B. Debugger

Diese weiteren Unterkapitel sind nicht Teil der Diplomarbeit enthalten. Sie sind jedoch eine hilfreiche Ergänzung und dokumentieren die gemeinsame Arbeit mit meinem Kommilitonen im Rahmen des ARMLAB Praktikums. Die Aufgabe bestand in der Schaffung einer Debugging Umgebung für das LPEC2001 Board.

Bei der Evaluation der verfügbaren Tools fielen vier besonders auf:

Insight[1] ist eine von RedHat programmierte graphische Oberfläche für den GNU Debugger gdb[2]. Die Version von Macraigor[3] ist speziell für ARM Prozessoren kompiliert und ermöglicht als Besonderheit das Debuggen über die JTAG Schnittstelle durch Closed Source Kernel Module.

Der ARMulator[4] ermöglicht die Simulation eines kompletten ARM Linux Systems auf einem normalen PC. Dies ermöglicht kurze Testzyklen, die besonders beim Debuggen des Kerns die Arbeit deutlich beschleunigen.

Openwince[5] ist ein universelles Tool zum Zugriff auf die JTAG Schnittstellen. Es unterstützt nur die grundlegenden JTAG Funktionen und ist deswegen nur als Ansatz für eine Spezialisierung auf ARM Prozessoren geeignet.

Jtag-Arm9[6] ist ein Tool, das den Zugriff auf die JTAG Schnittstelle und die Nutzung spezifischer Funktionen des ARM9 Prozessors ermöglicht. Das Programm ist etwas unübersichtlich programmiert, jedoch sehr kompakt. Die Implementierung erweist sich teilweise als fehlerhaft und führt zu Verbindungsabbrüchen und fehlerhaften Informationen.

Auf Basis des jtag-arm9 Projekts wurde ein GDB JTAG Server geschrieben, der mit dem Macraigor Debugger zusammen arbeitet. Im Prinzip wurde so der Closed Source Teil des Macraigor Debuggers neu implementiert oder andererseits hat das jtag-arm9 Projekt jetzt einen GDB Server bekommen.

In den folgenden Unterkapiteln wird zuerst die ARM JTAG Schnittstelle und deren Ansteuerung, sowie der Aufbau des gdb-jtag-arm Debuggers und seiner Backendfunktionen ausführlich erklärt. Es folgt die Dokumentation der Implementierung des Netzwerk GDB Servers und dem verwendeten Protokoll.

[1] Vgl. [RedHat 2004a]
[2] Vgl. [GNU 2004]
[3] Vgl. [Macraigor 2004]
[4] Vgl. [McCullough 2002]
[5] Vgl. [ETC 2003]
[6] Vgl. [Wood 2001]

B.1. JTAG ARM Backend

Die JTAG Schnittstelle (Joint Test Action Group) wurde in der Mitte der 80er Jahre entwickelt, um digitale Messungen an neuartigen IC Formen (BGA) möglich zu machen. Beim Boundary Scan Testing werden Testschaltungen in den Chip eingebaut, die die Grundlage für ein komplettes Testprotokoll bilden. Mit dieser Technologie ist auch ein In System Programming möglich.

Der Zugriff auf den Chip erfolgt durch drei Ebenen. Der Physical Layer stellt die Verbindung zum TAP (Test Access Port) Controller her. Die Verbindung ermöglicht es im JTAG Layer mittels verschiedener Scan Chains Leitungen aufzutrennen und manuell anzusteuern, sowohl nach innen, wie auch nach außen hin. Durch das Auftrennen der Adress- und Datenleitungen können im Complex Instruction Layer Anweisungen zur Ausführung an die CPU gesendet werden, die wiederrum Zugriff auf die Speicher und die Peripheriegeräte nehmen können. Die Schichten werden in den folgenden Unterkapiteln näher erklärt.

Der implementierte Debugger gdb-jtag-arm ist stark modular aufgebaut, um ihn möglichst übersichtlich zu halten. Ein Blockdiagramm der Module und deren Funktionen ist in der Abbildung B.1 zu sehen.

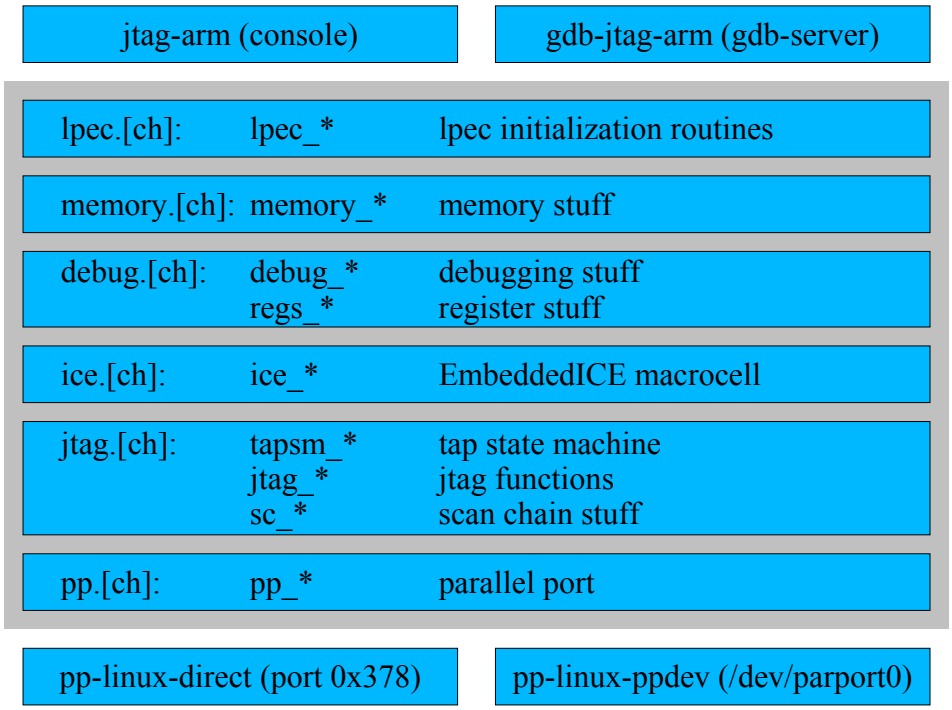

Abbildung B.1.: gdb-jtag-arm Module

B.1.1. Physical Layer

Die JTAG Schnittstelle ist ein Port mit fünf Leitungen. Sie sind in der Tabelle B.1 aufgeführt.

