Open On-Chip Debugger

Free and Open On-Chip Debugging, In-System Programming and Boundary-Scan Testing

Dominic Rath <Dominic.Rath AT gmx.de>

Open On-Chip Debugger

Free and Open On-Chip Debugging, In-System Programming and Boundary-Scan Testing

Dominic Rath

10 09 08 07 06 05 6 5 4 3 2 1

First edition:	11 July 2005
Second impression, with corrections:	18 July 2005
Second edition:	02 January 2006

Contents

List of Figures

List of Tables

1 Embedded Systems Debugging

Embedded systems in general, and ARM based system-on-chip (SOC) designs in particular, have seen an immense growth during the past years, with free and open software becoming an integral part of embedded systems development. A survey ran by linuxdevices.com [**?**] shows that 43% of the participants have used embedded Linux in their, or their companies, products, and 55% expect to do so within the next two years. The processor architecture used in most designs is ARM, being used by 30% of the developers, prior to x86 which has been used by 28%. Open source tools are the first choice for 59% of the participants, and more than 82% believe the tools available for embedded Linux development are either very good or acceptable.
While free and open source projects offer a high-quality toolchain for ARM development, debugging support is lacking behind, especially as far as system programming is concerned. The GNU Debugger (gdb) offers excellent debugging support, but covers only some areas of embedded systems debugging. Low level tasks require additional hard- and software, and existing open source solutions for these tasks are incomplete or at least partially deficient.
The goal of this diploma thesis is the design and implementation of a free solution for debugging of ARM7 and ARM9 family based SOC designs, making the use of proprietary commercial tools obsolete. The software written as part of this work is initially going to have support for selected members of these processor families, but extensibility to additional cores shall be simplified by an appropriate architectural design. The target interface will be based upon the IEEE Standard Test Access Port and Boundary-Scan Architecture [**?**]. Support for different interfaces between a host PC and an IEEE 1149.1 compatible target is an expressed goal.

Debugging embedded systems is different in many aspects from traditional application debugging. Compared to desktop systems, embedded systems have limited resources, such as main memory, processing power, or input and output capabilities. This makes it inconvenient or even impossible to run a software debugger together with the debuggee on the same system. Depending on the development task, there might be no software running on the target at all, like during bootloader development. In that case, there is usually no debugger on the system, too. During application development, the hardware is expected to be error-free, meaning that every subsystem (CPU, memory, storage, I/O) actually works. On embedded systems, the hardware itself could have errors, like an instable memory interface or untested system components. If the memory system is faulty or just untested, any code could fail, even a debugger.
Because of these restrictions, embedded systems are usually debugged using remote debugging: The debugger is running on a host computer, and controls the target either through hardware, or through a small software running on the target. It's possible for the developer to make use of all the comfort his workstation offers, while the target doesn't have to run a full featured debugger.
The purpose of debugging is to identify and remove defects in software programs. This can be achieved

by either passively watching the code-, and possibly the data flow, or by actively stopping the target at the point of interest. Passive debugging has the advantage of being non-intrusive, and allows program flow and timing to be inspected, while active debugging enables the developer to control the program flow, or alter the contents of target memory.
Some years ago, program code for embedded systems had to be loaded onto memory chips using external programmers. The chips had to be removed from the target, programmed, and put back into the system. Being able to download program code from the host to a target system while this is running greatly simplifies embedded systems development.

The following sections are meant to show some commonly used solutions for embedded debugging in genearal, and an overview of currently available solutions for ARM7/ARM9 debugging in particular. [**?**] gives an detailed review of the challenges of embedded systems design, the implications for debugging, and the various debug solutions used in embedded systems design.

1.1 Debug Solutions

Logic Analyzers, Trace Hardware

Logic analyzers and dedicated trace hardware, like the *ARM embedded trace macrocell* (ETM) [**?**], allow the program flow to be passively monitored. Logic analyzers monitor the target's data and address bus, and usually generate a listing of executed instructions, possibly annotated with the data accessed. While this works for older microcontrollers, where every instruction executed results in an access to the memory system, it's not possible to trace execution within modern cached architectures. Instructions contained inside the cache wont show up on the memory interface, making a complete trace impossible. Dedicated trace hardware is tightly coupled to the microcontroller core, and keeps track of every instruction executed, without having to rely on the memory interface. The amount of raw data can grow rapidly on systems with high clock rates, making it difficult to get the information out of the target, and hard to find the relevant parts. Advanced solutions like the ETM9 allow triggerpoints, for example instruction addresses, to be defined, at which the trace hardware starts monitoring the core. Filters further limit the amount of data that has to be transfered from the core to the debugging host.

In-Circuit Emulators, ROM 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's

possible to load code into the target, replacing instructions contained in ROM regions. The ICE watches the accessed memory space, and switches to it's included RAM when an access to overlayed memory occurs.

Debug Stubs

Debug stubs, often called "debug monitors", run on the target system, and connect to a host computer running the debug software. They require working initialization code, that sets up the target clocks, main memory, and a communication channel. This makes a Debug stub unsuitable for early development stages, where initialization code has to be debugged itself.
The stub utilizes an interrupt on the target to take control over program execution, a stub using RS232 communication for example would use the serial interrupt vector. When the host debugger sends data to the debugging stub, an interrupt is generated, giving the stub control over the target. The stub uses some kind of TRAP or BREAKPOINT instruction, or a SWI to replace breakpointed instructions. Once the target hits one of the breakpointed instructions, control is given to the debugging stub, which can inform the debugging host about the breakpoint.
The required initialization and the use of target resources are a major drawback of debugging stubs, but they require only little extra hardware, making them interesting for situations where development tools cost is important.

Integrated Debug Circuitry, On-Chip Debug

Integrated debug circuitry gives the power of in-circuit emulators at a much higher flexibility. Instead of having to replace the target's microcontroller with a special debug version, every chip shipped contains the debug functionality. A serial communication channel, able to operate at high clock speeds, is used to connect the debug circuitry to a host debugger, allowing low pin-count debug connections.

1.2 ARM Debug Solutions

Due to the popularity of SOC designs based on the ARM7 and ARM9 family there several vendors who provide tools to work with the integrated debug circuitry included in all ARM7 and ARM9 based microcontrollers. There are commercial tools available as well as free and open source implementations, offering a wide range of supported functionality. Some are offered as combined solutions, where hardware that interfaces between a host PC and the debug target comes together with debugging software, while others are pure hardware solutions, that can be combined with various software products. Here, debug software doesn't necessarily mean a full-featured debugger, but rather software that talks to the hardware, providing a set of debug functions to a debugging frontend. The following overview isn't meant as a comprehensive listing, but rather to show a few typical designs.

Combined Hardware and Software Solutions

ARM Multi-ICE

`http://www.arm.com/products/DevTools/MultiICE.html`
The *ARM Multi-ICE* allows debugging of a wide variety of ARM based cores and supports all possible functionality, including access to special system-control registers, semihosting and flash programming. It connects to the host computer using a PC parallel port and accesses the target with a JTAG clock of

up to 10 MHz. It comes together with a software called the *Multi-ICE server*, which contains the target specific debug functionality and provides the remote debugging interface (RDI) to a debugger frontend. The Multi-ICE server software requires a PC running a version of Microsoft Windows. Linux or other free operating systems are not supported.

ARM RealView RVI

`http://www.arm.com/products/DevTools/RVI.html`
The *ARM RealView ICE* supercedes the Multi-ICE, providing JTAG clock rates of up to 50 MHz, larger cable lengths, and host connection via Ethernet or Universal Serial Bus (USB), giving greater flexibility. The RVI contains an ARM9 processor which takes care of all the target specific debug functionality. It requires the *RealView Debugger (RVD)* as a frontend, which is available for Microsoft Windows, Linux and Solaris.

Abatron BDI2000

`http://www.abatron.ch/`
The *BDI2000* connects to a host computer via RS232 or 10BASE-T Ethernet, and supports JTAG clock rates up to 16 MHz. The hardware can be configured for a wide variety of target systems, including a configuration for ARM7 and ARM9 targets. The target specific debug functionality is contained inside the BDI2000. Various debugger frontends are supported, like *ARM RealView Tools*, *Metrowerks CodeWarrior* or the free GNU Project Debugger (GDB). An additional telnet interface provides direct access to hardware-specific debug functions. The BDI2000 gives access to almost all possible debug functionality, including system-control registers.

Hardware-only Solutions

Macraigor Wiggler, Parallel Port Wigglers

`http://www.macraigor.com/wiggler.htm`
The *Macraigor Wiggler* is a simple device that connects a PC parallel port to the target JTAG interface. The host PC simulates the target interface by switching signals on and off, a technique often called "bit-banging". It acts as a signal buffer, providing necessary level translation between the PC (5V Transistor-Transistor Logic (TTL)) and the target (1.5V...5V). Schematics for wiggler-compatible clones are freely available on the net and can be adjusted to special requirements, like adding or removing signals that are optional in the JTAG standard, but required by some targets. Figure 1.1 shows the functional equivalent of a complete wiggler clone. The Wiggler's speed is limited by the PC's parallel port, which requires a minimum of about 1 μs per `in` or `out` instruction [**?**]. A complete clock cycle requires at least 2 μs, limiting the maximum frequency to 500 kHz.

Macraigor Raven

`http://www.macraigor.com/raven.htm`
Like a Wiggler, the *Macraigor Raven* connects to the host using a PC parallel port, but it uses the enhanced parallel port (EPP) protocol and higher-level commands. Logic inside the Raven translates the parallel data from the host to a serial bitstream at up to 8 MHz. The internal design of the Raven is proprietary, schematics are not available. Binary drivers are available for Microsoft Windows and Linux.

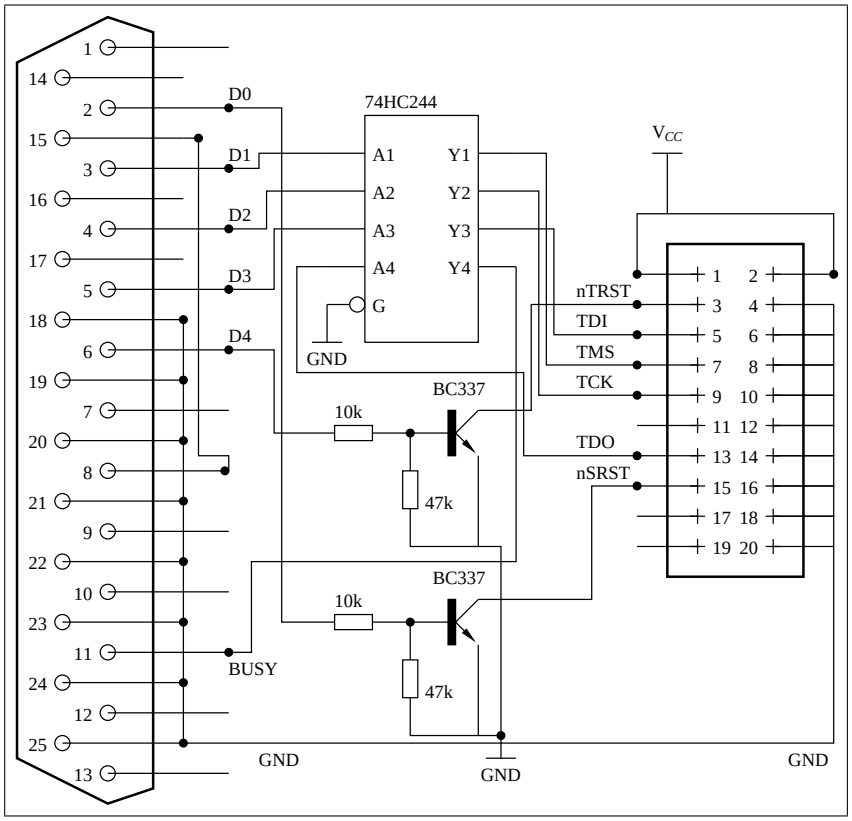

Figure 1.1: Wiggler schematics

Amontec Chameleon POD

`http://www.amontec.com/chameleon.shtml`
The *Amontec Chameleon POD* is based on a Xilinx Coolrunner (XPLA3) XCR3128XL-VQ100 Complex Programmable Logic Device (CPLD) (`http://www.xilinx.com`). It connects to the host using a PC parallel port, and supports many different configurations, including emulations of the Macraigor Wiggler and Raven. The configurations, that may be downloaded for free, are programmed into the Chameleon using proprietary software available only for Microsoft Windows.

