach described before. It requires a communications channel and potentially affects the system's realtime performance if the communication causes the target to stall.

3.5 Hardware Trace Support

Several approaches exist that support tracing with dedicated hardware. Hardware trace support usually has the advantage of being completely unintrusive, but depending on the implementation it might require physical access to device pins, overflows of the trace information could stall the target, or device pins used to output trace information might be shared with other target functionality.

Bus Trace

Tracing all accesses at bus level is possible when there is physical access to the bus, usually when using microprocessors that have no internal peripherals. Another requirement is that all memory accesses have to be visible on that bus – if intermediate on-chip memories and caches fulfill a request the trace equipment observing the bus wont see the access.

Logic analyzers with the ability to trace a large number of signals are required to capture the bus information of a modern 32 or 64 bit processor, because in addition to the data lines the address bus and control signals need to be traced as well. Filtering out specific accesses is often not possible because that would require the logic analyzer to decode all trace information on the fly, for example to identify the location of a memory read. This can be compensated by a large trace depth that allows irrelevant information to be filtered later during analysis.

An advantage of this trace technique is that it does not require support from the target, that is as long as the system doesn't use caches or on-chip memories, bus tracing will work.

Dedicated Trace Port

Targets with higher levels of integration often feature on-chip caches and memories as well as a large number of on-chip peripherals. Even if those system have an externally available bus there is little insight to be gained by tracing only accesses that occur on the external bus. Some targets therefore implement a dedicated trace port that outputs data similar to a bus trace in a compressed form.

Instruction traces are especially suited for compression, because the program counter increases by a predetermined amount on each cycle. Branches can also be compressed by outputting only the changes relative to the last known address, and only indirect branches[3] require an address to be output.

Data tracing requires a much higher trace port bandwidth, because often only the addresses can be compressed, while the data itself shows no recognizable pattern. An ARM9 system for example is able to read or write one 32 bit word per cycle[4] - if the trace port is narrower than that, data needs to be buffered using FIFOs, or the trace port overflows. Code that processes large amounts of data could still easily overflow a trace port and its FIFOs, unless the trace implementation provides means to filter the amount of data that is traced.

Because the trace information is compressed no filtering is possible within the trace capture unit. Only the target itself has the full address information available and can thus implement filtering, for example to exclude some known good library code from the trace. When filtering isn't possible a large trace depth is required to gain enough information about the system behaviour to be able to analyze the origin of a bug.

The Embedded Trace Macrocell (ETM) that can be connected to several ARM cores is an example of a dedicated trace port that outputs compressed trace information over an eight, twelve or twenty bit wide trace port.

On-Chip Trace

At higher core execution speeds getting the trace information out of the chip package can be be a difficult problem. The number of device pins that can be used as a trace port is often limited, because higher pin densities are a driving factor in chip cost and at core frequencies up to and above one gigahertz maintaining signal integrity is complex task.

Die area on the other hand is getting cheaper with the ever decreasing structure sizes that come with modern process technologies, which is why some targets provide on-chip memory that directly stores the compressed trace information, making it available for later analysis via a low-speed interface like JTAG.

Examples for on-chip trace capabilities are the Embedded Trace Buffer (ETB) from ARM that can be used together with an ARM ETM, or the trace buffer integrated in the XScale core.

[3]An indirect branch is branch whose target address can't be deduced from the program image because it depends on a register or memory value.

[4]Within limits, for example only with zero waitstate memory and when there is no immediate load-use interlock.

4 ARM

This chapter is going to provide an overview on the ARM technologies relevant to the trace debug techniques described in this master's thesis. ARM based microcontrollers can be classified by the architecture implemented, by the core family, and by the actual core used in a design. Table 4.1 has a list of letters used in ARM core and architecture names and their meaning.

Table 4.1: ARM core features

Letter	Description
T	Thumb mode support (compressed 16 bit instruction set)
D	Debug support
M	Enhanced Multiplier (multiply with 64 bit)
I	Embedded-ICE
E	ARM 'Enhanced' DSP instruction set
J	Jazelle Java acceleration technology

4.1 Architecture

The targets on which this document focuses implement the ARMv4T and ARMv5TE(J) architecture. The architecture specifies the programmer's model and available instruction set. With the exception of the Jazelle support only available on targets implementing the ARMv5TEJ architecture only small differences exist as far as the basic execution environment is concerned.
The ARMv5TE architecture adds support for an enhanced DSP instruction set, simplified ARM-Thumb interworking [1] and a few extra instructions. The Thumb instruction set was assigned a new version "THUMBv2" [DDI0100E, §7.1.1] which is not to be confused with the "Thumb-2" instruction set available in ARMv6 and ARMv7.

Programmer's Model

The execution environment was already described in [DR05, p.13], but some basic information will be provided here for reference. Detailed information can also be found in [DDI0100E, §2].
The ARM architecture (ARMv4 and ARMv5) defines 31 general purpose registers of which only 16

[1] Interworking means the ability to call ARM functions from Thumb code and vice versa.

User	FIQ	IRQ	Supervisor	Abort	Undefined	System
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12	R12
R13	R13_fiq	R13_irq	R13_svc	R13_abt	R13_und	R13
R14	R14_fiq	R14_irq	R14_svc	R14_abt	R14_und	R14
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_irq	SPSR_svc	SPSR_abt	SPSR_und	

Figure 4.1: ARM banked registers

registers are accessible at any time, the remaining registers are banked registers available only from within a particular processor mode. ARM core modes are *User*, *FIQ*, *IRQ*, *Supervisor*, *Abort*, *Undefined* and *System*. There are six program status registers, the current program status register (*CPSR*) and five saved program status registers (SPSR), one for each mode. *System* mode is special because it shares the whole register set with user mode, but as a privileged mode unlimited access to all system resources is possible. The ARM cores may be executing in either one of ARM, Thumb or Jazelle state, but Jazelle will be ignored for the purposes of this work as there is no information available publicly.

Figure 4.1 shows the ARM register file, the layout of the *PSR registers can be found in figure 4.2.

4.2 Implementations

Cores implementing one of the ARM architectures are available from a number of different vendors, most importantly ARM itself, but there are also companies holding an ARM architecture license[2] that allows them to roll their own CPU implementing the ARM architecture.

[2]See `http://www.arm.com/products/licensing/index.html` for the available ARM licensing schemes.

N	Z	C	V	Q			J	Reserved	I	F	T	M0	M3	M2	M1	M0
Negative flag	Zero flag	Carry flag	Overflow flag	DSP Overflow/Satuartion flag	Reserved	Reserved	Jazelle Bit J=1: Jazelle mode		IRQ interrupt disable flag I=1: IRQ disabled	FIQ interrupt disable flag F=1: FIQ disabled	Thumb Bit T=1: Thumb mode	M[4:0] 0b10000 USER 0b10001 FIQ 0b10010 IRQ 0b10011 SVC 0b10111 ABT 0b11011 UND 0b11111 SYS				

Figure 4.2: program status register format

ARM7

The ARM7 family consists of the ARM7TDMI, the ARM7TDMI-S, the ARM7EJ-S and the ARM720T. The ARM720T is a cached processor based on the ARM7TDMI core augmented with a MMU and a unified 8KB instruction and data cache. The ARM7TDMI employs a 3-stage pipeline with an instruction fetch stage, a decode stage and an execute stage. It is a Von-Neumann architecture with a single address space for both instructions and data, and only a single bus between the memory system and the core. The ARM7TDMI exists both as a hard macrocell tailored for a particular design process and a synthesisable variant ARM7TDMI-S. When working with synthesizable ARM cores it is important that the JTAG TCK needs to be synchronized with the core clock. Usually this means that the JTAG frequency must not exceed one sixth of the core frequency.
The ARM7 family members implement the ARMv4T architecture, with the exception of the ARM7EJ-S which is an ARMv5TEJ core. The ARM7EJ-S has little practical relevance because no cores based on this design are readily available at the time of this writing.

ARM9(E)

All ARM9(E) family members are based on a ARM9TDMI, ARM9E-S or ARM9EJ-S core. Only those cores that are based on the ARM9TDMI core implement the ARMv4T architecture, the newer designs based on the ARM9E(J)-S implement architecture version 5TEJ.
The ARM920T and ARM922T are ARM9TDMI cores with a MMU and separate instruction and data caches of 2x16KB (ARM920T) or 2x8KB (ARM922T) size.

- The ARM926EJ-S is a ARM9EJ-S based processor with a MMU and separate caches of variable sizes between 0MB and 1MB.
- The ARM946E-S is based on the ARM9E-S core with variable cache sizes and a MPU [3].
- The ARM966E-S is also based on the ARM9E-S but lacks support for a MMU.
- The ARM968E-S features a DMA slave interface.

[3]The memory protection unit (MPU). offers protection for memory regions without the need for a paged memory model that comes with a memory management unit (MMU).

The ARM9E(J)-S based processors feature tightly-coupled memory interfaces (TCM) for instructions and data that allow low latency access at zero or more waitstates. Other memories are typically connected via the advanced high-performance bus (AHB)and are more loosely coupled to the ARM9E(J)-S core. Tightly coupled memory is typically used for critical code sequences and critical data that requires deterministic accesses for which cache memory wouldn't be suited because of temporal or spatial locality of the accesses.

The ARM9(E) family uses a 5-stage pipeline with fetch, decode, execute, and writeback stages. The cores implement a modified Harvard architecture with separate buses for instructions and but still use a unified address space.

XScale

The XScale family implements the ARMv5TE architecture, and consists mainly of I/O processors (Intel IOP), network processors (Intel IXP) and application processors (Marvell PXA). In 2006 Intel's application processor business was acquired by Marvell, but the I/O processors and networking processors remained at Intel, making XScale based targets available from two different companies.

This document will only look at the PXA25x, PXA27x and IXP4xx processors for which sufficient information is available publicly. Other designs, especially the PXA3xx processors could differ significantly from the implementations described here.

The XScale core features a 7-stage pipeline that splits into a main execution pipeline, a memory pipeline and a MAC (multiply accumulate) pipeline. Two instruction fetch stages, IF1 and IF2, provide the following instruction decode stage (ID) with the next instruction that should be executed. The XScale's branch prediction logic, implemented in the branch target buffer (BTB), tries to predict the target of a branch instruction before it reaches the first execution stage (X1) to allow the right instructions to be fetched into pipeline. The ID stage identifies the instruction opcode and operands, detects undefined instruction exceptions, and expands instructions that span more than one cycle, for example LDM/STM[4], into a sequence of simpler instructions. The register fetch stage (RF) uses the decoded register information from the ID stage to supply the execution stages, the MAC stages, the data cache and the coprocessor interface with the required source operands. MAC instructions are then transferred to the MAC pipeline, all other instructions move on to the X1 stage. In X1 the branch target is calculated, and if it was mispredicted the pipeline needs to be flushed, because the wrong instructions were prefetched. X1 is also responsible for ALU (arithmetic-logical-unit) operations and for determining whether a conditional instruction passes its condition test based on the CPSR flags. During X2 the results that have to be written to the register file in the write-back stage (XWB) are determined. Once an instruction reached the XWB stage, it is treated as being completed.

[4]LDM: Load Multiple, STM: Store Multiple.

5 OpenOCD

OpenOCD, the Open On-Chip Debugger, started as a diploma thesis written at the University of Applied Sciences Augsburg (FH Augsburg) [DR05]. The diploma thesis was completed in July 2005 and the project got released under the terms of the GNU General Public License (GPL) using BerliOS to host the project's code repository: `http://developer.berlios.de/projects/openocd/`. The current source code is available from the subversion repository:

```
svn checkout svn://svn.berlios.de/openocd/trunk
```

In cases where access using the SVN protocol isn't possible (e.g. because of company firewalls), the code can alternatively be acquired using the HTTP protocol:

```
svn checkout http://svn.berlios.de/svnroot/repos/openocd/trunk
```

Authenticated access for developers with write privileges is possible using either the SVN+SSH protocol or via HTTPS:

```
svn checkout svn+ssh://developername@svn.berlios.de/svnroot/repos/openocd/trunk
svn checkout https://developername@svn.berlios.de/svnroot/repos/openocd/trunk
```

The OpenOCD project website at `http://openocd.berlios.de` attracts more than 10,000 unique visitors each month generating almost 55,000 page hits (see figure 5.1). As a convenience for Windows users a binary package with an installer is available from the yet another GNU toolchain (YAGARTO) project `http://www.yagarto.de`. The installer based on SVN revision 141 has been downloaded over 1,100 times during May 2007. Packages are available for the Debian and Ubuntu distributions. There is no data on the number of source downloads from the Subversion repository.

The OpenOCD project was used as a starting point for implementing the trace functionality that resulted from this master's thesis because of the author's familiarity with the OpenOCD codebase and because of the wide acceptance the OpenOCD gained during the past two years.

5.1 Current State

At the time of the first public release only ARM7TDMI, ARM720T and ARM920T based targets have been supported. The only supported JTAG interfaces were the Wiggler and the USBJTAG-1 [DR05, p.4-5].

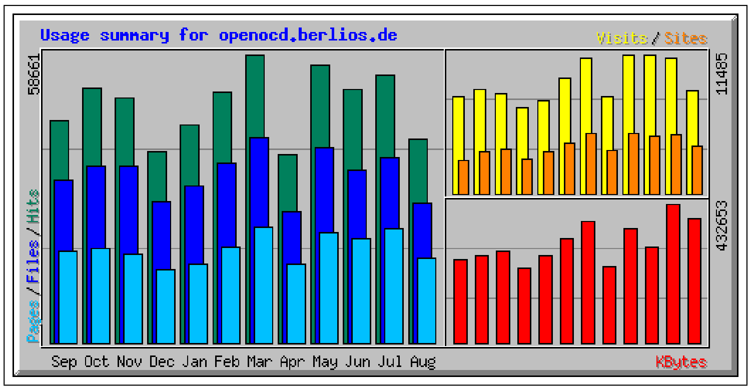


Figure 5.1: OpenOCD Website Usage Statistics

The following JTAG interfaces are supported:

- Parallel port wigglers.
  Simple PC parallel port interfaces that buffer the parallel port 5V TTL signals for use with lower voltage JTAG interfaces (typically 3.3V). Parallel port wigglers are available from a large number of vendors, and support includes cables originally designed to work with programmable logic devices such as CPLDs and FPGAs.
- Amontec JTAG Accelerator.
  A JTAG interface configuration for use with Amontec's Chameleon parallel port device. The chameleon consists of a IEEE1284 compatible bus interface and a Xilinx Coolrunner CPLD. See `http://www.amontec.com/jtag_accelerator.shtml` for additional information.
- Gateworks GW1602.
  A parallel port interface with a proprietary design using a CPLD. See `http://gateworks.com/avila_gw16012.htm` for additional information.
- FTDI FT2232 based USB JTAG interfaces. FTDI's FT2232 features a MPSSE (multiple protocol synchronous serial interface) that can be used to generate JTAGcompliant signals. FT2232 based devices are available from a number of vendors and can also be built using schematics available freely on the internet.
- ASIX PRESTO.
  A USB JTAG interface that can also be used to program various other microcontrollers. See `http://www.asix-tools.com/prg_presto.htm` for additional information.
- usbprog.
  The usbprog is a completely free design using a USBN9604 USB interface chip and an ATMega32 to implement various programmer protocols. See `http://www.embedded-projects.net/index.php?page_id=165` for additional information.

As of SVN revision 194 the OpenOCD supports the following targets:

- ARM7TDMI(-S)
- ARM9TDMI
- ARM920T
- ARM922T

- ARM926EJ-S
- ARM966E
- ARM Cortex-M3
- Intel/Marvell PXA25x
- Intel/Marvell PXA27x
- Intel/Marvell IXP42x

Target debugging is supported with the MMU and caches enabled, and cache coherency is ensured for targets where this is an issue (e.g. ARM9 based targets with separate instruction and data caches). When working with ARM920T targets the current cache and TLB content can be examined to identify potential performance bottlenecks.

Flash programming is supported for the following flashes:

- NXP LPC2000 internal flash
- Atmel AT91SAM7 internal flash
- STMicroelectronics STR7x and STR9x internal flash
- External CFI compatible flashes
- Several non-CFI compatible flashes using the AMD/Spansion algorithm
- NAND flashes connected to a NXP LPC3180 target

5.2 Further Development

So far only ARM based targets have been added to the OpenOCD, but the overall architecture should support other targets such as MIPS or PowerPC as well. The OpenOCD project constantly adds support for new flash memories, and support for other memory types like serial dataflash is being considered.

On the fastest interface currently available (FT2232) the USB 1.1 connection limits the OpenOCD's performance (download to memory, number of single-step cycles per second). A new design consisting of a single board computer (SBC) and a FPGA is currently being evaluated, where the OpenOCD would be running on the linux capable SBC. The OpenOCD's design with a telnet and GDB remote protocol interface always intended such a use case, but resource constraints on the SBC might require some changes.

The community that evolved around the OpenOCD project uses the services (SVN repository, patch tracker, mailing list) provided by BerliOS (`http://www.berlios.de`) and a forum hosted at `http://forum.sparkfun.com/viewforum.php?f=18` to organize the development efforts.

6 DCC Software Trace

In cases where no communication channel is left available on the target, for example because all UARTs are already used for application specific communication, the ARM7/9's debug unit EmbeddedICE offers an additional means of communication with a host, the debug communication channel (DCC). From the ARM core the DCC is accessible via coprocessor 14 using `MRC` (move to ARM register from coprocessor) and `MCR` (move to coprocessor from ARM register) instructions to read and write the coprocessor registers. The debug host uses JTAG to read and write the DCC control and data registers. Because the DCC is an unbuffered communication channel the target has to wait for the debugger to read the last transmitted word before it is able to transmit the next data item. The debugger needs to continuously poll the DCC to allow the target to continue its operation. This restriction imposes potential problems for time critical applications, where the additional delays could cause new problems to appear.

This chapter is going to outline how the ARM7/9 DCC can be used for debug output and software trace functionality.

6.1 DCC Registers

The DCC consists of three registers:

- Control Register: Read-Only.
- Data Read Register: Data sent from debugger to target.
- Data Write Register: Data sent from target to debugger.

The control register contains two bits for handshaking and a version field that indicates the implemented EmbeddedICE version.

R This bit is set to indicate that the DCC data read register contains a word to be read by the target.
W This bit is set to indicate that the DCC data write register contains a word to be read by the debugger.

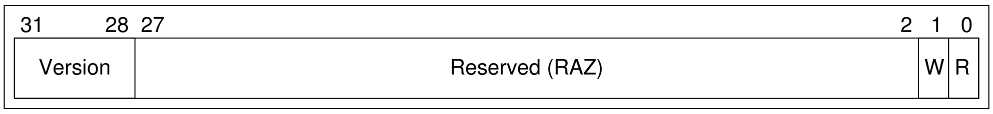


Figure 6.1: DCC Control Register

The data register is split into a read and a write register to reduce the amount of handshaking necessary during bidirectional transfers.

6.2 Accessing The DCC

Core Access To DCC

```
MRC CP14, 0, Rd, C0, C0
```

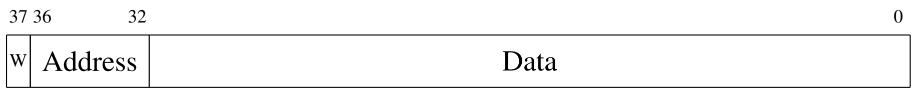
CP14 register c0 is the DCC control register. Accessing it via an `MRC` instruction returns the current register value in register `Rd`.

```
MCR CP14, 0, Rn, C1, C0 MRC CP14, 0, Rd, C1, C0
```

CP14 register c1 is the DCC data register as seen from the target. It can be written with a value from register `Rd` using an `MCR` instruction and is read into a register using an `MRC` instruction. Before the DCC data register can be written the target needs to poll the DCC control register until the `W` bit is clear. Reading the DCC data register requires the target to wait for the `R` bit to be set.

JTAG Access To DCC

The DCC is accessed via the ARM cores JTAGtest access port (TAP)[DR05, p.18]. After writing the EmbeddedICE scan chain's number (0x2) into the `SCANN` register and selecting the `INTEST` JTAG instruction the EmbeddedICE scan chain is connected between the TDI and TDO pins of the target:



After serially shifting in the desired bit pattern and passing through the JTAG Update-DR state the `W` bit specifies if the register selected by `Address` is to be read ($W = 0$) or written ($W = 1$). Because of this, two accesses are required to read a single EmbeddedICE register, but consecutive reads of the same register can be merged by keeping the `W` and the `Address` bits set.
The debugger has to wait for the DCC control reg's `W` bit to be set before reading the DCC data register via JTAG. This read resets the `W` bit, indicating that the current word has been consumed by the debugger and that the target may write the next word to the DCC data write register.
If the debugger writes a new word to the DCC data register the `R` bit is set to inform the target about the new data that can be read. The debugger has to wait until the `R` bit cleared again, unless it can be guaranteed that the target is always going to be fast enough to read any data written by the debugger.

6.3 OpenOCD Target Request Implementation

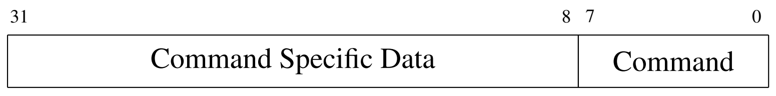
In addition to using the DCC as a debug and trace message channel, additional functionality can be implemented in a DCC aware debugger. One common use is called semihosting [ARMSEMI], a method to enhance target functionality using resources from a debug host. Using semihosting, a target could for example access a host's filesystem using standard C library functions:

```
 open(const char *pathname, int flags);
  read(int fd, void *buf, size_t count);
write(int fd, const void *buf, size_t count);
```

Semihosting itself doesn't necessarily require the use of the DCC, because it can be implemented using traditional start/stop debugging as well. The program code running on the target would use SWI calls that are caught by the debugger to request the debugger's attention. The debugger halts the target, reads necessary input from the target registers and memory space, executes the system call on behalf of the target, and writes any replies back to the target before it is resumed.

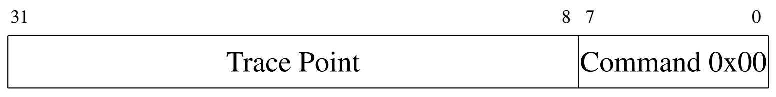
Target Request Protocol

Because semihosting and the output of debug and trace information both make use of the DCC, a common protocol was implemented to allow easy integration of semihosting functionality in the future:

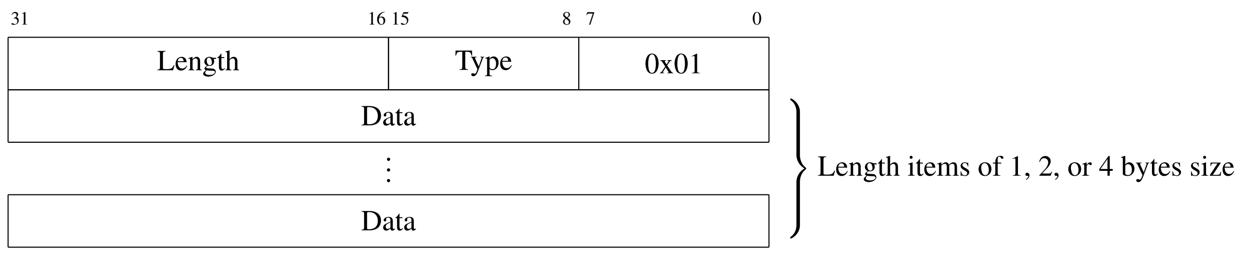


Currently two commands are implemented:

0x00 Trace message. Indicates to the host debugger that a certain part of the program executed using a 24 bit trace point number.



0x01 Debug message. Outputs ASCII strings or 8, 16 or 32 bit data streams on the debug host.



The `Type` field specifies whether ASCII data (0x0), single bytes (0x1), halfwords (0x2), or words (0x4) follow.

ARM7/9 Target Request Code

Because of the invasive nature of debugging using the DCC the target code needs to be modified to make use of the DCC as an additional communication channel. The `DCC_READY`6.1 and `DCC_OUTPUT`6.2 macros use GNU compiler collection (GCC) inline assembly syntax to access the EmbeddedICE coprocessor (CP14) because standard C language features have no support for such target specific facilities:

Listing 6.1: dcc_debug.c:DCC_READY

```
#define DCC_READY do {                                    \
    unsigned int dcc_status;                              \
    do {                                          \
        asm volatile("mrc p14, 0, %0, c0, c0" : "=r" (dcc_status) : );  \
    } while (dcc_status & 0x2);                           \
} while (0);
```

The DCC_READY macro waits in a busy loop until the W bit is cleared, indicating to the target that DCC data register is ready to accept a new word of data.

Listing 6.2: dcc_debug.c:DCC_OUTPUT

```
#define DCC_OUTPUT(x) do {                        \
    asm volatile("mcr p14, 0, %0, c1, c0" : : "r" ((x)));    \
} while (0);
```

The DCC_OUTPUT macro unconditionally outputs data using the DCC without waiting for the DCC data register to become empty. It is the responsibility of the calling code to ensure proper handshaking using the W bit.

Listing 6.3: dcc_debug.c:DCC_TRACEPOINT

```
void TRACEPOINT(unsigned int num)
```

If the program's execution flow is to be observed the TRACEPOINT(num) function can be used to output a trace point. In the current state the developer needs to manually examine the resulting binary if linking of trace point numbers to target addresses is desired. The objdump utility from the GNU toolchain can be used to disassemble an ELF image and the resulting output can be filtered for calls to the TRACEPOINT macro using the grep utility:

```
arm-none-eabi-objdump -d main.out |grep \<TRACEPOINT\> -B1
800004fc:       e3a00000        mov     r0, #0  ; 0x0
80000500:       eb00012c        bl      800009b8 <TRACEPOINT>
--
80000814:       e3a00001        mov     r0, #1  ; 0x1
80000818:       eb000066        bl      800009b8 <TRACEPOINT>
```

The above output shows trace point #0 at address 0x80000500 and trace point #1 at address as 0x80000818. A simple script could generate a list of trace points and addresses for use with the OpenOCD's trace feature.

An alternative approach could integrate the tracepoint mechanism with the code profiling support available in GCC, gprof and gcov, but that is beyond the scope of this work.

Listing 6.4: dcc_debug.c:DEBUG

```
void DEBUGASCII(const char *msg)
void DEBUGHEX32(unsigned int *data, int len)
void DEBUGHEX16(unsigned short *data, int len)
void DEBUGHEX8(unsigned short *data, int len)
```

When a developer wants to output debug information from the running program the `DEBUG*()` functions can be called to output the data on the debug host. Different functions are provided to print ASCII strings (`DEBUGASCII()`) and arbitrary binary data encoded as hex strings in quantities of one, two or four byte (`DEBUGHEX[8|16|32]()`.

6.4 OpenOCD DCC Software Trace Implementation

The support for software trace is implemented in `./src/target/trace.c`, which makes use of the generic target request code implemented in `./src/target/target_request.c`. The per target structure **`struct`** `target_s` was enhanced to include a pointer to the generic trace information that is stored in a structure of type **`struct`** `trace_s`.

Listing 6.5: ./src/target/trace.c

```
typedef struct trace_point_s
{
    u32 address;
    u64 hit_counter;
} trace_point_t;

typedef struct trace_s
{
    int num_trace_points;
    int trace_points_size;
    trace_point_t *trace_points;
    int trace_history_size;
    u32 *trace_history;
    int trace_history_pos;
    int trace_history_overflowed;
} trace_t;

typedef enum trace_status
{
    TRACE_IDLE = 0x0,
    TRACE_RUNNING = 0x1,
    TRACE_TRIGGERED = 0x2,
    TRACE_COMPLETED = 0x4,
    TRACE_OVERFLOWED = 0x8,
} trace_status_t;

extern int trace_point(struct target_s *target, int number);
```

The `trace_s` structure holds two arrays, one is the list of defined trace points (`trace_point_t *trace_points`) and the other contains the history of trace points encountered during the trace run (`u32 *trace_history`). Because trace points can be added dynamically from a config file or a command context[1] the number of currently defined trace points and the current size of the array is stored in two members of the `trace_s`

[1]An OpenOCD command context belongs to an open connection (currently GDB or telnet) and provides access to the OpenOCD's command interpreter.

structure. A trace point consists of the address at which the trace point is emitted and a hit counter that is used to keep track about how often a trace point was reached during the current trace run.

The current size of the trace history is kept together with the current position in the history buffer and a flag indicating a previous overflow. The current position is required when writing to the trace history buffer and is used together with the overflow flag to identify the oldest item available in the buffer.

The `trace_point()` function is provided as a callback for the *target request* subsystem. It gets called whenever the target reported a trace point and updates the trace information accordingly.

6.5 OpenOCD Target Request Usage

Using the ARM7/9 target request code from 6.4 the target can request services from the OpenOCD running on a debug host. The only feature currently available from the `target_request` family of commands is the display of debug messages:

```
> help target_request
  target_request    target_request commands
     debugmsgs -    enable/disable reception of debug messgages from target
```

Using `target_request debugmsgs ['enable'|'disable']` the user may enable or disable the output of debug messages to the current command context, which could be either a telnet connection or a GDB debug session (GDB, Insight, Eclipse).

6.6 OpenOCD Software Trace Usage

As of revision 195 the OpenOCD supports collecting trace information via the ARM7/9's DCC, but this could potentially be enhanced to support other target families as well, provided there is a suitable means of communication. One candidate architecture is XScale with its TX and RX debug registers that provide communication similar to the DCC.

```
> help trace
      trace    trace commands
  history -    display trace history, ['clear'] history or set [size]
    point -    display trace points, ['clear'] list of trace points, or add new
                                                          tracepoint at [address]
```

If the code executing on the target makes use of the `TRACEPOINT` (see listing 6.3) the OpenOCD collects data about how often each trace point got called and a history of trace points that executed in the past.

The `trace history` command allows the user to specify the number of previous trace points that should be kept in a cyclic buffer and is also used to display the trace history.