Abk.	Beschreibung
TCK	Test Clock (in)
TMS	Test Mode Select (in)
TDI	Test Data In (in)
TDO	Test Data Out (out)
TRST	Test Reset (in, optional)

Tabelle B.1.: JTAG Leitungen

Das übliche Pinout wird als EmbeddedICE Connector bezeichnet. Die Standardausführung bringt die fünf Leitungen auf 20 Pins unter. Die Kompaktversion, wie sie auch auf dem LPEC2001 Board vorhanden ist, hat nur 14 Pins. Die Abbildung B.2 zeigt die beiden Layouts von oben.

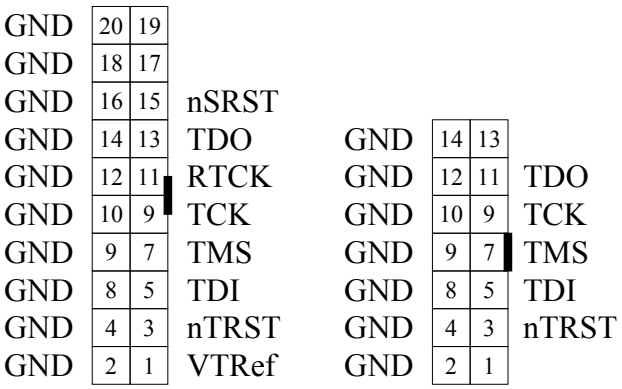

Abbildung B.2.: JTAG Pin Layouts (Oberseite)

JTAG Hardware

Es gibt etliche Schaltungsbeispiele[7], wie mittels des PCs eine Verbindung zwischen der JTAG Schnittstelle und dem PC aufgebaut werden kann. Die einfachsten bestehen nur aus einem Treiberbaustein, der mit dem bidirektionallen Parallelport des Rechners verbunden ist. Die professionellen Versionen besitzen einen eigenen Mikrocontroller[8] und benutzen häufig die schnellere USB Schnittstelle. So kann ein Großteil der Protokollarbeit reduziert und die Ausführungsgeschwindigkeit drastisch erhöht werden. Die maximale Geschwindigkeit ist dabei nur durch die maximale Frequenz der Taktleitung von 10 MHz limitiert.

[7] Vgl. [Khirman 2003]
[8] Vgl. [Atmel 2003]

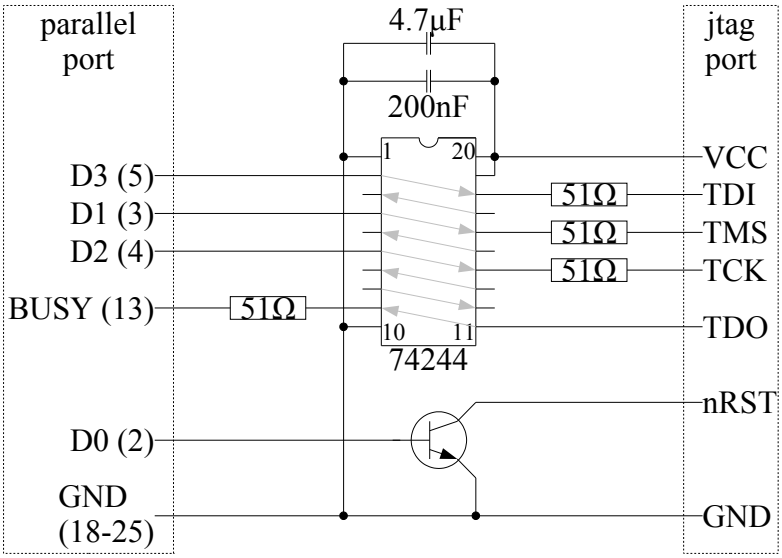

Abbildung B.3.: JTAG Wiggler Interface

Um keinen unnötigen Aufwand in die Hardware zu stecken und um Fehler zu vermeiden, fiel die Entscheidung für die einfachste Variante. Im Internet ist diese Beschaltung auch als Wiggler Interface zu finden. Die Abbildung B.3 zeigt den Schaltplan. Er besteht hauptsächlich aus dem Treiberbaustein 74244[9], einem Octal Buffer Line Driver with 3-State Outputs. Die 3-State Eigenschaft der acht enthaltenen Treiber wird nicht genutzt, die zwei Enable Eingänge liegen dauerhaft auf +5V. Die eine Richtung wird für die Signale `TCK`, `TMS` und `TDI` genutzt. Die Gegenrichtung ist nur mit `TDO` beschaltet. Die Resetleitung wird mittels eines Transistors DTC114[10] negiert. Er beinhaltet bereits die Widerstände, die eine optimale TTL Negierung ermöglichen. So ist sicher gestellt, dass die Resetleitung nicht auf Reset steht, auch wenn der PC nicht mit dem Interface verbunden ist.