USBJTAG-1

The device designed by Hubert Högl around a FTDI2232C (`http://www.ftdichip.com/FTProducts.htm#FT2232C`), connects to a host PC using a USB 1.1 Full-Speed (11 Mbit) interface. Using its Multi-Protocol Synchronous Serial Engine (MPSSE) [**?**], the FTDI chip is capable of JTAG clocks between 6 MHz and 93 Hz.

Figure 1.2 shows an example implementation using the DLP-2232M evaluation kit, available at `http://www.ftdichip.com/Products/EvaluationKits/DIPModules.htm#DLP-2232M`. The evaluation kit contains all circuitry necessary for the USB functionality, making it ideally suited for prototyping.

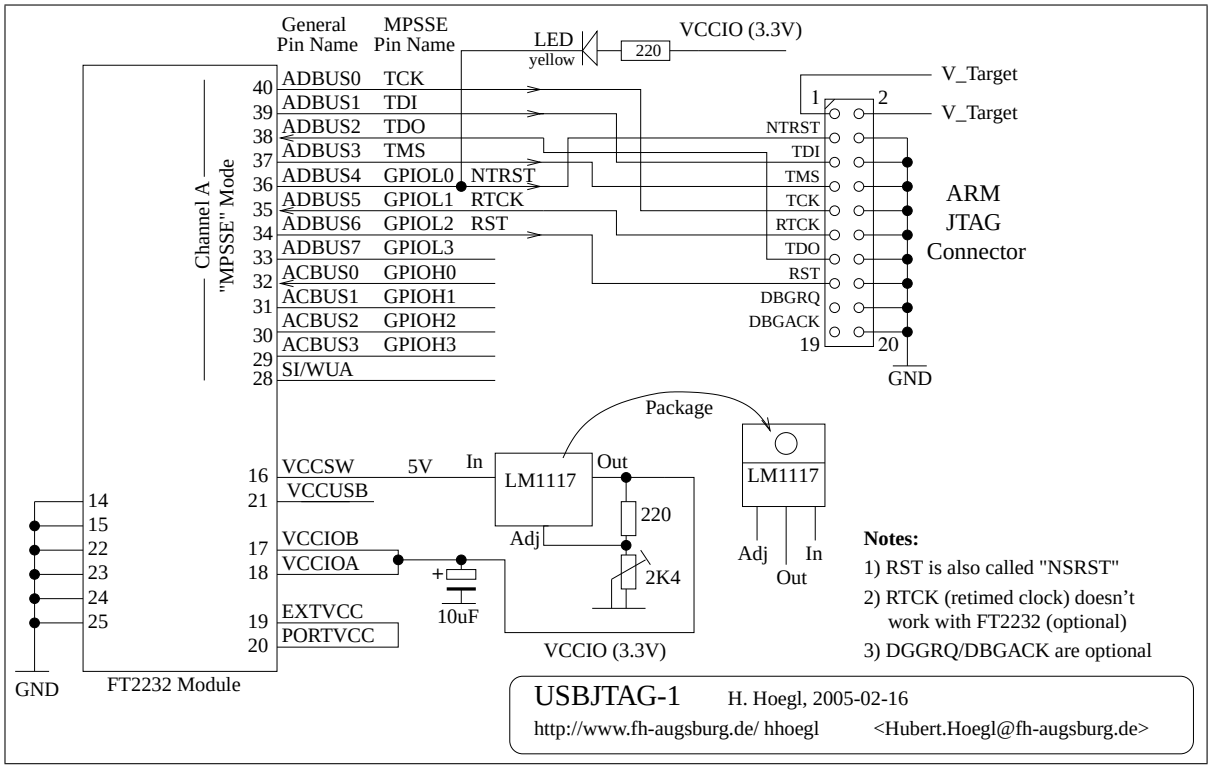

Figure 1.2: USBJTAG-1 schematics

Software Solutions

Macraigor OCD Commander, OCDRemote

`http://www.macraigor.com/ocd_cmd.htm`, `http://www.macraigor.com/full_gnu.htm`
The Macraigor software packages are intended to be used with the Macraigor Wiggler, Raven, usbDemon and mpDemon devices, but it's possible to use the software licensed as freeware together with Wiggler and Raven clones, too. Supported cores include many ARM7 and ARM9 members, as well as MIPS, Motorola PowerPC, and Intel XScale. Access to special registers is limited. There's no information available about the extent of cache and MMU handling. The OCD Commander is a complete graphical debugger, while the OCDRemote is a console application that interfaces the Macraigor hardware with the a GDB client. Both are available for Microsoft Windows and Linux. The Linux software comes with binary-only objectfiles that are linked into the provided kernel module. Writing Flash memory is not possible using the freeware programs, but Macraigor offers a (non-free) software called "Flash Programmer" that is able to program flash chips on ARM7 and ARM9 targets.

Open Source Software

There are a few open source projects to support ARM7 and ARM9 debugging, all licensed under the GNU General Public License (GPL). Functionality is limited compared to the available commercial solutions, and all but the gdb-jtag-arm seem to be unmaintained or no longer under active development. The

only JTAG hardware interfaces supported are Wiggler and compatibles. None of the projects provides handling of the MMU or caches found on cores like the ARM720t or ARM920t.

- JTAGER by Rongkai Zhan
  `http://jtager.sourceforge.net/`
  JTAGER supports ARM7TDMI, ARM720t, and ARM920t based targets. Flash memory write support is included for SST39LF/VF160 (`http://www.sst.com/`) and MBM29LV650 (`http://www.spansion.com/`) chips. A command line interface is implemented for user interaction, GDB support isn't included. Version 0.3.0 was released on October, 17th 2004, and is still in an early development stage. Bugs and shortfalls of the code can lead to target system crashes and memory inconsistencies.

- armtool by Erwin Authried (part of Midori Linux)
  `http://home.at/cgi-bin/viewcvs.cgi/midori/sources/armtool/`
  Armtool only supports ARM7TDMI based targets, is able to read and write target memory, and allows code downloaded to the target to be executed. It's suitable for batch usage, but doesn't allow user interaction, neither through a command line interface nor using a debugging frontend. Flash support isn't included.

- jtag-arm9 by Simon Wood
  `http://jtag-arm9.sourceforge.net/`
  Jtag-arm9 only supports ARM9 based targets, and contains a command line interface for user interaction. It is able to halt and resume the target, read and modify target registers, and supports memory read and write operations. Flash support isn't included.

- gdb-jtag-arm by Tobias Lorenz
  `http://gdb-jtag-arm.sourceforge.net/`
  Gdb-jtag-arm is the only open source project that supports the GNU Debugger (gdb) as a debugging frontend. It is based on jtag-arm9, fixing some, but not all, of the original software's bugs. Target system crashes or failures writing target memory can result from these defects.

2 IEEE 1149 - JTAG

The Joint Test Access Group (JTAG) was formed in 1985 to create printed circuit board (PCB) and integrated circuit (IC) test standards. The latest version of their proposal was approved by the Institute of Electrical and Electronics Engineers (IEEE) as IEEE Std. 1149.1-2001 [**?**], *IEEE Standard Test Access Port and Boundary-Scan Architecture*. The standard was created to support testing of component functionality, component interconnections and component interaction on assembled products. Subsequently, the term JTAG shall refer to the mentioned IEEE standard unless otherwise noted. This chapter is intended to give the reader enough understanding of the standard necessary for the operation of a JTAG based debugger. The main objective is therefor the design of a bus master, not the connected devices.

2.1 Test Logic

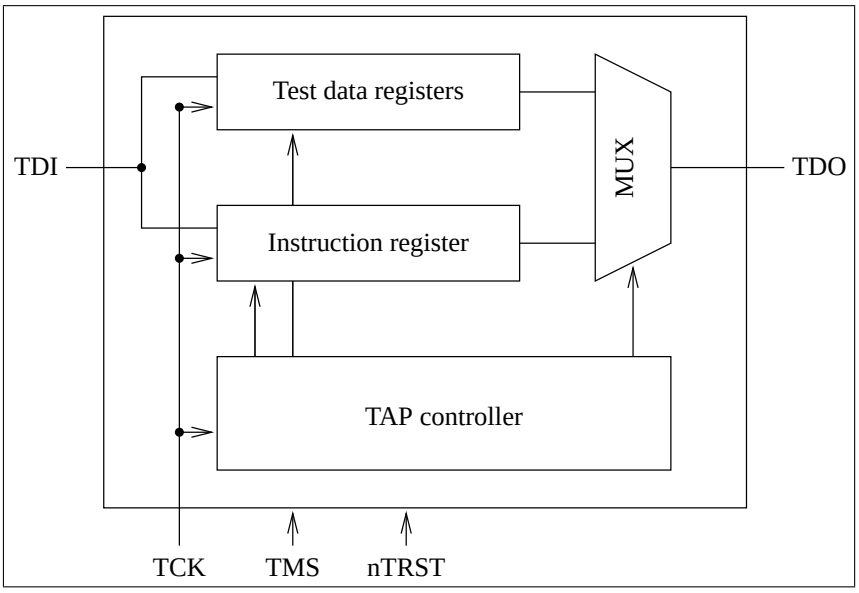

Figure 2.1: Test logic

A device that conforms to the JTAG standard contains one instruction register (IR), a number of test data registers (DR) and a test access port (TAP) controller which handles all test operations. The boundary-scan technique uses scan cells connected to a component's inputs and outputs, forming a serial

shift register. Test patterns are shifted (or "scanned") by a TAP bus master through the TAP into a component and apply known values to the scan cells. The scan cells' previous content is shifted out of the component and can be captured by the TAP bus master. The instruction register is required to be at least two bits long, and is used to control test functionality. The data registers are specific to a particular device, but the standard demands at least two register, a one-bit long bypass register, and a boundary-scan register. Figure 2.2 shows two components containing test logic like the conceptual example shown in Figure 2.1 connected to a single bus master.

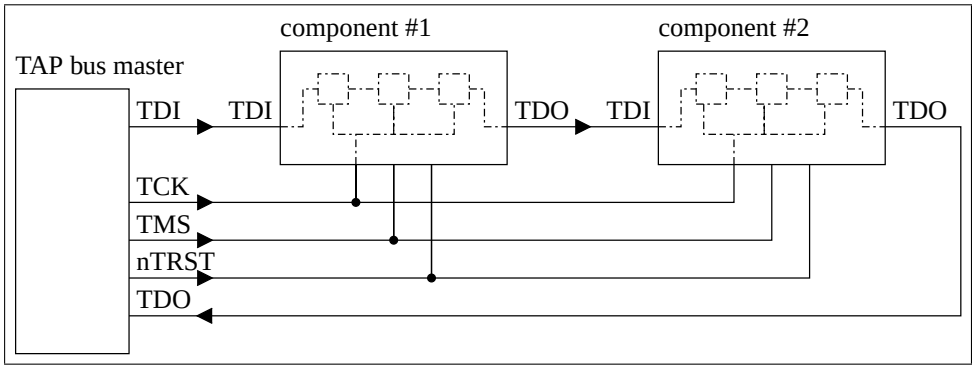

Figure 2.2: JTAG TAP / Boundary-Scan Architecture

2.2 JTAG/TAP Signals

Table 2.1: JTAG interface signals

Name (abbreviation)	Description	Direction
Test Clock (TCK)	Serial clock signal	out
Test Mode Select (TMS)	Controls movement of the JTAG state machine	out
Test Data Input (TDI)	Serial data fed into tested equipment	out
Test Data Output (TDO)	Serial data read back from tested equipment	in
Test Reset (nTRST)	Optional signal to asynchronously initialize test equipment	out

Table 2.1 shows the signals defined by the JTAG standard and their direction from the bus master's point of view. The TCK signal allows data to be scanned into multiple components independently from component specific system clocks. TCK may be stopped at 0 for an indefinite time, while test components are guaranteed to retain their current state, but not necessarily stopped at 1, which is permissible but not required by the standard.
The TMS signal selects the path taken in the JTAG state machine (see Figure 2.3). This signal is sampled at the rising edge of TCK, and is expected to be changed by the TAP bus master on the falling edge of TCK. The state machine is designed in a way that allows the Test-Logic-Reset state to be reached after five TCK cycles with TMS held high from every possible state. The standard requires circuitry to apply a logic 1 to TMS when the signal is undriven, ensuring normal operation when no test equipment is connected.

