



Java on the bare ARM metal

Masterthesis 2018

von

Marcel Gehrig

Referent:
Experte:
Abgabedatum:

Prof. Dr. Urs Graf
Bernhard Sprenger
10. August 2018

Abstract

The compiler from the *deep*-project can be used to compile Java code directly for an embedded PowerPC processor. It is possible to develop a simple motor controller or a complex robot controller with real-time capability in plain Java.

PowerPC processors are no longer in widespread use. They are now used only for some special applications. For this reason, *deep* is being developed for the use with ARM processors.

deep is currently used for education at the NTB and it should remain usable after the switch to the ARM micro-architecture.

For the development of the *deep* runtime environment, a hardware debugger is needed to read and write memory and processor registers.

gdb is a software debugger, which is popular for programming languages like C and C++. It has features like source code lookup, single stepping and breakpoints. These are really helpful features and they should also be provided for the development of *deep* applications.

In this paper an ARM processor and a matching development board will be evaluated. The development board should be cheap enough for NTB lessons. The processor has to have very good connectivity for an FPGA so that demanding robotic projects can be implemented with the same processor.

A toolchain is then designed, which can be used, to write a *deep* application to the memory of the processor.

The existing software interface for the hardware debugger will be adapted, so it can be used with the new toolchain.

A *deep* application will be supplemented with STABS debug information. The supplemented application can then be compiled and executed on the ARM processor. *gdb* can then be used on a host computer to debug the *deep* application on the ARM processor using features like source code lookup and single stepping.

Zusammenfassung

Das *deep*-Projekt ermöglicht es schon jetzt, eine Java-Applikation direkt für PowerPC-Prozessoren zu entwickeln. Dadurch ist es möglich, eine echtzeitfähige Regelung oder Robotersteuerung in Java zu entwickeln.

Prozessoren auf der Basis der PowerPC-Architektur sind aber nicht mehr weit verbreitet und zu einem Nischenprodukt geworden. Aus diesem Grund wird *deep* zurzeit für die ARM-Architektur weiterentwickelt.

deep wird für den Unterricht an der NTB verwendet und soll auch nach der Umstellung zur ARM-Architektur weiterhin für den Unterricht verwendet werden können.

Um die Laufzeitumgebung von *deep* entwickeln zu können, wird ein Hardware-Debugger benötigt, der den Speicher und die Register eines ARM-Prozessors lesen und schreiben kann.

Der *gdb*-Debugger ist ein weit verbreiteter Software-Debugger, der besonders für Sprachen wie C und C++ verwendet wird. Er bietet Features wie Sourcecode-Lookup, Single-Stepping und Breakpoints. Diese Features sind sehr nützlich und sollen auch für die Entwicklung von *deep*-Applikationen zur Verfügung stehen.

In dieser Arbeit wird ein ARM-Prozessor, inklusive Experimentierboard gesucht, der günstig genug für den Unterricht ist und eine sehr gute Anbindung an ein FPGA hat, damit auch anspruchsvolle Robotik-Projekte damit realisiert werden können.

Für diese Hardware wird dann eine Toolchain entwickelt, mit der ein von *deep* kompiliertes Programm in den Speicher des Prozessor geladen und gestartet werden kann.

Das bestehende Software-Interface für den Hardware-Debugger wird so angepasst, dass es mit der neuen Toolchain kompatibel ist.

Es wird gezeigt, dass der *gdb*-Debugger, inklusive der obengenannten Features, genutzt werden kann, um eine *deep*-Applikation zu debuggen, die auf dem ARM-Prozessor läuft. Dafür wird eine *deep*-Applikation mit STABS-Debuginformationen erweitert.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Stand der Technik	1
1.2	Motivation	1
1.3	Zielsetzung	1
2	Auswahl der Hardware	2
2.1	Soll-Kriterien und Muss-Kriterien bei der Auswahl der Hardware	2
2.2	Was ist ein Hardware Debugger	3
2.3	Übersicht über die ARM Mikroarchitekturen	3
2.4	Anbindung des FPGAs an den Prozessor	5
2.5	Fazit - Auswahl der Hardware	7
3	Überblick über das ganze System	8
3.1	Schematische Übersicht	8
3.2	Debugger Toolchains	10
4	Das SOC Zynq-7000	12
4.1	MIO und EMIO	12
4.2	Standard Zybo Workflow	13
4.3	Memory Mapping	17
4.4	Floating Point Unit	17
5	Die OpenOCD Software	21
5.1	Softwareinstallation der OpenOCD-Toolchain	21
5.2	OpenOCD CLI - Command Line Interface	22
5.3	OpenOCD Konfiguration	22
5.4	CLI-OpenOCD-Toolchain	25
6	Das ELF-Dateiformat mit STABS Debuginformationen	26
6.1	Nützliche Tools im Umgang mit ELF-Dateien	26
6.2	Grundlegender Aufbau	26
6.3	STABS Debuginformationen	27
6.4	Demoprogramm mit STABS	28
7	Der gdb-Debugger	34
7.1	Installation der gdb-Software mit der "GNU Embedded Toolchain"	34
7.2	<i>gdb</i> -Anwendungsbeispiel: "loopWithSTABS" auf das Zybo laden	34
7.3	Test der <i>gdb</i> -Funktionen	34