Implementierung

Linux bietet zwei Möglichkeiten, den Parallelport direkt anzusteuern. Die erste Variante funktioniert durch direkten Hardwarezugriff und ist deswegen nur als Superuser möglich. Sie ist in der Datei `pp-linux-direct.c` implementiert. Die bessere Variante ist die Benutzung der dafür vorgesehenen Device Files `/dev/parport*`. Sie ist in der Datei `pp-linux-ppdev.c` implementiert.

Beide Ansteuerungen sind als Shared Objects programmiert, so dass sie zur Laufzeit automatisch eingebunden werden können. Das ermöglicht die einfache Implementierung weiterer Treiber für andere Betriebssysteme (Solaris, Windows). Eine Funktion zur automatischen Auswahl sollte in der Datei `pp.c` implementiert werden.

[9] Vgl. [Philips 1990]
[10] Vgl. [ON 2004]

Beide Varianten stellen die vier Funktionen zur Verfügung, die in der Tabelle B.2 aufgeführt sind.

Funktion	Beschreibung
`pp_init`	Initialisierung des Parallelports
`pp_done`	Rücksetzen des Parallelports
`pp_inb`	Lesen der Statuspins des Parallelports
`pp_outb`	Setzen der Datenpins des Parallelports

Tabelle B.2.: Parallelport Zugriffsfunktionen

Die Funktionen `pp_init` und `pp_done` dienen der Initialisierung und Wiederherstellung des Parallelports.

Beim Direktzugriff wird hier mittels der Systemfunktion `ioperm` der Zugriff erbeten. Schlägt dieser fehl, terminiert das Programm sofort. Die Done Funktion enthält keine Anweisungen, weil beim Beenden des Programms automatisch die Berechtigung wieder zurückgesetzt wird.

Für den Zugriff über die Device Files muss das Device geöffnet und mit einem `ioctl(fd, PPCLAIM)` Kommando die Benutzung angemeldet werden. Auch hier führt ein Fehler zum Terminieren des Programms in der Anfangsphase. Die Done Funktion muss mit der `ioctl(fd, PPRELEASE)` Funktion das Ende der Benutzung melden und kann danach das Device File schließen.

Der Parallelport hat drei wichtige Register. Das Datenregister 0 wird verwendet, um die Leitungen nach außen zu setzen. Das Statusregister 1 wird verwendet, um die rückführenden Leitungen abzufragen, die an den Statuspins des Parallelports angeschlossen sind. Das Kontrollregister 2 wird nicht verwendet.

Das Auslesen der Leitungen durch die Funktion `pp_inb` und Setzen der Leitungen durch die Funktion `pp_outb` erfolgt beim direkten Zugriff mit den Assembleranweisungen `inb` und `outb`. Beim Auslesen wird ein `inb value, PP_Port+1` auf das Statusregister 1 ausgefüuhrt und der Rückgabewert in der Variable `value` gespeichert. Das Setzen wird mit einem `outb value, PP_Port+0` auf das Datenregister 0 ausgeführt.

Beim Zugriff über die Device Files funktioniert das ähnlich, allerdings mit dem Unterschied, dass die Register jetzt sinnvoll benannt sind und ein Zugriff nicht mit Assembleranweisungen erfolgt, sondern über ioctl Aufrufe. Gelesen wird durch die Anweisung `ioctl(fd, PPRSTATUS, &value)` und geschrieben wird durch die Anweisung `ioctl(fd, PPWDATA, &value)`.

B.1.2. JTAG Layer

Die vier Leitungen ermöglichen das Ansteuern des TAP Zustandsautomaten, wie er in der Abbildung B.4 gezeigt ist. Der Zustand des Automaten wird bei jeder positiven Flanke der Taktleitung `TCK` weitergeschaltet. Der neue Zustand ist abhängig von der `TMS` Leitung.