TDI transmits serial data shifted from the bus master to connected TAP controllers. Like TMS it is sampled on the rising edge of TCK, and in case of an undriven signal circuitry shall apply a logic 1 to TDI, too. Serial data from connected TAP controllers is shifted out of a component using the TDO signal. It changes its state on the falling edge of TCK, and should be sampled by the bus master on the rising edge of TCK.

2.3 JTAG State Machine

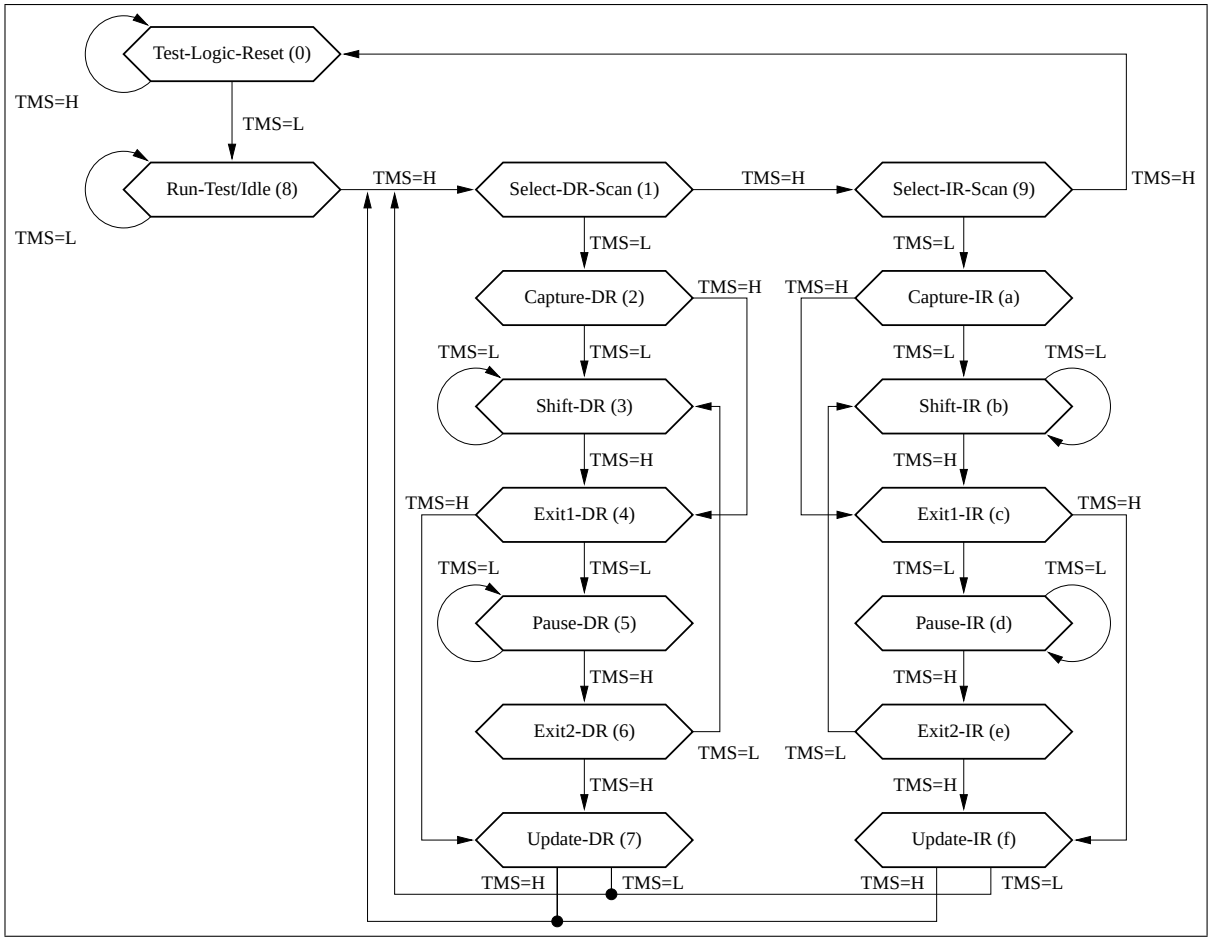

Figure 2.3: JTAG state machine

All JTAG operations are controlled through a state machine implemented in the TAP controller. The state machine is driven by TMS and is clocked by the rising edge of TCK. When a test session is initiated, the bus master has to initialize all connected TAP controllers by putting them into Test-Logic-Reset (TLR) state. TLR is reached by either forcing nTRST low or by executing five TCK cycles with TMS kept high. Once in TLR, the device identification register (IDCODE) or the bypass register (BYPASS) is selected, and all test functionality is reset. If TMS is low on a rising edge of TCK in TLR, the state

machine enters the Run-Test/Idle state. Depending on the currently selected instruction, test operations can be executed in the (RTI) state, or the test logic is left idle, and no operations occur. From RTI, Select-DR-Scan (SDS) is reached. SDS is, like Select-IR-Scan (SIS), Exit1-DR/IR (E1D, E1I) and Exit2-DR/IR (E2D, E2I), a temporary states where no test operations occur, used to select different paths through the state machine. In Capture-DR (CD), the currently selected test data register may be parallel loaded if appropriate, or left unchanged if the register doesn't have a parallel input or if the current instruction doesn't require the current value to be captured. During Capture-IR (CI), a fixed value of b01 is loaded into the least significant two bits of the IR, and design specific values may be put into any remaining IR bits. Once Shift-DR or Shift-IR is reached, the TAP bus master takes TMS low and starts outputting the desired value on each falling edge of TCK. The device under test will sample TDI on the rising edge and stay in Shift-IR while TMS is kept low. Pause-DR/IR may be used to indefinitely idle during - or between - scan operations. No test logic operations occur while a TAP controller is in Pause state. On the falling edge of TCK in Update-DR, the current value of the serial shift register is latched onto parallel outputs, if this is required for the currently selected test data register. Similarly, the current value of the instruction serial shift register is latched onto parallel outputs on the falling edge of TCK in Update-IR. Latching a new value on the IR parallel outputs makes this value the new current instruction.
Figure 2.4 shows an example session where a value of b0100 is scanned into a 4-bit long instruction register. During Capture-IR, a value of b0001 was loaded into the IR and can be captured by the bus master during the scan. Data register scans are similar, only a different path in the state machine is taken. The data register accessed depends on the current value of the instruction register and possibly on test operations executed earlier.

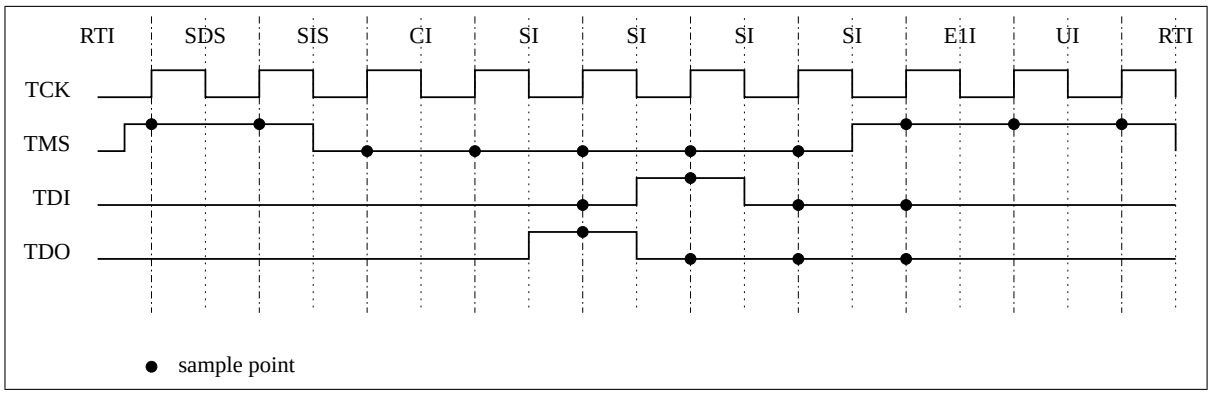

Figure 2.4: JTAG signals and states

2.4 JTAG Instructions

The JTAG standard requires several instructions to be available on a device compliant to standard IEEE 1149.1, but most of these are unimportant for the purpose of ARM debugging. The following instructions are used in debugging ARM7/ARM9 based systems:

Table 2.2: JTAG instructions (subset)

Name	Description
BYPASS	When the BYPASS instruction is selected on a device, the 1-bit wide bypass register is connected as the current test data register. This allows the scan chain in configurations with multiple successive devices to be shortened, making accesses faster. A device in BYPASS mode should not perform any test operation. One binary code for BYPASS shall be all ones (e.g. b1111 or 0xf for a device with a 4-bit wide instruction register), but additional codes may map to BYPASS, too.
EXTEST	The mandatory EXTEST instruction selects the boundary-scan register as the current test data register. Signals that are driven from outside of the component are loaded into the boundary-scan register during the falling edge of TCK in Capture-DR state, and signals that are driven from the component are loaded from the boundary-scan register on the falling edge of TCK in Update-DR state. This allows signals from the system to the component to be captured, and known values to be applied to signals driven from the component to the system. The binary code of the EXTEST instruction may be chosen by the component designer.
INTEST	The optional INTEST instruction also selects the boundary-scan register, but is used to capture signals driven out of the component, and known values to be applied to signals driven into the component. The binary code of the INTEST instruction may be chosen by the component designer.
IDCODE	IDCODE is an optional instruction that selects a device identification register as the current test data register. While IDCODE is selected, no other test data register shall be selected. The binary code of the IDCODE instruction may be chosen by the component designer.

3 ARM7 / ARM9 Architecture

This aim of this chapter is to describe the architecture implemented by ARM7 and ARM9 family targets, with a focus on aspects relevant for a debugger. These two core families share a great deal of debug functionality, making it possible to support both with a single debug solution.

3.1 Core Families

Currently available ARM7 family members, the ARM7TDMI, ARM710T, ARM720T, and ARM740T, are based on an ARM7TDMI core, with the exception of the ARM720T Rev 4, which is based on an ARM7TDMI-S synthesizable core. Older ARM7 members like the ARM700 or ARM750 are beyond the scope of this work and thus, ARM7 shall only refer to the above mentioned cores for the remainder of this document. The ARM9 family is based on the ARM9TDMI core, which is not available separately, but only as part of an ARM920T, ARM922T or ARM940T. Other ARM9 cores, like the ARM926EJ-S, ARM946E-S and ARM966E-S are based on the synthesizable ARM9E-S or ARM9EJ-S core, and contain slightly different debug functionality. This document is going to indicate these differences, but the prototype software resulting from this diploma thesis will be limited to ARM7TDMI, ARM720T and ARM920T cores. ARM9 cores with the letter E form a family of their own, but for the purposes of this document ARM9 shall refer to both families. See Table 3.1 for a list of letters used in ARM core and architecture names and their meaning.

The ARM architecture version implemented by the ARM7TDMI and ARM9TDMI based cores is

Table 3.1: 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

ARMv4T, while the newer ARM9E(J)-S based cores implement ARMv5TE or ARMv5TEJ. The major difference between the two architecture versions is the support of the ARM 'Enhanced' DSP instruction set, which is available on all ARMv5 cores, and the Jazelle Java acceleration technology, available only on ARMv5TEJ cores. From the debugger's point of view, the added DSP instructions don't require any special handling, as they don't affect the processor state, but if debug state is entered from Jazelle state, the core has to be switched to ARM state before the core and system state may be examined. All ARMv5

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 3.1: ARM banked registers

cores also support a software breakpoint instruction (BKPT) that forces the core into debug state when it's executed. On cores without support for this instruction, the software breakpoint behavior has to be simulated using a data dependent breakpoint, that triggers once a certain instruction is fetched from memory.