6.7 ARM7/9 Target Request Constraints

USB JTAG dongles based on the FTDI FT2232 chip are the most popular and versatile debug interfaces for use with the OpenOCD, but the USB communication has the drawback of introducing a delay of at least 1ms (USB 1.1 frame length). The handshaking necessary when reading or writing the DCC data register via JTAG would limit the DCC throughput to $500 * 4 byte/s$ if every control register read and subsequent data register access would execute within exactly one millisecond.

To improve the DCC transfer performance the OpenOCD DCC receive code pretends that the target is always going to be fast enough to write the next word into the DCC data register, making handshaking only a requirement for the first word of a transfer.

When doing continuous transfers the JTAG TAP moves from Update-DR state (where the EmbeddedICE register is actually read) to the Run-Test/Idle state (1 cycle), from there to Shift-DR (3 cycles) where it reads the next word (37 cycles) before it moves to Update-DR again (4 cycles), resulting in a minimum of 45 cycles per access (some JTAG interfaces might introduce additional cycles).

Listing 6.6: DEBUGHEX32

```
void DEBUGHEX32(unsigned int *data, int len)
{
        DCC_READY;
80000aac:       ee103e10        mrc     14, 0, r3, cr0, cr0, {0}
80000ab0:       e3130002        tst     r3, #2  ; 0x2
80000ab4:       1afffffc        bne     80000aac <DEBUGHEX32>

        DCC_OUTPUT(0x01 | 0x0400 | ((len & 0xffff) << 16));
80000ab8:       e1a03801        mov     r3, r1, lsl #16
80000abc:       e3833b01        orr     r3, r3, #1024   ; 0x400
80000ac0:       e3833001        orr     r3, r3, #1      ; 0x1
80000ac4:       ee013e10        mcr     14, 0, r3, cr1, cr0, {0}

        while (len > 0)
80000ac8:       e3510000        cmp     r1, #0  ; 0x0
80000acc:       d12fff1e        bxle    lr
        {
                DCC_READY;
80000ad0:       ee103e10        mrc     14, 0, r3, cr0, cr0, {0}
80000ad4:       e3130002        tst     r3, #2  ; 0x2
80000ad8:       1afffffc        bne     80000ad0 <DEBUGHEX32+0x24>
                DCC_OUTPUT(*data);
80000adc:       e5903000        ldr     r3, [r0]
80000ae0:       ee013e10        mcr     14, 0, r3, cr1, cr0, {0}
80000ae4:       e2511001        subs    r1, r1, #1      ; 0x1
                data++;
80000ae8:       e2800004        add     r0, r0, #4      ; 0x4
80000aec:       1afffff7        bne     80000ad0 <DEBUGHEX32+0x24>
80000af0:       e12fff1e        bx      lr

80000af4 <DEBUGHEX16>:
                len--;
        }
```

Listing 6.6 shows a disassembly of the `DEBUGHEX32` function. After the first handshaked transfer (the `DCC_OUTPUT` at 0x80000ab8) the DCC control register is read to see if the `W` bit is still set. In the worst case the `W` bit would be set when reading it first, and cleared on a subsequent read, so the read and test needs to be done twice, followed by a load of the next word to be transferred and sending that word to the coprocessor. A maximum of eleven instructions would thus execute between the transfer of two words. This allows for four cycles per instruction if the JTAG port was clocked at the same speed as the ARM core, and $n*4$ if the core is clocked at $1/n$ of the core frequency. With the requirement of running a synthesizable core like the ARM926EJ-S at a JTAG frequency of less than $1/6th$ of the core frequency plenty of cycles are available on the target without risking to loose synchronisation between the target and the debug host.

7 XScale On-Chip Trace

The XScale family of embedded processors features a 256-entry deep on-chip trace buffer to allow the program execution flow to be reconstructed. After enabling the trace buffer, trace messages are placed into the trace buffer whenever an indirect branch occurred or when some other event occurred that makes additional information for trace reconstruction necessary.
The XScale trace buffer can operate in fill-once and in wrap-around mode. Fill-once allows the whole program execution to be traced, but affects the target's real time execution, while wrap-around mode enables the developer to examine the instruction flow that lead to a breakpoint or watchpoint.

If the application is able to tolerate interruptions during target execution, the use of target RAM as an extended trace buffer allows the debugger to drastically increase the maximum trace depth.

Once the desired trace data has been acquired the collected trace is transferred to the debug host using the JTAG interface. The debug host then reconstructs the program flow with the help of the target image. Information about the target memory image is necessary because only target addresses of indirect branches are output, while direct branches only cause a branch message to be written to the trace buffer.

7.1 XScale Debugging

Debugging XScale based targets is substantially different from debugging ARM7/9 based targets. There are a number of different XScale implementations available, but for the purpose of this document XScale debugging will only be concerned with Intel/Marvell PXA25x, PXA27x and IXP4xx targets.

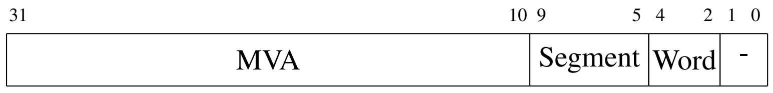
The XScale is an implementation of the ARMv5TE architecture. The targets come with separate data and instruction caches and an additional mini instruction cache[1]. The mini ICache is accessible only via the JTAGTAPand is used to load a debug handler code. When a debug exception occurs the target branches to the reset vector [2] and switches into a special DEBUG mode (also called special debug state (SDS)). The previous `CPSR` is saved in `SPSR_debug` and the return address is stored in $R14_debug$. The debugger must have loaded code at the reset vector which then communicates with the debug host using a set of transfer registers similar to the ARM7/9 DCC.

[1]Some XScale family members include a mini data cache, too.

[2]Exception vectors can reside at 0x0 and 0xffff0000 in the ARMv4 and ARMv5 architecture. The currently active location can be implementation defined or it is determined by the V bit in the system control processor's control register [DDI0100E, §2.6.9].

Mini ICache

XScale models with 32KB come with a 2KB mini ICache that is two-way set associative with a line length of eight words (32 bytes) and 16 sets.



A 32 bit address selects a word within a line with bits 2-4 and the segment with bits 5-9. Addresses that are 1024 byte apart go into the same segment, but because of the 2-way set associativity two lines can occupy the same segment. This is especially important when overloading the reset vectors at 0x0-0x1f and 0xffff0000-0xffff001f because both locations map to segment 0. No other address with bits 5-9 set to 0 can be loaded into the mini ICache without evicting the overloaded reset vectors from the cache.

The mini ICache is typically loaded while the XScale core is held in reset. That way the debugger gains control over that target right out of reset. Once the debug handler is loaded the target reset is deasserted and the core takes the overloaded reset vector and branches to the debug handler.

It is important to note that only code can be fetched from the mini ICache. If load or store operations access an address that is loaded into the mini ICache the request will still be fulfilled by either the data cache or main memory. This is a serious restriction because ARM code is typically interspersed with 32 bit constants in so called literal pools that are accessed using PC relative loads [3]. Code running from the mini ICache must always build immediate operands using data processing instructions.

OpenOCD XScale Debug Handler

The debug handler starts executing at the reset vector which must contain a branch instruction. In ARM mode a PC relative branch can specify a range of approximately +-32MB. The debug handler must therefor reside at an address that's within 32MB of both the high vectors (0xffff0000) and the low vectors (0x00000000). Furthermore it has to be aligned to a 2KB boundary (least significant eleven bits all zero) because of the mini ICache organisation. The OpenOCD restricts the base address of the debug handler to between 0x800-0x1fef800 and 0xfe000800-0xfffff800).

The source for the debug handler can be found in `./src/target/xscale/debug_handler.S`. To simplify building of the OpenOCD without the need for a cross compilation toolchain the SVN repository also holds a precompiled binary of the debug handler in `./src/target/xscale/debug_handler.bin`.

The debug handler initially sets the /GE! bit in the debug control and status register (DCSR), sends the current core state and waits for commands from the debug host. On reentry to the debug handler the /GE! bit is examined to see if the debug handler has been running before.

The protocol between the debug host and the target supports several commands:

[3]The ARM instruction set uses 32 bit instructions that leave no room for full 32 bit immediate operands. If a 32 bit constant is to be loaded this can be achieved using multiple data processing instructions with 12 bit immediate operands (8-bit + 4-bit rotation [DDI0100E, §5.1.3]) or by loading the constant from memory.

0x00	Read banked register of <mode>.
0x01	Set banked register of <mode>.
0x1n	Read memory from <address> using <count> n-byte accesses.
0x2n	Write memory from <address> using <count> n-byte accesses.
0x30	Resume execution (after restoring system state with 0x01 commands).
0x31	Resume execution with trace enabled (after restoring system state with 0x01 commands).
0x40	Read coprocessor register <n> (n is an index into a list of supported registers).
0x40	Write coprocessor register <n> (n is an index into a list of supported registers).
0x50	Clean main cache (write dirty entries back to main memory) using an otherwise unused <cache-clean-area>.
0x51	Invalidate complete DCache.
0x52	Invalidate complete ICache (not mini ICache).
0x53	Wait for outstanding CP15 operations to complete.
0x60	Clear sticky-abort bit in the DCSR.
0x61	Read trace buffer.
0x62	Clean trace buffer.

Inside the debug handler code care has been taken to ensure that the system state isn't corrupted before transferring it to the debug host on debug entry or before the target is resumed. A macro `m_send_to_debugger` is used when a register needs to be transferred without corrupting any other register, otherwise the macro `m_small_save_reg` calls a function that unconditionally sends the content of r0, saving one instruction (4 byte) per register write or read.

7.2 Trace Message Format

The XScale trace buffer compresses the amount of information that needs to be stored by outputting only control flow changes, i.e. after a taken branch or an exception. Other instructions increment a counter that keeps track of the number of executed instructions since the last control flow change. The trace buffer entries are 8-bit wide:

Table 7.1: XScale trace messages

Name	Format	Description
Exception	b0VVVCCCC	An exception caused a branch to address bVVV00.
Direct branch	b1000CCCC	A direct branch (B, BL) occurred.
Checkpointed direct branch	b1100CCCC	A direct branch that was additionally checkpointed.
Indirect branch	b1001CCCC	An indirect branch occurred (load to PC or data processing instruction).
Checkpointed direct branch	b1100CCCC	An indirect branch that was additionally checkpointed.
Roll-Over	b11111111	16 instructions have executed since the last control flow change.

The following listing shows the entry number `NNN`, the entry `EE` and whether the entry is a message byte (`00`) or part of an address `AA` (`01`). In order to determine if an entry belongs to an address the trace buffer needs to be parsed backwards. This is possible because the last entry read (the most recent one) is guaranteed to be a message byte.

```
NNN EE AA
  1 00 01
  2 00 01
  3 20 01
  4 00 01 Address: 0x00002000
  5 90 00 Indirect branch to 0x00002000
  6 80 00 Direct branch
  7 ff 00 Rollover
  8 ff 00 Rollover
  9 ff 00 Rollover
 10 00 01
 11 00 01
 12 88 01
 13 e0 01 address: 0x00088e0
 14 97 00 Indirect branch to 0x000088e0 (55 instructions previously executed)
 15 ff 00 Rollover
 16 ff 00 Rollover
 17 ff 00 Rollover
 18 ff 00 Rollover
 19 85 00 Direct Branch (5 instructions previously executed)
 20 82 00 Direct Branch (2 instructions previously executed)
 21 82 00 Direct Branch (2 instructions previously executed)
 22 82 00
...
247 82 00
248 82 00
249 82 00
```

The resulting trace output from the OpenOCD is shown in the next listing with an instruction count, the instruction's address, opcode and disassembly. The large number of instructions that executed between trace entries 4 to 55 and between 62 to 201 was stripped from the output because little could be learned from examining a continuous stream of instructions.

```
  1 0x00002000      0xea0019fe      B 0x00008800
  2 0x00008800      0xe59f0944      LDR r0, [r15, #0x944]
  3 0x00008804      0xe3e01000      MVN r1, #0x0
  4 0x00008808      0xe5801000      STR r1, [r0]
...
 55 0x000088d4      0xee120f10      MRC p15, 0x00, r0, c2, c0, 0x00
 56 0x000088d8      0xe1a00000      NOP
 57 0x000088dc      0xe24ff004      SUB r15, r15, #0x4
 58 0x000088e0      0xe1a00000      NOP
 59 0x000088e4      0xe1a00000      NOP
 60 0x000088e8      0xe1a00000      NOP
 61 0x000088ec      0xe1a00000      NOP
 62 0x000088f0      0xe3e00000      MVN r0, #0x0
...
201 0x000089f0      0xe1540002      CMP r4, r2
202 0x000089f4      0xcafffffc      BGT 0x000089ec
203 0x000089ec      0xe5932000      LDR r2, [r3]
204 0x000089f0      0xe1540002      CMP r4, r2
205 0x000089f4      0xcafffffc      BGT 0x000089ec
206 0x000089ec      0xe5932000      LDR r2, [r3]
```

The trace run was recorded during bootup of a PXA250 based target called Karo Triton. Tracing was enabled immediately out of reset, but trace recording couldn't trace the very first instruction at address 0x0, because due to the nature of XScale debugging the instruction at the reset vector was simulated inside the debugger (see 7.1).

Once the trace buffer fills up in fill-once mode a debug exception causes a branch to the reset vector, and the debug handler gains control of the target. It is important for the debugger to remember the last address that executed, otherwise any instructions since the last control flow change would be lost. In this case debug state was entered with the PC at 0x000089f0, but the last instruction flow change was a branch at 0x89f4 to address 0x89ec. The trace decoding code identified that instruction 0x89ec must have executed because of the gap between the last trace message and the current PC.

In fill-once mode or when the trace buffer didn't wrap around the first traced instruction is already known, and the instruction flow can be reconstructed starting with the first entry from the trace buffer, but if the trace wrapped around it is unknown which addresses the trace data refers to. The XScale trace buffer incorporates two checkpoint registers that take the address of a direct or indirect branch. These checkpoints will be placed about half the size of the trace buffer apart from each other to maximize the useable content of the trace buffer. If the address of the first instruction of a trace is unknown trace analysis skips through the trace buffer until either a checkpointed branch or an indirect branch is encountered.

The first four entries were identified as addressed and can be skipped when trying to reconstruct the program flow. The fifth entry is an indirect branch message whose target can be seen in the four previous entries, 0x2000 in this case. We now know that the next instruction executed will be at address 0x2000. In this case the indirect branch didn't specify an incremental word count, but even if one was specified the corresponding instructions couldn't be traced because no valid PC was acquired at this point.

Entry 6 is a direct branch, and again no instructions executed since the last control flow change. The image running on the target needs to be examined to identify the target of the direct branch. The opcode is 0xea0019fe, an unconditionally executed branch that specifies a 24 bit offset to the current PC. Reading the PC in ARM mode returns the address of the instruction + 8, and the offset has to be shifted to the left by two, giving a branch target of 0x2000 + 0x8 + (0x19fe « 2) = 0x8800.

The following three entries are rollover messages, each indicating that another sixteen instructions executed since the last control flow change for a total of 48. Entries 10 to 13 are part of an address, but entry 14 is an indirect branch with an incremental word count of 7, indicating that 55 (48 + 7) instructions executed since the last direct branch message in entry 6. These were output by the OpenOCD trace analysis as instructions 2 to 56, followed by the indirect branch as instruction 57. Subtracting the immediate operand four from the current value of the PC (address + 8) results in a branch to the instruction following the branch. Usually the source operands of a data processing operation that has the PC as its destination are not all known, so the XScale placed the indirect branch message together with the destination address (0x88e0) in the trace buffer.

The remainder of the trace buffer could be analyzed similar to the small part examined here. For each instruction included in the incremental word count the program counter is simply increased by the size of an instruction in the current operating mode[4], and in case of a direct branch message the opcode of the final instruction (the one that caused the message to be output) needs to be examined to determine the address of the next instruction. The target address of indirect branches is available by looking at the four trace buffer entries that precede the indirect branch message.

7.3 OpenOCD XScale Trace Implementation

Collecting and analyzing data from the XScale trace buffer is implemented in `./src/target/xscale.c` and `./src/target/xscale.h`, with the help of the debug handler code in `debug_handler.S`.

Listing 7.1: ./src/target/xscale.h

```
enum xscale_trace_entry_type
{
    XSCALE_TRACE_MESSAGE = 0x0,
    XSCALE_TRACE_ADDRESS = 0x1,
};

typedef struct xscale_trace_entry_s
{
    u8 data;
    enum xscale_trace_entry_type type;
} xscale_trace_entry_t;
```

When parsing the collected trace data for the first time in `xscale_read_trace()` each entry (held in the `data` member of the `xscale_trace_entry_t` type) is assigned an `xscale_trace_entry_type`, identifying it as either a trace message (`XSCALE_TRACE_MESSAGE`) or as part of an address (`XSCALE_TRACE_ADDRESS`). The code in `xscale_read_trace()` also determines the number of valid entries read from the trace buffer by identifying the first non-zero entry that isn't part of an address. If no valid entry could be found an error is returned.

Listing 7.2: ./src/target/xscale.h

```
typedef struct xscale_trace_data_s
```

[4]Four bytes in ARM mode, two in Thumb mode.

```
{
    xscale_trace_entry_t *entries;
    int depth;
    u32 chkpt0;
    u32 chkpt1;
    u32 last_instruction;
    struct xscale_trace_data_s *next;
} xscale_trace_data_t;

typedef struct xscale_trace_s
{
    trace_status_t capture_status;  /* current state of capture run */
    image_t *image;                 /* source for target opcodes */
    xscale_trace_data_t *data;      /* linked list of collected trace
 data */
    int buffer_enabled;             /* whether trace buffer is enabled */
    int buffer_fill;                /* maximum number of trace runs to
 read (-1 for wrap-around) */
    int pc_ok;
    u32 current_pc;
    armv4_5_state_t core_state;     /* current core state (ARM, Thumb,
 Jazelle) */
} xscale_trace_t;
```

Information about the XScale trace data collected is stored in a member `trace` of type `xscale_trace_t` that is part of the XScale specific data (`xscale_t`). This includes the current state of the trace run, the image used to read opcodes, a linked list of `xscale_trace_data_t` items, the mode the trace port currently operates in (disabled, wrap-around, fill once, fill n-times), and state information for the trace analysis.

A `xscale_trace_data_t` item contains a number of trace entries, the content of the checkpoint registers when the trace was collected, the address of the last instruction executed, and a pointer to the next item in a linked list.

The collected trace data is analyzed in `xscale_analyze_trace()`. This function iterates through all items in the `xscale_trace_t->data` linked list and cycles through the entries collected in each of the trace runs, skipping items marked as a branch target address (`XSCALE_TRACE_ADDRESS`). Depending on the type of the message entry information about the target of the control flow change is stored in `next_pc` and if a valid PC was acquired `next_pc_ok` is set to 1.

If trace analysis acquired the address of the program counter prior to the current control flow change it is able to print information about the instructions that executed up to this point, including all the instructions for which overflow message bytes were output. If the control flow change was because of a IRQ or FIQ exception the current instruction didn't execute, and the number of instructions that executed is decreased by one. Every instruction is fetched from the `image_t` used for trace analysis. When the last but one instruction is reached and the control flow change was a data abort, the instruction is examined to see whether it was a load to the program counter. If it was not, the number of instructions executed is further decremented by one, because only aborts on loads to the PC are included in the incremental word count, and this was actually the last instruction executed.

In case the control flow change was a direct branch, the last instruction that executed is examined to read the branch target address from the opcode and the program counter is updated accordingly. It is not necessary to identify the branch targets of instructions that were included in the incremental word count because if one of those was a branch it didn't pass its condition test, otherwise it would caused a trace message to be output.

Every instruction is then output to the current command context (a telnet or GDB connection), and if the trace buffer entry specified a valid new PC it is saved as the trace context's current PC.

7.4 OpenOCD XScale Trace Usage

```
           xscale    xscale specific commands
     trace_buffer -    <enable|disable> ['fill' [n]|'wrap']
dump_trace_buffer -    dump content of trace buffer
    analyze_trace -    analyze content of trace buffer
      trace_image -    load image from <file> [base address]
```

The OpenOCD allows the trace buffer to be enabled either in wrap-around or in fill-once mode. If fill-once mode is selected, an optional number of fills can be selected, in which case the target is immediately resumed after a debug entry if the reason for debug entry was a trace buffer full event and there are additional fills to be collected. This allows larger trace runs to be captured with reduced effects for realtime execution.

For analysis of the trace buffer an image is needed which can be specified by the `xscale trace_image` command. This image can be one of the supported image file formats or a pseudo image which reads the target's memory to analyze the executed instructions. The collected trace data may also be dumped to a file for analysis with external tools via the `dump_trace` command. Trace analysis is started using the `analyze_trace` command.

8 ETM - Embedded Trace Macrocell

The embedded trace macrocell (ETM)allows real-time tracing of instructions and data as an ARM core executes. It connects directly to the ARM core inside an ARM based ASIC and outputs trace data on a trace port. Trace data is generated at the full processor clock, i.e. every individual core cycle may be observed, but the trace port may be clocked at a lower frequency using half-rate clocking and demultiplexing.

This master's thesis focuses on tracing for ARM7 and ARM9 based targets and will be limited to the ETM variants currently implemented for these cores. Three major versions of the ETM architecture exist, but only ETMv1 is implemented by the ETM7 and ETM9 found on ARM7 and ARM9 based targets. There are also several minor versions ETMv1.0 to ETMv1.3 but all the targets used with the prototype implementation that results from this master's thesis implement ETMv1.2 or higher so this document will ignore the differences that come with older variants of the ETMv1 architecture. The differences between ETMv1.2 and ETMv1.3 are:

- Support for FIFOFULL. The use of FIFOFULL would allow the core to be stalled before the ETM's internal FIFO overflows, but FIFOFULL requires support from both the ETM and the system. FIFOFULL currently isn't used by the software written for this master's thesis so the difference between ETMv1 and ETMv2 doesn't matter here.
- Support for tracing Java code. This is required for cores with Jazelle code that have native support for Java bytecode. Because there is no public information available on the Jazelle technology there is no support for Java available, so this difference doesn't matter either.

Older versions of the ETM architecture have more important differences that affect how tracing is controlled and filtered, but these shall be ignored in this document.

The targets with included ETM functionality examined for the purpose of this thesis are the NXP LPC2294, an ARM7TDMI-S based microcontroller on an Olimex LPC-H2294 headerboard, the ST Microelectronics STR912 o a Hitex STR912 Evalboard and the NXP LPC3180, an ARM926EJ-S based microcontroller on a Phytec phyCORE-LPC3180 single board computer. See table 8.1 for the ETMs used in these targets.

Table 8.1: ETM implementations

Target	ETM
NXP LPC2294	ETM7 Rev1 implementing ETMv1.2
STM STR912	ETM9 r2p2 implementing ETMv1.3
NXP LPC3180	ETM9 r2p2 implementing ETMv1.3

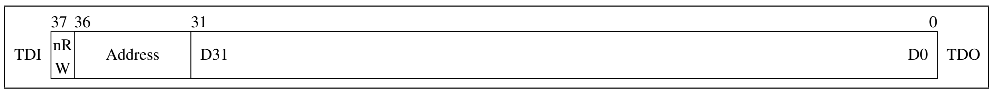


Figure 8.1: ETM Scan Chain

8.1 JTAG Access

The ETM registers can be programmed via the ARM core's TAP controller using scan chain 6 in way similar to the EmbeddedICE registers that are accessible via scan chain 2 [DR05, §3.3]. The debugger loads the SCAN_N instruction into the JTAG instruction register, scans the number of the ETM scan chain into the JTAG data register, and selects the INTEST JTAG instruction to read and write the ETM registers.

Figure 8.1 shows the layout of the ETM scan chain. Like the EmbeddedICE scan chain it consists of a 32 bit data field, a 5 bit address field, and a read/write bit.

8.2 Trace Port

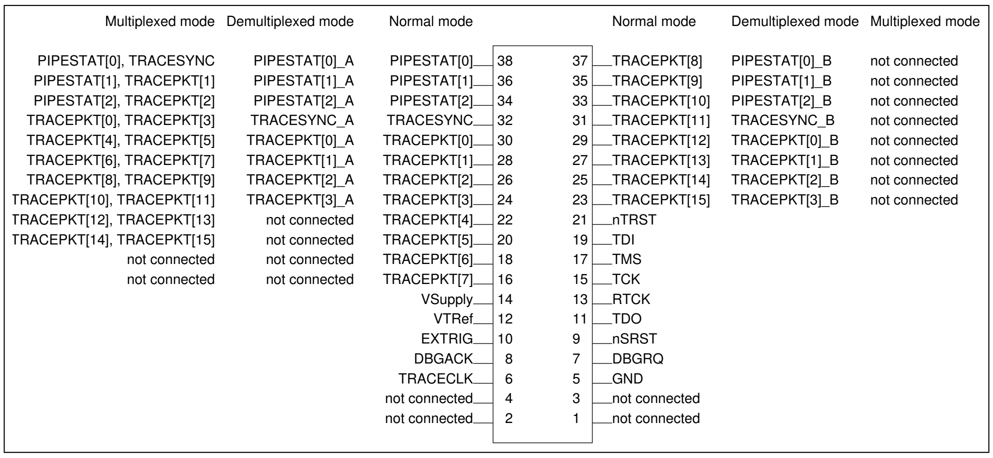
The trace port of ETMs implementing the ETMv1 architecture consists of the pins listed in table 8.2:

Table 8.2: ETMv1 trace port

Signal	Description
TRACECLK	The trace clock, running at the full processor speed, but optionally divided by 2 or 4.
PIPESTAT[2:0]	The current pipeline status, i.e. what the core actually did in this trace cycle.
TRACEPKT[n:0]	An output of 4, 8 or 16 bits from the ETM's FIFO containing trace information.
TRACESYNC	A synchronization signal used to match the TRACEPKT output to PIPESTAT cycles.

Because of the high frequency at which trace data is output a special connector type called "MICTOR" (matched impedance connector) is specified for use with ETM target connections. The use of Mictor connectors and cables allows for transmission at 200MHz and higher.

The separation of the PIPESTAT signals that closely follow the processor pipeline and the TRACEPKT signals allow a FIFO to be used when outputting the trace information. The TRACESYNC signal is used to synchronize between PIPESTAT and TRACEPKT.



Multiplexed mode	Demultiplexed mode	Normal mode			Normal mode	Demultiplexed mode	Multiplexed mode
PIPESTAT[0], TRACESYNC	PIPESTAT[0]_A	PIPESTAT[0]	38	37	TRACEPKT[8]	PIPESTAT[0]_B	not connected
PIPESTAT[1], TRACEPKT[1]	PIPESTAT[1]_A	PIPESTAT[1]	36	35	TRACEPKT[9]	PIPESTAT[1]_B	not connected
PIPESTAT[2], TRACEPKT[2]	PIPESTAT[2]_A	PIPESTAT[2]	34	33	TRACEPKT[10]	PIPESTAT[2]_B	not connected
TRACEPKT[0], TRACEPKT[3]	TRACESYNC_A	TRACESYNC	32	31	TRACEPKT[11]	TRACESYNC_B	not connected
TRACEPKT[4], TRACEPKT[5]	TRACEPKT[0]_A	TRACEPKT[0]	30	29	TRACEPKT[12]	TRACEPKT[0]_B	not connected
TRACEPKT[6], TRACEPKT[7]	TRACEPKT[1]_A	TRACEPKT[1]	28	27	TRACEPKT[13]	TRACEPKT[1]_B	not connected
TRACEPKT[8], TRACEPKT[9]	TRACEPKT[2]_A	TRACEPKT[2]	26	25	TRACEPKT[14]	TRACEPKT[2]_B	not connected
TRACEPKT[10], TRACEPKT[11]	TRACEPKT[3]_A	TRACEPKT[3]	24	23	TRACEPKT[15]	TRACEPKT[3]_B	not connected
TRACEPKT[12], TRACEPKT[13]	not connected	TRACEPKT[4]	22	21	nTRST		
TRACEPKT[14], TRACEPKT[15]	not connected	TRACEPKT[5]	20	19	TDI		
not connected	not connected	TRACEPKT[6]	18	17	TMS		
not connected	not connected	TRACEPKT[7]	16	15	TCK		
		VSupply	14	13	RTCK		
		VTRef	12	11	TDO		
		EXTRIG	10	9	nSRST		
		DBGACK	8	7	DBGRQ		
		TRACECLK	6	5	GND		
		not connected	4	3	not connected		
		not connected	2	1	not connected		

Figure 8.2: ETM Port Pinout

Trace Port Modes

The trace port may operate in normal, multiplexed or in demultiplexed mode:

- In normal mode all signals from the trace port are directly routed to device pins. The TRACECLK runs at either the full core clock or at half the core clock, if half-rate clocking is enabled. Half-rate clocking is used to reduce the signal transition frequency and requires the trace port analyzer (TPA)to capture data on both edges of the TRACECLK.
- Multiplexed mode is used when only few pins can be dedicated to outputting the trace information. The TRACECLK runs at the full core clock, but the pins output multiplexed data on both edges of the clock, i.e. a single pin outputs the information of two trace signals.
- Demultiplexed mode is used to reduce the signal transitioning rate by outputting trace data over twice as many pins at half the core's frequency. If half-rate clocking is enabled the TRACECLK is further divided by two and runs at $f_{core}/4$.

Figure 8.2 shows the pinout used with a single 38-pin ETM connector. If a demultiplexed ETM port would be used with a wider trace port (8 or 16 bit) a second ETM connector would be required, but this is beyond the scope of this document. If a trace port doesn't use the full 16 bit unconnected signals must be connected to ground (GND).

8.3 Event Resources

Resources

The ETM provides a wide variety of resources that can be used to control and filter tracing. These resources are based on memory accesses or are derived from memory accesses. A memory access can be either a data or an instruction fetch and may depend on an address, a data pattern, or combination of both.