8 Kundennutzen, Fazit und Ausblick	36
8.1 Rückblick und Kundennutzen	36
8.2 Ausblick	36
8.3 Fazit	36
9 Danksagung	37
10 Ehrenwörtliche Versicherung	38
11 Quellenverzeichnis	39

1 Einleitung

1.1 Stand der Technik

Das Projekt *deep*¹ ist eine Cross Development Plattform, die es erlaubt, ein Java Programm direkt auf einem Prozessor auszuführen. Es ermöglicht einem Entwickler ein Java Programm zu schreiben, welches direkt auf einem Prozessor läuft und echtzeitfähig ist. Zur Zeit wird dieses Projekt an der NTB für die Ausbildung von Systemtechnik-Studenten verwendet. Es erlaubt einfach und schnell Robotersteuerungen und Regelungen zu implementieren, ohne dass sich der Entwickler mit den Eigenarten von C und C++ Programmen auseinandersetzen muss.

deep unterstützt einige grundlegende Debugging-Funktionen. Mit einer mehreren tausend Franken teuren Abatronsonde kann der Speicher und die Register des Prozessors ausgelesen und auch geschrieben werden. Der aktuelle Debugger unterstützt keine Features wie *Breakpoints* oder *Sourcecode-Navigation*, die aus Debuggern wie dem *gdb*² bekannt sind.

1.2 Motivation

Aktuell ist *deep* nur mit der PowerPC-Architektur kompatibel. PowerPC Prozessoren sind aber nicht mehr weit verbreitet und sehr teuer. Die an der NTB verwendeten PowerPC-Prozessoren sind zwar leistungsstark, aber veraltet und teuer.

Aus diesem Grund wird *deep* zurzeit für die ARM-Architektur erweitert. Da die ARM-Architektur bei eingebetteten Prozessoren am weitesten verbreitet ist, ist auch die Auswahl an günstiger und leistungsstarker Hardware sehr gross. Dank der grossen Auswahl von ARM-Prozessoren können sehr günstige oder auch sehr leistungsstarke Prozessoren ausgewählt werden.

deep ist ein Open-Source-Projekt, welches auch für den Unterricht verwendet wird. Damit nicht für jeden Studenten teure Debugging-Hardware gekauft werden muss, ist eine kostengünstige Alternative wünschenswert.

Java ist im Gegensatz zu C und C++ eine sehr zielorientierte Sprache. Beim Java muss man sich nicht so detailliert um Ressourcen, wie Speicher und Hardwareschnittstellen kümmern, wie in C-orientierten Sprachen. Dieser Aspekt der Einfachheit soll auch beim Debugger beibehalten werden. Zusätzlich zum direkten Speicherauslesen sollen auch Variablen gelesen und geschrieben werden können. Eine native *Sourcecode-Navigation* in Eclipse vereinfacht die Entwicklung einer *deep*-Applikation sehr.

1.3 Zielsetzung

Bei dieser Arbeit werden mehrere Ziele verfolgt, die aufeinander aufbauen.

1. Passende Hardware (Experimentierboard) finden, welche auch im Unterricht verwendet werden kann.
2. Das grundlegende Debug-Interface, welches bereits für PowerPC existiert, für die ausgewählte Hardware anpassen. Dieses Interface soll für die Entwicklung von *deep* möglichst bald einsatzbereit sein.
3. Den GNU-Debugger (*gdb*) mit einem Programm verwenden, das vom *deep*-Compiler übersetzt wurde. Dazu soll vorerst das Command-Line-Interface (CLI) des *gdb* genutzt werden.
4. Den *gdb* in das Eclipse-Plugin von *deep* integrieren, damit der Debugger direkt aus Eclipse verwendet werden kann.