The basic execution context is the same for ARMv4T, ARMv5TE and ARMv5TEJ, and has to be restored by a debugger before control can be transfered back to system software.

- 31 general purpose registers, including the program counter (PC). Only 16 registers are accessible at any time, the remaining registers are banked registers available only from within a particular processor mode. Figure 3.1 shows the association between processor modes and banked registers. Note that System mode shares all registers with User mode. Register R15 is the PC, and its use is subject to special restrictions. Register R14 is the link register which stores the return address on function calls or exception entry. Register R13 is often used as a stack pointer, although this is not mandatory. The remaining registers may be used at the developer's or the compiler's choice.

- 6 status registers. The current program status register (CPSR) contains information about the current processor mode and state, while the saved program status registers contain the saved state from which an exception mode was entered. Only exception modes have a saved program status register. see Figure 3.2 for the program status register format in architecture versions up to ARMv5TEJ.

- The current processor mode is one of User (USR), Fast Interrupt (FIQ), Interrupt (IRQ), Supervisor

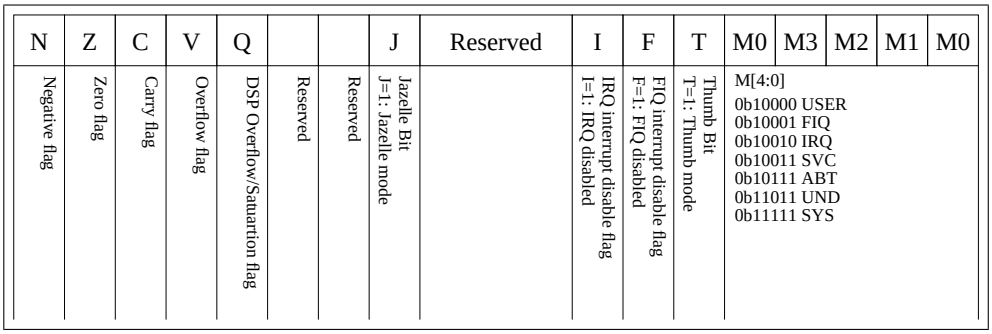

Figure 3.2: program status register format

(SVC), Abort (ABT), Undefined Instruction (UND), and System (SYS). All but the User mode are so called privileged modes, with full access to the hardware. Depending on the current mode, only a subset of the 31 general purpose registers and 6 status registers may be accessed.

- The current processor state is either ARM, Thumb, or Jazelle (on cores with Java support).

- A flat address space of 2^{32} 8-bit bytes.

Cores with a memory management unit (MMU) and caches require additional properties defining the current execution context. The MMU translates virtual addresses (VA) into physical addresses (PA) and may have a translation lookaside buffer (TLB) to store recently used or explicitly stored translations. The current content of the TLB should be considered part of the execution context, as additional page table walks, caused by evicted TLB entries, could have an impact on application critical timings. The same is true for caches that keep recently used memory blocks in high-speed memory tightly coupled to the processor. When accessing memory in a cached system, a debugger has to make sure that as much of the cache state as possible is preserved.

ARM7TDMI Implementation

The ARM7TDMI features a 3-stage pipeline with Fetch, Decode and Execute stages (see Figure 3.3), and a von-Neumann architecture memory system, where instructions and data are fetched from a unified bus. Implementation defined store instructions, that read the program counter [**?**, p. A2-7] (STR, STRT, and STM), store the address of the current instruction plus 12 bytes. On a data abort, instructions with addressing modes that update a base register will have their base register updated ("base updated" data abort model) [**?**, p. 2.2].
An instruction is fetched from the memory system during the Fetch stage, decoded (and possibly decompressed in case of Thumb instructions) in the Decode stage, and executed during one or more cycles in the Execute stage. Required registers are read during the first Execute cycle, and data memory is accessed during one or more subsequent Execute cycles.

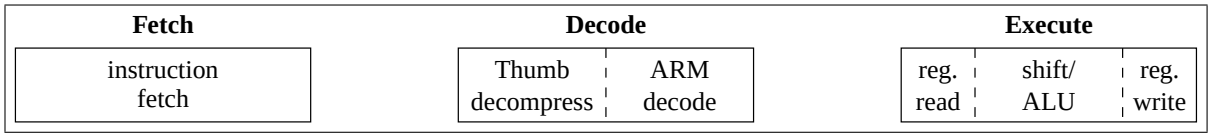

Figure 3.3: ARM7TDMI 3-stage pipeline

ARM9TDMI Implementation

The ARM9TDMI features a 5-stage pipeline that splits the ARM7TDMI's Execute stage into separate Execute, Memory and Write stages (see Figure 3.4). It has a modified Harvard architecture with two separate internal busses for instruction and data that connect externally to unified memory. Like the ARM7TDMI, implementation defined store instructions, that read the program counter [**?**, p. A2-7] (STR, STRT, and STM), store the address of the current instruction plus 12 bytes. The ARM9TDMI implements a "base restored" data abort model, where the base register will always be restored to the value before the aborted instruction was executed [**?**, p. 2.2].
Like the ARM7TDMI, an ARM9TDMI fetches an instruction in the Fetch stage, and decodes it in the Decode stage. Registers are read during the Decode stage, and additional logic ensures a behavior similar to older ARM cores where registers were read one stage later. During the Execute stage, shift and ALU operations are executed, generating results for data instructions, or addresses used in load/store instructions. Data memory is either read or written in the Memory stage, and registers are written in the Write stage. Because of the pipelined architecture, instructions may have to be stalled, if source operands of an instruction were written by an immediately preceding instruction which isn't yet finished. This case is called an interlock, and the core stops fetching new instructions until the results from the preceding instruction are available [**?**, p. 7.5].

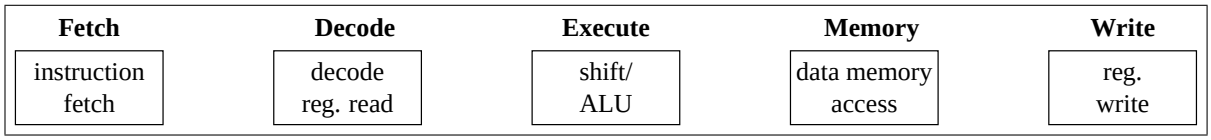

Figure 3.4: ARM9TDMI 5-stage pipeline

3.2 Core Debugging

All ARM7 and ARM9 cores have halt-mode debugging support, that allows the core to be completely stopped. During this debug state, a debugger may capture and modify core signals, allowing the core and system state to be examined and changed. While in debug state, the core is no longer clocked from its main clock (memory clock (MCLK) on ARM7TDMI, fast clock (FCLK) or bus clock (BCLK) on ARM9TDMI) but from a debug clock (DCLK) that's generated by the debug logic.
The ARM core macrocell is deeply embedded inside an ARM based SOC and core signals are not available on external pins. To still be able to debug these systems, ARM7 and ARM9 cores implement a JTAG compatible TAP controller with boundary-scan chains around the core signals. There are two scan chains available on hard macrocells (ARM7TDMI and ARM9TDMI based), one consisting of almost all

core signals, primarily intended for device testing, and another one that consists of a subset of the first, with signals especially important for debug. Figures 3.5 and 3.6 show the order of signals on the debug scan chains. When shifting data in or out of the device, the signal closest to TDO is the least significant bit. It is important to note that D[0:31] (ARM7TDMI) and I[0:31] (ARM9TDMI) are in reversed bit order. Systems based on the synthesizable ARM7TDMI-S and ARM9TDMI-S core lack the first scan chain but are otherwise similar to the hard macrocell implementations.

For the purpose of a debugger it's sufficient to use scan chain 1, which shall from now on be called the debug scan chain. This scan chain may be used in INTEST (see table 2.2) mode, allowing core signals to be captured, and known values to be scanned into the core, or in EXTEST mode, allowing signals from outside of the core to be captured, and known values to be driven to the outside of the core. During debug, the debug scan chain is used in conjunction with the INTEST instruction. The *scan path select register*, a special test data register used to select between several boundary-scan paths, is accessible by the ARM specific SCAN_N JTAG instruction. When SCAN_N is selected, a fixed value, with the most significant bit set to one and all others set to zero, is loaded into the scan path select register during Capture-DR, making it possible to recognize serial communication problems. After scanning a new value into the scan path select register, the new scan chain is made the currently active scan chain during Update-DR. From that point on, JTAG instructions accessing the boundary-scan register (INTEST, EXTEST) apply to the new scan chain.

ARM7TDMI Debug Scan Chain

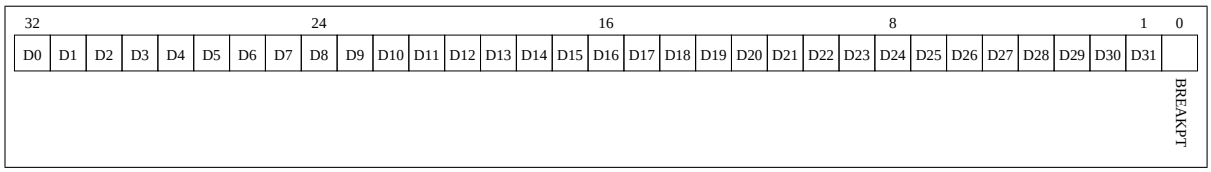

Figure 3.5: ARM7TDMI scan chain 1 (Debug)

Signals D[0:31] of the ARM7TDMI are connected to the core's data bus, and are used to fetch instructions or read/write data. The BREAKPT signal is used to mark instructions that have to be executed at system speed (clocked from MCLK, rather than DCLK), like instructions accessing memory or instructions that make the core return from debug state back to its normal state. The first time BREAKPT is scanned out of the core, it contains information about whether the core entered debug state due to a breakpoint (BREAKPT low) or because of a watchpoint (BREAKPT high).

ARM9TDMI Debug Scan Chain

Signals ID[0:31] of the ARM9TDMI are connected to the core's instruction bus, and are used to fetch instructions. The data lines DD[31:0] connected to the bi-directional data bus are used to read or write data. SYSSPEED is similar to the ARM7TDMI's BREAKPT signal, serving both as a flag for system speed instructions and as an indicator for the reason for debug entry. The WPTANDBKPT signal allows a debugger to determine if an instruction that triggered a watchpoint was immediately followed by a

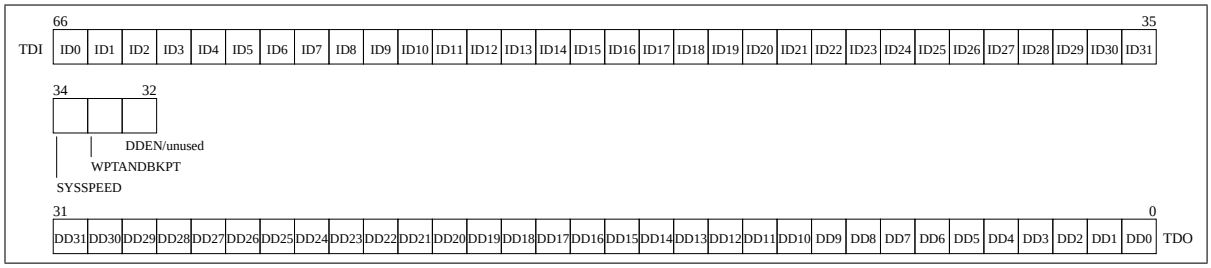

Figure 3.6: ARM9TDMI scan chain 1 (Debug)

breakpointed instruction. In that case, the breakpointed instruction wouldn't have been executed, and would be the next instruction after debug state is left. The DDEN signal is available only on hard macrocell cores, its bit position is unused on synthesizable cores. When DDEN is high, the core is driving data out on DD[31:0] which may be captured by a debugger.

Debug Instruction Execution

Once in debug state, a debugger may serially shift data into the debug scan chain by selecting scan chain 1 (via SCAN_N) and INTEST. While the debug scan chain is selected and INTEST is the current instruction, a DCLK cycle is pulsed on the rising edge of TCK when the TAP controller is in Run-Test/Idle state, making the core act upon the values currently contained in the debug scan chain. A debugger has to take the processor pipeline into account, that is the pipeline stages in which values appear on the databus or have to be written to the bus by the debugger, and possible interlocks in case of ARM9TDMI based cores.