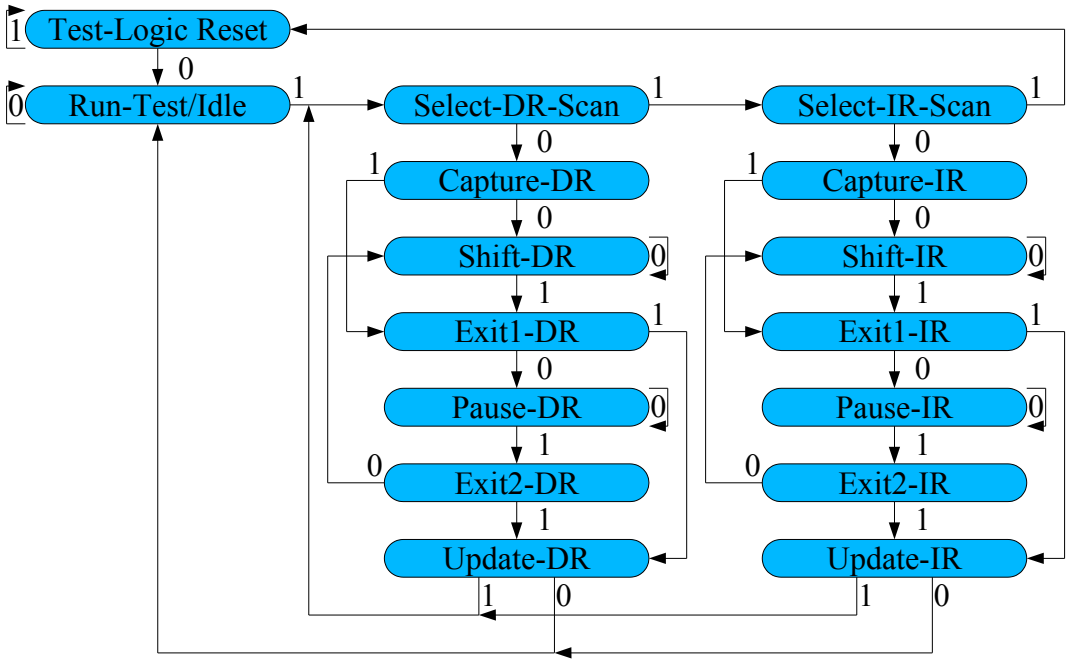

Abbildung B.4.: TAP Controller Zustandsübergänge

Durch fünfmaliges Senden von `TMS=1`, wird der Anfangszustand Test Logic Reset aus jedem beliebigen Zustand wieder erreicht. Ein Senden von `TMS=0` bringt den Automaten in den Wartezustand Run Test/Idle. Mit Hilfe des rechten IR Zweigs kann das Instruktion Register beschrieben werden. Der linke DR Zweig beschreibt das Data Register. Während der Capture Phase wird der aktuelle Zustand der Leitungen und der `IR` und `DR` Register im Chip festgehalten. Beim Traversieren der Shift Phase werden mit dem Zustand der `TDI` Leitung neue Daten Bit für Bit, zuerst LSB (Least Significant Bit), dann MSB (Most Significant Bit), in die Register und Daten bei der fallenden `TCK` Flanke auf die `TDO` Leitung raus geshiftet. In der Update Phase werden die neuen Daten dann gesetzt.

ARM9 JTAG Layer

Jede JTAG Schnittstelle hat spezifische Eigenschaften. Beim ARM9TDMI hat das `IR` Register eine Länge von 4 Bits. Bestimmte Kommandos können die CPU in den Test Reset Mode versetzen und damit anhalten, um den Zustand zu prüfen oder wieder zurück in den System Mode bringen. Der Rest des Systems kann dabei ungehindert weiterarbeiten, weil das Bussystem nach außen hin mit normaler Geschwindigkeit weiter arbeitet. Die Tabelle B.3 enthält eine kurze Übersicht über die bekannten Kommandos. Alle nicht belegten Instruktionskodes verhalten sich wie das `BYPASS` Kommando. Das Testen bestimmter Pins erfolgt über Scan Chains, in denen die Signale getrennt und in beide Richtungen einzeln getestet oder gesetzt werden können.

Kommando	Hexcode	Beschreibung
EXTEST	0000	places the selected scan chain in test mode and capture/-set the signals from the system logic to the output scan cells and the signals from the core logic to the input scan cells
SCAN_N	0010	selects the scan chain
SAMPLE_PRELOAD	0011	never use
RESTART	0100	restart the processor on exit from debug state
CLAMP	0101	set signals previously loaded into the currently loaded scan chain.
HIGHZ	0111	set all signals to high impedance state
CLAMPZ	1001	place all 3-state outputs in their inactive state
INTEST	1100	place the selected scan chain in test mode and capture/-set the signals from the core logic to the output scan cells and the signals from the system logic to the input scan cells
IDCODE	1110	read out the device identification register
BYPASS	1111	read out pins in system mode

Tabelle B.3.: ARM9 JTAG Kommandos

ARM9 Scan Chains

Im ARM9TDMI sind nur die ersten drei der 32 möglichen Scan Chains beschaltet. Nach einem Reset ist die Scan Chain 3 aktiv. In der Abbildung B.5 ist eine Übersicht der drei Scan Chains um den ARM9 Prozessorkern skizziert. Die Tabelle B.4 gibt einen Überblick über die Namen und der bei ARM üblichen weiteren Scan Chains und deren Signale. Eine Scan Chain wird mit Hilfe des SCAN_N Kommandos selektiert. Die INTEST und EXTEST Kommandos ermöglichen das Auftrennen und Testen der Leitungen nach innen und außen.

Nr.	Name	Beschreibung
0	Macrocell scan test	entire periphery of the ARM9TDMI core: data bus, control signals, address bus
1	Debug	only data bus and BREAKPT signal
2	EmbeddedICE programming	Embedded ICE unit control signals
3	External boundary scan	not ARM9TDMI
4	reserved	
8	reserved	
15	System Control Coprocessor	not ARM9TDMI

Tabelle B.4.: ARM9TDMI scan chains

Die Scan Chain 0 dient dem Macrocell Scan Test des ARM9TDMI. Sie liegt über allen Leitungen, die mit dem Prozessorkern verbunden sind, wie es in der Abbildung B.5 gezeigt ist. Sie ist mit 184 Bits die längste Scan Chain.