¹<http://www.deepjava.org/start>

²<https://www.gnu.org/software/gdb/>

2 Auswahl der Hardware

Die Auswahl von Hardware mit ARM-Prozessoren ist extrem gross. Ende September 2016 sind bereits über 86 Milliarden ARM-basierte Prozessoren verkauft worden.¹ Diese Zahl reflektiert zwar nicht direkt die Diversität der verschiedenen Prozessoren, aber sie zeigt recht gut wie enorm weit ARM-Prozessoren verbreitet sind.

In diesem Kapitel soll aus dem riesigen Angebotsdschungel die richtige Hardware ausgewählt werden, auf der diese Arbeit aufbauen kann. Die ausgewählte Hardware soll nicht nur für diese Arbeit genutzt werden, sondern später auch für den Robotik-Unterricht. Zusätzlich sollte der Prozessor auch leistungstark und flexibel genug sein, um ihn, oder eine Variante aus der gleichen Familie, in anspruchsvollen Robotikprojekten verwenden zu können.

2.1 Soll-Kriterien und Muss-Kriterien bei der Auswahl der Hardware

Für die Hardware sind folgende Soll-Kriterien und Muss-Kriterien ermittelt worden.

2.1.1 Muss-Kriterien

- Systemebene
 - FPGA: Der Prozessor muss mit einem FPGA kommunizieren können.
 - Hardware Debugger: Der Prozessor muss für die Entwicklung von *deep* einen Hardware Debugger, wie beispielsweise das BDI3000, von Abatron unterstützen.
 - Günstiger Programmierer: Wenn zusätzliche Hardware benötigt wird, um die *deep*-Applikation auf das Target zu schreiben, dann muss diese möglichst günstig sein.
 - Grosses Ökosystem: Das ausgewählte Produkt muss von einem grossen Ökosystem unterstützt werden. Aussterbende Produkte oder Nischenprodukte sind nicht akzeptabel.
 - Als fertiges Modul erhältlich: Für den Unterricht ein eigenes PCB entwickeln und herstellen ist keine Option.
 - Einbettbar: Der Prozessor muss auch bei einem selbstentwickelten PCB verwendet werden können. Wahlweise als SOM (*System On Module*) oder direkt als Prozessor im eigenen Package.
 - Die Hardware muss noch lange erhältlich bleiben.
 - FPU (*Floating Point Unit*): Für Gleitzahlenarithmetik.
 - Netzwerkschnittstelle: RJ-45 inklusive MAC (*Media Access Control*) und *Magetics*.
 - USB: USB Schnittstelle als Host und als Slave.
 - Flash: Mehr als 50kByte Flash.
 - RAM: Mehr als 100kByte RAM.
- Prozessorebene
 - ARMv7: Der Prozessor muss auf einer ARMv7 ISA (*Instruction Set Architecture*) basieren.
 - ARM Instruktionen: Der Prozessor muss ARM Instruktionen unterstützen. *Thumb* Instruktionen sind nicht ausreichend.

¹ Direkter Link: <https://www.arm.com/-/media/arm-com/news/ARM-media-fact-sheet-2016.pdf>
Archivierter Link: <https://web.archive.org/web/20180808122231/https://www.arm.com/-/media/arm-com/news/ARM-media-fact-sheet-2016.pdf>

2.1.2 Soll-Kriterien

- Systemebene
 - Einfach einbettbar: Der Prozessor ist als Prozessormodul erhältlich, sodass das Design von einem selbstentwickelten PCB einfacher wird.
 - Günstiger Hardware Debugger: Der Hardware Debugger kann auch für die Applikationsentwicklung mit *deep* eingesetzt werden.
 - Möglichst schneller Download der Applikation.
- Prozessorebene
 - Memory Mapped Bus für FPGA Schnittstelle.
 - FPU unterstützt *Double Precision*.
 - Integerdivision
 - Prozessortakt über 500MHz.