3.3 Embedded-ICE

The Embedded-ICE (formerly known as "ICEBreaker") macrocell available on all ARM7 and ARM9 cores provides on-chip debug functionality similar to an ICE (see §1.1). The Embedded-ICE unit is accessed through JTAG scan chain 2, which is selected through SCAN_N similarly to the debug scan chain (scan chain 1) and may only be used with the INTEST instruction. The Embedded-ICE scan chain (see Figure 3.7) is the same for ARM7 and ARM9 targets, and consists of 32 data bits, 5 address bits and a flag to distinguish between read (nRW low) and write (nRW high) accesses. Embedded-ICE features are accessible through registers, whose number is placed in the address field. The data field contains register data read or to be written, and is aligned to the least significant bit for registers with less than 32 bits. For register reads, the Embedded-ICE scan chain has to be accessed twice, once to program the nRW field for reading and the address of the register to be read, and once to capture the data of the selected register. Register writes are accomplished with a single access programming nRW for writing, the address, and the new register data. Register reads and writes are executed during the Update-DR state.

Every Embedded-ICE implementation provides a common set of supported features, with extensions or restrictions specific to certain families, cores or revisions. Embedded-ICE units contained in ARM7 and ARM9 family cores have two comparators that can be used to break on instruction fetches or data

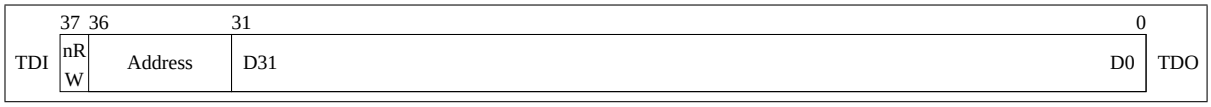

Figure 3.7: Embedded-ICE scan chain (Scan chain 2)

accesses. Each comparator consists of an address register, a data register, a control register, and a mask register for each of the three registers with a layout similar to the value register that may be used to make the comparator ignore masked bits in the comparison. The value/mask register combination allows three possible requirements to be set:

- A positive match. The respective bit is required to be 1. This is achieved by programming the value register bit to 1, and the mask register bit to 0.
- A negative match. The respective bit is required to be 0. This is achieved by programming the value register bit to 0, and the mask register bit to 0.
- An ignored bit. A match should occur whether the bit is 1 or 0. This is achieved by programming the mask register to 1, irrespective of the value bit.

The layout of the comparator registers is almost the same for all ARM7/ARM9 cores, with the exception of a slightly different layout of the control register found on ARM9TDMI based cores, which is due to the modified Harvard architecture of these cores.
The debug control register and the debug status register give access to debug signals that allow a core to be put into halt-mode debug state using an external request, and information about the core state to be examined by a debugger. The signals available through these registers depend on the exact core being used, but a common subset is provided by all implementations.
The Embedded-ICE debug communications channel allows a debugger to communicate with software running inside the core without using additional system resources like a RS232 port. It is accessible from a debugger via a control and a data register, and can be accessed from the core using coprocessor instructions. The control register is used to manage the communication between a debugger and the running core, and contains information about the Embedded-ICE version implemented.
Table 3.2 shows the available registers and their addresses. A detailed layout of the registers is given in figure 3.8, without registers of a flat 32 bit layout like the watchpoint data and address registers or the debug comms data register. The watchpoint control mask register has a layout similar to its control value register but is one bit shorter, as the Enable bit can not be masked.

Embedded-ICE Usage

Debug Request

Entering debug mode on the debugger's request works the same for all ARM7 and ARM9 targets. The debugger asserts DBGRQ by programming the debug control register with DBGRQ set to 1, and polls the debug status register until it reads a 1 in DBGACK. On ARM9 based cores DBGRQ could be left asserted, but ARM7 based cores require it to be deasserted in order to execute instructions at debug speed. The core is then in debug state and may be examined by the debugger. Debug state will only be

Table 3.2: Embedded-ICE register map

Address	Register name	Availability restrictions
0x0	Debug control register	
0x1	Debug status register	
0x2	Abort status register	only ARM7 cores with monitor mode debug (ARM7TDMI Rev 4, ARM7TDMI-S Rev4, and ARM720t Rev4)
	Vector catch register	all ARM9 cores
0x4	Debug comms control register	
0x5	Debug comms data register	
0x8	Watchpoint 0 address value	
0x9	Watchpoint 0 address mask	
0xa	Watchpoint 0 data value	
0xb	Watchpoint 0 data mask	
0xc	Watchpoint 0 control value	
0xd	Watchpoint 0 control mask	
0x10	Watchpoint 1 address value	
0x11	Watchpoint 1 address mask	
0x12	Watchpoint 1 data value	
0x13	Watchpoint 1 data mask	
0x14	Watchpoint 1 control value	
0x15	Watchpoint 1 control mask	

entered after one TCK cycle has been spent in Run-Test/Idle, allowing debug requests to be set up in multiple targets that are then halted at the same time.

Hardware Breakpoints

Hardware breakpoints are realized using one of the two comparators. The address value and mask register should be programmed to match the desired address, with the least significant bit masked for Thumb breakpoints (16 bit instructions) or the two least significant bits masked for ARM breakpoints. This ensures that the breakpoint triggers even with undefined signal levels on unused address lines.
A variant of HW breakpoints would be the use of a data dependent breakpoint. By programming the watchpoint's data register to match a certain instruction value, the breakpoint would only trigger if that instruction is fetched. Together with the address value and mask registers this could be used to set a breakpoint on the execution of a certain instruction in a specified part of the address space.
On ARM7 targets, a comparator may be programmed to match on instruction and data accesses by masking the nOPC field in the watchpoint control register. That's not possible on ARM9 targets due to their modified Harvard architecture, as the comparator can only watch a single bus. A generic breakpoint layout that works for ARM7 and ARM9 targets programs nOPC to a negative match.
If breakpoint matches should be restricted to Thumb instruction fetches, MAS[0] (ITBIT on ARM9) may be programmed to require a positive match. MAS[1:0] is used to determine the size of a memory access, with b00 meaning byte accesses, b01 being a half word access (16 bit) and b10 being a word access (32 bit). An instruction fetch with MAS[1:0] set to b01 is therefor a 16-bit Thumb instruction fetch.
Using the Range, Chain and Extern fields in the watchpoint control register allows more complex break-

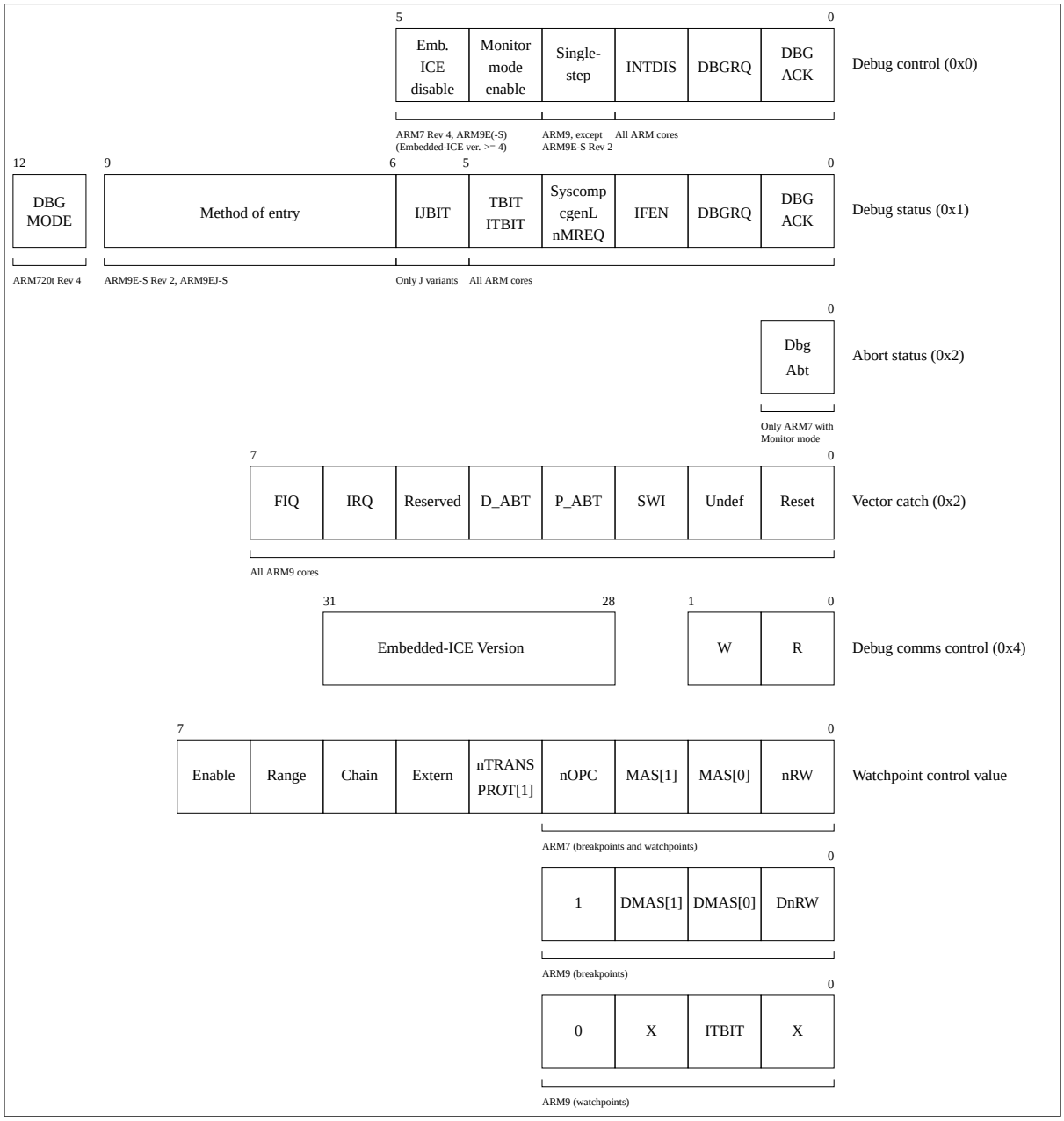

Figure 3.8: Embedded-ICE register layout

points to be defined. The technical reference manuals of each ARM core give information on the possibilities and some usage guidelines.

Software Breakpoints

Software breakpoints work by replacing an instruction in the target memory with a special instruction that forces the core to enter debug state. Cores implementing the ARMv5TE(J) architecture may use the dedicated BKPT instruction for this purpose, while older cores use a value defined in a data dependent instruction breakpoint. The watchpoint address mask register is programmed to ignore the address (all bits 1), and the data register is set to match on a certain instruction value. To be able to use the same instruction value for ARM and Thumb state breakpoints, a symmetric pattern with the same value in the upper and lower 16 bit of a 32 bit word should be chosen. 0xDEEEDEEE (32 bit ARM breakpoint) and 0xDEEE (16 bit Thumb breakpoint) is a possible implementation that fits to the Thumb state breakpoint instruction available on ARMv5TE(J) BKPT (0xDExx) [**?**]. The control register should be programmed to require a negative match on nOPC and ignore all other bits.

Watchpoints

Hardware watchpoints are similar to HW breakpoints, but monitor the data bus. This is achieved by programming the nOPC field in the watchpoint control register to a positive match. Using the nRW field, a watchpoint can be limited to reads (negative match), writes (positive match) or any access (ignore).

Vector Catching

It may be important for a debugger to catch all or certain exceptions generated in the target. On ARM7 targets, this has to be achieved using a HW or SW breakpoint, requiring the use of one of the two comparators. ARM9 targets contain a dedicated vector catch register that allows breakpoints to be set on all or selected exceptions. The vector catch register only triggers on exception mode entry, not on a regular fetch caused by a branch to an exception address.