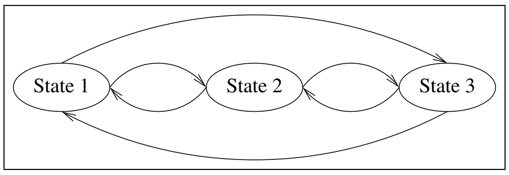


Figure 8.3: ETM Sequencer

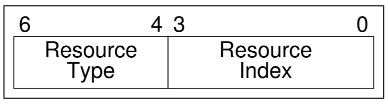


Figure 8.4: ETM Resource

The following list shows the resources available in the ETMv1 architecture:

- Memory access resources
  - Single address comparators that match on instruction or data accesses, optionally combined with a data comparator to match only when a certain word is read from an address.
  - Address range comparators are like single address comparators but match on a whole address range instead of a single address. An addres range comparator matches when an address is $>=$ address A and $<$ address B.
  - EmbeddedICE watchpoint comparators can be used if the RANGEOUT feature supported. They operate similar to single address comparators.
  - Memory map decoders divide the ASIC specific memory map for example to identify on-chip RAM, ROM and registers. Using the memory map decoder requires further information from the chip vendor.
- Derived resources Derived resources allow more complex terms to be formulated by combining memory access resources with additional logic:
  - 16 bit counters that count down at the full system clock with a counter enable event based on any of the other event resources.
  - The sequencer is a three state statemachine as shown in figure 8.3. This allows multi-stage triggers to be set up, for example to trigger only inside function X if function Y was called before. The state machine advances to another state based on events from other resources.
- External resources
  - Hardwired “true” is a resource that’s always true.
  - External inputs are ASIC specific inputs that can be true or false.

ETM resources as shown in figure 8.4 are defined by the *type* of resoure, encoded in bits six to four, and a four bit *index* in bits three to zero. The resource type is one of the types listed in table 8.3, and the index selects a particular instance of the selected resource, like an address comparator pair or one of the counters. Table 8.3 shows how the *Resource Type* encoding maps to the ETM resources described above.

Only indexes from the defined range may be used when selecting a resource as an input to an event.

Table 8.3: ETM resource types

Encoding	Description
000	Single address comparator 1-16, indexed as 0-15
001	Address range comparator 1-8, indexed as 0-7
010	EmbeddedICE comparator 1-2, indexed as 0-1
011	Memory map decode 1-16, indexed as 0-15
100	Counter 1-4 at zero, indexed as 0-3
101	Sequencer in states 1-3, indexed as 0-2
110	External inputs 1-4, indexed as 0-3, or hardwire 'true', index 15
111	*Reserved*

The resources available in a particular ETM implementation can be read from the *ETM configuration code register*, and should be used in conjunction with the *System configuration register* to determine the capabilities offered by the combination of ETM and ASIC:

- ETM configuration code register

[23]	FIFOFULL present
[22:20]	Number of ext. outputs
[19:17]	Number of ext. inputs
[16]	Sequencer present
[15:12]	Number of counters
[11:8]	Number of memory map decoders
[7:4]	Pairs of data comparators
[3:0]	Pairs of address comparators

- System configuration register

[8]	FIFOFULL supported
[7]	Demultiplexed trace data format supported
[6]	Multiplexed trace data format supported
[5]	Normal trace data format supported
[4]	Full-rate clocking supported
[3]	Half-rate clocking supported
[2:0]	Maximum port size. See table 8.5 for the supported encoding.

The targets chosen for the prototype implementation of ETM trace analysis offer a very different amount of resources, one implementing the large standard and two implementing only the small standard [IHI0014N, §2.12.2].

Events

An ETM event is the boolean combination of up to two resources, allowing complex conditions to be set up. The layout for all ETM event registers is the same and is shown in figure 8.5.

Table 8.4: ETM configurations

Feature	LPC2294	STR912	LPC3180
Pairs of address decoders	1	1	8
Data value comparators	0	0	8
Memory map decoders	4	4	16
Counters	1	1	4
Sequencer present	No	No	Yes
External inputs	2	2	4 (not wired)
External outputs	0	0	4 (not wired)
FIFOFULL present	Yes	Yes	Yes
FIFODEPTH	10	9	45
Max. port size	4	4	16 (wired to ETB)
Port mode	Half-Rate	Full-Rate	n/a
Trace data format	Normal	Normal	Normal

- ETM Event registers

[16:14]	The boolean function that defines how the two resources are combined to generate the event output.
[13:7]	Resource B, the second operand to the boolean function.
[6:0]	Resource A, the first operand to the boolean function.

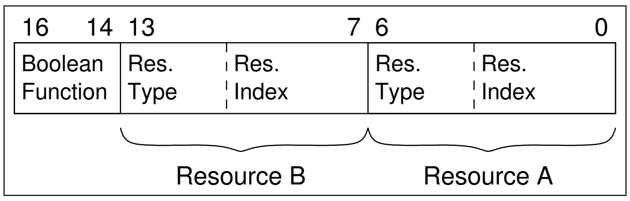


Figure 8.5: ETM Event

The following encodings for boolean functions exist:

000	A (Event is true when resource A is true)
001	Not(A) (Event is true when resource A is false)
010	A And B (Even is true when both resources A and B are true)
011	Not(A) And B (Event is true when A is false and B is true)
100	Not(A) And Not(B) (Event is true when neither A nor B is true)
101	A Or B (Event is true when A or B is true)
110	Not(A) Or B (Event is true when A is false or B is true)
111	Not(A) Or Not(B) (Event is true when A or B is false)

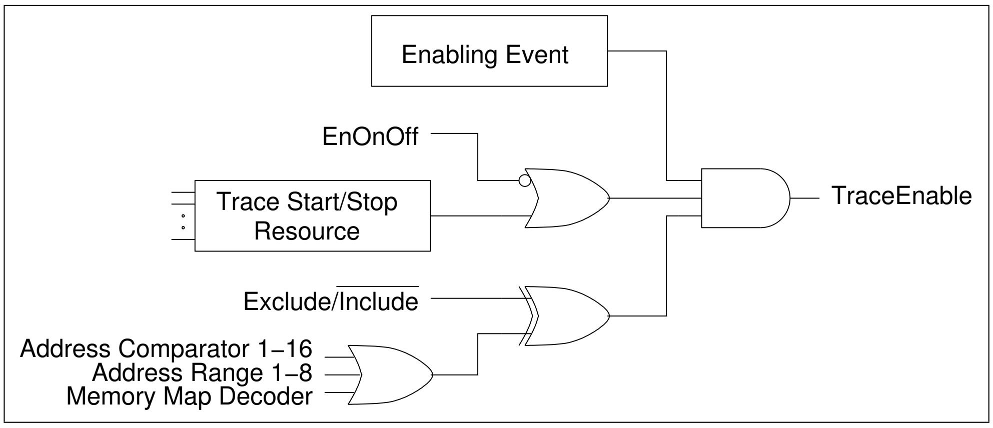


Figure 8.6: ETM TraceEnable Signal

8.4 Trace Control And Filtering

Various options are provided to configure which aspects of the program execution should be traced.

The ETM resources can be used to control starting and stopping of the trace and to trigger around a certain point of interest. Trace start/stop allows the tracing to be limited to an area of interest, for example to addresses in which a user's application is located, preventing areas that are part of the operating system or some library from being traced.

Triggering is important when the trace data can't be examined on the fly and all trace information is written into a buffer for later examination. A trigger counter starts counting down to zero once the trigger condition is met. Once the counter reaches zero the trace capture is disabled. If the trigger counter is initialized with a low value compared to the buffer depth the useable trace contains data from before the trigger point. A value around half of the buffer's depth is used to trace around the point of interest, while a large value allows what happened after the trigger to be traced. There will only be one trigger output during a trace run, and the ETM needs to be programmed again to reenable generation of the trigger signal.

The *TraceEnable* (see figure 8.6) signal is used to decide when core execution should be traced. The ETM continuously outputs data on the trace port, but formats the output so that a trace port analyzer (TPA) only captures data when *TraceEnable* is high and there is data to be traced.

The *ViewData* (see figure 8.7) signal indicates whether data for a data load or store instruction is to be traced. Especially the amount of data tracing needs to be carefully limited via *ViewData* to avoid overflowing a narrow trace port. An ARM9 core is capable of loading one word per cycle from zero waitstate memory (usually cache or DTCM [1]), but the trace port is only 16 bit wide at most. Tracing the address of a data access causes five packets of eight bits size to be output, and if the data content is to be traced, too, another four packets will be required to trace a 32 bit access.

[1]Data Tightly Coupled Memory, see 4.2

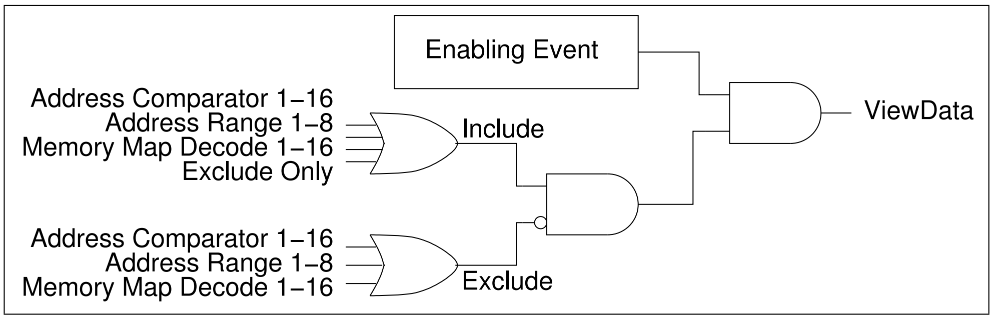


Figure 8.7: ETM ViewData Signal

Address And Data Comparator Registers

For every address comparator implemented by an ETM an *Address Comparator Value Register* and an *Address Access Type Register* is provided, allowing the address and type of the access to be specified that should be monitored by the comparator. If the ETM additionaly features data comparators two additional registers per address comparator pair take the data value (*Data Comparator Value Register*) and a mask (*Data Comparator Mask Register*) to make the comparator dependant on a particular data pattern.

- Address Comparator Value Register

[31:0] Address Value. The address that should be monitored by this comparator.

- Address Access Type Register

[6:5] Data comparison control. When a data comparator is available this field can be set to b00 to ignore the data, b01 limits the comparator to accesses where the data matches, and b11 causes the address comparator to match only when the data comparator doesn't match. The encoding b10 is reserved and must not be used.

[4:3] Size. This field can be set to b00 to trace Java instructions or byte data, b01 to trace Thumb code or half-word accesses, or b11 to trace ARM code or word accesses. The encoding b10 is reserved and must not be used.

[2:0] Access type. This field has several encodings that specify the type of access this comparator should monitor:

  ∗ b000: Instruction fetched
  ∗ b001: Instruction executed (condition test ignored)
  ∗ b010: Instruction executed with condition test passed
  ∗ b011: Instruction executed with condition test failed
  ∗ b100: Data access (load or store)
  ∗ b101: Data load
  ∗ b110: Data store

- Data Comparator Value Register

[31:0]	Data Value. The data value that should be monitored by this comparator.

- Data Comparator Mask Register

[31:0]	Data Mask. Bits programmed as one are ignored when comparing the data.

The data value and mask must be programmed to compare only valid bytes in a transfer depending on the access size, address and endianness of the memory system. For example to watch the byte at address 0x1001 for value 0xVV on a little-endian system the value has to be set to 0x0000VV00 with a mask of 0xffff00ff.

TraceEnable Registers

The *TraceEnable* signal is controlled by the following registers:

- Trace Start/Stop Resource Control Register

[31:16]	Select single address comparator 1-16 as stop address
[15:0]	Select single address comparator 1-16 as start address

- TraceEnable Control 2 Register

[15:0]	Select single address comparator 1-16 for include/exclude control

- TraceEnable Event Register

[16:0]	TraceEnable Event

- TraceEnable Control Register

[25]	EnOnOff (1: Tracing controlled by on-/off-address)
[24]	Exclude/notInclude (0: Include, 1: Exclude)
[23:8]	Select memory map decode 1-16 for include/exclude control
[7:0]	Select address range comparators 1-8 for include/exclude control

The *EnOnOff* control bit causes the start/stop resource to be don't care if 0 (EnOnOff is a negated input to the OR element). The *Exclude/notInclude* bit is an input to an XOR element causing the output to be the negation of the preceding OR element if high.

ViewData Registers

The *ViewData* signal is controlled by the following registers:

- ViewData Event Register

[16:0]	ViewData Event

- ViewData Control 1 Register

[31:16]	Select single address comparator 1-16 for exclude control
[15:0]	Select single address comparator 1-16 for include control

- View Data Control 2 Register

[31:16]	Select memory map decode 1-16 for exclude control
[15:0]	Select memory map decode 1-16 for include control

- View Data Control 3 Register

[16]	Exclude only (0: Mixed mode, 1: Exclude only)
[15:8]	Select address range comparators 1-8 for exclude control
[7:0]	Select address range comparators 1-8 for include control

The *Exclude only* bit set to 1 causes the *Include* signal to be unconditionally high, resulting in *ViewData* matching all accesses that are not specifically excluded. If *Exclude only* is 0 the *Include* signal is controlled by matches of the address comparators and memory map decodes, resulting in matches only when an access is included but not excluded.

ETM Trace Control Registers

- ETM Control Register

[17:16]	Port mode. Whether the port operates in normal mode (b00), multiplexed mode (b01) or in demultiplexed mode (b11).
[15:14]	Context ID Size. Number of bits from the context ID that should be traced. None (b00), bits [7:0] (b01), bits [15:0] (b10) or bits [31:0] (b11).
[13]	Half-rate clocking. Whether the TRACECLK should be divided by two, resulting in a frequency of $f_{core}/2$ (in normal port mode) or $f_{core}/4$ (in demultiplexed port mode).
[12]	Cycle accurate tracing. Causes the trace data to be formatted so that every cycle with *TraceEnable* high will be traced, even when there is nothing to be traced (i.e. waitstates or internal cycles and no trace packet).
[11]	ETM port selection. Needs to be set to 1 to enable use of the ETM pins as the trace port mode. If 0 these bits can be used as GPIO.
[10]	ETM programming. This bit has to be set to 1 before ETM registers can be programmed.
[9]	Debug request control. Allows the ETM trigger event to be used as a debug request. If set to 1 and a trigger occurs the DBGRQ signal is asserted until the core acknowledges debug via the DBGACK signal.
[8]	Branch output. Setting this bit causes all branches to output an address. This allows the execution flow to be reconstructed without an available image.
[7]	Stall processor. The FIFOFULL signal allows the core to be stalled when the FIFO is about to overflow (using a programmable high-water-mark) if this bit is programmed to 1.
[6:4]	Port size. Specifies the width of the TRACEPKT output. See table 8.5 for valid port size encodings.
[3:2]	Data access. Causes nothing (b00), data portion (b01), address portion (b10) or both parts (b11) of a data access to be traced.

[1] Monitor CPRT. If set to 1 coprocessor register transfers will be traced.
[0] ETM power down. Out of reset the ETM is powered down and only the `ETM control register` may be accessed to program this bit to 1. If set to 1 all parts of the ETM can be accessed.

- Trigger Event Register

[16:0] The event used to trigger the trace capture. See figure 8.5 for the encoding.

- ETM Status Register

[2] The current status of the trace start/stop resource. If high, a start address has been traced but no stop trace was encountered yet.
[1] Allows the value of the *ETM control register*'s, *ETM programming* bit to be read back. This bit needs to be polled until it goes high before ETM registers may be programmed.
[0] FIFO overflow. This bit is high if an overflow occurred that hasn't been traced already.

Table 8.5: ETM port size encoding

Encoding	Description
b000	4 bit port
b001	8 bit port
b010	16 bit port

8.5 ETM Trace Format

The ETM outputs the current *PIPESTAT[2:0]* for each cycle executing, see table 8.6 for the encoding. Trace packets are always 8 bits wide, no matter what the current ETM port size is set to. In case of a 16 bit trace port up to two trace packets can be output in one cycle, and if a 4 bit port is used a trace packet is output in two consecutive cycles.

A trace packet can be output on each cycle except in *TR* or *TD* cycles. The packets that belong to one PIPESTAT (e.g. a *BE* PIPESTAT with its associated branch address) will be output as a continuous block with no other packets in between. Gaps in the output only exist when the PIPESTAT would be *WT* which is turned into a *TD* and FIFO draining is deferred. Packets for a particular instruction wont start together with or before the PIPESTAT for the previous instruction was output, but may follow immediately after the preceding instruction, before the PIPESTAT to which the packets belong is output.

Special rules apply to 16 bit wide ports where it is possible to output two trace packets in a single cycle. The first packet of a branch will always be output on TRACEPKT[7:0]. A single packet will only be output if the PIPESTAT is not *WT* - during *WT* cycles there will usually be two packets, otherwise draining is stopped until there is another functional PIPESTAT or until there are at least two trace packets to be output. There are exceptions that cause a single packet to be output even though the current PIPESTAT is *WT*. This is the case when the FIFO is being drained after an overflow, if the ARM core is in debug state, or when the FIFO is being drained after tracing was disabled.

Table 8.6: ETM PIPESTAT encoding

Encoding	Mnemonic	Description
b000	IE	Instruction executed. An instruction executed but generated no associated trace packet.
b001	ID	Instruction executed with data. A load or store executed and the data, address or both were placed into the FIFO.
b010	IN	Instruction not executed. An instruction didn't execute because its condition code test failed.
b011	WT	Waitstate. No instruction executed and the pipeline didn't advance, for example because the memory system delayed an access or because the core executed an internal cycle. A trace packet is output in this cycle.
b100	BE	Branch executed. An indirect branch executed, or a direct branch required the branch address to be output.
b101	BD	Branch executed with data. A data access had *ViewData* enabled and loaded the program counter.
b110	TR	Trigger. A trigger occurred and replaced the original PIPESTAT which is output in TRACEPKT[2:0] instead. No other trace packet is output in this cycle.
b111	TD	Trace disabled. This PIPESTAT is output when *TraceEnable* is low or when there is no trace packet to be output. In cycle accurate tracing TD is output with *TRACEPKT[0]* high to indicate that *TraceEnable* was high.

Exceptions are traced as branches to the corresponding vector. Depending on the type of exception this means that the current instruction was interrupted (Reset, IRQ, or FIQ) or executed and was then turned into a branch to the exception vector (Prefetch Abort, Data Abort, Undefined Instruction or SWI).

Branch addresses and addresses of data accesses are output as up to five packets with the eighth bit indicating that more packets follow (8th bit set). If less than five packets are output those bits that were output replace the bits from the previous address. This means that after tracing started a full branch address output is needed to acquire a complete 32 bit address. Earlier branches that output only part of the address can't be traced because the branch target address will be unknown.

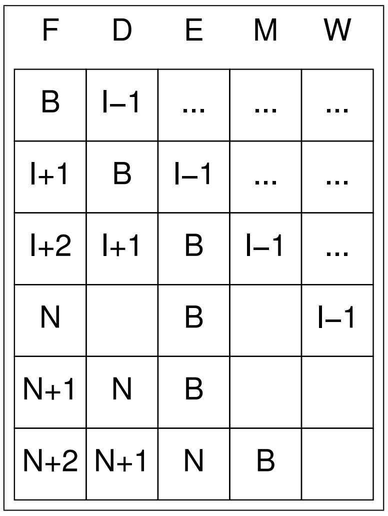
		32			24		16		8		0
J	Reason	[31:28]	1	[27:21]	1	[20:14]	1	[13:7]	1	[6:0]	

The *Reason* given in a full branch address output provides further information about the branch. The valid reason codes are given in table 8.7, other codes are reserved.

Following a branch two instructions that have already been fetched will have to be discarded, and the pipeline needs to be refilled. Figure 8.8 shows how a branch was fetched, followed by fetching two instructions (I+1 and I+2) from the current instruction stream. By the time the branch reached the execute stage the destination of the branch is calculated and instruction I+2 is fetched, because at that point is already too late to prevent the fetch. The branch instruction remains in the execute stage, the fetch and decode stages are flushed, and a new instruction is fetched from the new location (N). In the next cycle another instruction is fetched (N+1), the previously fetched instruction advances into the decode stage,

Table 8.7: ETM Branch Reason Code

Encoding	Description
b000	Normal PC change
b001	Tracing enabled
b010	Trace restarted after FIFO overflow
b011	Exit from debug state
b100	Periodic synchronization point



F	D	E	M	W
B	I–1	...	...	...
I+1	B	I–1	...	...
I+2	I+1	B	I–1	...
N		B		I–1
N+1	N	B		
N+2	N+1	N	B	

Figure 8.8: ARM Pipeline during a branch

while the branch still occupies the execute stage. The branch is then free to move into the memory stage (nothing happens there for a branch) and instruction N gets executed. While figure 8.8 refers to a 5-stage ARM9 pipeline the same mechanisms apply to a 3-stage ARM7 pipeline where the execute stage is the third stage, too.

These two prefetch cycles during which the branch instruction occupies the execute pipeline stage are reused by the ETM to output an address packet offset (APO) that is used to synchronize between the PIPESTAT and TRACEPKT signals. During the first APO cycle the lower two bits of the address packet offset are output on PIPESTAT[1:0], the second APO cycle outputs the higher two bits. PIPESTAT[2] isn't used during an APO to be able to identify a *BE/BD* PIPESTAT that is immediately followed by another *BE/BD* PIPESTAT, discarding the previous branch. A *TR* PIPESTAT might also occur at any time and can be identified similarly. The APO specifies the number of branch addresses (packets with TRACESYNC high) that have to be skipped to get to the address that belongs to the current branch (the one that caused the APO).

Example ETM Trace

The following trace data was collected while running a blinking LED example on the NXP LPC3180 board. The code was loaded into the microcontroller's internal SRAM at address 0x08000000 and the target was resumed out of reset. The example code then executed the following steps:

- Prepare stacks for each core mode
- Enable PLL to run at 208MHz
- Enable SDRAM running at 104MHz
- Relocate text section to SDRAM and jump to address in SDRAM
- Generate page table to directly map memory areas with appropriate cacheable (IRAM/IROM, SDRAM) and bufferable (IRAM, SDRAM) settings.
- Enable MMU and caches (both instruction and data cache)

The example toggles LEDs in an endless loop, waits between toggles by counting up to one million, and uses inline assembly to generate an instruction stream that causes the ETM FIFO to grow full enough to cause trace packets to be delayed. After toggling through all LEDs a hundred times an invalid access causes a data abort exception to allow verification of the ETM analysis exception handling. The complete code and precompiled binaries can be found on the accompanying CD in the folder `examples/lpc3180_gcc_blinky_sdram_cached_abort/`.

The trace run was set to collect both the address and the data of all accesses (the *ViewEnable* signal was set to include the whole memory area), cycle accurate tracing was enabled, and only the `wait()` function was excluded from the trace by programming a pair of address comparators to exclude this area. The trigger was programmed to the location of the abort handler (0x10) and the trigger counter was programmed to use only ten percent of the available buffer space after the trigger occurred, allowing the events that lead to the data abort to be examined. The LPC3180 features a combination of an ETM and an ETB [2] to allow trace data to be captured without the need for a separate trace capture unit. The ETM was used in normal trace format mode with a 16 bit wide trace port. The data read from the ETB was preprocessed to include the TRACESYNC bit in `FL[0]` (flags), it has the `TR` PIPESTAT replaced by the real PIPESTAT read from TRACEPKT[2:0] and uses `FL[1]` to indicate a trigger cycle. The PIPESTAT column is abbreviated as `PI` and TRACEPKT as `TPKT`, and contiguous cycles with `TD` PIPESTAT were stripped from the trace information.

```
### PI TPKT FL NOTES
  1 07 8001 00 TD
...
  6 07 8001 00 TD
  7 00 8000 00 IE
  8 03 8004 00 WT
  9 01 0000 00 ID
 10 07 ea01 00 TD
 11 07 ea01 00 TD
```

[2]ETB: Embedded Trace Buffer, see chapter 9.

```
 12 07 ea01 00 TD
 13 00 ea00 00 IE
 14 07 14e9 00 TD
 15 04 8ffc 01 BE --- tracing enabled at 0x800007fc ---
 16 00 8080 00 APO1 (00)
 17 00 1418 00 APO2 (00)
 18 07 f011 00 TD
 19 04 11b8 01 BE
 20 00 e599 00 APO1 (00)
 21 00 e599 00 APO2 (00)
```

The instructions that executed during cycles 1 to 14 couldn't be reconstructed because no address was known at that point. In cycle 15 a branch with destination 0x800007fc and reason code 0x1 executed, indicating that tracing was enabled at that point. The reason for enabling tracing at that point is that the address pair comparators were programmed to match between 0x800007b4 and 0x800007fc, which excludes the last address of the wait function (address comparator pairs match *start* $<= X <$ *end*). Following the BE PIPESTAT two APO cycles specifying an offset of zero can be seen, so the address packet that starts at cycle 15 (TRACESYNC was set) holds the first part of the address that belongs to the current branch. The instruction at 0x800007fc is a *branch and exchange* (`BX`) to the address held in r14 (also called link register (lr), the return address). This causes another BE to be output, again with an APO of zero. This time only two packets were output (0xb8 and 0x11, 0x11 has the MSB set to zero to indicate that no packets follow) which form the address 0x08b8(/14)[3]. The next instruction is therefor found at address 0x800008bc (bits 31 to 14 were taken from the previous full address output, bits 13 to 0 are from the current branch target).

```
### PI TPKT FL NOTES
 22 00 e599 00 IE 0x800008b8      0xe3a0390a      MOV r3, #0x28000 (3 cycles)
 23 00 e599 00 IE 0x800008bc      0xe2833121      ADD r3, r3, #0x40000008 (1 cycle)
 24 07 e599 00 TD
 ...
 36 07 e599 00 TD
 37 00 e599 00 IE 0x800008c0      0xe3a02080      MOV r2, #0x80 (14 cycles)
 38 01 8088 00 ID 0x800008c4      0xe5832000      STR r2, [r3] (1 cycle)
                                                  address: 0x40028008
                                                  data: 0x00000080
 39 00 808a 00 IE 0x800008c8      0xe3a03902      MOV r3, #0x8000 (1 cycle)
 40 00 8004 00 IE 0x800008cc      0xe2833004      ADD r3, r3, #0x4 (1 cycle)
```

In cycles 22 and 23 two data processing instructions[4] executed to form the address for the following store. Cycle 37 loads register r2 with the word to be written, and in cycle 38 the *ID* PIPESTAT indicates that a load or store instruction executed and that an address, data or both were output. Because this trace run had both address and data tracing enabled the trace analysis expects to find up to five address packets

[3]The notation 0xXXXX(/n) is used to show that only *n* bits of the hex word are valid

[4]Because the ARM instruction set is 32 bits wide there is no room to hold 32 bit immediate operands. When such operands are required they are often built using a sequence of dataprocessing instructions with rotated 8 bit immediates instead.

followed by four data packets (the number of data packets can be deduced by looking at the instruction opcode, in this case a word store (32 bit)). The cycle count in brackets is the number of ETM cycles between two instructions in a cycle accurate trace. Obviously the `MOV` didn't take 14 cycles to execute while the store to address 0x40028008[5] reportedly took only one cycle. The problem here is that the waitstates and internal cycles associated with some instructions cause the PIPESTAT to be output too late or too early, but it is still possible to examine the number of cycles a block of code took to execute.

The next block from the trace dump is located a few cycles further down the execution. Here multiple load/store instructions generated enough trace packets to occupy at least part of the FIFO, and two subsequent branch instructions have their trace packets output late so that an APO greater than zero was generated.

```
### PI TPKT FL NOTES
115 00 8001 00 IE 0x8000083c      0xe3a02004      MOV r2, #0x4 (14 cycles)
116 01 8084 00 ID 0x80000840      0xe5832000      STR r2, [r3] (1 cycle)
                                                  address: 0x40028004
                                                  data: 0x00000004
117 03 808a 00 WT
118 03 0404 00 WT
119 00 0000 00 IE 0x80000844      0xe3a00102      MOV r0, #0x80000000 (3 cycles)
120 03 8000 00 WT
121 03 8080 00 WT
122 03 0880 00 WT
123 01 000d 00 ID 0x80000848      0xe890001e      LDMIA r0, {r1, r2, r3, r4}
                                                  (4 cycles)
                                                  address: 0x80000000
                                                  data: 0xea00000d
                                                  data: 0xe59ff014
                                                  data: 0xe59ff014
                                                  data: 0xe59ff014
124 04 ea00 00 BE 0x8000084c      0xe24ff004      SUB r15, r15, #0x4 (1 cycle)
125 00 f014 00 APO1
126 00 e59f 00 APO2
127 04 f014 00 BE 0x80000850      0xe24ff004      SUB r15, r15, #0x4 (3 cycles)
128 01 e59f 00 APO1
129 00 f014 00 APO2
130 03 e59f 00 WT
131 03 10d0 01 WT
132 00 0054 01 IE 0x80000854      0xebffffd6      BL 0x800007b4 (5 cycles)
133 07 0001 00 TD
134 04 8ffc 01 BE --- tracing enabled at 0x800007fc ---
135 00 8080 00 APO1
136 00 0018 00 APO2
```

[5]Address 0x40028008 is the "Output Pin Set Register" (PIO_OUTP_SET), used to set GPIO pins on the LPC3180 [UM10198, §5.2]

The address for the store in cycle 116 was output starting in cycle 116 on TRACEPKT[7:0] until cycle 118 (TRACEPKT[7:0]), followed by the data in cycle 118 (TRACEPKT[15:8]) to cycle 120 (TRACEPKT[7:0]). Outputting the packets earlier wasn't possible because the packets for an instruction can't start at the same time or before the preceding instruction which was the IE in cycle 115.

The address for the `LDMIA` in cycle 131 immediately follows after the last data packet from the previous store. Cycle 120 was a *WT* cycle which is why we know that there were two packets output. The `LDMIA` base address was output from cycle 120 (TRACEPKT[15:8]) to cycle 122 (TRACEPKT[15:8]), followed by 16 packets of data (four 32 bit words) up to cycle 130. In cycle 124 a branch executed with an APO of zero, but this time the TRACESYNC signal wasn't high, so the target address must start later in cycle 131 where TRACESYNC is high for the first time since the *BE* PIPESTAT. Immediately after the first branch another one follows in cycle 127, but because there was already a branch address pending an APO of one was output. The address output in cycle 131 (packets 0xd0 and 0x10) is 0x0850(/14), followed by address 0x54(/7) in cycle 132. Again only the valid bits from the branch target addresses replace the corresponding bits from the previously output address, resulting in branch targets of 0x80000850 and 0x80000854. The trace then continue with a branch to the `wait()` function at address 0x800007b4 where tracing is disabled until it's reenabled at address 0x800007fc.