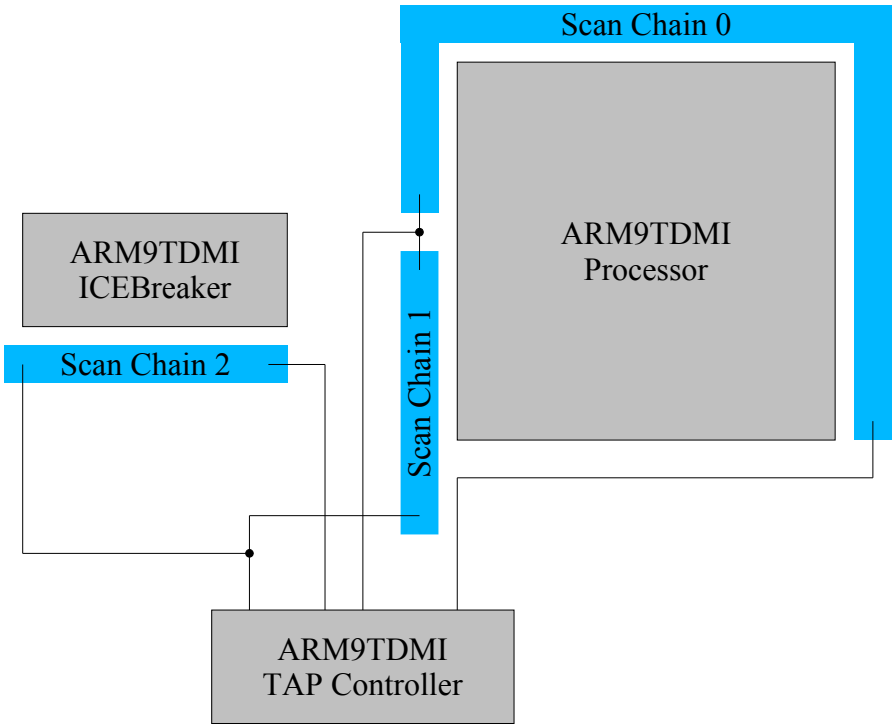

Abbildung B.5.: ARM9TDMI scan chain arrangement

Die Scan Chain 1 dient dem Debugging der Software. Sie liegt nur über dem Datenbus und dem `BREAKPT` Signal, zum Anhalten und Fortsetzen des Prozessors. Diese Leitungen reichen aus, um dem Prozessor Assembleranweisungen zur Ausführung zu übergeben. Neben den Instruktionen werden auch die Daten über den Bus transferiert, so dass in der richtigen Pipelinestufe ein Zugriff darauf möglich ist.

Die Scan Chain 2 ist eine weitere hilfreiche Scan Chain, die zum Debuggen der Software dient. Sie steuert die Signale der ICEBreaker Unit und ermöglicht eine Konfiguration der enthaltenen Register. Sie ermöglicht u.a. das Programmieren von Hard- und Software Break- und Watchpoints.

Die Nutzung der Scan Chain 1 zum Debuggen der Software und dem Zugriff auf Speicher und weitere Hardwarekomponenten wird im Kapitel B.1.3 beschrieben.

Implementierung

Die Funktionen des JTAG Layers wurden in der Datei `jtag.c` implementiert. Drei Funktionsklassen sind vertreten: `tapsm` Funktionen steuern die Pulse zum Durchschreiten der TAP State Machine und dem Auslesen und Setzen eines Registers. Die `jtag` Funktionen ermöglichen speziell das Auslesen und Programmieren der Instruktions- und Datenregister. Die aktuelle Scan Chain wird über die `sc` Funktionen ermittelt und ausgelesen.

B.1.3. Complex Instruction Layer

Im Complex Instruction Layer werden komplexere Instruktionen mit Hilfe der o.g. Scan Chains ausgeführt. Die folgenden Unterkapitel beschreiben die Funktionen der EmbeddedICE Macrocell und den Zugriff auf die internen Prozessorfunktionen und -register. Zuletzt wird der Prozessor benutzt, um Zugriff auf externe Komponenten (Speicher) zu nehmen.

EmbeddedICE Macrocell

Der Zugriff auf die EmbeddedICE Register erfolgt durch einen lesenden oder schreibenden Zugriff auf die Scan Chain 2. Sie hat eine Länge von 38 Bits. Die ersten 32 Bits enthalten den EmbeddedICE Registerinhalt. Die folgenden fünf Bits adressieren das Register. Und das letzte Bit gibt an, ob es sich um einen Lese- (0) oder Schreibzugriff (1) handelt.

Die EmbeddedICE Registerbelegung und deren Funktionen sind in der Tabelle B.5 beschrieben.

Adr.	**Bits**	**Beschreibung**
0	4	debug control
1	5	debug status
2	8	vector catch control
4	6	debug comms control
5	32	debug comms data
8	32	watchpoint 0 address value
9	32	watchpoint 0 address mask
10	32	watchpoint 0 data value
11	32	watchpoint 0 data mask
12	9	watchpoint 0 control value
13	8	watchpoint 0 control mask
16	32	watchpoint 1 address value
17	32	watchpoint 1 address mask
18	32	watchpoint 1 data value
19	32	watchpoint 1 data mask
20	9	watchpoint 1 control value
21	8	watchpoint 1 control mask

Tabelle B.5.: EmbeddedICE Register

Die Debug Control Register ermöglichen das Versetzen des Prozessors in den o.g. Debug Modus (Test Reset Mode). Das Erreichen dieses Zustands und das Prüfen auf Beendigung aller Speicherzugriffe wird durch das Debug Status Register angezeigt. Gegenüber dem Control Register des ARM7TDMI, besitzt der ARM9 die Fähigkeit über dieses Register Assembler Anweisungen im Single Step Modus auszuführen.

Das Vector Catch Register ermöglicht das Anhalten des Prozessors beim Zugriff auf bestimmte Exception Vektoren. Diese Vektoren zeigen auf Routinen, die bei Ausnahmebehandlungen ausgeführt werden. Folgende Ausnahmebedingungen existieren: Reset, Undefined Instruction, Software Interrupt, Prefetch Abort, Data Abort, IRQ und FIQ.

Die Debug Comms Register ermöglichen eine serielle Kommunikation über die JTAG Schnittstelle mit dem Programm. Die Software muss dafür entsprechende Routinen besitzen, die Ein- und Ausgaben über das spezielle Koprozessorregister 14 ermöglichen.

Hard-/Software Break-/Watchpoints	**Anhalten durch**
Hardware Breakpoint	das Ausführen einer beliebigen Instruktion an einer angegebenen Adresse
Hardware Watchpoint	den Zugriff auf ein beliebiges Datum an einer angegebenen Adresse
Software Breakpoint	das Ausführen einer angegebenen Instruktion an einer beliebigen Adresse
Software Watchpoint	den Zugriff auf ein angegebenes Datum an einer beliebigen Adresse

Tabelle B.6.: Hard-/Software Break-/Watchpoints

Die EmbeddedICE Macrocell hat zwei Watchpoint Register. Die Tabelle B.6 zeigt die vier Konfigurationsmöglichkeiten. Die Instruktionen und Daten werden mit Hilfe der Address und Data Value Register festgelegt. Durch die Address und Data Mask Register lassen sich Masken definieren, die auch bestimmte Abweichungen und Adressbereiche der definierten Muster ermöglichen. Über das Control Register wird der Watchpoint aktiviert und die Betriebsart und Größe des Datenzugriffs festgelegt.