2.2 Was ist ein Hardware Debugger

Der Begriff *Hardware Debugger* ist nicht eindeutig definiert. Im einfachsten Fall kann ein Hardware Debugger nur einen *Boundary Scan* durchführen, wie es ursprünglich für JTAG vorgesehen war. Bei *Boundary Scan* können die I/O Pins von einem Prozessor gelesen und auch gesetzt werden. Mit solch einem Scan kann während der Produktion bei den bestückten PCBs überprüft werden, ob alle Lötstellen Kontakt herstellen und dabei keine Kurzschlüsse bilden. Für diesen Scan wird der Prozessor-Kern nicht verwendet, sondern eine separate Peripherie im Prozessor. Über das JTAG-Interface kann der Scan ausgeführt werden, ohne dass eine Applikation auf dem Prozessor ausgeführt werden muss.

Moderne Prozessoren erweitern diese grundlegendsten Funktionen mit einigen sehr hilfreichen Features. So bieten ARM-Prozessoren mit der *CoreSight*-Technologie noch viel mehr als nur einen *Boundary Scan*. Die untenstehende Liste zeigt einige Funktionen dieser Technologie, aber nicht alle. Die für diese Arbeit relevanten Funktionen sind **fett** geschrieben.

- **Prozessor Register lesen und schreiben**
- **RAM lesen und schreiben**
- **Externer Flash Speicher lesen und schreiben**
- **Hardware Breakpoint auf den Program Counter**
- Hardware Breakpoint auf einer Speicherstelle (Watchpoint)
- Debug Trace (ETM Program Trace)
- Debug Trace Buffer

Da ein Hardware Debugger keine funktionsfähige Software auf dem Prozessor benötigt, kann er auch gut verwendet werden, um die grundlegendsten Funktionen, wie beispielsweise den Bootvorgang, vom *deep* Laufzeit System zu entwickeln.

2.3 Übersicht über die ARM Mikroarchitekturen

In diesem Kapitel werden die verschiedenen ARM-Architekturen untersucht und beurteilt. Tabelle 2.3.5 fasst alle Vor- und Nachteile zusammen.

	Vorteile	Nachteile
A	<ul style="list-style-type: none"> * Sehr leistungsstark * Support für vollwertige Betriebssysteme * Grosse Variation erhältlich (energiesparend / sehr leistungsstark) * Reichhaltiger Funktionsumfang * NEON und FPU-Unterstützung 	<ul style="list-style-type: none"> * Langsamer Context-Switch * Relativ hoher Stromverbrauch * Relativ teuer * Mit GPU erhältlich * Keine DSP-Unterstützung * Keine HW-Division
R	<ul style="list-style-type: none"> * Sehr gut geeignet für Echtzeitanwendungen * Sehr schneller Context-Switch * DSP-Unterstützung 	<ul style="list-style-type: none"> * Kleiner Funktionsumfang * Nicht so leistungstark wie Cortex A * Keine Linux-Unterstützung
M	<ul style="list-style-type: none"> * Sehr schneller Context-Switch * Sehr energiesparend * DSP-Unterstützung 	<ul style="list-style-type: none"> * Geringe Rechenleistung * Keine Linux-Unterstützung * Unterstützt nur Thumb-Instruktionen

Tabelle 2.1: Übersicht ARM Mikroarchitekturen

2.3.1 Cortex-A

Prozessoren der Cortex-A Familie sind gut geeignet für die Verwendung mit einem vollen Betriebssystem, wie Windows, Linux oder Android. Cortex-A Prozessoren bieten den umfangreichsten Support für externe Peripherien, wie USB, Ethernet und RAM. Sie sind auch die leistungsstärksten ARM-Cortex Prozessoren.

2.3.2 Cortex-R

Cortex-R Prozessoren werden entwickelt für Echtzeitanwendungen und sicherheitskritische Applikationen, wie Festplattenkontroller und medizinische Geräte. Sie sind normalerweise nicht mit einer MMU (*Memory Management Unit*) ausgerüstet. Mit einer Taktrate von über 1GHz und einem sehr schnellen Interruptverhalten eignen sich Prozessoren mit einem Cortex-R sehr gut, um auf externe Stimuli schnell zu reagieren.

2.3.3 Cortex-M

Die Prozessoren aus der Cortex-M Familie sind mit einer Taktrate um 200Mhz relativ langsam. Sie sind sehr stromsparend und durch die kurze Pipeline haben sie eine deterministische und kurze Interruptverzögerung. Die Prozessoren aus der Cortex-M Reihe unterstützen aber nur die Thumb-Instruktionen und kommen deshalb nicht in Frage.