The following trace dump sequence is from the area around the trigger which was output in cycle 1844 (flag 0x2 set). The LED blinking loop executed one hundred times and the deliberately wrong store at address 0x80000930 caused a data abort exception resulting in a branch to address 0x10 that resulted in the trigger.

```
1704 07 0401 00 TD
1705 00 0400 00 IE 0x80000910      0xe3520000      CMP r2, #0x0 (35 cycles)
1706 07 8001 00 TD
1707 02 8001 00 IN 0x80000914      0x0affffc5      BEQ 0x80000830 (not executed)
                                                   (2 cycles)
1708 07 8001 00 TD
1709 00 8001 00 IE 0x80000918      0xe3e02585      MVN r2, #0x21400000 (4 cycles)
1710 07 8001 00 TD
1711 00 8001 00 IE 0x8000091c      0xe2422949      SUB r2, r2, #0x124000 (2 cycles)
1712 07 8001 00 TD
1713 00 8001 00 IE 0x80000920      0xe2422e11      SUB r2, r2, #0x110 (2 cycles)
1714 07 8001 00 TD
1715 00 8001 00 IE 0x80000924      0xe3a036ba      MOV r3, #0xba00000 (2 cycles)
1716 07 8001 00 TD
1717 00 8001 00 IE 0x80000928      0xe2833937      ADD r3, r3, #0xdc000
                                                   (2 cycles)
1718 07 8001 00 TD
...
1758 07 8001 00 TD
1759 00 8001 00 IE 0x8000092c      0xe28330de      ADD r3, r3, #0xde (42 cycles)
```

```
1760 03 fdef 00 WT
1761 03 f5b6 00 WT
1762 03 de0d 00 WT
1763 03 adc0 00 WT
1764 07 0009 00 TD
...
1843 07 0009 00 TD
1844 03 000b 02 WT (trigger)
1845 05 900b 00 BD 0x80000930     0xe5823000     STR r3, [r2] (86 cycles)
                                                data abort
                                                address: 0xdeadbeef
                                                data: 0x0badc0de
1846 00 8090 01 APO1
1847 00 8080 00 APO2
1848 07 1801 00 TD
...
1918 07 1801 00 TD
1919 04 9c00 00 BE 0x00000010     0xe59ff014     LDR r15, [r15, #0x14] (74 cycles)
1920 00 939c 01 APO1
1921 00 8080 00 APO2
1922 03 f408 00 WT
1923 01 ffff 00 ID 0x8000099c     0xe52dc004     STR r12, [r13, #-0x4]! (4 cycles)
1924 00 088f 00 IE 0x800009a0     0xe1a0c00d     MOV r12, r13 (1 cycle)
1925 03 ff3c 00 WT
1926 03 81ff 00 WT
1927 03 6864 00 WT
1928 01 ffff 00 ID 0x800009a4     0xe92dd800     STMDB r13!, {r11, r12, r14, r15}
                                                (4 cycles)
                                                address: 0x81ffffe4
                                                data: 0x81ffff68
                                                data: 0x81fffff4
                                                data: 0x80000938
                                                data: 0x800009b0
```

The compare[6] that executed in cycle 1705 caused the zero flag to be cleared. The branch if equal (BEQ) instruction failed its condition test and wasn't executed (PIPESTAT is *IN*), therefor the instruction flow continued without branching. During cycles 1709 to 1759 registers r2 and r3 were loaded with the values 0xdeadbeef (r2) and 0xbadc0de (r3), and the store instruction in cycle 1845 caused a data abort exception because there is no valid memory at address 0xdeadbeef[7]. The store was changed into a branch with data by the ETM and the branch target (0x10), the address of the access and the data were output as trace packets. The instruction at the abort vector is a PC relative load instruction that is output as a *BE* in cycle 1919. The TRACESYNC signal wasn't high because there was a single packet left in the FIFO, the last packet from the *BD* branch target address that started in cycle 1846 (TRACESYNC high).

[6]The ARM *CMP* instruction subtracts the second operand from the first and updates the flags accordingly, without storing the result.

[7]Address 0xdeadbeef is marked as a reserved area in the LPC3180 user's manual.

This packet couldn't be output earlier because no functional packets were generated until cycle 1919 and a *WT* cycle always outputs two packets on a 16 bit port. The PC relative load branched to address 0x8000099c were the usual C function prologue (store base pointer, move stack pointer to base pointer, store registers to stack) executed.

8.6 OpenOCD ETM Implementation

The OpenOCD implements ETM support in `./src/target/etm.c` using declarations from `./src/target/etm.h` and `./src/target/trace.h`. `trace.h` is meant to provide some generally useful defines for embedded systems tracing, but is currently limited to defining possible states of a trace run:

Listing 8.1: ./src/target/trace.h

```c
typedef enum trace_status
{
    TRACE_IDLE = 0x0,
    TRACE_RUNNING = 0x1,
    TRACE_TRIGGERED = 0x2,
    TRACE_COMPLETED = 0x4,
    TRACE_OVERFLOWED = 0x8,
} trace_status_t;
```

The register handling in `etm.c` is done the same way that `embeddedice.c` handles the EmbeddedICE registers, using the register cache functionality to provide a defined interface for the user to program the ETM registers. A special hook in `etm_buildregister_cache` is provided for the ETB capture driver to allow it to add its registers to the target's register cache.

If a target has an ETM configured the ARM7/9 specific `arm7_9_common_t->etm_ctx` field is initialized with a pointer to a structure of type `etm_context_t`:

Listing 8.2: ./src/target/etm.h

```c
typedef struct etm_context_s
{
    target_t *target;               /* target this ETM is connected
to */
    reg_cache_t *reg_cache;         /* ETM register cache */
    etm_capture_driver_t *capture_driver;   /* driver used to access
ETM data */
    void *capture_driver_priv;      /* capture driver private data */
    u32 trigger_percent;            /* percent of trace buffer to be
filled after the trigger */
    trace_status_t capture_status;  /* current state of capture run */
    etmv1_trace_data_t *trace_data; /* trace data */
    u32 trace_depth;                /* number of trace cycles to be
analyzed, 0 if no trace data available */
    etm_portmode_t portmode;        /* normal, multiplexed or
demultiplexed */
```

```
    etmv1_tracemode_t tracemode;    /* type of information the trace contains (data, addres, contextID, ...) */
    armv4_5_state_t core_state;     /* current core state (ARM, Thumb, Jazelle) */
    image_t *image;                 /* source for target opcodes */
    u32 pipe_index;                 /* current trace cycle */
    u32 data_index;                 /* cycle holding next data packet */
    int data_half;                  /* port half on a 16 bit port */
    u32 current_pc;                 /* current program counter */
    u32 pc_ok;                      /* full PC has been acquired */
    u32 last_branch;                /* last branch address output */
    u32 last_branch_reason;         /* branch reason code for the last branch encountered */
    u32 last_ptr;                   /* address of the last data access */
    u32 ptr_ok;                     /* whether last_ptr is valid */
    u32 context_id;                 /* context ID of the code being traced */
    u32 last_instruction;           /* index of last instruction executed (to calculate cycle timings) */
} etm_context_t;
```

This ETM context is used to hold all information about an ETM in a per-target structure, including the current state of the trace analysis to reduce the number of parameters that need to be passed around during trace analysis.

The ETM code only implements functionality common to all ETM solutions without support for a particular TPA. Similar to the JTAG interface, target and flash support already available in the OpenOCD a capture driver model was created to allow the generic ETM code to be used with different TPAs by simply implementing the `struct etm_capture_driver_s` interface.

Listing 8.3: ./src/target/etm.h

```
typedef struct etm_capture_driver_s
{
    char *name;
    int (*register_commands)(struct command_context_s *cmd_ctx);
    int (*init)(struct etm_context_s *etm_ctx);
    trace_status_t (*status)(struct etm_context_s *etm_ctx);
    int (*read_trace)(struct etm_context_s *etm_ctx);
    int (*start_capture)(struct etm_context_s *etm_ctx);
    int (*stop_capture)(struct etm_context_s *etm_ctx);
} etm_capture_driver_t;
```

The `name` field is used to reference an ETM capture driver, the `register_commands()` function registers driver specific configuration and user commands, and `init()` is called to initialize the capture interface. The `status()` function should return one of the `trace_status_t` states and is also responsible for updating the `etm_context_t->capture_status` field. `read_trace()` is called when the captured trace data is required, for example when the trace is about to be analyzed or when the trace should be saved to a dump file. It is the capture driver's task to allocate the `etm_context_t->trace_data` array, fill it with the captured frames, and report the number of valid frames in the `etm_context_t->trace_depth` member variable.

In order to start the trace capture the ETM code calls the `start_capture()` function which should check if the currently selected ETM port mode is supported by the capture driver and then program the capture device to enable trace capture. A `stop_capture()` function is used to end a trace run and update the trace status but doesn't necessarily have to retrieve the collected information. The `read_trace()` function is provided specifically for this purpose to avoid reading trace data that isn't going to be used, for example because the user decided to immediately start another trace run.

Listing 8.4: ./src/target/etm.h

```c
enum
{
    ETMV1_TRACESYNC_CYCLE = 0x1,
    ETMV1_TRIGGER_CYCLE = 0x2,
};

typedef struct etmv1_trace_data_s
{
    u8 pipestat;    /* bits 0-2 pipeline status */
    u16 packet;     /* packet data (4, 8 or 16 bit) */
    int flags;      /* ETMV1_TRACESYNC_CYCLE, ETMV1_TRIGGER_CYCLE */
} etmv1_trace_data_t;
```

The trace data is stored in an array of type `etmv1_trace_data_t` to allow preprocessing of the collected trace cycles. In case of a trigger cycle the actual PIPESTAT is copied from `packet` to the `pipestat` field, and the TRACESYNC signal is stored in a `flags` field which is also used to mark the cycle in which the trigger was observed. This preprocessing allows trigger cycles to be handled just like any other cycle while the marker ensures that the packet associated with a trigger cycle isn't used (no valid FIFO data was output in a trigger cycle).
Two separate indexes are maintained for `pipestat` and `packet`, with an extra flag to index the first or second half of a 16 bit port.

The major part of ETM analysis functionality is implemented in the `etmv1_analyze_trace` function. It iterates through the captured trace cycles up to `etm_context_t->trace_depth`, examining the `pipestat` field until a branch (*BE* or *BD*) is encountered. `etmv1_branch_address()` extracts the target address of a branch using the APO and TRACESYNC flags to synchronize the `data_index` with the current `pipe_index`. The `etmv1_next_packet()` function is used to read the next packet from the `trace_data` array according to the rules described in section 8.5.

At the beginning of the trace analysis `etm_context_t->pc_ok` is initialized to zero, indicating that no valid PC has been acquired yet. `etmv1_branch_address()` sets this field to one once a full address output with five address packets has been observed.
As soon as the full address of the current instruction is known the ETM analysis code reads the instructions that executed from the `etm_context_t->image`, disassembles it using the `arm_evaluate_opcode()` and `thumb_evaluate_opcode` functions provided by the OpenOCD disassembler[8], and outputs the instruction stream to the current command context (telnet or GDB session).

[8]See ./src/target/arm_disassembler.c for the disassembler source.

8.7 OpenOCD ETM Trace Usage

In order to use an ETM the OpenOCD's configuration file has to specify several properties of the ETM, like which target it is connected to, the port width and mode, and which capture driver to use.

```
#etm config <target> <port_width> <port_mode> <capture_driver>
etm config 0 16 normal full etb
```

The above configuration enables an ETM connected to target #0 using a 16 bit wide trace port operating in normal mode with full speed clocking. An embedded trace buffer (ETB) is connected to the ETM and shall be used to read the collected trace data.

If there is an ETM configured the `etm` commands are registered and can be used to set up a trace, start the trace, and to analyze trace.

```
                 etm    Embedded Trace Macrocell
         tracemode -    configure trace mode <none|data|address|all>
                        <context id bits> <cycle accurate> <branch output>
              info -    display info about the current target's ETM
trigger_percent <percent> -     amount (<percent>) of trace buffer to be filled
                        after the trigger occurred
            status -    display current target's ETM status
             start -    start ETM trace collection
              stop -    stop ETM trace collection
           analyze -    analyze collected ETM trace
             image -    load image from <file> [base address]
              dump -    dump captured trace data <file>
              load -    load trace data for analysis <file>
```

The `tracemode` command allows the user to configure if data, instructions or both should be traced, the amount of context id bits that should be recorded, whether the trace is to be cycle accurate, and if the address of all branches, including direct branches, should be traced. Using the `trigger_percent` command, the amount of trace buffer space that should be used after a trigger occurred may be configured (default is 50 to trace an equal amount of code before and after the trigger). Analyzing a trace requires an image of the executed code, but if the target can tolerate being halted the use of a pseudo image using reads from the target memory space is possible, too. Tracing can be started (`start`), stopped (`stop`), and the resulting capture can be dumped to a file (`dump`) for analysis by external tools or for later analysis using the OpenOCD (`load`). The `analyze` command causes the OpenOCD to start analyzing the trace.

Programming the trace control and filtering currently needs to be done manually because the numerous possibilities for combining the various trace resources and events make command line configuration of these items difficult. The ETM registers are accessible via OpenOCD's default register support, allowing for example the trigger event register to be programmed via `reg ETM_TRACE_EN_EVENT 0x6f`.

9 ETB - Embedded Trace Buffer

At core frequencies of 400MHz and more retrieving trace data via an ETM trace port becomes a serious problem because of signal integrity issues. The ETB is an on-chip trace buffer that connects to an ARM7/9 ETM, allowing the trace data to be collected via JTAG without the need for the high-pincount, high-bandwidth ETM trace port.

The ETB connects to the system via a JTAG TAP port of its own, an ETM trace port, and via an AHB slave memory-mapped peripheral. This chapter is going to use the ETM + ETB combination available on the NXP LPC3180 that was already used to capture the example trace data used in chapter 8 to show how the ETB fits into an ARM SoC. Figure 9.1 shows the connections of the ETB inside the LPC3180.

In the LPC3180 the ARM926EJ-S core is connected to the JTAG TDI pin, the ARM's TDO pin is connected to the ETB's TDI, and the ETB's TDO is connected to the JTAG TDO pin. When viewing the whole scan chain as a long shift register with the least significant bit on the right the ETB is therefor the first (rightmost) in the chain, followed by the target. This is also how the JTAG chain layout has to be specified in the OpenOCD configuration file:

```
#jtag_device <IR length> <IR capture> <IR capture mask>
jtag_device 4 0x1 0xf
jtag_device 4 0x1 0xf
```

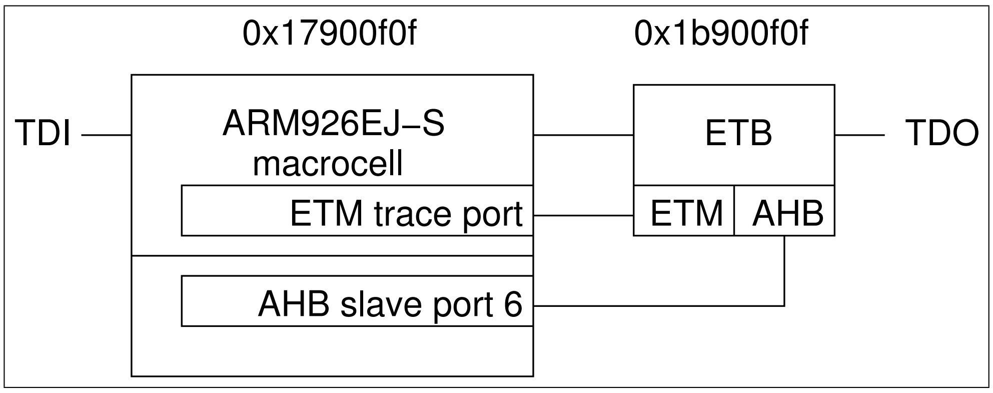


Figure 9.1: ETB Connections Inside The LPC3180

```
#target <type> <endianness> <reset mode> <chain_pos>
target arm926ejs little reset_halt 1

#etb config <target> <chain_pos>
etb config 0 0
```

The ETB's AHB slaved peripheral interface is connected to the LPC3180's AHB bus matrix on slave port 6 [UM10198, §3.1, Fig. 4]. Contrary to what the user's manual says on page 14 the ETB data RAM isn't mapped at address 0x311e0000 but at address 0x310e0000, the ETB's control registers are mapped at 0x310c0000. In order to be able to access the ETB registers the ETM needs to be powered up (*powerdown* bit in the *ETM control register* (see subsection 8.4) needs to be cleared), and the *SoftwareCntl* bit in the *ETB control register* needs to be set[1]. The ETB registers are then accessible using the AHB slave interface only, until the *SoftwareCntl* bit gets cleared again.

9.1 ETB Registers

Table 9.1 shows the ETB registers, their address when accessed via JTAG, the offset when accessed via the AHB slave peripheral, whether the register is read-only or writeable, and a description of the register content.

Table 9.1: ETB Registers

Num (Offset)	Type	Description
0 (0x00)	Read-Only	32 Bit Identification Register (0x1b900f0f)
1 (0x04)	Read-Only	RAM Depth (number of entries in ETB RAM)
2 (0x08)	Read-Only	RAM Width (size of a single ETB entry)
3 (0x0c)	Read-Only	Status Register
4 (—-)	Read-Only	RAM Data
5 (0x14)	Read-Write	RAM Read Pointer
6 (0x18)	Read-Write	RAM Write Pointer
7 (0x1c)	Read-Write	Trigger Counter
8 (0x20)	Read-Write	Control Register

- The status register holds four bits:
  - [0] Full. This bit indicates whether the RAM write pointer wrapped around at least once.
  - [1] Triggered. The triggered is high once the *TR* PIPESTAT has been captured.
  - [2] AcqComp. This bit reads as one when the trigger counter reaches zero, indicating that the trace acquisition completed.
  - [3] DFEmpty. The data formatter pipeline might contain data that still needs to be written to RAM. Waiting for this bit to be set guarantees that all outstanding data has been written to RAM.

[1] The bit is set upon reset but cleared by the first JTAG access to the ETB port. This allows the device to self-test the trace functionality if a core-accessible ETM is connected, too.

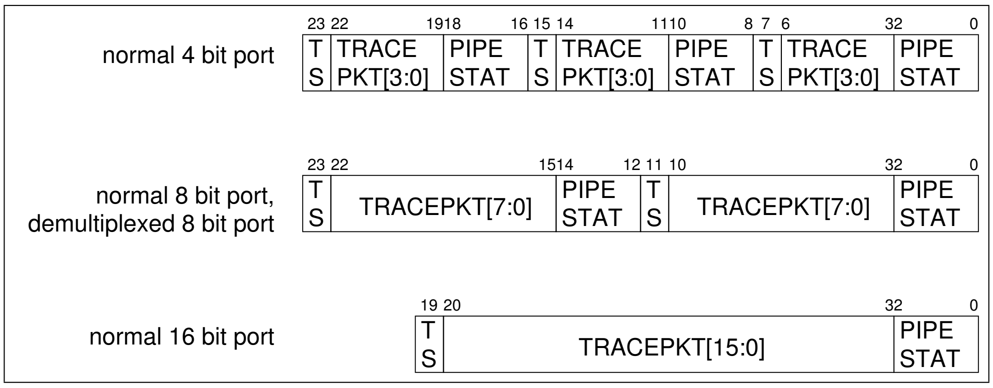


Figure 9.2: ETB RAM Format

- The layout of the control register with three bits width is:

  [0] TraceCaptEn. Setting this bit enables the trace capture. The ETB will write capture data to the RAM as long as the trigger counter is greater than zero (acquisition not finished) and this bit is high.

  [1] Demux. If the ETM operates in demultiplexed mode this bit needs to be set to enable demultiplexed support in the ETB.

  [2] SoftwareCntl. Setting this bit transfers control over the ETB registers to the AHB slave peripheral. It needs to be cleared again to reenable JTAG access to the ETB.

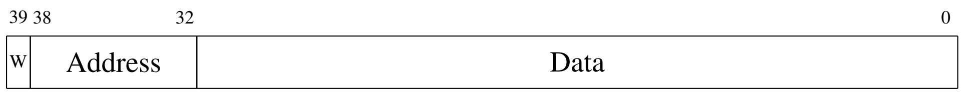
ETB RAM

The ETB RAM is at least 24 bits wide but can be implemented as a full 32 bit memory if it should be made available as a general purpose RAM via the AHB slave when ETB functionality isn't required. When connected to an ETM implementing the ETMv1 architecture the ETB supports the normal trace data format with 4, 8 and 16 bits port width, and the demultiplexed trace data format with an 8 bit port. By selecting different ETM port widths a user may balance trace depth versus the amount of information that's traced. Figure 9.2 shows the ETB RAM layout used for the various supported port modes.

Two pointers are provided to index the ETB RAM, a *RAM Read Pointer* and a *RAM Write Pointer*. The *RAM Read Pointer* is autoincremented on any JTAG access to the ETB RAM, the *RAM Write Pointer* is incremented internally by the ETB trace data formatter whenever the ETB RAM is written.

JTAG Access

The ETB registers can be programmed via the ETB TAP controller using scan chain 0 in way similar to the EmbeddedICE or ETM registers that are accessible via the ARM Core's TAP controller scan chain 2 and 6[DR05, §3.3]. The debugger loads the SCAN_N instruction into the JTAG instruction register, scans the number of the ETB scan chain into the JTAG data register, and selects the INTEST JTAG instruction to read and write the ETB registers:



Writing the ETB registers is achieved by scanning the new *Data* value, the register's *Address* and the *W* bit set to 1 into the JTAG data register and moving the TAP statemachine through Update-DR. In order to read a register two consecutive accesses are required, one that programs the desired *Address* and which sets the *W* bit to 0, and another one that scans out the requested data after moving through Update-DR and back to Shift-DR to execute the register read. Just like the EmbeddedICE DCC data register the ETB data register needs to be handled with care, because every access to it causes the ETB RAM read or write pointer to increment. The other registers can be read multiple times without negative sideeffects so one should set the address field for example to zero (*ID register*) whenever there is no further register read required. Reading the ETB RAM can be accelerated by keeping the *W* bit low and the address set to 4, requesting a new ETB RAM read everytime the previous value is scanned out, reducing the number of accesses to $N+1$ in order to read N words instead of $N*2$ accesses.

9.2 Trace Capture

In order to program the ETB for trace capture the *RAM Write Pointer* needs to be initialized with a known start value, usually 0x0, and the *Control Register*'s *TraceCaptEn* bit has to be written as one. Once tracing started the *ETB Status Register* can be polled to determine whether the trace triggered already and when trace acquisition completed.

Once the *AcqComp* and *DFEmpty* bits are high the *TraceCaptEn* bit should be cleared again to allow the collected trace data to be retrieved. The first step necessary is determining what the oldest entry (the first to be read) from the trace buffer is and how many entries were captured. If the *Full* bit is clear, indicating that the ETB RAM didn't overflow, the oldest frame is at index 0x0 and the *RAM Write Pointer* holds the number of valid entries in the trace buffer. If the ETB RAM overflowed the oldest entry is the one at the *RAM Write Pointer* (the one that would have been overwritten next), and the trace buffer's depth is the number of valid entries.

A debugger then has to read the ETB data RAM, either via JTAG or by reading from the memory-mapped peripheral, but JTAG usually means less overhead. When reading via JTAG the *RAM Read Pointer* has to be initialized with the index of the oldest trace entry. The debugger can simply read the number of entries needed while the ETB logic automatically increments the *RAM Write Pointer*, wrapping back to 0x0 in case the trace overflowed. Depending on the trace data format an ETB entry contains one (16 bit port), two (8 bit port) or three (4 bit port) trace cycles from the ETM.

9.3 OpenOCD Integration

ETB support is implemented in `./src/target/etb.c` using declarations from `./src/target/etb.h`. The `etb_t` structure holds information about the ETB on a per target basis and is accessible via the driver specific `etm_context_t->capture_driver_priv` pointer.

A global variable of type `etm_capture_driver_t` called `etb_capture_driver` holds pointers to the ETB functions and implements the ETM capture driver interface defined for the OpenOCD.

The ETB driver requires one configuration statement specifying the target the ETB (and thus the ETM) is connected to and its position in the JTAG scan chain:

```
#etb config <target> <chain_pos>
etb config 0 0
```

There are no user acessible commands registered by the ETB, because all provided functionality is accessed implicitly by ETM commands.

10 OpenOCD+trace

The OpenOCD+trace was created as an example implementation of an ETM trace capture device that allows analysis of trace data from ARM7 and ARM9 based targets. It is built using an existing FPGA development board with a Xilinx Virtex-2 FPGA and enough SDRAM to allow capture of large trace runs. The current implementation is limited to 4-bit wide trace ports like they are used on ARM7 and small ARM9 targets like the NXP LPC2000 series or ST Microelectronics' STR91x series. Capture is limited to medium frequencies around 50MHz to allow an undedicated hardware platform to be used. Trace capture at higher frequencies would have required high-frequency aware board layout and special connectors, but the basic design principles should be transferable to such applications as well.

The design is limited to analysing the captured trace data offline, as a design with support for on the fly analysis would require a very high speed connection to the host PC which wasn't available on the chosen FPGA development board. On the fly analysis would also double the buffer memory bandwidth requirements, making the use of either double data rate (DDR) memory or a twice as wide memory bus necessary.

The Hitex STR912 Evalboard was selected as the testing platform because it provided easy access to the ETM signals. A simple breadboard connects the ETM signals coming from the STR912 board via a 20 pin ribbon cable to the FPGA board which plugs directly into the breadboard via two 50 pin connectors.

10.1 Requirements

A STR912 device running at 48MHz generates a considerable amount of data that needs to be stored in the trace buffer memory: $8\frac{bit}{cycle} * 48 * 10^6\frac{cycle}{s} = 384{,}000{,}000\frac{bit}{s} \approx 46\frac{MB}{s}$. The maximum frequency of a STR912 is 96MHz, generating about $92\frac{MB}{s}$. A target like the AT91RM9200 with its 16 bit trace port running at 180MHz requires even more transfer bandwidth to the buffer memory: $20\frac{bit}{cycle} * 180 * 10^6\frac{cycle}{s} = 3{,}600{,}000{,}000\frac{bit}{s} \approx 430\frac{MB}{s}$. The amount of buffer space necessary depends on the desired trace depth. One megabyte of RAM is enough to store over a million of cycles from a four bit trace port or about 400,000 cycles from a 16 bit trace port.

The time available for processing a trace cycle is determined by the clock cycle length. At 48MHz a trace clock cycle is $20{,}8\overline{3}ns$ long, at 96MHz it's $10{,}41\overline{6}ns$, and at 180MHz it is only $5{,}\overline{5}ns$. If the trace buffer memory is wider than the trace port multiple trace cycles can be merged into a single memory access, allowing more time for the memory accesses.

10.2 Hardware

The Virtex-2 development board used is from a small series production and not available commercially. It features a Xilinx Virtex-2 XC2V250-5FG256 FPGA, a FTDI FT245BM USB interface chip for communication with a host PC, 16MB of Micron MT48LC8M16A2-7E SDRAM for data storage, and a EA DIP204-4 LCD module for displaying status information. A 4MB Spansion AM29LV320MB flash memory is available to store FPGA configurations and a Xilinx XC9572XL-VQ64 CPLD is used to control the configuration process. Four debounced buttons are available for user interaction with the device.

Xilinx Virtex-2 XC2V250 FPGA

The FPGA is one of the smaller members of the Xilinx Virtex-2 family of devices which has the benefit of being supported by the free[1] Xilinx ISE WebPack edition whereas larger variants like the XC2V1000 require the use of the Xilinx ISE Foundation edition that isn't available for free. The device contains the equivalent of 250,000 system gates[2], 432Kb block RAM, eight digital clock managers and supports frequencies of up to 420MHz.

Configuration

On power up or after a reset the FPGA is configured in Master SelectMAP mode via the CPLD and flash memory, but a JTAG interface is available as well for runtime configuration without having to rewrite the flash. Xilinx FPGAs can be configured via JTAG using the Xilinx Impact utility and a simple parallel port interface but the driver necessary for working with the parallel port interface under Linux were not functional on the chosen development platform. The OpenOCD was extended with support for loading Xilinx bitstream files into the Virtex-2. The file format for Xilinx `.bit` files is undocumented but information available on the internet revealed that it consists of a preamble and several sections describing the file content. A section is made up of a single ASCII character designating the section and a length field followed by the section data.

- 13 bytes preamble (00 09 0f f0 0f f0 0f f0 0f f0 00 00 01)
- Section 'a', the source file name: Contains an ASCII string with the name of the source file used to generate this bitstream.
- Section 'b', device name: String holding the name of the target device.
- Section 'c', creation date: String with the creation date of the bitstream.
- Section 'd', creation time: String with the creation time of the bitstream.
- Section 'e', bitstream: The bitstream length is encoded in 4 bytes, followed by the actual bitstream data.

[1]Xilinx ISE WebPack can be downloaded for free from `http://www.xilinx.com`.

[2]FPGA logic density measurements are often marketing driven and can't necessarily be compared among different vendors [FPGA01].

Xilinx bitstreams should be shifted in MSB first which is why all bytes need to be swapped (bit 0->7, 1->6, ...) before being sent to the device via the OpenOCD's JTAG subsystem that shifts out data LSB first.

Programming the device via JTAG is achieved by first selecting the JPROG_B JTAG instruction that has the same effect as manually taking the *PROG_B* signal low, causing the FPGA configuration to reset and cleaning the configuration memory to be able to load a new configuration. After waiting some time to allow configuration memory to be cleared the CFG_IN instruction is selected and the bitstream is scanned into the FPGA in one go. The FPGA is then started by resetting the TAP via a sequence of at least five TCK cycles with TMS held high (move to Test-Logic-Reset state) and selecting the JSTART instruction followed by at least 12 TCK cycles in Run-Test/Idle state.