Die Software bietet den Zugriff über die `ice` Funktionen des Moduls `ice.c` an.

Debugging- und Registerfunktionen

Dem Ausführen von Instruktionen im Prozessor geht immer das Anhalten und das Sichern der Register vorweg, damit das Programm später unverändert fortgesetzt werden kann. Das Listing B.1 zeigt diese Instruktionssequenz.

```
STR     r0, [r0]        // Save R0 before use
MOV     r0, pc          // Copy PC into R0
STR     r0, [r0]        // Now save the PC in R0
NOP                     // Read out R0
NOP
NOP                     // Read out PC

STMIA   r0, {r1-r14}    // Save R1-R14
NOP
NOP
NOP                     // Save R1
NOP                     // Save R2
...
NOP                     // Save R14
```

Listing B.1: Ausschnitt aus debug.c: Register sichern

Wie an diesem Codeausschnitt gut zu sehen ist, erfolgen die Transfers von `R0` und `PC` drei Pipelinestufen später während der `NOP` Anweisungen. Zu diesem Zeitpunkt können die Daten auf dem Bus gelesen werden. Ähnlich funktioniert das Auslesen der weiteren Register `R1` bis `R14`. Auch hier ermöglicht eine Verzögerung durch zwei `NOP` Anweisungen das Auslesen der Daten auf dem Datenbus.

Das Rückschreiben funktioniert in ähnlicher Weise, allerdings unter Verwendung der `LDMIA` und `LDR` Anweisungen.

Die `debug` Funktionen wurden im gleichnamigen Modul `debug.c` implementiert. Die `regs` Funktionen dienen dem Auslesen und Setzen der Register.

Speicherzugriff

Der Speicherzugriff kann nach dem Sichern der Register durch die im Listing B.2 gezeigten Instruktionen erfolgen.

```
LDR     r0, [r0]        // put address in r0
NOP
NOP
NOP                     // scan in address
LDR     r1, [r0]        // load r1 from location
NOP                     // run this cycle at system speed
STR     r1, [r0]        // output the data value
NOP
NOP
NOP                     // read out
```

Listing B.2: Ausschnitt aus memory.c: Speicher lesen

Drei Taktzyklen nach den Befehlen wird die Adresse auf dem Datenbus überschrieben und somit stattdessen das übergebene Datum in das Register `R0` geschrieben. Der Zugriff auf die Speicherzelle durch das `LDR` Kommando muss in Systemgeschwindigkeit erfolgen, da sonst keine Synchronisation mit dem externen Bus- und Speichersystem zustande kommt. Die `STR` Anweisung ermöglicht drei Taktzyklen später das Auslesen des Speicherinhalts.

Das Schreiben von Speicher erfolgt in ähnlicher Weise, allerdings durch die Verwendung der Instruktionen `LDMIA` und `STR`.

Die `memory` Funktionen wurden im Modul `memory.c` implementiert.

B.1.4. spezielle LPEC Funktionen

Zwar sind die JTAG Funktionen für jede ARM9 CPU benutzbar, jedoch sind spezielle Funktionen für den Betrieb mit dem LPEC2001 Board erforderlich. Diese sind explizit in das Modul `lpec.c` ausgelagert.

Der P2001 besitzt einen Watchdog Timer, der bei jedem Nulldurchgang des Zählers einen Reset auslöst. Das Abschalten des Watchdogs erfolgt durch das Setzen eines Bits im Timer Register, also durch einen Speicherzugriff.

Weiterhin besitzt das LPEC Board externen SDRAM. Um diesen zu benutzen, muss zuvor der Memory Controller aktiviert und konfiguriert werden. Auch dies erfolgt durch einen Registersatz, der mit Hilfe der Speicheroperationen konfiguriert wird.

B.1.5. Konsolenfrontend

Diese Backend Funktionen werden durch zwei Frontends dem Benutzer angeboten. Das `jtag-arm` Frontend stellt im Prinzip dieselbe Funktionalität, wie das ursprüngliche `jtag-arm9` Programm zur Verfügung und dient primär dem Debuggen der neuen und erweiterten Funktionen. Das zweite Frontend `gdb-jtag-arm` wird im folgenden Kapitel B.2 beschrieben.

Das Konsolenfrontend ermöglicht die Nutzung aller JTAG Funktionen, sowie das Auslesen von Speicherbereichen und das Uploaden und Starten von Programmen. Eine Befehlsliste ist in der Tabelle B.7 aufgeführt.

Kommando	**Beschreibung**
`#`	Kommando ignorieren
`script`	Kommandos aus externem Skript ausführen
`echo`	Ausgabe von Text
`pause`	Auf Eingabe warten
`quit`	Programm beenden
`help`	Hilfe anzeigen
`jtag`	JTAG Reset durchführen
`idcode`	IDCode ausgeben und aufschlüsseln
`stop`	Prozessor anhalten
`step`	Single Step ausführen
`restart`	Prozessor fortsetzen
`run <address>`	Fortfahren des Programms ab Adresse
`read <address> [<length>]`	Speicher ab Adresse ausgeben
`write <address> <data>`	Speicher schreiben
`load <filename> <address> [<length>]`	Image laden
`regs [<reg> <value>]`	Register anzeigen oder setzen
`bphw <0..1> [<addr> <mask>]`	Hardware Breakpoint setzen
`bpsw <0..1> [<pattern>]`	Software Breakpoint setzen

Tabelle B.7.: jtag-arm Befehle

B.2. GDB Server

Normalerweise wird der GNU Debugger[11] zur Fehlerfindung eines lokal laufenden Programms genutzt. Durch ein Textinterface sind das Anhalten, das Setzen von Breakpoints, der Speicherzugriff und alle sonst üblichen Funktionen zum Debuggen eines Programms gegeben[12].