Single-Stepping

Single-stepping is implemented in hardware on most ARM9 cores with the exception of ARM9E-S Rev 2 and ARM9EJ-S based designs, where this capability can't be found. The *Single Step* bit contained in the debug control register of cores supporting single-stepping forces the core to re-enter debug state after a single instruction has been fetched and executed.
Cores without hardware single-stepping capability have to simulate this behavior using a breakpoint combined out of two comparators. The two comparators form an inverse breakpoint, that breaks on everything but the current address:

- Both watchpoint units are programmed for HW breakpoint usage, requiring a negative match on nOPC and ignoring the data value (data mask registers set to all ones).
- Watchpoint 1 matches the address of the current instruction (the one to be executed), but isn't enabled. Inside the ARM7/ARM9 Embedded-ICE unit, the Range output of watchpoint 1 is derived from its address comparator and connected to watchpoint 0's Range input. An address match on watchpoint 1 appears as a positive value on watchpoint 0's Range field.

- Watchpoint 0 is enabled and set to match on any address, but is required to have a negative match on its Range field.

- The core resumes execution from debug state. On the first instruction fetch, watchpoint 1 matches the address but doesn't trigger as it's not enabled. Watchpoint 0 matches the address, but its Range input is high (from watchpoint 1), preventing it from triggering.

- On the second instruction fetch, watchpoint 1 no longer matches the address. Watchpoint 0 still matches the address, this time with a low Range input, making it trigger. The core enters debug mode after executing one instruction.

This method doesn't work for instructions that branch back to themself, a combination that's probably rarely seen in reality. In that case, watchpoint 1's address comparator would match forever, preventing the core from re-entering debug state. A debugger should take care of that possibility by implementing a timeout by which the core should have reentered debug state.

3.4 Debug State Entry

Debug state may be entered as a result of the following conditions:

- Debug request. Either an external debug request (EDBGRQ) or as a result of programming the Embedded-ICE control register with DBGRQ set to 1. The core is forced to enter debug state after it finished executing the current instruction. On ARM7 cores, the program counter (PC) contains the address of the instruction to be executed next plus two addresses (8 byte in ARM state, 4 byte in Thumb state), whereas on ARM9 systems it contains the address plus three addresses (12/6 bytes).

- Breakpoint. A breakpoint can be triggered by an Embedded-ICE watchpoint, a software breakpoint instruction on ARMv5TE(J) targets or an external breakpoint signal (IEBKPT). If an instruction fetch causes a breakpoint to trigger, the instruction is still fetched into the pipeline and marked as breakpointed. If the instruction reaches the execute stage (i.e. it's not flushed due to a branch or exception entry), the core enters debug state without executing the breakpointed instruction. On both ARM7 and ARM9 systems, the PC contains the address of the breakpointed instruction plus three addresses. BREAKPT on ARM7 cores or SYSSPEED on ARM9 cores are low the first time they're scanned out of the debug scan chain after a breakpoint occurred.

- Watchpoint. Either an external watchpoint (DEWPT) or an Embedded-ICE watchpoint (nOPC positive match). The instruction causing the memory access and the immediately following instruction have been executed after a watchpoint triggered. Just as it's the case for a breakpoint, the PC contains the value of the next instruction plus three addresses on both ARM7 and ARM9 systems. A watchpoint is signaled by high values of BREAKPT and SYSSPEED the first time these bits are scanned out.

- Watchpoint + Breakpoint. The instruction immediately following a watchpoint may be breakpointed, in which case it's not going to be executed. This can't be detected on an ARM7 system, but on an ARM9 WPTANDBKPT may be examined to detect such a situation. The PC contains the address of the instruction to be executed next plus three addresses.

In addition to the debug reason detection via BREAKPT/SYSSPEED, newer ARM9 cores like the ARM9E-S Rev 2 and the ARM9EJ-S offer a Method of entry field in the debug status register (see Figure 3.8) with detailed information about the condition that caused debug entry [**?**].

3.5 Core State

Once in debug state, a debugger may start to examine the core state using instructions scanned into the debug scan chain (see §3.2). The core registers of the current processor mode can be read using a "store multiple" (STM) instruction. The debugger puts the `STM` instruction in the processor pipeline, clocks the core by moving through Run-Test/Idle, and loads two additional "no operations" (`NOP`) into the pipeline. During the 4th cycle, the values of the registers referenced by the `STM` instruction start to appear on D[0:31] (ARM7) and DD[31:0] (ARM9), and can be capture by the debugger [**?**, p. A4-84].
The current program status register (CPSR) may be read using a "move PSR to general purpose register" (`MRS`) instruction that moves the CPSR into one of the general purpose registers followed by a "store register" instruction that makes the value of that register appear on the data bus (D[0:31]/DD[31:0]). Saved program status registers of exception modes are handled similarly using the R bit of the `MRS` instruction that moves the SPSR instead of the CPSR [**?**, p. A4-60].
Registers of other modes require a change to that mode which can be done using a "move to status register from core register" (`MSR`) instruction. While during normal execution, the current processore mode may only be changed when in privileged modes or on exception entry, there's no such restriction while the core is in debug state. A debugger can put a `MSR` instruction with an immediate operand in the processor pipeline, and may start examining registers of the new mode once the `MSR` instruction is completed [**?**, p. A4-64].

3.6 System State

It's not possible to access system memory while the core is in debug state and clocked from DCLK, so the core must resynchronize to its main clock (BCLK/FCLK/MCLK, see §3.2). ARM cores define a JTAG instruction RESTART that's used to restart the core from debug state. The core resynchronizes to the memory system once the TAP controller reaches the Run-Test/Idle state with RESTART as the current instruction. Using load multiple (`LDM`) instructions executed at system speed to read system memory and store multiple (`STM`) instructions executed at debug speed to capture the read values a debugger may examine the system state. If system memory is to be modified, the operations may occur in the opposite order, using `LDM` at debug speed to load the new values into core registers and using `STM` to write them.
On ARM7 based cores, the instruction prior to the one that is to be executed at system speed has to be scanned into the core with the BREAKPT bit set high. ARM9 based cores require the instruction that should be executed at system speed to be scanned in with the SYSSPEED bit low, followed by a `NOP` with SYSSPEED high.
After the necessary instructions have been put into the processor pipeline, RESTART is loaded into the TAP controller, and the state machine is moved to Run-Test/Idle state. The core resynchronizes to the memory clock, executes the system speed access, and reenters debug state. A debugger should poll the debug status register to determine when the operation completed.

Table 3.3: LDR operation cycle timing (ARM7) with PC as destination

Pipeline stage	Cycle number	Action
Fetch	1	LDR instruction is fetched
Decode	1	instruction fetched, LDR is decoded
Execute	1	instruction fetched, LDR source address is calculated
Execute	2	nothing fetched, new PC is loaded from memory
Execute	3	nothing fetched, PC register is written
Execute	4	instruction fetched from new PC
Execute	5	instruction fetched from new PC+4

3.7 Exit from Debug State

Exit from debug state is similar to system state accesses, but instead of a load/store instruction a branch is loaded into the processor pipeline. A debugger has to restore the execution context (see list 3.1) before it may exit from debug state. This may be achieved using `MSR` instructions to modify the CPSR and `LDM` instructions that load the core registers. Finally, the PC has to be reloaded, as it got incremented on every instruction executed during debug. `LDR` instructions that load the PC are similar to a branch, as they require the core pipeline to be flushed, and new instructions have to be fetched from memory. Table 3.3 shows the cycles executed on ARM7 cores [**?**, p. 6-13], but the order of operations is the same for ARM9 systems. During Execute cycles 4 and 5, new instructions are fetched, and the PC is incremented. ARM9 systems may fetch the branch with SYSSPEED set and the following `NOP` during these cycles, requiring a branch back to the current instruction (-2 addresses), but ARM7 systems need an additional `NOP` during the 4th Execute cycle. This results in a `NOP` fetched during `LDR's` execute cycle 4, a `NOP` with BREAKPT set during `LDR's` execute cycle 5, and the final branch back to the last but two instruction (-4 addresses). After the branch has been clocked into the pipeline, RESTART is selected as the current JTAG instruction. Once the TAP controller reaches Run-Test/Idle, the processor starts executing from the restored PC.

4 ARM MMU/Cache Handling

ARM cores with a MMU or Caches require special treatment. The debugger has to ensure coherency between the caches and main memory, while as much as possible of the cache and MMU state has to be preserved. This chapter is going to look at the implications for ARM720t cores (MMU and unified cache) and ARM920t cores (MMU and separate instruction and data caches). The same considerations are true for other cores, but the support provided by the on-chip debug facilities may be different, requiring different actions by a debugger. A basic understanding of MMU/Cache implementations in general are required, detailed information is given in [**?**] and [**?**].

4.1 Unified versus Separate Cache Implementations

Unified Cache (e.g. ARM720t)

Figure 4.1 shows the basic organization of a unified cache system like the ARM720t. The ARM7TDMI processor core issues virtual addresses to the MMU and the unified cache. While the MMU is turned off, the core still issues its addresses to the MMU, which passes them through unaltered, giving a flat 32 bit address space.
ARM7TDMI data reads from addresses that are already contained in the cache are satisfied from there. If the data isn't in the cache, the MMU checks its TLB to see if there's already a translation for the required virtual address. In cases where the TLB contains the required translation, and the virtual address is in a cacheable memory area, a line fetch is executed to fill the cache with data from the bus interface, which is then transfered to the ARM7TDMI. Uncacheable memory is transfered directly from the bus interface to the core. On a TLB miss, the MMU executes a page table walk to find a translation for the virtual address, which is written to the TLB. The MMU generates a translation abort, if no valid translation is found, and the core enters the data abort exception handler. Instruction fetches are similar, but an instruction abort is generated instead of a data abort.
The ARM720t has a write-through cache and operates on read-miss allocation [**?**, p. 4.2]. Memory writes update the cache on a hit, but will always be written to main memory, too. Cache lines are only loaded or replaced on read operations - write operations that don't generate a hit inside the cache wont alter it.
Coherency isn't an issue for ARM720t based systems due to the unified write-through cache. It's sufficient to disable the caches during memory reads to ensure that cache content is preserved. Read hits are still served from the cache if it's disabled, [**?**, p. 4.2], and cache misses lead to system memory accesses.

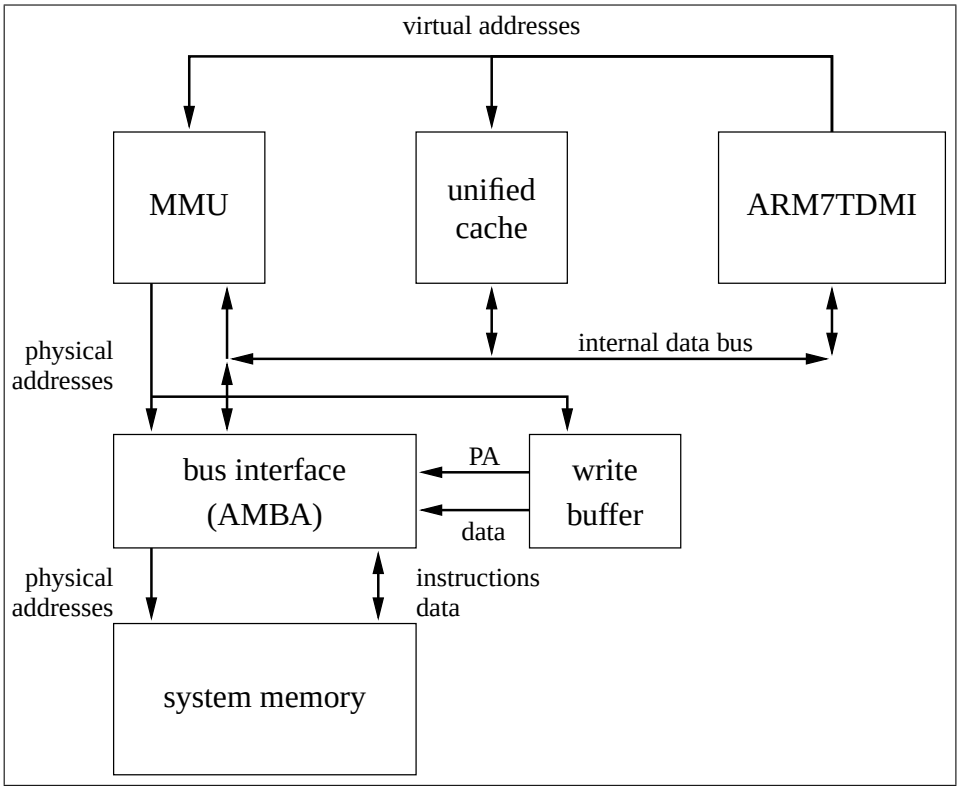

Figure 4.1: ARM unified cache organization (ARM720t)