FTDI FT245BM

The FTDI FT245BM [FTDI01] is an USB interface chip with a FIFO connected to an eight bit bus interface. The device supports USB1.1 and USB2.0 full-speed ($12 * 10^6$b/s) hosts and has a 128 byte FIFO receive buffer (host->device) and a 384 byte FIFO transmit buffer (device->host). An EEPROM interface allows personalization of the USB VID/PID, serial number and product description.

On the host the FT245BM is accessible either via FTDI's own FTD2XX library, libftdi as a GPLed replacement for FTD2XX, or as a serial interface using the ftdi_sio module on Linux or the virtual COM port (VCP) drivers on Windows.

The FIFO interface uses the signals listed in table 10.1 to interface with a microcontroller, FPGA or some other device on the target using an asynchronous host bus.

Table 10.1: FT245 FIFO Interface Signals

Name	Direction	Description
D[7:0]	IN/OUT	FIFO Data Bus
nRD	IN	Read Strobe (active low)
WR	IN	Write Strobe
nTXE	OUT	Transmit FIFO Empty (active low)
nRXF	OUT	Receive FIFO Full (active low)
SI/WU	IN	Send Immediate / Wake-Up

Figures 10.1 and 10.2 show the timing of FIFO bus read and write cycles. Read cycles on the FT245's FIFO bus can only start when the nRXF signal is low (active), indicating that at least one word is available in the receive FIFO. After taking the nRD line low it takes between 20ns and 50ns (T3) until D[7:0] holds valid data. The read strobe has to stay low for at least 50ns (T1) before it has to remain inactive for a minimum of 50ns (T2) plus the time it takes for nRXF to go low again. The output on D[7:0] is held valid for 0ns (T4) and can thus not be relied upon after taking nRD high again. Up to 25ns (T5) can pass before nRXF goes inactive to indicate processing of the next word, so it's necessary to wait at least that long before polling nRXF to see if new data is available. Once high nRXF will remain

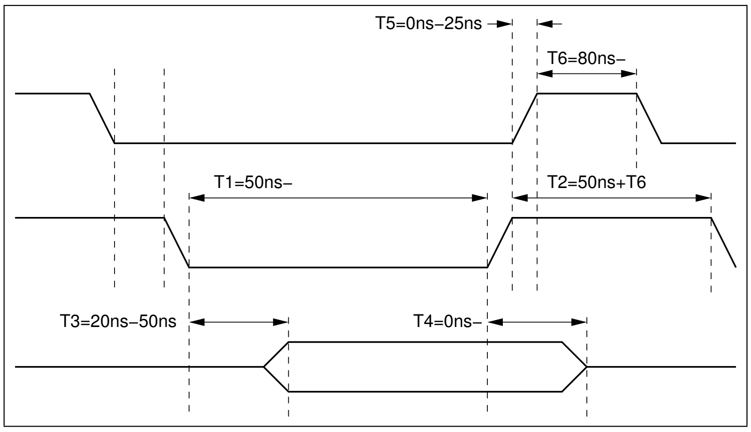


Figure 10.1: FT245 Read Cycle

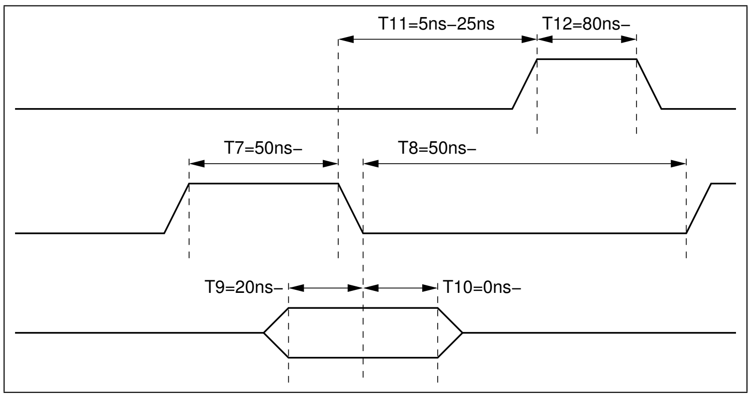


Figure 10.2: FT245 Write Cycle

inactive for 80ns or more (T6).

In order to start a write cycle the nTXE line has to be checked to see if there is room in the FIFO to accept a new byte. The WR line can then be taken low for a minimum of 50ns (T7), and must remain inactive after that again for at least another 50ns (T8). The minimum setup time of D[7:0] when writing is 20ns (T9), a hold time from the falling edge of WR is not necessary (T10). It may take between 5ns and 25ns (T11) for nTXE to go high, indicating that the current byte is being processed, and the signal wont go low for at least 80ns (T12), inhibiting any further writes during that time.

Micron MT48LC8M16A2-7E

The Micron MT48LC8M16A2-7E [MICRON01] is a 16 bit wide single data rate (SDR) SDRAM that allows operation at up to 143MHz with a CAS latency of 3 or 133MHz with a CAS latency of 2. The memory is organized as 4 banks of 2M x 16 bit for a total of 128Mb. Each bank consists of 4096 rows by 512 columns (see figure 10.3). The signals used to interface the SDRAM with a microcontroller or

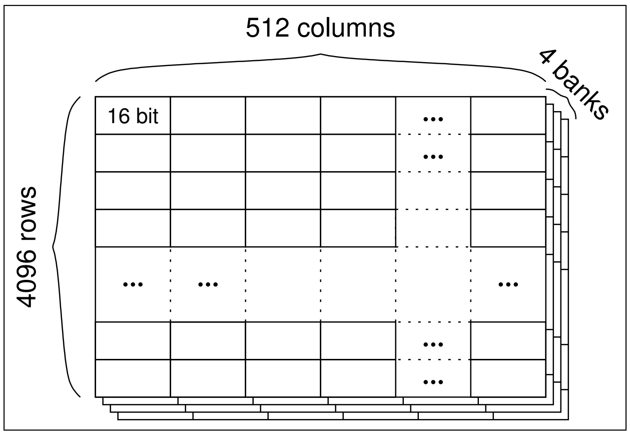


Figure 10.3: SDRAM Organization

FPGA are listed in table 10.2. The memory support bursts of one (i.e. no burst), two, four or eight words per access. It requires 4096 refresh cycles per 64ms or one refresh cycle every 15.625us.

Table 10.2: SDRAM Signals

Name	Direction	Description
DQ[15:0]	IN/OUT	Data
A[11:0]	IN	Address
BA[1:0]	IN	Bank Address
DQML, DQMH	IN	Input/Output Mask
nWE, nCAS, nRAS	IN	Command Inputs (active low)
nCS	IN	Chip Select
CKE	IN	Clock Enable
CLK	IN	Clock

A SDRAM is a Synchronous Dynamic RAM - all signals are synchronous to a clock signal CLK, and the memory is built using capacitors to store the information rather than with transistors used in SRAM (Static RAM). The synchronous interface makes the SDRAM an ideal choice for use in an FPGA based system where the whole design usually operates synchronously. The dynamic nature of SDRAM makes it necessary to refresh the memory at regular intervals to prevent the memory cells from loosing their charge.

While SDRAM still uses the traditionally named control signals nWE, nCAS, nRAS, and nCS, SDRAM is usually controlled via commands that translate to a certain pattern on the control lines (see 10.3. During start up a predefined sequence of commands needs to be applied in order to set up the memory for correct operation:

- After power has been applied and the clock stabilized 100us need to be spent with either NOP or COMMAND INHIBT.

Table 10.3: SDRAM Commands

SDRAM Command	nCS	nRAS	nCAS	nWE	DQM	Address	DQ
COMMAND INHIBIT	H	-	-	-		-	-
NOP	L	H	H	H	-	-	-
ACTIVE	L	L	H	H	-	Bank & Row	-
READ	L	H	L	H	MSK	Bank & Column	-
WRITE	L	H	L	L	MSK	Bank & Column	Data
BURST TERMINATE	L	H	H	L	-	-	-
PRECHARGE	L	L	H	L	-	Code	-
AUTO REFRESH	L	L	L	H	-	-	-
LOAD MODE REGISTER	L	L	L	L	-	Op-Code	-

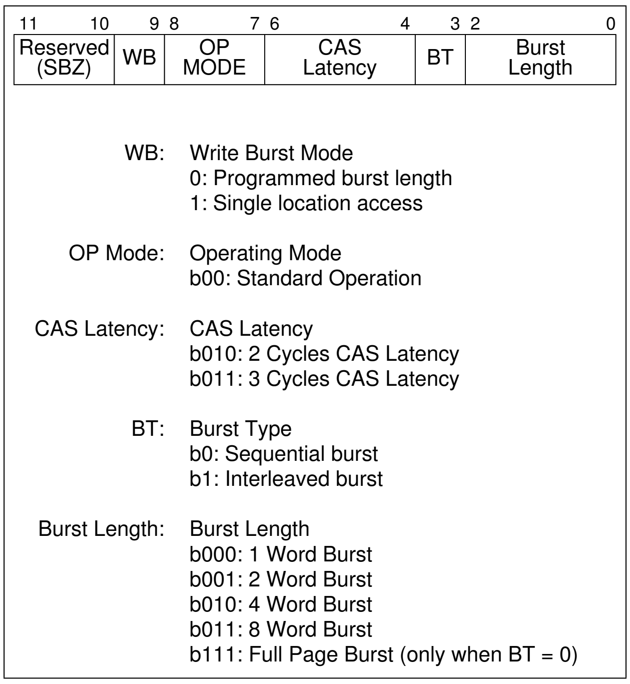


Figure 10.4: SDRAM Mode Register

- Following the start up period a PRECHARGE command should be applied with line A10 high to precharge all banks, placing them in idle state.
- Once in idle state two AUTO REFRESH cycles should be executed.
- The LOAD MODE REGISTER must be used to configure the memory. A bitpattern according to the *SDRAM Mode Register* layout given in figure 10.4 must be placed on the address lines *A[11:0]*. This is a necessary step because the mode register content is unknown out of reset, i.e. there are no defined defaults.
- The SDRAM is now operational.

Because of its dynamic nature SDRAM needs to be refreshed at regular intervals to prevent information loss in the memory cell capacitors. When idle the memory can be put in self-refresh mode to save power, but during operation 4096 AUTO REFRESH cycles are required per 64ms, or one evenly distributed AUTO REFRESH cycle every 15.625us. In order to perform the refresh cycle all four banks have to be placed in idle state by applying a PRECHARGE command to every open bank (or one PRECHARGE ALL command by keeping A10 high during the PRECHARGE command). A minimum delay of t_{RP} (PRECHARGE command period) has to elapse before the AUTO REFRESH command may be used.

The MT48LC8M16A2-7E features four banks that may each have an open row (also called page). Rows are opened by applying the ACTIVE command together with the desired bank (BA[1:0]) and row address (A[11:0]). After the delay specified as t_{RCD} (ACTIVE to READ or WRITE delay) the newly opened line can be read or written, but it must be closed within the limits specified by t_{RAS} (ACTIVE to PRECHARGE command). A row on bank b can be opened while bank a is still being accessed, as long as at least t_{RRD} (ACTIVE bank a to ACTIVE bank b command) elapsed between to consecutive ACTIVE commands.

An open row can be read by issuing the READ command and the requested column (A[8:0]). The configured CAS latency is the number of cycles that pass between issuing the READ command and the data being available on DQ[15:0]. At a CAS latency of two there is a single cycle delay and starting with the second cycle after the READ command the requested data can be read.

Writing happens with zero cycle delay which means that the address (BA[1:0], A[8:0]) needs to be applied together with the data (DQ[15:0]) at the same time as the WRITE command. If the *WB* bit is set to one in the *SDRAM Mode Register* memory can be written via single accesses, if it is set to zero the device expects data for the programmed number of cycles (*Burst Length*), but a burst may be truncated at any time via a PRECHARGE, READ or another WRITE command.

The DQM[HL] signals allow the two byte lanes to be masked, DQML is used with DQ[7:0], DQMH affects DQ[15:8]. During a memory read DQM operates with a two cycle latency, allowing the DQ output buffers to be placed in high-impedance state two cycles after the corresponding DQM was high. For write operations DQM operates with no latency, masking the data from being read into the input buffer to preserve the previous content of the byte lane.

EA DIP204-4 LCD Module

The EA DIP204-4 [EADIP204] is an alphanumeric LCD module with four lines and twenty characters per line. The LCD uses a Samsung KS0073 controller chip that supports a four or eight bit MCU data bus and a SPI interface, but on the XC2V250 FPGA development board the LCD module is configured to operate in SPI mode only. Table 10.4 lists the signals used to connect the display in SPI mode.

The display uses 5V levels on its SPI interface but the Virtex-2 FPGA operates at voltages of 3.3V or lower (see I/O standards in [XILDS031]), making level conversion necessary. A HEX inverting Schmitt trigger (74HC14) powered with 5V is used to raise the voltage on signals coming from the FPGA and a 270 Ohm series resistor limits the current on the 5V coming from the LCD on the SDO line. A side effect of this is that all signals coming from the FPGA are inverted, making it necessary for the FPGA to invert the signals itself before putting them on output pins.

Table 10.4: LCD Signals

Name	Direction	Description
nCS	IN	Chip Select (active low)
nRST	IN	Reset (active low)
SID	IN	Serial data input
SOD	OUT	Serial data output
SCLK	IN	Serial clock

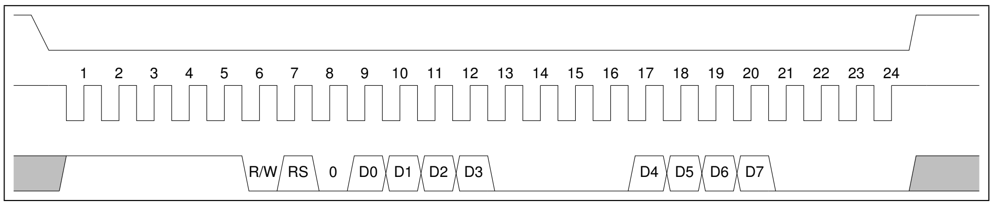


Figure 10.5: LCD Write Cycle

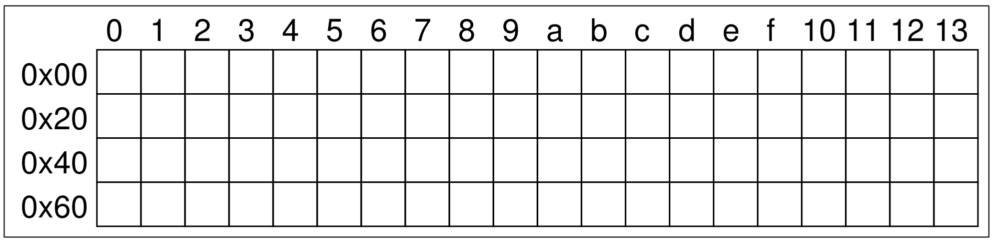


Figure 10.6: LCD Memory Layout

The serial interface supports both read and write operations, but for the purposes of outputting data on the display there is usually no need for reading data back. The timing when writing eight bits of data is shown in figure 10.5. At the beginning of a write a starting byte has to be sent, consisting of five subsequent ones, the read/$\overline{write}$ bit set to zero (write), the value of the *RS* bit, and a zero. After transferring four bits of data four subsequent zeros must be inserted to guarantee a safe data transfer, followed by the next four bits of data and another four zeros.

The display controller's data memory (DDRAM) is 80 bytes long (4x20 characters) with the addressing shown in figure 10.6.

Data sent to the controller has eight bits per byte, but an additional *RS* bit designates the input as an instruction (RS = 0) or as data (RS = 1). The display controller needs to be initialized for the connected display module and to configure options such as cursor blinking and data entry mode. An internal bit, the extension bit *RE*, is used to access additional registers. This bit can be set and cleared by the *Function set* command.

The display is configured to output data sent with the *RS* bit set to 1 on the display in increasing

Table 10.5: LCD Initialization Commands

Command	Description
0x34	Function set. Configures the controller for a 8 bit data bus (ignored with serial interface), one-line display (later overwritten for four-line display), sets the *RE* and chooses normal (non-inverted) display mode.
0x09	Extended function set (requires RE bit set). Configures the controller for a 5-dot display, normal cursor and four-line mode.
0x30	Function set. Clears the *RE* bit again.
0x0F	Display on/off control. Turns the display, cursor and cursor-blinking on.
0x01	Clear display. Clears the DDRAM and returns the cursor to home (address 0x00).
0x06	Entry mode select. Configures data entry to increase the cursor and address and disables shifting.

addresses. If data should be output on an address other than the one immediately following, the DDRAM address can be set using an instruction ($RS = 0$) with bit 7 high and the address in bits 6 to 0:

7	6	0
1	Address	

Commands take between 39us and 1.53ms (only "Clear Display" and "Cursor Home") during which the controller is busy internally and wont accept new data. Data input requires 43us internal processing time. The display supports serial communication with a minimum of 0.5us SCLK cycle time at which a single write can complete in 12us. A busy flag can be read to determine when the last operation finished, but simply waiting long enough for internal operations to complete is fine.

10.3 FPGA - Field Programmable Gate Array

An FPGA is a programmable logic device used to implement combinatorial and synchronous logic designs. The smallest element[3] usually consists of a D-type flip-flop and a look-up table (LUT) that can be used to implement every possible boolean function with up to a certain number of inputs[4]. These logic elements are placed in a regular array, often enhanced with additional functionality such as clock managers, RAM resources and more dedicated functionality like multipliers or even complete DSP blocks. Routing resources provide local interconnects between logic elements and global connections for signals like clocks that are driving a large number of inputs (i.e. they have a high "fan-out") using dedicated routing lines. The configuration of all logic and routing resources is controlled using static memory cells.

[3]FPGA architectures vary among different vendors. Altera for example calls their smallest design entity "logic element", Xilinx calls it a "logic cell", but the basic concepts apply to both.

[4]Typically 4-input LUTs are used, but newer devices like the Virtex-5 series move towards 6-input LUTs.

Workflow

The programming file for an FPGA is a bitstream that's loaded into the FPGA's static memory cells. The workflow used to create the bitstream using electronic design automation (EDA) tools consists of the following steps:

- Design entry.
  The intended design needs to be entered into the EDA tool, either as a schematic or using a hardware description language such as VHDL or Verilog.
- Synthesis.
  Designs entered in a hardware description language combine gate and netlist descriptions (e.g. signal assignments, combinatorial statements) with functional descriptions. Synthesis translates these functional descriptions to a netlist and optimizes the complete netlist for use with a particular FPGA.
- Mapping.
  The netlist has to be mapped to the resources available on an FPGA. The result of the mapping step is a description of which design elements were mapped to the FPGAs logic elements. If a design uses more resources than available on the FPGA this will be identified during mapping were the device utilization is calculated.
- Place & Route (PAR).
  The mapping associated design elements with FPGA resources. During PAR the elements are placed into a particular location on the FPGA and the required connections are made using the router. Multiple steps of place and route execute until all timing constraints are met.
- Programming file generation.
  A completely placed and routed design contains information about the exact configuration of every FPGA element and the interconnections. This is used together with information about the specific device to generate the bitstream that can be loaded into the FPGA.

VHDL

VHDL, the VHSIC[5] Hardware Description Language is one of the languages used to describe hardware that can be implemented in FPGAs or ASICs. VHDL evolved from a project run by the United States Department of Defense during the 1970s and early 1980s [DLP02, p. 1] and is now an IEEE standard published as IEEE Std 1076. There are several variants of the language like VHLD87 and VHDL93 designated by the year they were standardized. Alternative languages used in FPGA design are for example Verilog, Abel or Handel-C.

This document is going to provide only a short overview about the language, for additional information see [IEEE1046], [IEEE1046.6], or for example [DLP02].

VHDL is a hardware description language, that is it provides a formalized view of hardware, but it supports constructs that can't be translated into a netlist by the synthesis tool. A subset of VHDL defined in [IEEE1046.6] can be used for descriptions that can be implemented in FPGA designs.

Listing 10.1: trace_capture_behavioral.vhd

```
CLKGEN: PROCESS (sys_CLK0)
```

[5]VHSIC is short for Very High Speed Integrated Circuit.

```
BEGIN
    if sys_CLK0 = '0' then
        sys_CLK0 <= '1' after 6.250 ns;
    else
        sys_CLK0 <= '0' after 6.250 ns;
    end if;
END PROCESS;
```

The PROCESS shown in listing 10.1[6] is valid VHDL code describing a clock running at 80MHz with a 50/50 duty cycle[7], but no hardware exists that could be configured to perform the described behaviour. When synthesizing the code from listing 10.2 on the other hand the synthesis tool will correctly identify a D flip-flop clocked by the rising edge of *clock* with *enable* as the clock enable signal.

Listing 10.2: D–Flip–Flop

```
process(clock, enable, data)
begin
if (enabled and clock'event and clock = '1') then
    output <= data;
end if;
end process
```

A VHDL design's primary design unit is the entity. It specifies the name of the entity, its ports, and other information.

Listing 10.3: VHDL Entity

```
entity entity-name is
    Port (
        outputsignal : out std_logic;
        outputbus : out std_logic_vector (7 downto 0);
        ...
        inputsignal : in std_logic
    );
end trace_capture;
```

An entities body is defined by an architecture that either describes structure, dataflow or behaviour.

Listing 10.4: Architecture

```
architecture Dataflow of entity-name is
    signal internalbus : std_logic_vector (7 downto 0);
begin
    internalbus <= inputsignal & "1110001";
    outputbus <= internalbus;
end Behavioral
```

A structural description combines instantiations of available submodules and describes how they are interconnected. A dataflow describes the data transferred from signal to signal. The behavioral description allows the intended behavior to be defined in terms of concurrent and sequential statements.

[6]Listing 10.1 is part of the testbench used when simulating the OpenOCD+trace design.

[7]The duty cycle describes the relation between a signal's high and low period.

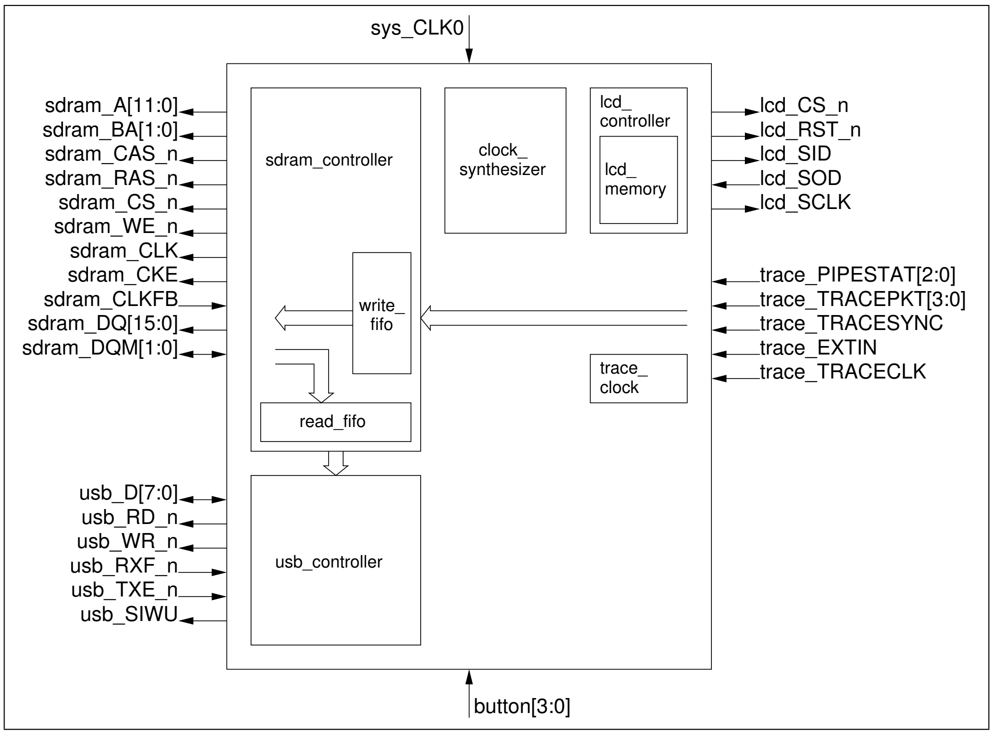


Figure 10.7: OpenOCD+trace trace_capture.vhd

10.4 FPGA Design

The OpenOCD+trace FPGA design is implemented in the top-level design file `trace_capture.vhd`. The top-level design defines the interface to the system (SDRAM, clock, USB, ETM trace port, LCD module, and user buttons), includes submodules that implement reusable components like the interface to the SDRAM and USB controllers, and controls the overall behavior. All components except for the FIFOs used to smooth out SDRAM accesses and the LCD string memory were written from scratch in VHDL. The FIFOs and LCD string memory were generated using the Xilinx Core Generator. Figure 10.7 shows the interface to the top level entity *trace_capture* and the internal components.

User Interface

The interface to the OpenOCD+trace design is realized via a few commands and a set of registers accessible via USB. Figure 10.8 shows the layout of the commands available. Bits 7 to 4 designate the command, bits 3 to 0 allow up to four bits of data payload per command. Additional data can be sent and received using more USB transfers.

The NOP command 0x0 is implemented to be able to flush the FT245BM FIFO until a defined state is reached in which no further data bytes are expected and the OpenOCD+trace is ready to accept a new command. When a register is to be read or written the register's number has to be placed in bits 2 to 0, allowing up to 8 registers to be specified, and bit 3 describes the direction of the access ($\overline{r}/w = 0$ is a

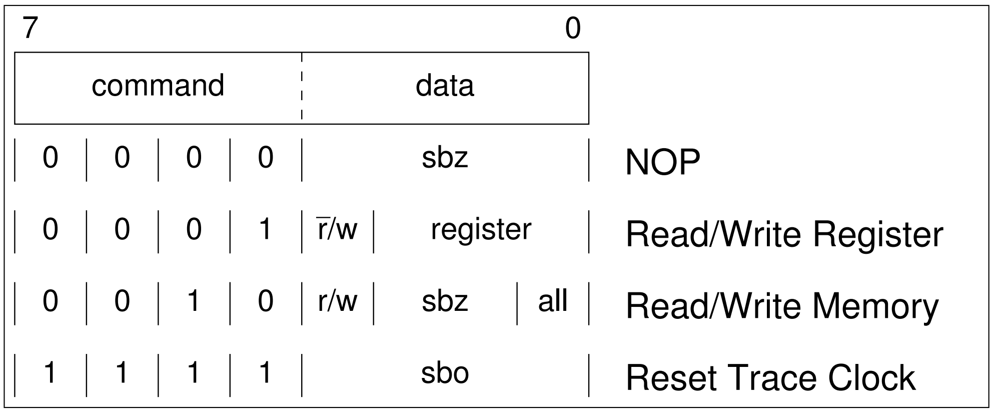

7
0
command
data
0 0 0 0 sbz NOP
0 0 0 1 r̄/w register Read/Write Register
0 0 1 0 r/w sbz all Read/Write Memory
1 1 1 1 sbo Reset Trace Clock

Figure 10.8: OpenOCD+trace Command Definitions

read, 1 is a write). The Read/Write SDRAM command operates on either all of the SDRAM memory, implicitly reloading the SDRAM counter with its maximum and the address with 0x0, or on the range selected by the current address and counter setting. The Trace Clock Reset command is provided to reset the DCM used to generate the internal trace clock, for example when the target clock frequency changed because the PLL got enabled or disabled. Table 10.6 shows the list of implemented registers.

Table 10.6: OpenOCD+trace Registers

Num	Size	Description
b000	20 bit	SDRAM Address Register
b001	21 bit	Trigger Counter Register
b010	2 bit	Control Register
b011	4 bit	Status Register
b100	21 bit	SDRAM Counter Register
b111	32 bit	Identification Register

The control register consists of an *Enabled* bit ([0]) and the *half-rate* bit ([1]) that doubles the ETM TRACECLK clock frequency before using it as `trace_clk` to capture data.

The status register has a *Complete* bit ([0]) indicating completion of the trace capture run (trigger counter reached zero), a *Triggered* bit ([1]) (*TR* cycle observed), a *Full* bit ([2]) (the SDRAM overflowed), and a *Clock Good* bit ([3]) indicating that the ETM TRACECLK quality allowed the DCM to lock.

Clock Domains

The input clock on sys_CLK0 is fed into the clock_synthesizer module that uses a DCM (digital clock manager) to generate the 100MHz system clock (sys_MAINCLK) out of the 80MHz crystal connected to the FPGA. The system clock is used to drive the USB controller, the LCD interface and the SDRAM memory.

The ETM trace clock trace_TRACECLK is fed through a DCM to provide duty cycle corrected versions of the original clock and a clock running at twice the original frequency. The resulting trace_clk is used to clock in data coming from the ETM port, format the data, and to control the trigger logic.

Because there is no relationship between the two clock domains used in the design they have to be treated as being asynchronous, requiring every signal to be synchronized when passing from one domain to another. A 16KB write FIFO implemented with a Xilinx CoreGen generated asynchronous FIFO is used to send captured trace data from the trace_clk domain to the sys_MAINCLK domain. The FIFO uses the dual port capabilities of the integrated block RAM resources to implement synchronized access from both clock domains.

Another DCM is used to deskew the external SDRAM clock with regard to the internal sys_MAINCLK. The sdram_CLK signal is routed back to the FPGA on the sdram_CLKFB pin to provide a feedback of the clock signal used to drive the SDRAM. Because the SDCLK signal is synchronous to the sys_MAINCLK signal there is no need for synchronization of signals passing to and from the SDRAM controller.

USB Controller

The USB controller is implemented in `usb_controller.vhd`. Its internal interface consists of the signals READ, READ_VALID, WRITE, WRITE_ACK, BUSY, DATA_IN, and DATA_OUT, all prefixed with usb_ when the component is instantiated in trace_capture.vhd[8].