Mittlerweile gibt es neben dem normalen textuellen Interface, auch einfach zu bedienende graphische Oberflächen, wie die GDB/Insight[13] Kombination.

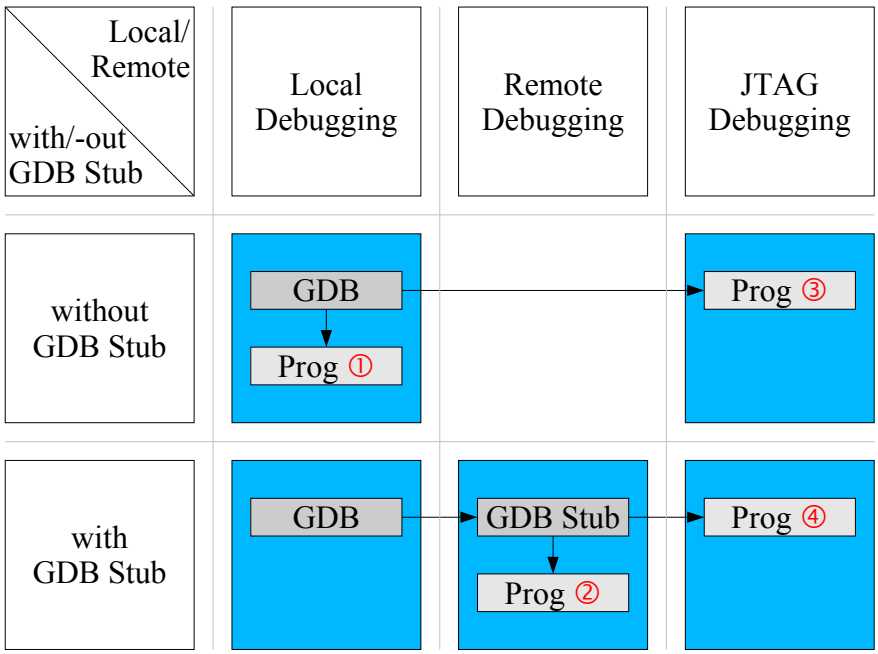

Abbildung B.6.: Matrix der GDB (-Stubs)

Die Abbildung B.6 zeigt die vier Möglichen Situation, wie der GDB eingesetzt werden kann.

Wie erwähnt läuft sowohl der GDB als auch das zu analysierende Programm normalerweise auf einem Rechner (Punkt 1). Es ist jedoch auch möglich, den GDB als TCP Server (GDB Stub) auf einem weiteren Rechner laufen zu lassen (Punkt 2). Das Frontend lässt sich dann über das Netzwerk anbinden. Diese Kombination bringt eine hohe Sicherheit, dass ein fehlerhaftes Programm nicht zum Absturz der Entwicklungsstation mit dem darauf laufenden Frontend führt.

Die Möglichkeit der Trennung von Frontend und Backend nutzt auch der Macraigor GDB/Insight Debugger[14], um ein Debuggen der Zielplattform über die JTAG Schnittstelle

[11] Vgl. [GNU 2004]
[12] Vgl. [Stallman et al. 2004]
[13] Vgl. [RedHat 2004a]
[14] Vgl. [Macraigor 2004]

durch einen GDB Stub zu ermöglichen (Punkt 3). Dieser GDB Stub ist leider nur begrenzt einsetzbar, weil er auf spezielle Kernelmodule aufsetzt, die nicht allgemein benutzbar sind.

Andere JTAG Debugger haben anstatt einen speziellen GDB Stub zu schreiben, den GDB direkt erweitert (Punkt 4)[15]. Diese Lösung ist allerdings nicht gut nutzbar, weil sich das Programm in einer permanenten Weiterentwicklung befindet. Das Netzwerkprotokoll wird dagegen nur erweitert und ist abwärtskompatibel.

Natürlich ist es auch möglich, den GDB Stub und das Frontend auf dem selben Rechner zu betreiben und sie durch einen Netzwerksocket kommunizieren zu lassen. Für die Nutzung der implementierten Lösung mit dem JTAG-fähigen GDB Stub ist daher auch die Nutzung eines PCs ausreichend.

B.2.1. TCP Server

Der GDB Stub stellt seine Dienste über einen TCP Server zur Verfügung. Unter Linux gibt es mehrere Varianten, einen TCP Dienst zu implementieren. Die einfachste Variante ist durch die Nutzung eines Internet Super Servers[16] gegeben. Dieser startet beim Verbindungsaufbau mit dem Netzwerksockel ein angegebenes Programm und leitet die Ein- und Ausgaben (STDIO, STDOUT) auf Netzwerksockel um. Wird eine Verbindung via Telnet mit diesem Netzwerkport aufgebaut, kann das Programm benutzen werden, so als wäre es lokal gestartet worden.

Die bessere Variante ist die Implementierung eines eigenen TCP Servers. Dieser integriert sich bei diesem Projekt auch besser in die bestehende Struktur. Es gibt mehrere Varianten der Umsetzung, je nachdem, ob er nach einer Verbindung terminieren oder bei jeder Anfrage einen weiteren Bearbeitungsprozess erzeugen soll[17].