Separate Caches (e.g. ARM920t)

Figure 4.2 shows the organization of an ARM920t processor with its separate data and instruction caches. Virtual/physical address translation is handled similar to the ARM720t, but two independent MMUs take care of instruction and data virtual addresses.
The ARM920t data cache can operate on a write-through or a write-back policy, depending on the configuration of the particular memory location, while naturally the instruction cache doesn't write any memory. Writes to write-through memory regions update the cache and are sent to the write buffer, too, while writes to write-back memory only update the data cache entry and mark it as dirty. Dirty cache lines are written to main memory if the cache line is to be replaced or if an explicit data cache flush was initiated. Both caches implement a read-miss allocation like the ARM720t.
Keeping the instruction and data cache in a coherent state is an important task of a debugger designed for ARM920t based systems, as accesses to the data caches and main memory wont affect the instruction cache. When the debugger modifies program code, for example by setting a software breakpoint, a write could go to the data cache only, in case of a write-back region. As soon as that instruction is to be fetched, the instruction cache is queried, and might return an instruction it fetched before the debugger modified the code. On an instruction cache miss, a line fill is executed, loading instructions from system memory, which may have outdated code, too. To ensure coherency and preserve as much of the cache state as possible in every possible case, it is important to carry out the following steps in case of a memory write while caches are enabled:

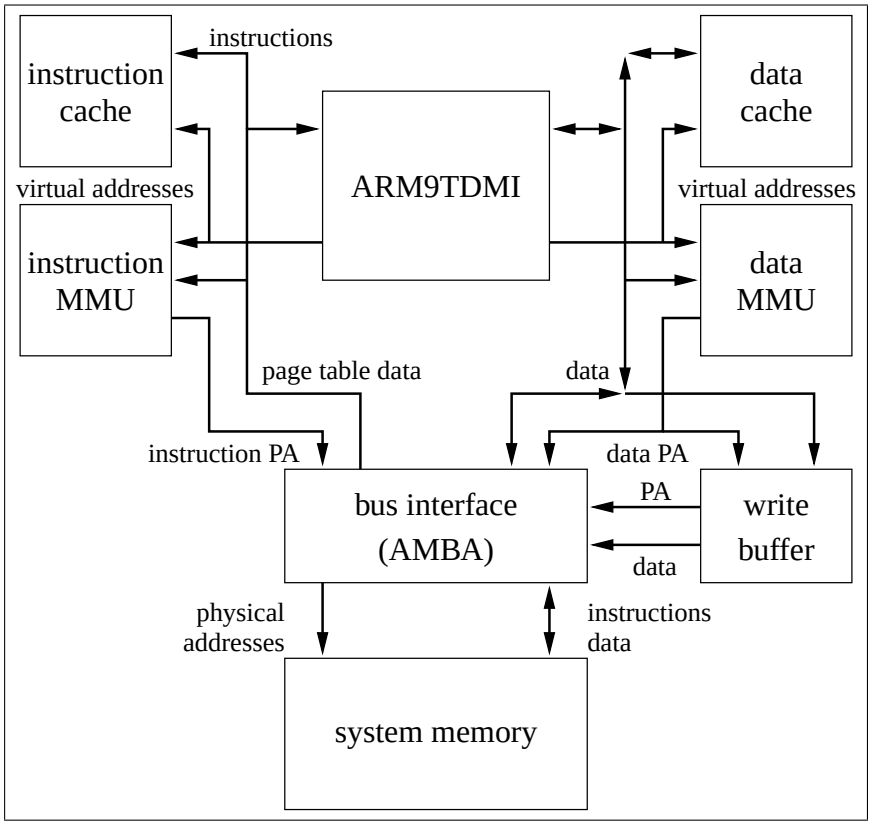

Figure 4.2: ARM separate instruction and data caches (ARM920t)

- Disable line fills for instruction and data caches on debug entry. This ensures that no cache lines are replaced during debugging.

- Examine the cacheable/bufferable bits for a memory location that is to be written. If the region is marked as write-back cacheable, execute either a cache flush, change the memory region temporarily to write-through, or write the memory twice, once using the virtual address while MMU and caches are enabled, and once using the physical address while MMU and caches are disabled. This guarantees that the data cache and system memory are in a coherent state.

- Invalidate the instruction cache for every address that was written. The core is going to execute a line fetch if it has to execute an instruction from an address that was invalidated before, fetching the modified code from system memory.

These steps are time consuming, if larger memory blocks have to be written, so a debugger might apply them only to small modifications, like writing of half-words and words. This ensures that breakpoints always affect instructions possibly contained in the instruction cache, while keeping the overhead for other operations to a minimum. Larger transfers typically affect either modified data, where coherency isn't an issue, or code download, in which case the user can explicitly specify that coherency is to be ensured, if this is desired.

4.2 System Control Coprocessor

The MMU, caches and other system features available only on cached systems are controlled by coprocessor 15 (CP15). It has a common programmer's model available on all implementations, while special features or restrictions only apply to selected cores. [**?**, p. B2-1 ff.] gives detailed information about the system control coprocessor's programmer's model. This section is going to explain the use of functionality necessary or useful for a debugger.

- Main ID register. Accessible as register 0 with opcode2 set to 0. Contains information about the processor core like architecture version, part number and core revision number. Useful to determine if revision dependent features are available.
- Cache Type register. If available (like on ARM9 cores) this register gives detailed information about the cache type (write-back/through, supported functions), whether it's a unified cache or separate I/D caches, and the size and organization of the caches (line length, associativity, number of cache sets). This is especially important on cores with configurable cache sizes like the ARM926EJ-S. The Cache Type register is accessed as register 0 with opcode2 set to 1.
- Control register, CP15 Register 1. Used to control system features like the MMU and caches. See [**?**, p. B2-13] and a particular core's technical reference manual for a list of available configuration options. This register should be made user-accessible through a debugger.
- MMU translation table base register, CP15 Register 2. Used as the offset to the first-level page table for a first-level descriptor fetch. Only bits 31 to 14 are used, the remaining 14 bits should be zero. The first-level page table is therefor aligned to a 16 kB boundary.
- Fault status register, CP15 Register 3. Contains information about the abort reason. On ARM9 cores there are two registers available, one for data aborts and one for prefetch aborts (instruction fetch abort). The instruction fault registers are only accessible from a debugger.
- Fault address register, CP15 Register 4. The address that caused an abort. ARM9 cores have two registers, one for data aborts and one for prefetch aborts.
- MMU, cache and write buffer control registers. The use of these registers is mostly implementation defined.
- FCSE ID register, CP15 Register 13. This register contains the fast context switch extension process id (PID) of the current process in its top seven bits. FCSE allows process memory to be relocated by replacing the top seven bits of a virtual address with the PID. This gives 128 process memory blocks of 32 MB size that can be switched without having to modify the virtual-to-physical address translation. Virtual addresses (VA) are first translated using the FCSE, producing a modified virtual address (MVA), which is then fed to the caches and MMU.

ARM720t CP15 Accesses

Coprocessor 15 registers may be accessed through a debugger using the JTAG boundary-scan chain 15 together with the INTEST instruction. Figure 4.3 shows the layout of scan chain 15, with data bits CPDATA[0:31] and a flag indicating whether the value represents data or an instruction. Coprocessor

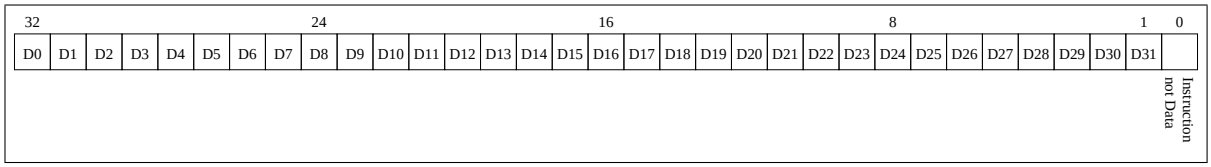

Figure 4.3: ARM720t scan chain 15 (CP15)

Table 4.1: ARM720t CP15 read operations

CPDATA31..0 (in)	CPDATA31..0 (out)	Instruction bit	Clock
coprocessor instruction	ignored	1	yes
NOP instruction	ignored	1	yes
NOP instruction	ignored	1	no
0x0	ignored	0	yes
0x0	read value	0	yes
NOP instruction	ignored	1	yes

instructions are executed by serially shifting them into scan chain, and moving the TAP controller to Run-Test/Idle where the coprocessor is clocked. [**?**] gives an example on how to access coprocessor 15 using JTAG accesses. The CP15 follows the ARM7TDMI pipeline with its Fetch, Decode and Execute stages. An instruction that should be executed has to be scanned into the pipeline with the instruction bit high, followed by two `NOP` instructions. The last access to scan chain 15 before the value is read has to have the instruction bit low, indicating a data access. The coprocessor instruction is executed in the second Execute cycle, during which the debugger can capture the data. Coprocessor register writes are similar and require the new value to be scanned into scan chain 15 during the second Execute cycle. Table 4.1 shows the process of executing a coprocessor 15 instruction that reads a coprocessor register. The coprocessor instruction has to be built according to the register that should be accessed. The final `NOP` was necessary during all tests to ensure that CP15 operations worked properly. The only public documentation about ARM720t CP15 accesses is the FAQ entry [**?**].

ARM920t CP15 Accesses

There are two access types for CP15 registers on ARM920t cores, physical access mode and interpreted access mode. Both use the JTAG boundary-scan chain 15, which may only be used together with the INTEST instruction. [**?**, p. 9-32 ff.] lists the registers accessible by each of the two methods. The layout of scan chain 15 is shown in figure 4.4. The mode of operation is selected by bit 0 (next to TDO), with a 0 indicating an interpreted access and 1 a physical access.

Physical Access Mode

Scan chain 15 behaves similar to the Embedded-ICE scan chain when used in physical access mode with bit 0 set high. The data, address and nRW bits are serially shifted into the scan chain. During Update-DR the register is read or written, requiring an additional pass for register reads, where the value of the selected register is shifted out of the boundary-scan register.

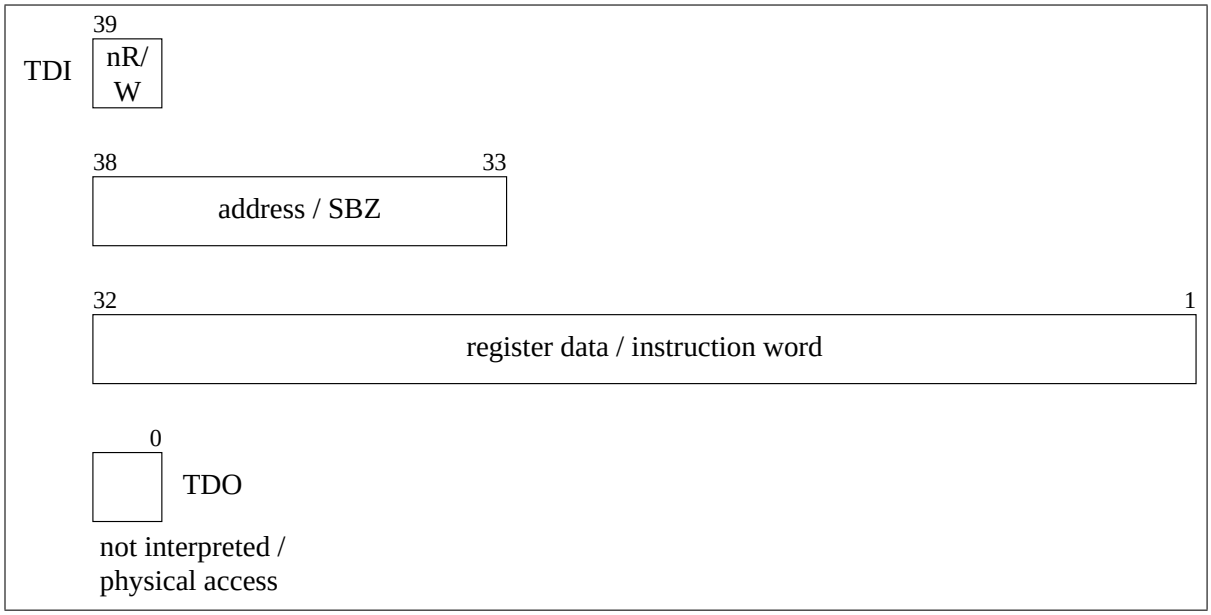

Figure 4.4: ARM920t scan chain 15 (CP15)

Interpreted Access Mode

Before an interpreted access may be executed, the CP15 test state register (register 15) has to be modified using a physical access to set the CP15 interpret mode bit [**?**, p. B-4]. The desired coprocessor instruction is then scanned into scan chain 15 with bit 0 low to select interpreted access mode. The ARM9TDMI is used to execute a system speed load (CP15 register read) or store (CP15 register write) operation, which executes the coprocessor instruction, reading or writing the core register specified in the system speed load/store. After the coprocessor accesses are finished, the CP15 test state register has to be restored in order to disable interpreted access mode.