Listing 10.5: usb_controller.vhd

```vhdl
entity usb_controller is
    Port (
        CLK : in STD_LOGIC; -- Main clock, 100 MHz
        CLK_locked : in STD_LOGIC; -- Main clock DCM locked

        READ : in STD_LOGIC;
        READ_VALID : out STD_LOGIC;
        WRITE : in STD_LOGIC;
        WRITE_ACK : out STD_LOGIC;
        BUSY : out STD_LOGIC;
        DATA_IN : out STD_LOGIC_VECTOR (7 downto 0);
        DATA_OUT : in STD_LOGIC_VECTOR (7 downto 0);

        D : inout STD_LOGIC_VECTOR (7 downto 0);
        RD_n : out STD_LOGIC;
        WR : out STD_LOGIC;
        RXF_n : in STD_LOGIC;
        TXE_n : in STD_LOGIC;
        SIWU : out STD_LOGIC
        );
end usb_controller;
```

[8]That is, the ports from the component are bound to top level entity signals of the same name with an additional prefix.

The READ signal indicates a read request from the upstream logic that will be acknowledged by the USB controller setting BUSY to a logic one once processing the request started. Once the data is available from the FT245 interface chip the READ_VALID is used to indicate completion of the transfer. The upstream logic can then read from DATA_IN and later deasserts the READ signal to complete the request. The BUSY signal will go low again when the USB controller is ready for another access.

The WRITE signal is used to signal a write request. Processing the request is similar to a read, that is after WRITE went high the controller will assert its BUSY signal once it is able to start the transfer, and WRITE_ACK is used to acknowledge successful completion of the transfer. After WRITE is low again the controller will deassert the BUSY signal once it is ready for the next request.

Listing 10.6: usb_controller.vhd

```vhdl
    signal D_output_enable : std_logic;
    signal DATA_IN_register : std_logic_vector (7 downto 0);

    signal RXF_n_reg : std_logic;
    signal TXE_n_reg : std_logic;
    signal RXF_n_reg_s : std_logic;
    signal TXE_n_reg_s : std_logic;

    type BUS_STATE_TYPE is (IDLE, READ_SETUP, DO_READ, READ_IS_VALID,
        READ_DELAY, DO_WRITE, WRITE_HOLD, WRITE_DELAY);

    signal bus_state : BUS_STATE_TYPE := IDLE;

    signal delay : std_logic_vector (2 downto 0);
```

The USB controller is implemented with a state machine whose current state is stored in the signal bus_state of type BUS_STATE_TYPE, an enumeration of the possible USB controller states. A delay counter is used to ensure timing requirements of the FT245BM interface chip are observed. Because the USB interface data bus is a bidirectional tri-state bus a `D_output_enable` signal is used to switch between input (`D` in high-Z mode) and output mode.

Listing 10.7: usb_controller.vhd

```vhdl
    -- usb_D is a tri-state bus
    DATA_IN <= DATA_IN_register;
    D <= DATA_OUT when D_output_enable = '1' else (others => 'Z');

    BUSY <= '1' when bus_state /= IDLE else '0';
```

The BUSY signal is a combinatorial output generated from the current bus state, and will be low only when the controller is in IDLE state, all other states assign a high level to the signal.

Listing 10.8: usb_controller.vhd

```vhdl
        elsif CLK'event and CLK = '1' then

            -- register asynchronous FIFO state signals
            RXF_n_reg_s <= RXF_n;
            TXE_n_reg_s <= TXE_n;
```

```
                RXF_n_reg <= RXF_n_reg_s;
                TXE_n_reg <= TXE_n_reg_s;
```

A process clocked by the rising edge of CLK (sys_MAINCLK) is used to control the component's behaviour. While the DCM used to generate the clock didn't lock all signals are assigned safe defaults.

The RXF_n signal used to notify the USB controller that the FT245BM FIFO has new data available and the TXE_n signal that indicates free space in the transmit FIFO are asynchronous signals that need to be registered before being used in a synchronous design. Two flip-flops per signal (*_n_reg_s and *_n_reg) are used to synchronize the incoming signals, causing a two cycle latency of the FT245BM FIFO state signals. This has a negative impact on performance, but fast USB transfers aren't necessary in the OpenOCD+trace design anyway.

During IDLE state the acknowledge signals of the internal interface (READ_VALID, WRITE_ACK) and the external strobe signals (RD_n, WR) are set to inactive state. A new transfer cycle will only be started by moving to READ_SETUP or DO_WRITE states when the FT245BM is able to accept the transfer.

In case of a read the read strobe is set to low, a delay counter is initialized to b110, and the controller moves to READ_SETUP state until the delay counter reaches zero again, giving the FT245BM 60ns to access the data[9]. The data is then registered in DO_READ state, a delay of another 60ns[10] is set, and the controller moves on to READ_IS_VALID state. Here the RD_n signal is deasserted again, READ_VALID is set high to indicate completion of the transfer, and the controller waits for READ to go low and the delay to reach zero. READ_DELAY will be entered after the READ request got deasserted, and the controller will move back to IDLE state after the delay completed.

For a write the write strobe is asserted, the tri-state D_output_enable is set to enable the output buffer, and a delay of 60ns[11] is programmed. The controller moves to DO_WRITE state until the delay counter reaches zero, after which the write strobe is deasserted again and the WRITE_HOLD state is entered for one cycle, after which WRITE_ACK indicates completion of the transfer, another delay of 60ns[12] is set, and the WRITE_DELAY state is entered. In WRITE_DELAY the tri-state output buffer is disabled again and when both the WRITE signal got deasserted and the delay elapsed the state machine moves back to IDLE.

LCD Controller

The controller for the 4x20 character LCD module is implemented in `lcd_controller.vhd`. Its internal interface consists only of the status flags that describe the current trace capture state. Changes on one of those flags cause the LCD controller to clear the display and to output the new status information.

[9]This should take 50ns at most, see T1 in figure 10.1.

[10]Minimum RD_n to RD_n precharge time is 50ns, see T2 in figure 10.1.

[11]WR minimum pulse width is 50ns, see time T7.

[12]WR to WR precharge minimum is 50ns, see time T8.

Listing 10.9: lcd_controller.vhd

```vhdl
entity lcd_controller is
    Port (
        -- clock
        CLK : in std_logic; -- Main clock, 100 MHz
        CLK_locked : in std_logic; -- Main clock DCM locked

        -- internal interface
        enabled : in std_logic;
        completed : in std_logic;
        full : in std_logic;
        triggered : in std_logic;
        trace_clk_good : in std_logic;

        -- external interface
        -- CS_n, RST_n, SID and SCLK are inverted by a Schmitt-Trigger
        -- used for level shifting from 3.3V to 5V
        CS_n : out  std_logic;
        RST_n : out  std_logic;
        SID : out  std_logic;
        SOD : in  std_logic;
        SCLK : out  std_logic
    );
end lcd_controller;
```

The LCD controller uses a Xilinx CoreGen generated single port block RAM (lcd_memory) with nine bit wide entries and a depth of 2048 characters. Again a state machine was used to implement the controller's behaviour. The lcd_state signal of type LCD_STATE_TYPE holds the current state. The input signals are aggregated in a five bit wide vector `status`, and a delayed copy of that vector (`status_delayed`) is used to detect any changes on the status flags. The LCD memory is interfaced using `lcd_address` and `lcd_data`. The LCD supports writes of instructions (RS = 0) and data (RS = 1). To simplify the design of the LCD controller all instruction writes are output using a delay large enough to complete all possible commands (1.53ms), while data writes use a shorter delay that only meets a minimum time of 43us. The type of access currently executing is stored in the `delayed_rs` signal, and an eighteen bit wide counter `lcd_delay` implements the necessary delays. The large counter is necessary because of the huge discrepancy between LCD speed (delays of up to 1.53ms) and the FPGA operating frequency (100MHz, 10ns cycle time). The `init` and `line_done` signals control the LCD output. `int_SCLK` and `int_CS_n` are copies of the respective output signals to allow them to be read back[13].

Listing 10.10: lcd_controller.vhd

```vhdl
component lcd_memory
    port (
        addra: IN std_logic_vector(10 downto 0);
        clka: IN std_logic;
        douta: OUT std_logic_vector(8 downto 0)
    );
```

[13]Output signals of an entity can only be assigned to but not read back.

```
end component;

type LCD_STATE_TYPE is (
    IDLE, LINE1, LINE2, LINE3, LINE4
);
signal lcd_state : LCD_STATE_TYPE := LINE1;
signal status : std_logic_vector(4 downto 0);
signal status_delayed : std_logic_vector(4 downto 0);
signal lcd_address : std_logic_vector (10 downto 0);
signal lcd_data : std_logic_vector (8 downto 0);
signal delayed_rs : std_logic;
signal shift_register : std_logic_vector (23 downto 0);
signal lcd_delay : std_logic_vector (17 downto 0);

signal init : std_logic;
signal line_done : std_logic;   -- current line completed

signal int_SCLK : std_logic; -- internal version of SCLK
signal int_CS_n : std_logic; -- internal version of CS_n
```

The current OpenOCD+trace status is divided upon the four lines of the LCD module. The first line statically displays the text **"OpenOCD+trace"** to indicate that the FPGA configuration was loaded successfully. The second line displays the main state, that is whether the trace is currently **"IDLE"**, **"RUNNING"** or **"COMPLETED"**. The third line optionally contains the flags **"OVERFLOWED"**, **"TRIGGERED"**, or **"OVERFLOWED, TRIGGERED"**. The last line displays the current state of the TRACECLK signal coming from the ETM trace port, either **"TRACECLK"** or **"NO TRACECLK"**.

In IDLE state the LCD controller waits for a change of the status vector, an additional signal `init` is used to trigger the initial output of data on the display. The init signal is deasserted after it was first observed high. The state machine sets `line_done` low and assigns the address of the desired string to the `lcd_address` on the beginning of a new line. It then moves to the state corresponding to the line that's currently output (e.g. from IDLE to LINE1, while the first line is being output) where it stays until the `line_done` signal goes high again. The address for the next string is selected based on the flags relevant to the current line, and `line_done` is taken low again. After outputting the fourth line the state machine moves back to IDLE state.

Listing 10.11: lcd_controller.vhd

```
if line_done = '0' then
    if lcd_delay = "000000000000000000" then
        if shift_register = "000000000000000000000000"
and int_SCLK = '0' then -- start next character
            if lcd_data = "000000000" then
                line_done <= '1';
            else
                shift_register <= not("111110" & lcd_data(8) & '0'
& lcd_data(0) & lcd_data(1) & lcd_data(2) & lcd_data(3) & "0000"
& lcd_data(4) & lcd_data(5) & lcd_data(6) & lcd_data(7) & "0000");
                delayed_rs <= lcd_data(8); -- remember the access type
(RS == 0 means command, 1 means data)
```

```
                lcd_address <= lcd_address + 1;
            end if;
```

Outputting a line is handled outside of the state machine. When `line_done` is low and the delay counter expired the previous write completed and depending on the next byte from the LCD string memory either the line finished (terminating NULL character) and `line_done` is asserted or the `shift_register` is loaded with the data for the next write. The whole shift register content is inverted because of the hex inverting schmitt trigger (see subsection 10.2) and consists of the starting byte with the five subsequent ones used for synchronisation, the nR/W bit set to zero because all accesses are writes, the RS bit, and the data bits padded with zeros for a safe transfer. The type of access is remembered in `delayed_rs`, and the `lcd_address` is incremented.

Listing 10.12: lcd_controller.vhd

```
if delayed_rs = '0' then
    lcd_delay <= "100111000100000000";
-- use max. delay (2.62ms) for all commands
else
    lcd_delay <= "000001001110001000";
-- delay data writes by 50us (5000 cycles of 10ns)
end if;
int_CS_n <= '0';
```

Depending on the type of the previous access a large enough delay is inserted to guarantee that the previous access completed, and the chip select is deasserted (it is an active low signal, but inverted by the buffer).

Listing 10.13: lcd_controller.vhd

```
else -- process current character
     if int_CS_n = '0' then
         int_CS_n <= '1';
         lcd_delay <= "000001001110001000";
     else
         if int_SCLK = '0' then
             SID <= shift_register(23); -- change data on falling edge
             shift_register <= shift_register(22 downto 0) & '0';
-- shift left
             int_SCLK <= '1'; -- falling edge
             lcd_delay <= "000000000001100100"; -- 1us
         else
             int_SCLK <= '0'; -- rising edge
             lcd_delay <= "000000000001100100"; -- 1us
         end if;
     end if;
end if;
```

When a character is to be processed the chip select line is asserted and a delay is inserted. The SCLK line is toggled and data is shifted out on the falling edge of the LCD clock. The SCLK is generated with a 50/50 duty cycle and 2us period length to achieve a 500KHz frequency.

SDRAM Controller

The SDRAM controller is implemented in `sdram_controller.vhd`. Its internal interface consists of an address register and a counter register, both with a load signal, an input, and an output, a read and write command input, a busy signal, and the interfaces to the read and write FIFOs.

Listing 10.14: sdram_controller.vhd

```
entity sdram_controller is
    Port (
        CLK : in STD_LOGIC; -- Main clock, 100 MHz
        CLK_locked : in STD_LOGIC; -- Main clock DCM locked

        load_address : in std_logic;
        in_address : in std_logic_vector (19 downto 0);
        out_address : out std_logic_vector (19 downto 0);
        load_counter : in std_logic;
        in_counter : in std_logic_vector (20 downto 0);
        out_counter : out std_logic_vector (20 downto 0);
        read : in std_logic;
        write : in std_logic;
        busy : out std_logic;

        -- read_fifo for reading from SDRAM
        rd_fifo_rd_en: IN std_logic;
        rd_fifo_dout: OUT std_logic_VECTOR(15 downto 0);
        rd_fifo_empty: OUT std_logic;
        rd_fifo_valid: OUT std_logic;

        -- write_fifo for writing to SDRAM
        wr_fifo_din: IN std_logic_VECTOR(15 downto 0);
        wr_fifo_wr_clk: IN std_logic;
        wr_fifo_wr_en: IN std_logic;
        wr_fifo_full: OUT std_logic;
        wr_fifo_wr_ack: OUT std_logic;

        A : out  STD_LOGIC_VECTOR (11 downto 0);
        BA : out  STD_LOGIC_VECTOR (1 downto 0);
        CAS_n : out  STD_LOGIC;
        RAS_n : out  STD_LOGIC;
        CS_n : out  STD_LOGIC;
        WE_n : out  STD_LOGIC;
        CKE : out  STD_LOGIC;
        SDCLK : out  STD_LOGIC;
        SDCLKFB : in  STD_LOGIC;
        DQ : inout  STD_LOGIC_VECTOR (15 downto 0);
        DQM : out  STD_LOGIC_VECTOR (1 downto 0)
    );
end sdram_controller;
```

The controller is specifically designed for the purpose of storing sequential data as fast as possible. Arbitration between reads and writes is only implemented insofar as that reads take precedence but it

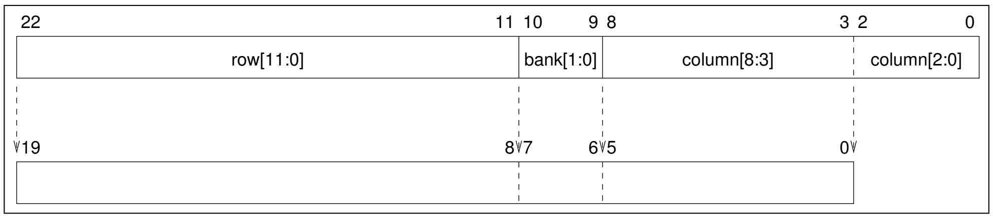


Figure 10.9: SDRAM Addressing

should generally be considered an error if both `read` and `write` are asserted at the same time. The SDRAM is accessed in bursts of eight 16 bit words, which is also the smallest addressable unit. The address register is twenty bits wide, allowing $2^{20} = 1048576$ items of 128 bits to be addressed. The counter has an additional bit to difference between the last accessed entry (0x1) and completion (0x0). In order to start a transfer the address register needs to be loaded with the address of the first access, and the counter has to be programmed with the number of 128 bit items that should be transferred.

In case of a read the controller is then going to transfer data from the memory to the read FIFO (32 entries deep) until there are less than 16 entries, the size of two bursts, available. The 16 entries were chosen as the FIFO's programmable full threshold (`rd_fifo_prog_full`) because a FIFO should never grow completely full. As the SDRAM interface fills the read FIFO a lot faster than the USB interface could possibly empty it there should always be between 16 and 24 entries in the FIFO. Once `rd_fifo_prog_full` is low again the next data burst will be read from SDRAM, until the transfer `counter` reaches zero.

Writes are buffered using a much larger FIFO with 1024 entries of 16 bits size each. The controller transfers data from the write FIFO to the SDRAM memory as long as there are at least eight words available in the FIFO. The programmable empty threshold (`wr_fifo_prog_empty`) was set to 8 because after starting a SDRAM burst the controller needs to be able to provide a data word on every clock cycle (10ns) with zero delays. The large FIFO was chosen to be able to buffer transfers while the SDRAM is busy with opening a bank or executing an AUTO REFRESH cycle.

SDRAM Addressing

SDRAM memory is addressed using bank (BA[n:0]), row (A[r:0], and column (A[c:0]) addresses, see subsection 10.2. The MT48LC8M16A2-7E consists of $8 * 2^{20}$ 16 bit words, that can be addressed using a 23 bit linearized address LA[22:0]. Internally, the memory consists of four banks (bank[1:0], each with 4096 rows (row[11:0]) by 512 columns (column[8:0]) of 16 bit words. The SDRAM controller implemented for the OpenOCD+trace uses an interleaving scheme that places bank[0], row[0] at the beginning of the address space, followed by bank[1], row[0], bank[2], row[0], and bank[3], row[0], before the next bank[0]'s row[1] starts. Figure 10.9 shows how the linearized address maps to the internal SDRAM organization, and also the lowest three bit from the address are practically ignored, because the memory is always accessed in bursts of eight, where the first three bits are always zero.

This addressing scheme allows the memory to be accessed sequentially without incurring a speed penalty for having to reopen a row on a bank that was just closed, because at the end of any row there will always follow a row on a different bank.

VHDL Implementation

Listing 10.15: sdram_controller.vhd

```
architecture Behavioral of sdram_controller is
    component read_fifo
        port (
            clk: IN std_logic;
            din: IN std_logic_VECTOR(15 downto 0);
            rd_en: IN std_logic;
            rst: IN std_logic;
            wr_en: IN std_logic;
            dout: OUT std_logic_VECTOR(15 downto 0);
            empty: OUT std_logic;
            full: OUT std_logic;
            prog_full: OUT std_logic;
            valid: OUT std_logic;
            wr_ack: OUT std_logic
        );
    end component;

    component write_fifo
        port (
            din: IN std_logic_VECTOR(15 downto 0);
            rd_clk: IN std_logic;
            rd_en: IN std_logic;
            rst: IN std_logic;
            wr_clk: IN std_logic;
            wr_en: IN std_logic;
            dout: OUT std_logic_VECTOR(15 downto 0);
            empty: OUT std_logic;
            prog_empty: OUT std_logic;
            full: OUT std_logic;
            valid: OUT std_logic;
            wr_ack: OUT std_logic
        );
    end component;
```

The templates for the FIFOs were generated by the Xilinx Core Generator. Because the `read_fifo` will only be read and written from the `sys_MAINCLK` domain it is implemented as a common clock FIFO using distributed RAM (Distributed SelectRAM). The `write_fifo` is written from the `trace_clk` domain and read from the `sys_MAINCLK` domain and is therefor implemented using independent clocks and block RAM (Block SelectRAM). Each of the FIFOs is sixteen bits wide to match the size of the SDRAM memory the controller is connected to. It would have been possible to implement the `write_fifo` write port only eight bits wide, allowing the data from the ETM port to be stored as is, but as the data needs to be examined anyway to identify trigger and trace-disabled cycles the 16 bit port was chosen because it can be written at half or less the ETM port's data transition rate[14].

When the `wr_en` signal on the FIFO's write port is high the data available at its `din` port will be placed into the FIFO. If the `rd_en` signal is registered high the next word of data will be output on its `dout` port in the next cycle. The programmable empty and full thresholds might indicate an empty or full condition even if the FIFO fill level hasn't reached the critical value already. It is guaranteed that if the `read_fifo`'s `prog_full` flag is cleared there is room for at least 16 entries, and if the `write_fifo`'s `prog_empty` signal is low there are at least eight entries available to store the next burst from SDRAM.

Listing 10.16: sdram_controller.vhd

```
    signal DQ_output_enable : STD_LOGIC;
    signal DQ_input : std_logic_vector (15 downto 0);
    signal DQ_output : std_logic_vector (15 downto 0);

    signal address : std_logic_vector (19 downto 0);
    signal counter : std_logic_vector (20 downto 0);
    signal int_busy : std_logic;
```

Because the SDRAM data bus is a bidirectional tri-state bus a `DQ_output_enable` signal is used to switch between input (`D` in high-Z mode) and output mode. The `address`, `counter`, and `int_busy` registers make the addressing , number of remaining bursts, and busy state available to the SDRAM controller.

Listing 10.17: sdram_controller.vhd

```
    type SDRAM_INT_STATE_TYPE is (
        INIT_POWERUP_DELAY, INIT_PRECHARGE, INIT_PRECHARGE_NOP,
        INIT_AUTOREFRESH1, INIT_AR1_NOP,
        INIT_AUTOREFRESH2, INIT_AR2_NOP,
        INIT_LOAD_MODE_REGISTER, INIT_LOAD_MODE_REGISTER_NOP,
        IDLE,
        PRECHARGE, PRECHARGE_ALL, PRECHARGE_NOP,
        AUTO_REFRESH, AUTO_REFRESH_NOP,
        ACTIVE, ACTIVE_NOP,
        READ_CMD, READ_NOP, READ_DATA,
        WRITE_CMD, WRITE_DATA
    );
    signal int_state : SDRAM_INT_STATE_TYPE;
```

The SDRAM controller is implemented with a state machine whose current state is stored in signal `int_state` of type BUS_STATE_TYPE, an enumeration of the possible SDRAM controller states. States prefixed with INIT_ are only entered during the controller's initialization phase, the other states are entered upon an access request or a selfrefresh timeout. When no accesses are pending the controller will remain in IDLE state. The initialization sequence is shown in figure 10.10 and follows the requirements outlined in 10.2. Where more than one cycle had to be spend outputting NOPs a delay counter was used to reduce the number of states required to describe the state machine. This is the case in INIT_-POWERUP_DELAY where 20,000 cycles of 10ns are spent for a total of 200us powerup delay[15], in

[14]Two ETM port cycles fit into one FIFO entry, but there can be ETM cycles where no data needs to be traced.

[15]The MT48LC8M16A2-7E datasheet specifies a minimum powerup delay of 100us, but the delay was increased as a safeguard for reliable operation.

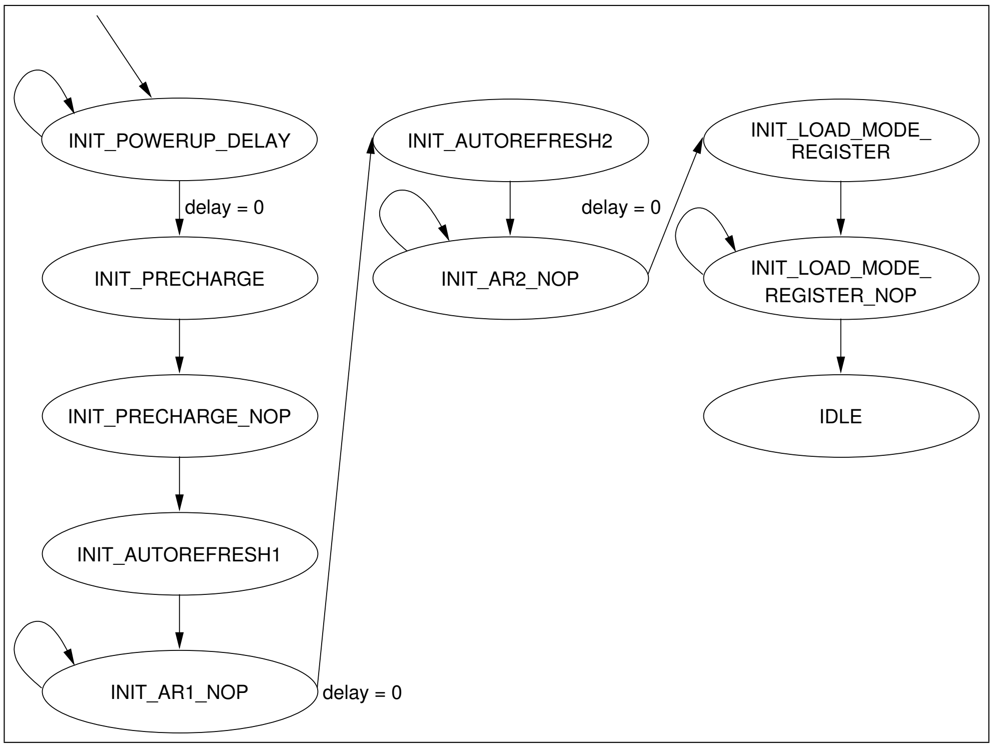


Figure 10.10: SDRAM Initialization Sequence

INIT_AR1_NOP and INIT_AR2_NOP where 6 cycles are spent for a total autorefresh period of 70ns[16], and in INIT_LOAD_MODE_REGISTER_NOP where one additional cycle is spent for a total of two clock cycles[17].

Listing 10.18: sdram_controller.vhd

```
    signal precharge_bank : std_logic_vector (1 downto 0);

    signal active_address : std_logic_vector (13 downto 0);

    signal bank0_idle : std_logic;
    signal bank1_idle : std_logic;
    signal bank2_idle : std_logic;
    signal bank3_idle : std_logic;

    signal bank0_row : std_logic_vector (11 downto 0);
    signal bank1_row : std_logic_vector (11 downto 0);
    signal bank2_row : std_logic_vector (11 downto 0);
    signal bank3_row : std_logic_vector (11 downto 0);

    signal refresh_counter : integer range 0 to 1500;
```

[16]The minimum autorefresh period t_{RFC} is 66ns.

[17]t_{MRD} is the minimum time between a mode register load operation and an ACTIVE or PRECHARGE command.

```

    signal DCM_SDRAM_locked : std_logic;
    signal DCM_SDRAM_rst : std_logic;
    signal DCM_SDRAM_rst_n : std_logic;
```

Several signals are used to store information about currently pending operations. The `precharge_bank` signal holds the number of the bank that needs to be precharged. `active_address` is the concatenation of the bank (bits 13 to 12) and row (bits 11 to 0) that should be opened. `bankN_idle` (N = 0...3) identifies idle banks[18], and `bankN_row` (N = 0...3) holds the number of the row that's currently open on a given bank if the corresponding `bankN_idle` signal is low.

The `refresh_counter` keeps track of the elapsed time since the last AUTO REFRESH cycle. The MT48LC8M16A2-7E requires 4096 refresh cycles every 64ms, and the SDRAM controller implemented for the OpenOCD satisfies that demand by executing an AUTO REFRESH roughly every 15us[19]. Refresh takes precedence only when the controller is in IDLE state, because interrupting a running burst would have a huge impact on the design complexity while only optimizing the refresh cycle period. The statemachine spends a maximum of ten cycles out of IDLE, when it is currently executing a read request (READ_CMD + READ_CMD_NOP + 8x READ_DATA). Before executing an AUTO REFRESH any open banks have to be closed, possibly adding six cycles to the maximum delay. This means that AUTO REFRESH cycles are guaranteed after at most 15us + 16ns = 15.016us, which is easily within the maximum time allowed.

The second DCM (`DCM_sdram`) used to deskew the external SDRAM clock must start only after the main system clock (the one generating `sys_MAINCLK`) locked and the feedback signal (`SDCLKFB`) is available, because it needs its input to be stable to achieve optimal locking and a minimized jitter. The FPGA configuration process could still have the `SDCLK` pin in tri-state mode by the time the SDRAM DCM tries to lock, causing the feedback on `SDCLKFB` to be unavailable (see [XILUG02, p.84]).

Listing 10.19: sdram_controller.vhd

```
    -- keep DCM_sdram reset until CLK locked
    SRL16_inst : SRL16 port map (
     Q => DCM_SDRAM_rst_n, A0 => '0', A1 => '0', A2 => '1', A3 => '0',
     D => CLK_locked, CLK => CLK);
    DCM_SDRAM_rst <= not(DCM_SDRAM_rst_n);
```

The `DCM_sdram` is therefor held in reset until four cycles after the main DCM locked by connecting a sixteen bit shift-left-register's (`SRL16_inst`) `D` input to the `CLK_locked` signal coming from the main system DCM and the shift register's output `Q` to the reset input of the SDRAM DCM. The address inputs `A[3:0]` determine the position from the sixteen bit shift register that is output on `Q`, in this case selecting the fourth bit.

Listing 10.20: sdram_controller.vhd

```
    -- sdram_DQ is a tri-state bus
    DQ_input <= DQ;
    DQ <= DQ_output when DQ_output_enable = '1' else (others => 'Z');
```

[18]A bank with no open row is called an idle bank

[19]Maximum delay between two consecutive AUTO REFRESH cycles is 64ms / 4096 = 15.625us.

```

    -- Read FIFO always loads from SDRAM
    rd_fifo_din <= DQ_input;

    -- Write FIFO always outputs to SDRAM
    DQ_output <= wr_fifo_dout;

    rd_fifo_rst <= not(CLK_locked and DCM_SDRAM_locked);
    wr_fifo_rst <= not(CLK_locked and DCM_SDRAM_locked);

    CKE <= CLK_locked and DCM_SDRAM_locked;
    CS_n <= not(CLK_locked and DCM_SDRAM_locked);

    -- address and counter visible outside
    out_address <= address;
    out_counter <= counter;

    -- complete visible outside
    busy <= int_busy;
    int_busy <= '0' when counter = '0' & x"00000" and load_counter = '0'
  else '1';
```

Listing 10.20 shows how the SDRAM data bus is implemented as a tri-state bus controlled by the `DQ_output_enable` signal. Because the only data written to the read FIFO comes from SDRAM its `rd_fifo_din` input is directly connected to the data coming from `DQ_input`, just like the write FIFO output is directly connected to `DQ_output`. Both FIFOs are held in reset until both clocks locked, because nothing could be done with accumulated data during that time anyway.