Für dieses Projekt macht nur ein einmaliges Starten Sinn, weil die Platine nur durch einen GDB simultan bearbeitet werden kann. Linux orientiert sich bei den Netzwerkfunktionen an denen von BSD Unix. Die Funktionen sind weitgehend identisch. Zuerst muss ein TCP (Stream) Socket erstellt werden. Dafür gibt es die Funktion `socket`. Dem nun erstellten Socket wird mit dem `bind` Kommando ein Port zuweisen. Der GNU Stub nutzt den TCP Netzwerkport 8888. Mit `listen` wird auf eine Anfrage gewartet und mit `accept` wird diese akzeptiert. Danach kann die Verbindung als normaler Kommunikationskanal genutzt werden. Geschlossen werden Kommunikationskanäle immer mit `close`. Die wichtigsten Kommandos sind im Listing B.3 gezeigt.

```
socket(AF_INET, SOCK_STREAM, 0);

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(8888);
bind(socket_desc, (struct sockaddr *)&address, sizeof(address));

addrlen = sizeof(address);
listen(socket_desc, 1);
```

[15] Vgl. [Mohor 2004]
[16] Vgl. [Ermer und Meyer 2004]
[17] Vgl. [Pont 2002] [Frost 1996] [SCO 1997]

```
client_socket = accept(socket_desc, (struct sockaddr*)&address, &addrlen);

while(client_socket) {
        read(client_socket, &byte, 1);
        write(client,socket, &byte, 1);
}

close(client_desc);
```

Listing B.3: Implementierung eines einfachen TCP Servers

B.2.2. GDB Protokoll

Das GDB Remote Serial Protocol, welches zur Kommunikation zwischen dem GDB Frontend und dem Backend benutzt wird, ist im GDB User Manual dokumentiert[18]. Das GDB Internals Manual[19] beschreibt dessen Implementierung.

Im Internet finden sich zudem Zusammenfassungen der wichtigsten Kommandos[20] und weitere Projekte, die das Protokoll direkt nutzen, um mit dem GDB Frontend zu kommunizieren[21].

Das Protokoll selbst ist textorientiert aufgebaut. Alle GDB Kommandos und Antworten (außer Bestätigungen) werden als Pakete versendet. Ein Paket beginnt mit dem Buchstaben $, gefolgt von den eigentlichen Paketdaten, und schließt mit einem #, gefolgt von zwei Prüfzeichen ab. Die Prüfsumme besteht aus der Summe aller Zeichen zwischen dem beginnenden $ und dem abschließenden # modulo 256. Bestätigungen werden entweder positiv als + oder negativ als - gesandt.

Die Tabelle B.8 zeigt die wichtigsten und implementierten Kommandos.

Kommando	**Antwort**	**Beschreibung**
`c<addr>`	stop reply packet	continue
`D`	no response	detach
`g`	`<X><X>...`	read registers
`G<X><X>...`	`OK` or `E<NN>`	write registers
`m<addr>,<length>`	`<X><X>...`	read memory
`M<addr>,<length>:<X><X>...`	`OK` or `E<NN>`	write memory
`z<type>,<addr>,<length>`	`OK`, no response or `E<NN>`	remove breakpoint or watchpoint
`Z<type>,<addr>,<length>`	`OK`, no response or `E<NN>`	insert breakpoint or watchpoint

Tabelle B.8.: GDB Remote Serial Protocol

[18] Vgl. [Stallman et al. 2004]
[19] Vgl. [Gilmore und Shebs 2004]
[20] Vgl. [Gatliff 1999]
[21] Vgl. [Finneran 2003]

Eine kurze Protokollsequenz ist im Listing B.4 zu sehen. Sie zeigt die Abfrage der Registerbank (`+$g67`), das Ausführen einer Assembleranweisung im Single Step Modus (`+$s73`), das Setzen eines Breakpoints (`+$Z0,3a4,4de`), das Lesen (`+$m3a4,495`) und Schreiben (`+$M3a4,4:e7ffdefeab`) eines Softwarebreakpointmusters an der Speicheraddresse `0x3a4` und das Fortsetzen des Programms (`+$c64`).

```
+$g#67
Command: g - Checksum IN:67, CA:67
Registers:
  R00 - 0x00180000   R08 - 0x00000000
  R01 - 0xe7ffdefe   R09 - 0x00000000
  R02 - 0x00000000   R10 - 0x00000000
  R03 - 0x0000b80a   R11 - 0x00000000
  R04 - 0x000000fe   R12 - 0x00000821
  R05 - 0x000085b8   R13 - 0x000002ec
  R06 - 0x00000000   R14 - 0x00000000
  R07 - 0x00000000   R15 - 0x000003d8

  PC  - 0x000003d8
Sending: $00180000e7ffdefe000000000000b80a000000fe000085b800000000000000000000000
0000000000000000000000000000000821000002ec00000000000003d800000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000040000050#5f
(read registers)

+$s#73
Command: s - Checksum IN:73, CA:73
Stepping over software breakpoint 0
debug control is 000C (interrupt disable)(single stepping)
debug status is 0000C (interrupt enable)(system complete)
System HALTED in 32bit State
R0: 00180000   PC: 000003dc   CPSR: 40000050
Sending: $S05#b8
(step)

+$Z0,3a4,4#de
Command: Z0,3a4,4 - Checksum IN:de, CA:de
(set breakpoint)
Unbekanntes BP-Kommando: @, Addresse: bfffd304, Bereich 0804cbf6, Maske 4014a040
Sending: $#00

+$m3a4,4#95
Command: m3a4,4 - Checksum IN:95, CA:95
Sending: $e59f505c#36
(read memory)

+$M3a4,4:e7ffdefe#ab
Command: M3a4,4:e7ffdefe - Checksum IN:ab, CA:ab
Sending: $OK#9a
(write memory)

+$c#63
```

Listing B.4: GDB RDB Protokollsequenz