5 OpenOCD Command Line Arguments

OpenOCD accepts the following command line arguments:

Table 5.1: OpenOCD Command Line Arguments

Long option	Short option	Arguments	Description
help	h		Display help text.
debug	d	level	Set debug level 0-3, default is 2. Without the optional argument debug level is set to 3.
file	f	filename	Use configuration file `filename`.
log_output	l	filename	Redirect log output to `filename`.
interface	i	name	Use the JTAG interface driver `name`.

6 OpenOCD Commands

6.1 Server

Configuration Commands

- `telnet_port <port>`
  Listen for telnet connections on `port`.

- `gdb_port <port>`
  Listen for GDB connections for the first target on `port`, subsequent targets listen on `port + n`.

User Commands

- `shutdown`
  Shut the server down.

- `exit`
  Exit telnet session. Leaves server running.

6.2 Interpreter

The interpreter commands may be used to define variables used within other subsystems like JTAG.

User And Configuration Commands

- `var <name> [num_fields|'delete'] [size1] [sizeN]`
  Allocate, display or delete variable. Allocation has to define the `size` for all `num_fields` elements.

- `field <var> <field> [value|'flip']`
  Display or modify variable field.

- `script <file>`
  Execute commands from `file`.

6.3 JTAG

Configuration Commands

- interface <name>
  Use JTAG interface driver name. If a matching driver is found, its command handlers are registered, and can be used from then on. Only one interface may be specified. Currently supported interfaces are:
  - parport Parallel port bitbanging, e.g. using Wiggler(-clones).
  - amt_jtagaccel Amontec Chameleon in its JTAG Accelerator configuration.
  - ftdi2232 FTDI FT2232C devices using the open source libftdi.
  - ftd2xx FTDI FT2232C devices using the FTDI libftd2xx.

- jtag_device <IR length> <IR capture> <IR mask> <IDCODE instruction>
  Defines the next JTAG device in the daisy chain. The first jtag_device is the one closest to TDO. IR capture is the value that is loaded into the instruction register during Capture-IR, IR mask specifies which bits of the IR capture value have to match (bits [1:0] (0x3) are mandatory). The IDCODE operand allows the JTAG subsystem to identify devices.

- reset_config <type>
  Defines the type of reset configuration supported by the JTAG interface and the connected devices. Possible values are none, trst_only, separate, and combined. Philips LPC2000 devices for example hold the test logic in reset when the system reset is asserted, so configurations containing a LPC device should use combined.

Parport

- parport_port <port|num>
  If compiled to use direct port I/O, use port as the base address for the parallel port. If compiled with --enable-parport_ppdev to support parallel port access through the ppdev module, use /dev/parportnum.

- parport_cable <name>
  Use parport cable definition <name>. Currently available cable configurations are
  - wiggler For cables that implement full Wiggler compatibility (Macraigor Wigglers, Olimex ARM-JTAG, "new" Amontec Chameleon Wiggler configuration).
  - old_amt_wiggler For the "old" Amontec Wiggler configuration that came with the Chameleon-Programmer. This configuration has nTRST and nSRST exchanged and both reset lines inverted.
  - chameleon The pin definition for reconfiguring the Amontec Chameleon. The configuration switch has to be set to configuration mode.

Amt_jtagaccel

- parport_port <port>
  Use port as the base address for the parallel port.

- rtck enabled
  Enable use of RTCK. This slows the interface down, but ensures reliable communication with -S targets.

User Commands

- scan_chain
  Print current scan chain configuration.
- endstate <tap_state>
  Finish JTAG operations in tap_state>.
- jtag_reset <trst> <srst>
  Toggle reset lines <trst> <srst>.
- runtest <num_cycles>
  Move to Run-Test/Idle, and execute num_cycles.
- statemove [tap_state]
  Move to current endstate or tap_state.
- irscan <device> <instr> [devN] [instrN]
  Execute IR scan. For each device in the scan chain, device and instr have to be listed.
- drscan <device> <var> [devN] [varN]
  Execute DR scan. Only devices not in bypass state have to be listed. Variables may be referenced by name or by ordinal number.

User And Configuration Commands

- jtag_speed <value>
  Limit TCK speed. The meaning of value depends on the interface driver. Parport doesn't implement speed limiting, the FTDI drivers use 6MHz / value, and the Amontec JTAG Accelerator operates at 8MHz / 2^value.
- verify_ircapture <enable|disable>
  Verify value captured during Capture-IR. This increases speed on some JTAG interfaces, where reading results requires additinal commands, but prevents early detection of communication problems.

6.4 Target

Configuration Commands

- target <type> <startup_mode>
  Configure targets. This command takes additional arguments depending on the target type. Currently supported target types are:
  - arm7tdmi

  - arm9tdmi

startup_mode may be one of:

  - attach Attach to the target, but don't take any action.
  - reset_halt Reset the target and request immediate halt.
  - reset_run Reset the target and let it run.
  - init_halt Reset the target, halt it, and execute target initialization (setting up memory, disabling watchdogs, ...).
  - early_halt Reset the target, and halt it as early as possible (but don't do anything during reset.

Some startup modes interfere with others, i.e. two targets may not specify a startup mode that requires a reset and a startup mode that doesn't. Some targets may not support several startup modes, like the LPC2000 devices that can't halt immediately out of reset.

User Commands

- targets [num]
  Display list of configured targets, or make num the current target.

- reg [#|name} [value|'force']
  Display or modify registers. If called with no arguments, a list of all registers defined for the current target is displayed. A single register may be accessed by its ordinal number or its name. If force is specified, the register is read from the target even if a cached value exists.

- poll [on|off]
  Print information about the current target's state. Continous polling is enabled by default, but can be disabled or reenabled with on or off.

- halt
  Request target halt.

- resume <address>
  Resume the target at the current position or at address.

- step <address>
  Single-step at the current position or at address.

- reset [halt|init]
  Reset the target, and optionally halt it or halt and init.

- md[whb] <address> [count]
  Display count words (32 bit), half-words (16 bit) or bytes at address. If count is omitted, one element is displayed.

- mw[whb] <address <value>
  Write value at the word, half-word or byte location address.

- `bp <address> <length> [hw]`
  Set a breakpoint of `length` bytes at `address`. Default is setting a software breakpoint, unless `hw` is specified. The interpretation of the `length` argument depends on the target. For ARM targets, length may be 2 for Thumb state breakpoints or 4 for ARM state breakpoints.

- `rbp <address>`
  Remove breakpoint at `address`.

- `wp <address> <length> <r|w|a>`
  Set a watchpoint of `length` bytes at `address`. Watchpoints are either read, write or access (trigger on both reads and writes).

- `rwp <address>`
  Remove a watchpoint at `address`.

- `load_binary <file> <address>`
  Load binary `file` into target memory at `address`.

- `dump_binary <file> <address> <size>`
  Dump target memory of `size` bytes at `address` into `file`.

ARM v4/5 Architecture

- `armv4_5 reg`
  Display all banked ARM core registers.

- `armv4_5 core_state [arm|thumb]`
  Display the current core state, or switch between `arm` and `thumb` state.

ARM 7/9 Family

- `armv7_9 write_xpsr <value> <spsr>`
  Write the program status register. `spsr` selects between the current program status register (0) and the saved program status register (1) of the current mode.

- `arm7_9 write_xpsr_im8 <8bit immediate> <rotate> <not cpsr|spsr>`
  Same as `write_xpsr`, but use the immediate operand opcode.

- `arm7_9 write_core_reg <num> <mode> <value>`
  Write core register `num` of `mode` with `value`.

- `arm7_9 sw_bkpts <enable|disable>`
  Enable or disable the use of software breakpoints. On ARMv5 cores (ARM7E-S, ARM9E-S) with support for the BKPT instruction this has no effect, on other cores it controls the use of one of the watchpoint units for implementing software breakpoints.

- `arm7_9 force_hw_bkpts`
  Force the use of hardware breakpoints. This may be used with Insight to supports breakpoints in ROM or older versions of GDB that use software breakpoints for single-stepping.

Table 6.1: LPC2000 device matrix

Number	Flash Size (kB)	Ram Size (kB)	LPC2000 Driver Variant	Block Layout
2104	128	16	lpc2000_v1	16 x 8kB
2105	128	32	lpc2000_v1	16 x 8kB
2106	128	64	lpc2000_v1	16 x 8kB
2114	128	16	lpc2000_v1	16 x 8kB
2119	128	16	lpc2000_v1	16 x 8kB
2124	256	16	lpc2000_v1	8x8kB + 2x64kB + 8x8kB
2129	256	16	lpc2000_v1	8x8kB + 2x64kB + 8x8kB
2131	32	8	lpc2000_v2	8x4kB
2132	64	16	lpc2000_v2	8x4kB + 1x32kB
2134	128	16	lpc2000_v2	8x4kB + 3x32kB
2136	256	16	lpc2000_v2	8x4kB + 7x32kB
2138	512	32	lpc2000_v2	8x4kB + 14x32kB + 5x4kB (500kB)
2141	32	8	lpc2000_v2	8x4kB
2142	64	16	lpc2000_v2	8x4kB + 1x32kB
2144	128	16	lpc2000_v2	8x4kB + 3x32kB
2146	256	40	lpc2000_v2	8x4kB + 7x32kB
2148	512	40	lpc2000_v2	8x4kB + 14x32kB + 5x4kB (500kB)
2194	256	16	lpc2000_v1	8x8kB + 2x64kB + 8x8kB
2210	0	16	n/a	n/a
2212	128	16	lpc2000_v1	16x8kB
2214	256	16	lpc2000_v1	8x8kB + 2x64kB + 8x8kB
2290	0	16	n/a	n/a
2292	256	16	lpc2000_v1	8x8kB + 2x64kB + 8x8kB
2294	256	16	lpc2000_v1	8x8kB + 2x64kB + 8x8kB

6.5 Flash

Configuration Commands

- `flash bank <driver> <base> <size> <chip_width> <bus_width>`
  Configure a flash bank at address `base` of `size` bytes with a bus of `bus_width` bits formed by chips of `chip_width` bits size using `driver`.

LPC2000

- `flash bank lpc2000 <base> <size> 0 0 <lpc_variant> <target#> <cclk>`
  The internal flash of LPC2000 devices doesn't require chip- and buswidth to be defined. The `lpc_variant` specifies the supported IAP commands of the device. The flash bank is part of `target#` which runs at `cclk`kHz.

User Commands

- `flash banks`
  Display list of configured flash banks.

- `flash info <bank>`
  Display information and list of blocks of flash `bank`.
- `flash probe <bank>`
  Probe flash `bank` if it matches the configured bank. This may also update information about the state of flash blocks.
- `flash erase <bank> <first> <last>`
  Erase blocks `first` to `last` of flash `bank`.
- `flash write <bank> <file> <offset>`
  Write `file` to flash `bank` at `offset`.

LPC2000

- `lpc2000 part_id <num>`
  Display the device Part ID of LPC2000 flash bank `num`.

6.6 XSVF

User Commands

- `xsvf <num> <file>`
  Program the device `num` using the specified xsvf `file`. This currently only works for Xilinx Coolrunner devices (tested only with Amontec Chameleons).