The SDRAM clock enable (`CKE`) is high once both clocks locked, and the chip select will be driven low (active) at the same time. This is possible because the OpenOCD+trace design makes no use of the COMMAND INHIBIT or SELF REFRESH commands available when those two lines are deasserted.

The last concurrent assignments in `sdram_controller.vhd` make the current `address`, `counter` value, and `busy` signal accessible outside of the sdram_controller entity. The SDRAM controller is in busy state whenever the counter is not zero (more data to transfer) or when the counter is currently being loaded.

Figure 10.11 shows the state machine after initialization completed, when the controller is in IDLE state waiting for requests or AUTO REFRESH cycles.

Listing 10.21: sdram_controller.vhd

```
when IDLE =>
    RAS_n <= '1';
    CAS_n <= '1';
    WE_n <= '1';
    int_state <= IDLE;
    -- refresh takes precedence
    if refresh_counter = 0 then
```

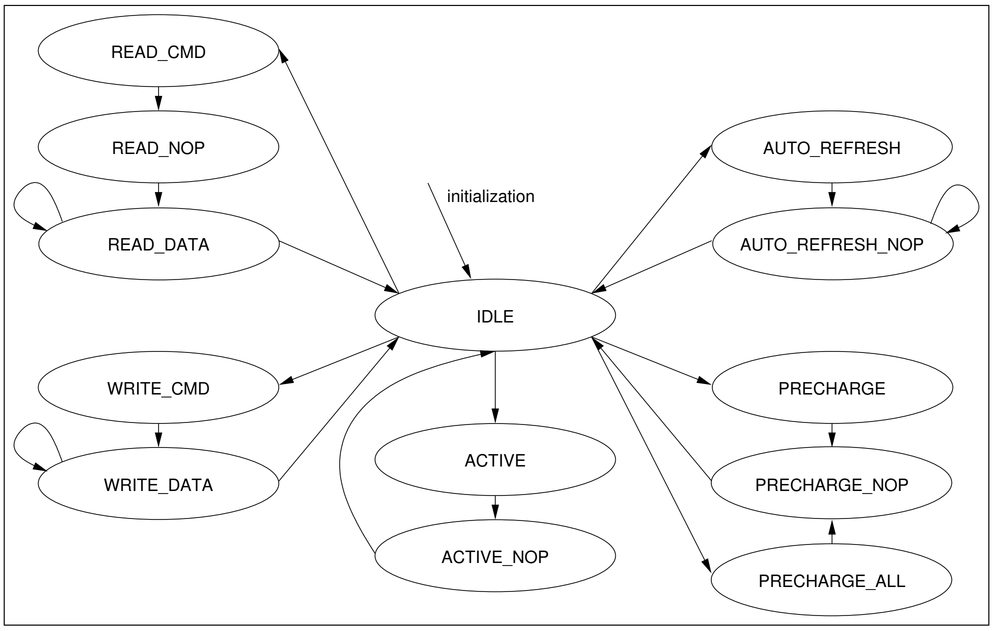


Figure 10.11: SDRAM Access Sequence

```
        if bank0_idle = '0' or bank1_idle = '0' or bank2_idle = '0'
or bank3_idle = '0' then
            int_state <= PRECHARGE_ALL;
        else
            int_state <= AUTO_REFRESH;
        end if;
```

In IDLE state the controller outputs a NOP command. If the `refresh_counter` reaches zero the controller checks whether one more more banks are not idle and executes a PRECHARGE ALL cycle if necessary. After the state machine passes through PRECHARGE_ALL and PRECHARGE_NOP it will reenter the IDLE state, this time with all banks idle, and because the refresh counter stopped counting at zero it will then enter the AUTO_REFRESH state.

Listing 10.22: sdram_controller.vhd

```
else
    if int_busy = '1' then
        if (read = '1' and rd_fifo_prog_full /= '1')
            or (write = '1' and wr_fifo_prog_empty /= '1') then
            -- we want to access the memory (read or write and
FIFO available)
            if address(7 downto 6) = "00" then
                if bank0_idle = '0' and bank0_row /=
address(19 downto 8) then
                    -- need to close the current row
                    precharge_bank <= address(7 downto 6);
                    int_state <= PRECHARGE;
                elsif bank0_idle = '1' then
                    -- need to open a row
                    active_address <= address(7 downto 6)
& address(19 downto 8);
                    int_state <= ACTIVE;
                else
                    -- ready to read/write data
                    if read = '1' then
                        int_state <= READ_CMD;
                    elsif write = '1' then
                        wr_fifo_rd_en <= '1';
                        int_state <= WRITE_CMD;
                    end if;
                end if;
```

If no refresh cycle is necessary, if the controller is currently busy (counter != 0), and if either a read request with available space in the read FIFO or a write request with data available in the write FIFO is pending the controller will check if the current row needs to be closed, if a new bank needs to be opened, or it will proceed with accessing the memory.

Address bits `address[7:6]` identify the bank that needs to be accessed next[20]. If the bank isn't idle and the currently open row doesn't match the row that should be accessed (`address[19:8]`) the controller will enter PRECHARGE state, followed by PRECHARGE_NOP and the return to IDLE state. At that point the bank will be idle, and the controller progresses with opening the new row. If the bank was already idle previously the PRECHARGE step can be skipped, and the new row is opened immediately. The controller stores the desired bank and row in `active_address` and passes through ACTIVE and ACTIVE_NOP where the row gets opened before the state machine returns to IDLE. The controller can then read or write data in the open row depending on the request type. In case of a write the `wr_fifo_rd_en` signal is immediately asserted to have the data that should be written available on the next cycle.

Listing 10.23: sdram_controller.vhd

```
when READ_CMD =>
    -- READ (assert bank and column address)
    BA <= address(7 downto 6);
    A <= "000" & address(5 downto 0) & "000";
    RAS_n <= '1';
    CAS_n <= '0';
    WE_n <= '1';
    int_state <= READ_NOP;
when READ_NOP =>
    -- NOP (CAS delay 2)
    RAS_n <= '1';
    CAS_n <= '1';
    WE_n <= '1';
    int_state <= READ_DATA;
```

[20]The code from listing 10.22 is repeated for banks one, two and three, but for documentation purposes it is going to be enough to examine operation of bank 0

```
        delay <= 8;
    when READ_DATA =>
        -- NOP (capture data)
        RAS_n <= '1';
        CAS_n <= '1';
        WE_n <= '1';
        rd_fifo_wr_en <= '1';
        if delay = 0 then
            rd_fifo_wr_en <= '0';
            counter <= counter - 1;
            address <= address + 1;
            int_state <= IDLE;
        else
            delay <= delay - 1;
        end if;
```

In case of a read request the READ_CMD state is entered, and the READ command is output together with the bank (from `address[7:6]`) and column address (`address[5:0]` and the three least significant bits zero because of the eight word bursts). The controller then passes through READ_NOP where only one cycle is spent because the MT48LC8M16A2-7E is fast enough to operate at two cycles CAS latency. If the memory were slower, requiring three cycles of CAS latency, an additional cycle would have to be spent in READ_NOP. In READ_DATA the data available on `DQ_input` is placed into the read FIFO by asserting the FIFO's `rd_fifo_wr_en` signal for eight consecutive cycles. Once the delay reached zero the controller increments the `counter` and `address`, deasserts the write enable signal, and moves back to IDLE state, ready for the next access.

Listing 10.24: sdram_controller.vhd

```
    when WRITE_CMD =>
        -- WRITE (assert bank and column address)
        BA <= address(7 downto 6);
        A <= "000" & address(5 downto 0) & "000";
        RAS_n <= '1';
        CAS_n <= '0';
        WE_n <= '0';
        delay <= 7;
        DQ_output_enable <= '1';
        int_state <= WRITE_DATA;
    when WRITE_DATA =>
        -- NOP (output data)
        RAS_n <= '1';
        CAS_n <= '1';
        WE_n <= '1';
        if delay = 0 then
            DQ_output_enable <= '0';
            counter <= counter - 1;
            address <= address + 1;
            int_state <= IDLE;
        else
            if delay = 1 then
                wr_fifo_rd_en <= '0';
            end if;
            delay <= delay - 1;
        end if;
```

The controller already outputs data in WRITE_CMD, because SDRAM writes require no delay between the WRITE command and the first data word. Seven cycles are then spent in WRITE_DATA state outputting the remaining words of the burst. By the time the `delay` counter reaches one the write FIFO read enable is deasserted, and on the last cycle the output enable is deasserted, the counter and address are incremented, and the IDLE state is reentered.

Listing 10.25: sdram_controller.vhd

```
        when others =>
            int_state <= INIT_POWERUP_DELAY;
            delay <= 9999;
    end case;
```

As a matter of good FPGA design practice a catch-call case statement is used to recover from any unexpected problems in the state machine by moving back to `INIT_POWERUP_DELAY`.

The SDRAM controller requires 100ns (10 cycles) to write a burst of eight words (16 bytes) for an aggregate bandwidth of 160 million B/s. The refresh counter is programmed to execute an AUTO REFRESH cycle every 15us, a time frame long enough for 150 burst cycles. The necessary PRECHARGE ALL prior to applying the AUTO REFRESH command requires 30ns, the AUTO REFRESH itself takes 80ns, and opening the bank with an ACTIVE command requires another 20ns, for a total of 150ns. This means that instead of 150 burst cycles only about 148 burst cycles are going to fit in between two subsequent refresh cycles. 148 bursts of 16 byte per 15us result in an effective bandwidth of 157,866,666 B/s.

The SDRAM controller doesn't implement a counter to keep track of the time a bank has been opened already, but this timing isn't critical for the OpenOCD+trace's SDRAM controller because the refresh cycle period of 15us is substantially lower than the maximum ACTIVE to PRECHARGE (t_{RAS}) of 120us anyway.

Trace Capture

The main control logic and the capturing are implemented in `trace_capture.vhd`. It instantiates all the submodules shown in figure 10.7 and connects the external interfaces to the submodule ports (e.g. USB, SDRAM, LCD). The top level entity `trace_capture` defines the interface to the peripherals connected to the FPGA. The `jtag_*` signals are commented out because a JTAG controller is not yet implemented, but could be added as a future enhancement.

Listing 10.26: trace_capture.vhd

```
entity trace_capture is
    Port (
        sdram_A : out std_logic_vector (11 downto 0);
        sdram_BA : out std_logic_vector (1 downto 0);
        sdram_CAS_n : out std_logic;
        sdram_CKE : out std_logic;
        sdram_CLK : out std_logic;
        sdram_CS_n : out std_logic;
        sdram_RAS_n : out std_logic;
        sdram_WE_n : out std_logic;
        sdram_CLKFB : in std_logic;
        sdram_DQ : inout std_logic_vector (15 downto 0);
        sdram_DQM : out std_logic_vector (1 downto 0);

        sys_CLK0 : in std_logic;

        usb_D : inout std_logic_vector (7 downto 0);
        usb_RD_n : out std_logic;
        usb_WR : out std_logic;
        usb_RXF_n : in std_logic;
        usb_TXE_n : in std_logic;
        usb_SIWU : out std_logic;

        trace_PIPESTAT : in std_logic_vector (2 downto 0);
        trace_TRACESYNC : in std_logic;
        trace_TRACEPKT : in std_logic_vector (3 downto 0);
        trace_EXTIN : out std_logic;
        trace_TRACECLK : in std_logic;

        --jtag_TCK : out std_logic;
        --jtag_TMS : out std_logic;
        --jtag_TDI : out std_logic;
        --jtag_TDO : in std_logic;
        --jtag_nTRST : out std_logic;
        --jtag_nSRST : out std_logic;

        lcd_CS_n : out std_logic;
        lcd_RST_n : out std_logic;
        lcd_SID : out std_logic;
        lcd_SOD : in std_logic;
        lcd_SCLK : out std_logic;

        button : in std_logic_vector (3 downto 0)
    );
end trace_capture;
```

The control logic is implemented as a state machine whose current state is stored in signal `controller_-state` of type `CONTROLLER_STATE_TYPE`. After reset the state machine comes up in IDLE state, waiting for command input from the USB controller. It supports reading and writing the OpenOCD+trace registers and controls data transfers from SDRAM to the host via the USB controller.

Listing 10.27: trace_capture.vhd

```
    type CONTROLLER_STATE_TYPE is (
        IDLE, COMMAND,
        READREG, MUXREG, SENDREG,
        WRITEREG, RECEIVEREG, STOREREG,
        TRANSFER_READ, TRANSFER_READ_FIFO_VALID, TRANSFER_READ_FIFO_REQUEST,
        TRANSFER_READ_HIGH, TRANSFER_READ_LOW,
        TRANSFER_WRITE,
        RESET_TRACE
    );
    signal controller_state : CONTROLLER_STATE_TYPE := IDLE;
```

The only combinatorial assignment generates the `sys_trace_clk_good` signal out of the synchronized versions of `trace_clkin_stopped`, `trace_clkfx_stopped`, and `trace_clk_locked`.

Listing 10.28: trace_capture.vhd

```
    sys_trace_clk_good <= (not sys_trace_clkin_stopped(1))
        and (not sys_trace_clkfx_stopped(1))
        and sys_trace_clk_locked(1);
```

A process clocked by the rising edge of `sys_MAINCLK` is used to implement the state machine. While the DCM used to generate the clock didn't lock, all signals are assigned safe defaults.

Listing 10.29: trace_capture.vhd

```
elsif sys_MAINCLK'event and sys_MAINCLK = '1' then

    -- assert register load signals only for one clock cycle
    sdram_load_address <= '0';
    sdram_load_counter <= '0';

    -- keep track of signal changes
    delayed_button <= button;
    usb_READ_VALID_delayed <= usb_READ_VALID;
    usb_WRITE_ACK_delayed <= usb_WRITE_ACK;
    sdram_busy_delayed <= sdram_busy;
    sys_trace_enabled_delayed <= sys_trace_enabled;
```

The code exploits the sequential description of behavioural VHDL designs and assigns zeros to the signals used to load the SDRAM address and counter values. If a one is assigned later within the same process that assignment takes precedence, but if no assignment is made the signals will default to zero.

In order to identify changes to some of the signals, the design maintains delayed versions of those and later compares the current state of the signal with the delayed copy.

Listing 10.30: trace_capture.vhd

```
    -- synchronize status signals form trace_clk domain
    sys_trace_triggered <= sys_trace_triggered(0) & trace_triggered;
    sys_trace_completed <= sys_trace_completed(0) & trace_completed;
    sys_ack_trigger <= sys_ack_trigger(0) & trace_ack_trigger;
    sys_trace_clk_locked <= sys_trace_clk_locked(0) & trace_clk_locked;
    sys_trace_clkin_stopped <=
sys_trace_clkin_stopped(0) & trace_clkin_stopped;
    sys_trace_clkfx_stopped <=
sys_trace_clkfx_stopped(0) & trace_clkfx_stopped;
```

Because all signals coming from the `trace_clk` domain are asynchronous to the control logic they are synchronized prior to being used in the `sys_MAINCLK` domain using two flip-flops each. These registers should be accessed for example as `sys_trace_triggered(1)`.

Whenever the *enabled* bit in the *OpenOCD+trace control register* is set, the SDRAM counter is loaded with its maximum value, the overflow bit *full* (`sys_trace_full`) is cleared, and the SDRAM write request is asserted. Clearing the the *enabled* bit again deasserts the write request and allows the captured data to be retrieved from the buffer memory.

Listing 10.31: trace_capture.vhd

```vhdl
    -- enabling trace reloads the SDRAM counter
    if sys_trace_enabled /= sys_trace_enabled_delayed and sys_trace_enabled = '1' then
        sdram_in_counter <= '1' & x"00000";
        sdram_load_counter <= '1';
        sys_trace_full <= '0';
        sdram_write <= '1';
    elsif sys_trace_enabled /= sys_trace_enabled_delayed and sys_trace_enabled = '0' then
        sdram_write <= '0';
    end if;

    -- when tracing is enabled, reload the SDRAM counter as soon as it runs down
    if sys_trace_enabled = '1' then
        if sdram_busy /= sdram_busy_delayed and sdram_busy = '0' then
            sdram_in_counter <= '1' & x"00000";
            sdram_load_counter <= '1';
            sys_trace_full <= '1';
        end if;
    end if;
```

When tracing is enabled and the SDRAM controller's busy flag changes to zero (transfer completed) the SDRAM wrapped around once. The `sys_trace_full` signal is asserted and the counter is reloaded with its maximum value. The write FIFO has room for 2048 ETM port cycles which provides enough time to reenable the SDRAM controller write request. Reloading the counter implicitly switched the SDRAM controller to busy state, and the controller will continue writing data from the write FIFO to the SDRAM.

Listing 10.32: trace_capture.vhd

```vhdl
case controller_state is

    when IDLE =>
        if usb_BUSY = '0' then
            usb_READ <= '1'; -- request data from USB
        else
            if usb_READ_VALID /= usb_READ_VALID_delayed and usb_READ_VALID = '1' then
                usb_READ <= '0'; -- clear read request
```

```
                    usb_input <= usb_DATA_IN;
                    controller_state <= COMMAND;
                end if;
            end if;
```

In IDLE state a USB read request is signaled if the USB controller completed the previous access (`usb_BUSY` low), otherwise the `usb_READ_VALID` signal is monitored for a change to one, indicating completion of the access. Once data is available from the USB controller the read request is cleared, allowing the USB controller to complete the current request. The data is registered in the `usb_input` register and the state machine transitions to COMMAND state.

In the COMMAND state the byte received via USB is evaluated and the next state is selected. In case of a register access the desired register is stored in `active_register`, and if the *all* bit was set on a SDRAM transfer request the SDRAM counter is loaded with its maximum value. A sixteen cycle delay is used to ensure that the DCM reset signal is asserted long enough to allow the logic running from `trace_clk` to register the reset request.

A register is read in READREG state and then sent using multiple transfers via `usb_output` in MUXREG and SENDREG state. While in SENDREG state the control logic waits for the USB controller to become idle. After `usb_BUSY` was low the USB write request is signaled, the data is placed on `usb_DATA_OUT` and the state machine moves back to MUXREG, where either the next byte is transferred or the read process is completed by moving back to IDLE state. The control logic will always wait in SENDREG for the previous USB transfer to complete before initiating a new one.

In WRITEREG state the USB read request is asserted once the USB controller is idle. When the USB transfer completed and the USB data is valid the RECEIVEREG state is entered. The data received via USB is stored in `input_register` and the controller moves back to WRITEREG state until the complete 32 bits were received successfully. The controller then moves to STOREREG where the data from `input_register` is used to load the SDRAM address or counter register, the trigger counter, or to set the control register bits. A handshaking mechanism is used to synchronise write access to the trigger counter register: Once the write reached the STOREREG state the `sys_load_trigger` signal is asserted. The logic in the `trace_clk` domain registers that signal change at some point and asserts the `trace_ack_trigger` signal, indicating that the new value has been acknowledged. The control logic then releases the `sys_load_trigger` signal, and the new trigger counter is set. Because of this complex handshaking mechanism required to synchronize access to a large register reading of the trigger counter was not implemented, but if it were to be added at a later point the same synchronisation scheme could be used.

Listing 10.33: trace_capture.vhd

```
        when TRANSFER_READ =>
            -- can't access memory while trace is enabled
            if sys_trace_enabled = '1' then
                controller_state <= IDLE;
            else
                if sdram_busy = '0' and sdram_rd_fifo_empty = '1' then
                    sdram_read <= '0';
                    controller_state <= IDLE;
                else
                    sdram_read <= '1';
                    -- request next entry from SDRAM read FIFO
                    if sdram_rd_fifo_empty <= '0' then
                        sdram_rd_fifo_rd_en <= '1';
                        controller_state <= TRANSFER_READ_FIFO_REQUEST;
                    end if;
                end if;
            end if;

        -- only request one FIFO entry
        when TRANSFER_READ_FIFO_REQUEST =>
            sdram_rd_fifo_rd_en <= '0';
            controller_state <= TRANSFER_READ_FIFO_VALID;

        -- FIFO output valid, register value
        when TRANSFER_READ_FIFO_VALID =>
            output_register(15 downto 0) <= sdram_rd_fifo_dout;
            controller_state <= TRANSFER_READ_LOW;

        -- transfer first byte from FIFO
        when TRANSFER_READ_LOW =>
            if usb_BUSY = '0' then
                usb_DATA_OUT <= output_register(7 downto 0);
                usb_WRITE <= '1';
            else
                if usb_WRITE_ACK /= usb_WRITE_ACK_delayed and usb_WRITE_ACK = '1' then
                    usb_WRITE <= '0';
                    controller_state <= TRANSFER_READ_HIGH;
                end if;
            end if;

        -- transfer second byte from FIFO
        when TRANSFER_READ_HIGH =>
            if usb_BUSY = '0' then
                usb_DATA_OUT <= output_register(15 downto 8);
                usb_WRITE <= '1';
            else
                if usb_WRITE_ACK /= usb_WRITE_ACK_delayed and usb_WRITE_ACK = '1' then
                    usb_WRITE <= '0';
                    controller_state <= TRANSFER_READ;
                end if;
            end if;
```

SDRAM reads are handled in the TRANSFER_READ state. Because the SDRAM controller doesn't support concurrent read and write accesses the control logic immediately returns to IDLE state if tracing is currently enabled. It also returns to IDLE when the SDRAM controller isn't busy anymore and no entries are left in the read FIFO, indicating completion of the requested SDRAM read. Otherwise the SDRAM controller read request signal is asserted, and the TRANSFER_READ_FIFO_REQUEST state is entered once the read FIFO isn't empty (i.e. it has at least one entry available). The read FIFO read enable (`sdram_rd_fifo_rd_en`) is asserted for one cycle when moving to TRANSFER_READ_FIFO_-REQUEST where it's disabled again. In the next cycle the control logic is in the TRANSFER_READ_-FIFO_VALID state and the FIFO's output is registered in `output_register`, which is then sent to the host with two USB write requests, similar to a register read.

SDRAM writes were only meant as a test case to allow verification of the SDRAM controller but were later abandoned because the SDRAM controller proved to work reliably. Adding SDRAM writes from the USB controller would have required extensive synchronization because the write FIFO's write port operates in the `trace_clk` domain while the USB controller operates in the `sys_MAINLCK` domain.

Listing 10.34: trace_capture.vhd

```vhdl
elsif trace_clk'event and trace_clk = '1' then

    -- keep track of signal changes
    trace_enabled_delayed <= trace_enabled(1);
    trace_load_trigger_delayed <= trace_load_trigger(1);

    -- synchronize signals from sys_MAINCLK domain
    trace_enabled <= trace_enabled(0) & sys_trace_enabled;
    trace_half_rate <= trace_half_rate(0) & sys_trace_half_rate;
    trace_load_trigger <= trace_load_trigger(0) & sys_load_trigger;

    -- enabling trace clears the status signals
    if trace_enabled(1) /= trace_enabled_delayed
and trace_enabled(1) = '1' then
        trace_completed <= '0';
        trace_triggered <= '0';
    end if;

    -- handshaking to load trigger counter */
    if trace_load_trigger(1) /= trace_load_trigger_delayed
        and trace_load_trigger(1) = '1' then

        trace_trigger_counter <= sys_trigger_counter & "000";
        trace_ack_trigger <= '1';
    elsif trace_load_trigger(1) /= trace_load_trigger_delayed
        and trace_load_trigger(1) = '0' then

        trace_ack_trigger <= '0';
    end if;
```

The trace capture itself is implemented as a separate process and is clocked by the rising edge of `trace_clk`, a clock generated by the trace_clock component running at either the same frequency as the ETM TRACECLK or at twice that frequency, if the ETM port operates in half-rate clocking mode.

Signals coming from the `sys_MAINCLK` domain are synchronized and changes of the `trace_enabled(1)` and `trace_load_trigger(1)` signals are monitored. The `trace_completed` and `trace_triggered` status signals are cleared when tracing is enabled to indicate the start of a new trace run.

As described above a synchronization mechanism using synchronized handshake signals is used to transfer the trigger counter value (`trace_trigger_counter`) from the `sys_MAINCLK` domain to the `trace_clk` domain. The value from the trigger counter register is concatenated with three zeros because the SDRAM is accessed in bursts of eight, but the trace capture logic accesses the write FIFO one word at a time.

Listing 10.35: trace_capture.vhd

```vhdl
    if trace_trigger_counter = x"00000"
and trace_load_trigger(1) = '0' then
        trace_completed <= '1';
    end if;

    -- pipeline trace data
    trace_data <= trace_TRACESYNC & trace_TRACEPKT & trace_PIPESTAT;

    -- pipelined trace data is not valid if PIPESTAT is TD
and TRACEPKT 0 is not set
    if trace_PIPESTAT = "111" and trace_TRACEPKT(0) = '0' then
        trace_data_valid <= '0';
    else
        trace_data_valid <= '1';
    end if;
```

The `trace_completed` signal is asserted once the trigger counter reached zero, but only if the counter isn't currently being loaded. The signals coming from the ETM port are registered in `trace_data` and `trace_PIPESTAT` is analyzed, but the data isn't yet stored to the FIFO. This pipelining improves the timing of the design because there is a whole cycle until the `trace_data_valid` signal is needed to decide whether the data from the previous cycle should be stored to the FIFO or not.

Listing 10.36: trace_capture.vhd

```vhdl
    -- while the trace is not completed (trigger counter not zero)
    if trace_completed = '0' and trace_data_valid = '1' then
        -- multiplex trace data onto fifo bus
        if trace_port_half = '0' then
            sdram_wr_fifo_din(7 downto 0) <= trace_data;
            trace_port_half <= '1';
        else
            sdram_wr_fifo_din(15 downto 8) <= trace_data;
            sdram_wr_fifo_wr_en <= '1';
            -- start decrementing trigger counter on every write
to fifo if triggered
            if trace_triggered = '1' then
                trace_trigger_counter <= trace_trigger_counter - 1;
            end if;
            trace_port_half <= '0';
        end if;
    end if;

    if trace_PIPESTAT = "110" then
        trace_triggered <= '1';
    end if;
```

While the trace run isn't completed and if the data captured during the last cycle was valid the write FIFO data input is written. Each half of `sdram_wr_fifo_din` is written every other valid trace cycle, and after writing the second half the write enable signal is asserted for a single cycle. If a *TR* (trigger) cycle was observed since the start of the current trace run the trigger counter will be decremented after storing the next entry in the write FIFO.

Constraints

Placement and timing constraints are defined in `trace_capture.ucf`. The UCF (user constraints file) fileformat uses a Xilinx specific syntax and can be edited by hand (it is ASCII text) or using the UCF wizard. For the purposes of the OpenOCD+trace manually specifying the necessary constraints was sufficient and the wizard wasn't used.

The UCF file assigns package pins to the signals used in the design, configures electrical parameters of some signals for improved signal quality, specifies the clock periods the design flow should expect on clock inputs, and specifies some timing constraints that will be used by the place and route step to optimize timings.

Listing 10.37: trace_capture.ucf

```
NET sdram_A[*] OFFSET = OUT : 2.0 : BEFORE : sys_CLK0 ;
NET sdram_DQ[*] OFFSET = OUT : 2.0 : BEFORE : sys_CLK0 ;
NET sdram_RAS_n OFFSET = OUT : 2.0 : BEFORE : sys_CLK0 ;
NET sdram_CAS_n OFFSET = OUT : 2.0 : BEFORE : sys_CLK0 ;
NET sdram_WE_n OFFSET = OUT : 2.0 : BEFORE : sys_CLK0 ;
NET sdram_BA[*] OFFSET = OUT : 2.0 : BEFORE : sys_CLK0 ;
NET sdram_DQ[*] OFFSET = IN : 6.3 : AFTER : sys_CLK0;

NET trace_TRACEPKT[*] OFFSET = IN : 2.5 : BEFORE : trace_TRACECLK;
NET trace_TRACESYNC OFFSET = IN : 2.5 : BEFORE : trace_TRACECLK;
NET trace_PIPESTAT[*] OFFSET = IN : 2.5 : BEFORE : trace_TRACECLK;
```

The MT48LC8M16A2-7E specifies a minimum setup time for all signals entering the memory of 1.5ns. The constraints file places a timing constraint on all outputs to the SDRAM that requires the signals to be valid at least 2ns before the rising edge of sys_CLK0, allowing up to 500ps of board delay. The *Access time from CLK (pos. edge)*, $t_{AC(2)}$ maximum is 6ns when running at two cycles CAS latency, the constraints file therefor specifies an `OFFSET=IN` of 6.3ns, still giving up to 300ps of timing margin for the board delay. An additional timing constraint was placed on the ETM port signals to have the PAR stage optimize for a small setup time of only 2.5ns.

10.5 OpenOCD Integration

OpenOCD+trace support is implemented in `./src/target/oocd_trace.c` using declarations from `./src/target/oocd_trace.h`. The `oocd_trace_t` structure holds information about the OpenOCD+trace on a per target basis and is accessible via the driver specific `etm_context_t->capture_driver_priv` pointer.

The OpenOCD+trace driver requires one configuration statement specifying the target whose ETM trace port the unit is connected to and the path to the serial device:

```
#oocd_trace config <target#> <TTY path>
oocd_trace config 0 /dev/ttyUSB0
```

A global variable of type `etm_capture_driver_t` called `oocd_trace_capture_driver` holds pointers to the OpenOCD+trace functions and implements the ETM capture driver interface specified for the OpenOCD.

10.6 STR912 Example Capture

The following trace was captured from a Hitex STR912 evalboard running a sample program that displays a counter on the two 7-segment LED displays mounted on the board. Because the two 7-segment displays share the enable lines for the segments two additional signals are used to select the currently active display. This makes it necessary to time multiplex the output, and a short delay loop counting up to 10,000 was used to keep the output steady between switching the displays.

ETM port pins are often multiplexed with other functionality. In case of the STR912FW44 in the LQFP128 package the trace pins are available on multiple different pins.

Table 10.7: STR912FW44 LQFP128 ETM Port

ETM Signal	Pins
TRACEPKT[3:0]	P0.[3-0], P2.[3-0], P4.[3-0], P7.[3-0]
PIPESTAT[2:0]	P0.[6-4], P2.[6-4], P4.[6-4]
TRACESYNC	P0.7, P2.7. P4.7
TRACECLK	P1.5, P1.7, P5.0, P6.6
EXTRIG	P1.0, P1.7, P5.3, P6.7, P7.5

On the Hitex STR912 evalboard a Mictor connector can be fitted for use as an ETM port (X27 TRACE), but the OpenOCD+trace design uses the pin 2.54mm pin header (X28 ETM) which is originally intended to take a set of jumpers that connect the ETM lines to an extension header (X24 PORT B) (see figure 10.12). Pin 10 on connector X24 (PORT B) connects to ground, and a jumper between pins 8 and 10 was used to make that ground connection available on X28 pin 2.

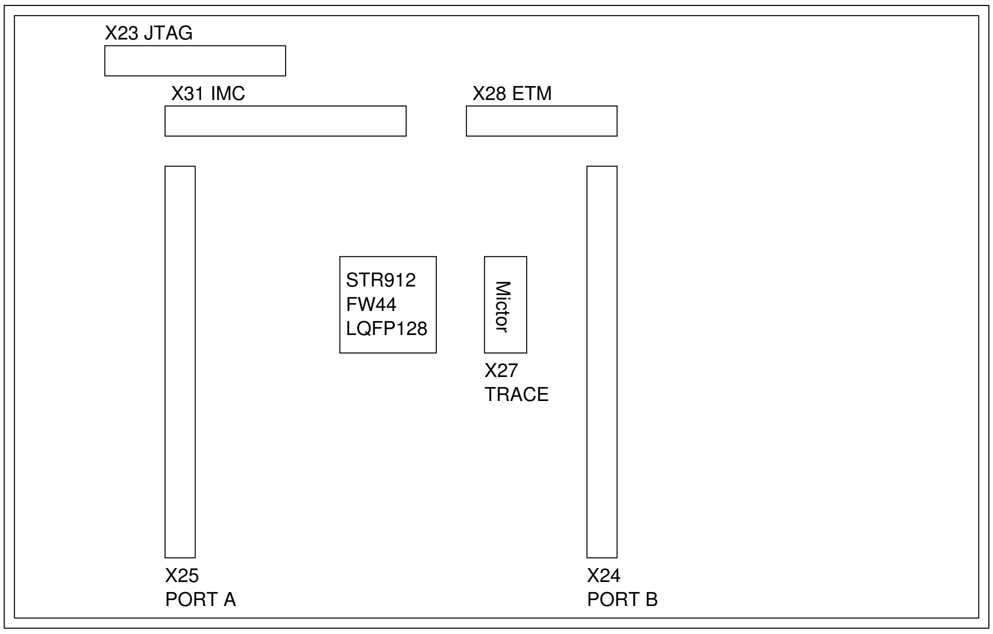


Figure 10.12: Hitex STR912 Evalboard

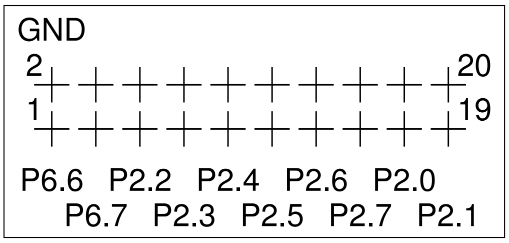


Figure 10.13: Hitex STR912 ETM Port

Pins P2.[3-0], P2.[6-4], P2.7 and P6.6 were assigned their alternate output 3 (TRACEPKT[3:0], PIPESTAT[2:0] and TRACESYNC, TRACECLK), P6.7 was assigned its alternate input 1 (EXTRIG), resulting in the X28 pinout shown in figure 10.13.

The necessary initialisations were made using OpenOCD memory write commands after halting the target:

```
# GPIOOUT6 0x5c00205c, P6.6 as alternate output 3
mww 0x5c00205c 0x3000
# GPIOOUT2 0x5c00204c, P2.[7-0] as alternate output 3
mww 0x5c00204c 0xffff
# GPIOIN6 0x5c00207c, P6.7 as alternate input 1
mww 0x5c00207c 0x80
```

The ETM was configured for instruction trace only with zero bits of the context id, cycle accurate tracing, and normal branch output behaviour. The address comparators were configured to match on ARM mode instruction fetches from the beginning to the end of the `delay()` function. The trace enable event was always enabled, and the trace control register was configured to exclude the area selected by address comparator pair 1. The trigger event was set to be always disabled, because no address comparator was available anymore. This could have been overcome by selecting one of the EmbeddedICE comparators via its range feature, but for the purposes of testing the OpenOCD+trace implementation use of a trigger was not necessary.

```
# instruction-tracing only, no context ID tracing
etm tracemode none 0 enable disable

# ETM_ADDR_COMPARATOR_VALUE1
reg 70 0x268
# ETM_ADDR_ACCESS_TYPE1
reg 86 0x19

# ETM_ADDR_COMPARATOR_VALUE2
reg 71 0x2a8
# ETM_ADDR_ACCESS_TYPE2
reg 87 0x19

# ETM_TRACE_EN_EVENT
reg 62 0x6f
# ETM_TRACE_EN_CTRL1
reg 63 0x01000001
# ETM trigger event
reg 56 0x406f
```

Tracing was enabled only for about one second to limit the amount of collected trace data, but during that short period already 39616 trace cycles were acquired, because only the relatively short delay loop got filtered, while the multiplexed LED display output was included in the trace.

Listing 10.38 shows a disassembly of the endless loop from the example application. The current value of `val` is divided by 10 and the remainder (first decimal digit) is output on 7-segment display one, followed by a call to the delay loop. The second decimal digit is output on 7-segment display two, another call to the delay loop is made, and the `divide` counter is incremented. If `divide` reached 100 `val` is increased by one and the `divide` counter is reset. At that point the loop starts from the beginning with a branch back to address 0x39c.

Listing 10.38: str912_hitex_blink/main.out

```
        while (1)
        {
                output_7segment(val % 10, 1);
 39c:   e51b1014        ldr     r1, [fp, #-20]
```

```
3a0:   e59f30c0        ldr     r3, [pc, #192]  ; 468 <.text+0x468>
3a4:   e0c32391        smull   r2, r3, r1, r3
3a8:   e1a02143        mov     r2, r3, asr #2
3ac:   e1a03fc1        mov     r3, r1, asr #31
3b0:   e0632002        rsb     r2, r3, r2
3b4:   e50b201c        str     r2, [fp, #-28]
3b8:   e51b301c        ldr     r3, [fp, #-28]
3bc:   e1a03083        mov     r3, r3, lsl #1
3c0:   e1a02103        mov     r2, r3, lsl #2
3c4:   e0833002        add     r3, r3, r2
3c8:   e0631001        rsb     r1, r3, r1
3cc:   e50b101c        str     r1, [fp, #-28]
3d0:   e51b001c        ldr     r0, [fp, #-28]
3d4:   e3a01001        mov     r1, #1  ; 0x1
3d8:   ebffffb3        bl      2ac <output_7segment>
                delay();
3dc:   ebffffa1        bl      268 <delay>
                output_7segment((val / 10) % 10, 2);
3e0:   e51b1014        ldr     r1, [fp, #-20]
3e4:   e59f307c        ldr     r3, [pc, #124]  ; 468 <.text+0x468>
3e8:   e0c32391        smull   r2, r3, r1, r3
3ec:   e1a02143        mov     r2, r3, asr #2
3f0:   e1a03fc1        mov     r3, r1, asr #31
3f4:   e0631002        rsb     r1, r3, r2
3f8:   e59f3068        ldr     r3, [pc, #104]  ; 468 <.text+0x468>
3fc:   e0c32391        smull   r2, r3, r1, r3
400:   e1a02143        mov     r2, r3, asr #2
404:   e1a03fc1        mov     r3, r1, asr #31
408:   e0632002        rsb     r2, r3, r2
40c:   e50b2018        str     r2, [fp, #-24]
410:   e51b3018        ldr     r3, [fp, #-24]
414:   e1a03083        mov     r3, r3, lsl #1
418:   e1a02103        mov     r2, r3, lsl #2
41c:   e0833002        add     r3, r3, r2
420:   e0631001        rsb     r1, r3, r1
424:   e50b1018        str     r1, [fp, #-24]
428:   e51b0018        ldr     r0, [fp, #-24]
42c:   e3a01002        mov     r1, #2  ; 0x2
430:   ebffff9d        bl      2ac <output_7segment>
                delay();
434:   ebffff8b        bl      268 <delay>

                if (++divide == 100)
438:   e51b3010        ldr     r3, [fp, #-16]
43c:   e2833001        add     r3, r3, #1      ; 0x1
440:   e50b3010        str     r3, [fp, #-16]
444:   e51b3010        ldr     r3, [fp, #-16]
448:   e3530064        cmp     r3, #100        ; 0x64
44c:   1affffd2        bne     39c <main+0x28>
                {
                        divide = 0;
450:   e3a03000        mov     r3, #0  ; 0x0
454:    e50b3010        str     r3, [fp, #-16]
                        val += 1;
458:    e51b3014        ldr     r3, [fp, #-20]
45c:    e2833001        add     r3, r3, #1      ; 0x1
460:    e50b3014        str     r3, [fp, #-20]
                }
        }
464:    eaffffcc        b       39c <main+0x28>
```

Below is the result from letting OpenOCD analyze the captured trace data. The target was resumed at address 0x3e0 where the value of the second digit is calculated before `output_7segment()` gets called (address 0x430) to output the value on display two. The last two instructions show the C function prologue where the stack pointer is copied to the frame pointer and some registers are pushed on the stack.

```
> etm analyze
--- tracing enabled at 0x000003e0 ---
0x000003e0      0xe51b1014      LDR r1, [r11, #-0x14] (12 cycles)
0x000003e4      0xe59f307c      LDR r3, [r15, #0x7c] (3 cycles)
0x000003e8      0xe0c32391      SMULL r3, r2, r1, r3 (6 cycles)
0x000003ec      0xe1a02143      MOV r2, r3, ASR #0x2 (4 cycles)
0x000003f0      0xe1a03fc1      MOV r3, r1, ASR #0x1f (4 cycles)
0x000003f4      0xe0631002      RSB r1, r3, r2 (9 cycles)
0x000003f8      0xe59f3068      LDR r3, [r15, #0x68] (10 cycles)
0x000003fc      0xe0c32391      SMULL r3, r2, r1, r3 (6 cycles)
0x00000400      0xe1a02143      MOV r2, r3, ASR #0x2 (1 cycle)
0x00000404      0xe1a03fc1      MOV r3, r1, ASR #0x1f (2 cycles)
0x00000408      0xe0632002      RSB r2, r3, r2 (1 cycle)
0x0000040c      0xe50b2018      STR r2, [r11, #-0x18] (2 cycles)
0x00000410      0xe51b3018      LDR r3, [r11, #-0x18] (10 cycles)
0x00000414      0xe1a03083      MOV r3, r3, LSL #0x1 (1 cycle)
0x00000418      0xe1a02103      MOV r2, r3, LSL #0x2 (1 cycle)
0x0000041c      0xe0833002      ADD r3, r3, r2 (1 cycle)
0x00000420      0xe0631001      RSB r1, r3, r1 (1 cycle)
0x00000424      0xe50b1018      STR r1, [r11, #-0x18] (2 cycles)
0x00000428      0xe51b0018      LDR r0, [r11, #-0x18] (2 cycles)
0x0000042c      0xe3a01002      MOV r1, #0x2 (9 cycles)
0x00000430      0xebffff9d      BL 0x000002ac (3 cycles)
                                /* 90 cycles since tracing was enabled */
0x000002ac      0xe1a0c00d      MOV r12, r13 (1 cycle)
0x000002b0      0xe92dd800      STMDB r13!, {r11, r12, r14, r15} (5 cycles)
...
```

The same code was traced again, this time after removing waitstates for accesses to SRAM by clearing the *WSR_DTCM* and *WSR_AHB* bits in the system configuration register 0 (*SCU_SCR0*). The cycle times improved reproducibly, but only by two cycles for the code that executes between enabling the trace (address 0x3e0) and the branch to the `delay` routine at address 0x430. There are seven instructions that load or store data to and from memory, but only five of these access the SRAM, the other two are PC relative loads that read data from the literal pool that is embedded in the program code in flash memory. Two of those operations are stores that should already go through the write buffer, so one could expect to save three cycles in this part of the code. It is generally not possible to count on the exact cycle numbers reported for a given instruction, making it hard to see where the additional cycles come from, but it seems that at least one cycle that could have been saved by removing the waitstates was spent for a load-use interlock.

```
--- tracing enabled at 0x000003e0 ---
0x000003e0      0xe51b1014      LDR r1, [r11, #-0x14] (11 cycles)
0x000003e4      0xe59f307c      LDR r3, [r15, #0x7c] (10 cycles)
0x000003e8      0xe0c32391      SMULL r3, r2, r1, r3 (6 cycles)
0x000003ec      0xe1a02143      MOV r2, r3, ASR #0x2 (1 cycle)
0x000003f0      0xe1a03fc1      MOV r3, r1, ASR #0x1f (2 cycles)
0x000003f4      0xe0631002      RSB r1, r3, r2 (9 cycles)
0x000003f8      0xe59f3068      LDR r3, [r15, #0x68] (10 cycles)
0x000003fc      0xe0c32391      SMULL r3, r2, r1, r3 (6 cycles)
0x00000400      0xe1a02143      MOV r2, r3, ASR #0x2 (1 cycle)
0x00000404      0xe1a03fc1      MOV r3, r1, ASR #0x1f (2 cycles)
0x00000408      0xe0632002      RSB r2, r3, r2 (1 cycle)
0x0000040c      0xe50b2018      STR r2, [r11, #-0x18] (1 cycle)
0x00000410      0xe51b3018      LDR r3, [r11, #-0x18] (10 cycles)
0x00000414      0xe1a03083      MOV r3, r3, LSL #0x1 (1 cycle)
0x00000418      0xe1a02103      MOV r2, r3, LSL #0x2 (1 cycle)
0x0000041c      0xe0833002      ADD r3, r3, r2 (1 cycle)
0x00000420      0xe0631001      RSB r1, r3, r1 (1 cycle)
0x00000424      0xe50b1018      STR r1, [r11, #-0x18] (1 cycle)
0x00000428      0xe51b0018      LDR r0, [r11, #-0x18] (1 cycle)
0x0000042c      0xe3a01002      MOV r1, #0x2 (9 cycles)
0x00000430      0xebffff9d      BL 0x000002ac (3 cycles)
                                /* 88 cycles since tracing was enabled */
0x000002ac      0xe1a0c00d      MOV r12, r13 (1 cycle)
0x000002b0      0xe92dd800      STMDB r13!, {r11, r12, r14, r15} (5 cycles)
```

Using the information gained from the cycle accurate ETM trace a developer could start reworking the code to avoid load-use conflicts. The compiler provides switches like `-On` to optimize for speed (`n` = 1-3) or for code size (`n` = ’s’), and options that instruct it to generate code for a particular core, like `arm966e-s`. Timecritical part of the code could be further optimized using inline assembly to make use of knowledge about the target’s memory subsystem, like the STR912’s TCM, write buffer, and the exact cycle counts from the ETM Trace.

11 Conclusions

Three different trace variants have been analyzed as part of this master's thesis: Software trace that stores the trace information on the development host, the XScale's on-chip trace buffer, and flow trace using the ARM ETM (optionally in conjunction with the ETB).

As outlined in chapter 6, the software trace functionality offers an easy way for developers to observe their target's execution without seriously affecting realtime performance, as long as the rate of trace points is below the JTAG interface's polling frequency. The `DEBUG*` macros further allow the content of variables to be observed without having to halt the target in order to examine the memory content. The obvious drawback is the intrusiveness, because the approach requires the code from `dcc_debug.c` to be linked with the target code, and explicit calls to the trace and debug output functions.

The support code written for the XScale's on-chip trace buffer allows developers to trace the last instructions that executed before the target entered debug state. Alternatively the upcoming instruction flow can be observed by using fill-once mode, optionally for a longer period of time if repeated entries to debug state can be tolerated. The maximum depth of the 256-entry trace buffer heavily depends on the number of branches included in the traced code. A branch message plus the branch address requires five bytes, leaving room for only 51 branches in the buffer. If the maximum of 15 sequential instructions executes prior to every branch the buffer could trace 816 instructions. If the traced code consists exclusively of sequential statements with no intermediate branches, a maximum of 4096 instructions could be traced. Observing the accessed data isn't possible at all with the XScale's trace buffer. The lack of a trigger could be mitigated by using one of the two XScale breakpoints and a special debug handler that immediately resumes execution without reenabling trace (trace up to the trigger), or by enabling the trace in fill-once mode once the breakpoint was reached (trace from the trigger).

The large standard ETM9 r2p2 included in the LPC3180 in conjunction with the 2048 entry (8KB) ETB allows sophisticated filtering inside the ETM to be set up. The eight pairs of address decoders allow areas of interest to be included while less interesting parts like known-good libraries can be excluded from a trace run. The ability to trace every cycle executed inside the core provides a level of insight into potential performance bottlenecks that can't be achieved by merely looking at the instruction cycle count and memory system documentation. The small standard ETM implementations included in the LPC2000 series or the STR912 allows only limited trace filtering inside the ETM. A TPA with large buffers is required to be able to analyze the captured trace data later on the host. The OpenOCD+trace design with its 16MB trace buffer that is good for up to 16 million trace cycles allows large parts of a program to be traced, making it a suitable choice for tracing relatively low speed (< 100MHz), deeply embedded targets.

11.1 Further Development

The XScale trace buffer support could be improved to use target memory as a buffer to store trace data, thus mitigating the negative effects on real time performance. The necessary code would have to be included in the debug handler located in the mini ICache where tight resource constraints could require dynamic loading of debug handler functionality, which is why this wasn't included in the current XScale trace support implemented in the OpenOCD.

OpenOCD+trace, the ETM trace protocol analyzer (TPA) developed as part of this master's thesis, can only be a prototype for further work in this area. The Virtex-2 device used is rather expensive, and many of its capabilities wont be required when tracing hardware running at 100MHz or less. SDR SDRAM usage is declining, and often DDR or even DDR2 SDRAM is used with newer FPGA families. DDR RAM provides twice the bandwidth or alternatively allows RAM of only half the width to be used, offering either increased performance or reduced costs.

Currently, the ETM resources have to be programmed manually by accessing the ETM registers via OpenOCD's `reg` command. For targets supporting more complex ETM resources and events this can become inconvenient to use. A graphical user interface (GUI) would make the available ETM functionality more accessible.

Trace analysis would benefit from a GUI as well. Especially when using the OpenOCD+trace the amount of trace data that can be captured exceeds what can be reasonably displayed on a text interface by several orders of magnitude. With cycle accurate trace disabled the 16MB trace memory could hold trace data for several millions of traced instructions. A GUI could link a trace protocol which lists executed instructions with a data window that keeps track of accessed memory and a mixed source/disassembly window that shows the executed instructions in the context of surrounding code.

Additional functionality that could be implemented using the OpenOCD+trace's large trace buffer is code coverage analysis. The trace analysis would have to track every executed address to provide the developer with an overview about parts of target code that never executed, indicating unnecessary leftovers or dead ends in the code that are never reached, presumably indicating a bug or wrong assumptions about the input that is presented to the system.

Finally, the trace information generated by the different backends (software trace, XScale trace buffer, and ETM) could be combined in a generic trace representation that documents the flow of instructions. This level of abstraction would be required in order to implement a single GUI that works for all sources of trace information.

A Utilized Software

The following chapter is going to list the tools utilized and what they've been used for.

A.1 Development Platform

A Kubuntu 7.04 "Feisty Fawn" GNU/Linux system has been used for developing the software and FPGA design and for typesetting this document. The system ran Linux kernels up to 2.6.22-ck1, compiled from the sources available at `http://www.kernel.org` with patches aimed at improving desktop Linux performance from `http://members.optusnet.com.au/ckolivas/kernel/`. The K Desktop Environment (KDE) (`http://www.kde.org`), Release 3.5.7, served as the desktop environment.

Eclipse 3.3 "Europa" and the C/C++ development tools (CDT) version 4.0 were used as the integrated development environment with the GNU autotools (autoconf, automake) providing the underlying build infrastructure for the OpenOCD.

The Xilinx ISE WebPack 9.1 was used for developing the OpenOCD+trace FPGA design, including synthesis and implementation of the design inside the FPGA.

A.2 Typesetting

This entire document was typeset using teTeX (`http://www.tug.org/teTeX/`), a TEX distribution for Unix systems, consisting only of free software.

Instead of one of the document classes that come with teTeX, the Memoir class by Peter Wilson has been used. Memoir is a flexible class for typesetting general fiction, non-fiction and mathematical works as books, reports, articles or manuscripts. Memoir is available from the Comprehensive TeX Archive Network (CTAN) at `http://www.ctan.org/tex-archive/macros/latex/contrib/memoir/`.

The glossary package which is available at `http://www.ctan.org/tex-archive/macros/latex/contrib/glossary/?action=/tex-archive/macros/latex/contrib/`) was used to create the glossary and to maintain abbreviations used in the text. All listings have been typeset with the help of the listings package (part of teTeX).

A.3 Figures

All figures have been created using Xfig, a drawing program for the X Window System. Xfig offers built-in support for integrating figures within TEX documents by exporting them to combined Postscript/LaTeX and PDF/LaTeX formats. This combines TeX's typesetting flexibility with the drawing capabilities offered by Xfig.

B Source Code

The printed edition of the master's thesis includes a CD-ROM with the source code of the Open On-Chip Debugger, the VHDL and project files of the OpenOCD+trace capture design, the ARM7/9 DCC target code, and the test programs used to verfiy correct operation of the trace capture and analysis software.

The following directories can be found on the CD-ROM:

- `dcc_debug` - The target code required for the OpenOCD DCC target request and trace functionality.
- `openocd` - The current OpenOCD source code (Revision 207).
- `test_programs`
  - `lpc3180_gcc_blinky_dccdebug` - Example code demonstrating the use of DCC target request and trace functionality.
  - `lpc3180_gcc_blinky_sdram_cached_abort` - The example that generated the example ETM trace capture in subsection 8.5.
  - `str912_hitex_blink` - The example that generate the example ETM trace capture in section 10.6.
- `trace_capture` - The Xilinx ISE WebPack 9.1 project with the OpenOCD+trace design.

The latest version of the OpenOCD source code can also be obtained from the project's SVN repository, see chapter 5 for details.

C GNU Free Documentation License

Glossary

AMBA Advanced High-Performance Bus	The AHB is a high-bandwidth bus used to connect the ARM core to on-chip peripherals, DMA controllers, internal memories and other high-speed resources.
Background Debug Mode	The debug interface found for example on Freescale (ex. Motorola) 68000 family cores.
Common On-Chip Processor	The debug interface used by IBM and Freescale (ex. Motorola) "Power" designs like the PowerPC and PowerQUICC families. COP can be configured to operate in a IEEE1149.1 (JTAG) compatible mode.
Debug Communication Channel	Allows communication between an ARM7/ARM9 based target and a debug host during target program execution through a JTAG connection.
Debug Control and Status Register	A register inside the XScale's debug hardware to control debug functionality.
Double Data Rate	A technique used with memories or data buses where data is transferred on both edges of a clock signal, doubling the data rate while keeping the clock signal transition rate constant.
Electronic Design Automation	Electronic design automation refers to methods used to enter and process digital and analog electronic designs for simulation and implementation.
Embedded System	A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. Contrast with general-purpose computer.

Embedded Trace Buffer	An on-chip buffer that stores data from an ETB in RAM for later retrieval via JTAG [DDI0242B].
Embedded Trace Macrocell	A hardware macrocell that outputs instruction and data trace information on a trace port [IHI0014N].
Flash memory	A type of non-volatile memory often segmented into blocks that can be individually erased and re-programmed.
GNU Compiler Collection	The GNU compiler collection is a suite of compiler for several programming languages such as C and C++. GCC was created by the GNU project and is released under the terms of the GNU GPL.
GNU General Public License	The GNU GPL is a free software license originally written by Richard Stallman.
In-Circuit Emulator	Debug hardware that connects to a target system instead of the original microcontroller.
Jazelle	The Jazelle Java acceleration technology speeds up processing of Java bytecode by executing most Java instructions directly in hardware, without Emulation using a virtual machine.
JTAG	Joint Test Access Group, but commonly used to describe the IEEE Standard Test Access Port and Boundary-Scan Architecture, IEEE 1149.1 [IEEE1149].
Random Access Memory	Memory which can be read and written without restrictions on the number of read and write operations or the order of successive operations.
Read Only Memory	Memory with fixed content, which can be read but not written.
SBC	Single board computer. A complete computer system implemented on a single printed circuit board, consisting of a microprocessor together with memory, storage, communication interfaces and other peripherals.
Schmitt Trigger	A comparator circuit used in electronics. The Schmitt trigger implements two different thresholds, an upper threshold that must be reached for the output to become high, and another, lower threshold for when the output should go low again.

Single Data Rate	Used as a disambiguation to avoid confusion with double data rate (DDR) interfaces, especially between SDR SDRAM and DDR SDRAM. SDR is a synchronous design technique where data is transferred on a single clock edge only.
Special Debug State	XScale debugging in halt mode enters an additional core mode "DEBUG" when a debug exception occurred.
Semihosting	Some ARM debuggers support a feature known as semihosting to enable a target system which doesn't support various features required by the ANSI C library to use the features of the host instead [ARMSEMI].
Software Interrupt	A software-generated interrupt, often used to call system functions from user space.
Test Access Port	A general-purpose port that can give access to many test support functions built into a component [IEEE1149, p. 17]. Defined by the IEEE standard 1149.1. Proposed by the Joint Test Access Group as a way to test component functionality, component interconnections, and component interaction.
Thumb	Thumb is a compressed 16 bit instruction set extension available on all current ARM7 and ARM9 family cores. It works with the full 32-bit length of ARM registers, but limits access to eight general purpose registers. The remaining registers may be accessed using special transfer instructions, but not with general data processing instructions.
Thumb-2	Thumb-2 is a new instruction set available since ARMv6 (optional) and ARMv7 (obligatory). Thumb-2 combines 16 bit Thumb instructions with 32 bit instructions for improved performance while maintaining Thumb's high code density.
Trace Port Analyzer	A trace port analyzer captures the data from an ETM's trace port. A TPA may be specifically designed for tracing but it could also be a general purpose logic analyzer.
USB VID/PID	The USB vendor ID (VID) and product ID (PID) allow automatic identification of devices connected to a USB host.

Yet Another GNU Toolchain	Yagarto is a precompiled GNU crosscompilation toolchain for ARM based targets that runs on the Microsoft Windows operating system.

Bibliography

[ARMSEMI] ARM Ltd. *Semihosting* Available from `http://www.arm.com/products/DevTools/Semihosting.html`

[DDI0100E] ARM Ltd., David Seal *ARM Architecture Reference Manual* Addison-Wesley Professional 2000 2nd edition

[DDI0242B] ARM Ltd. *Embedded Trace Buffer TRM* Available from `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0242b/DDI0242.pdf`, and on the ARM Technical Publications CD 2002

[DLP02] Douglas L. Perry *VHDL : Programming By Example* McGraw-Hill Professional ISBN-13: 978-0071400701 2002

[DR05] Dominic Rath *Design and Implementation of an On-Chip Debug Solution for Embedded Target Systems based on the ARM7 and ARM9 Family* Available from `http://www.fh-augsburg.de/~hhoegl/da/da-25/thesis.pdf` 2005

[EADIP204] Electronic Assembly *EA DIP204-4 Datasheet* Available from `http://lcd-module.de/deu/pdf/doma/dip204-4.pdf` 2005

[FPGA01] FPGA and Programmable Logic Journal, Kevin Morris *Terminology Tango 101. From Dog Gates to Marketing Megahertz* Available from `http://www.fpgajournal.com/articles/20040706_tango.htm` 2004

[FTDI01] Future Technology Devices International Limited *FT245BM USB FIFO Datasheet* Available from `http://ftdichip.com/Documents/DataSheets/DS_FT245BM.pdf` 2005 v1.7

[IEEE1046] IEEE *IEEE Standard 1046-2002 VHDL Language Reference Manual* Available as Print (ISBN 0-7381-3247-0) or PDF (ISBN 0-7381-3248-9) edition, 2002

[IEEE1046.6] IEEE *IEEE Standard 1046.6-1999 for VHDL Register Transfer Level (RTL) Synthesis* Available as Print (ISBN 0-7381-1819-2) or PDF (ISBN 0-7381-1820-6) edition, 1999

[IEEE1149] IEEE *IEEE Standard 1149.1-2001 Test Access Port and Boundary-Scan Architecture* Available as Print (ISBN 0-7381-2944-5) or PDF (ISBN 0-7381-2945-3) edition, 2001

[IHI0014N] ARM Ltd. *Embedded Trace Macrocell Architecture Specification* Available from `http://www.arm.com/pdfs/IHI0014N_etm_v34_architecture_spec.pdf`, and on the ARM Technical Publications CD 2006

[INTEL278796] Intel Corporation *Intel XScale Microarchitecture for the PXA255 Processor - User's Manual* 2003

[MICRON01] Micron Technology, Inc. *128Mb: x4, x8, x16 SDRAM* Available from `http://download.micron.com/pdf/datasheets/dram/sdram/128MSDRAM.pdf` 2001 Rev. K

[UM10198] Philips Semiconductors (now NXP) *LPC3180 User Manual* Available from `http://www.nxp.com/` 6 June 2006 Rev. 01

[XILDS031] Xilinx Inc. *Virtex-II Platform FPGAs: Complete Data Sheet* Available from `http://direct.xilinx.com/bvdocs/publications/ds031.pdf` 2005 v3.4

[XILUG02] Xilinx Inc. *Virtex-II Platform FPGA User Guide* Available from `http://direct.xilinx.com/bvdocs/userguides/ug002.pdf` 2007 v2.1

Index