2.3.4 ARM-Prozessoren ausserhalb der Cortex Reihe

Seit 2004 werden die meisten Kerne in eine der Cortex-Familien eingeteilt. Ältere Kerne, sogenannte "Classic cores", haben Namen wie z.b. ARM7 oder ARM1156T2F-S. Da solche Designs meist aus einer Zeit vor 2004 stammen, gilt das Design als veraltet und wird bei dieser Arbeit nicht berücksichtigt.

2.3.5 Fazit über die Wahl der ARM-Mikroarchitekturen

Die Prozessoren, die auf der Cortex-A Mikroarchitektur basieren, bieten die grösste Flexibilität. Zusätzlich ist das Angebot bei den Cortex-A-Prozessoren am grössten. Die anderen Cortex-Reihen bieten keine Vorteile, die für dieses Projekt von Nutzen sind. Aus diesen Gründen wird die Auswahl auf die Prozessoren aus der Cortex-A-Reihe begrenzt.

2.4 Anbindung des FPGAs an den Prozessor

FPGAs haben typischerweise einen sehr hohen *Pin-Count* und werden in *BGA-Packages* ausgeliefert.

Es gibt verschiedene Möglichkeiten, wie ein FPGA mit einem Prozessor verbunden werden kann. Die Vor- und Nachteile der verschiedenen Bauarten werden in diesem Kapitel abgewogen und in der Tabelle 2.4.4 zusammengefasst. Bild 2.1 gibt eine schematische Übersicht über die verschiedenen Bauarten.

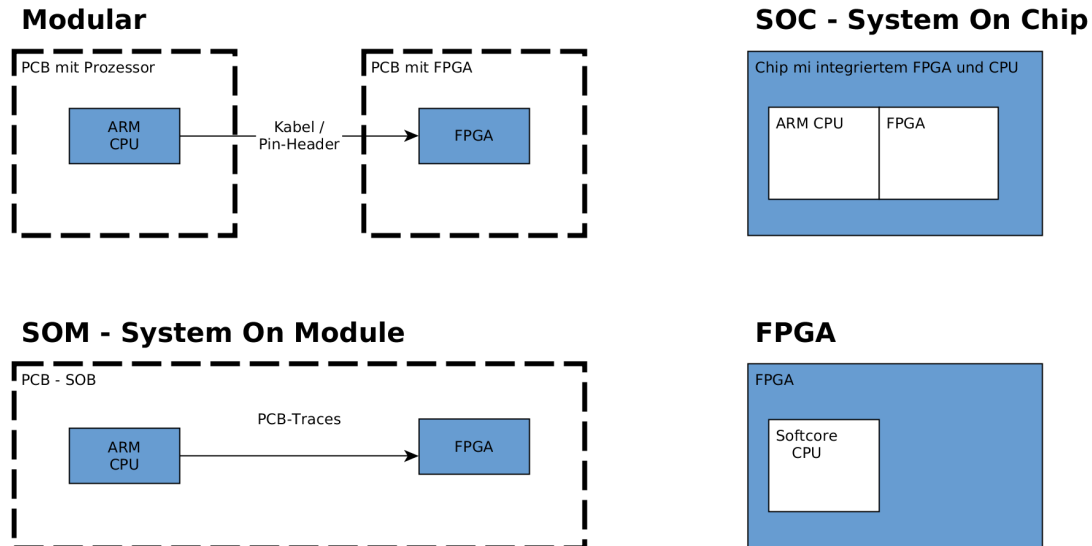


Abbildung 2.1: Mögliche Anbindungen des FPGA an die CPU

2.4.1 FPGA als Zusatzplatine zum Prozessorboard - Bauweise "Modular"

Das "FPGA Development Board CAPE for the BEAGLEBONE"² ist eine Aufsteckplatine für den Beaglebone Black. Wenn sie auf den Beaglebone Black aufgesteckt wird, erweitert sie den ARM-basierten Linux PC um einen "Spatran 6 LX9" FPGA, inklusive einiger I/O-Peripherien und SDRAM.

Vorteile:

- Relativ günstig.
- Funktioniert "Out of the Box"
- Schnelles GPMC-Interface (General-Purpose Memory Controller) zwischen Prozessor und FPGA.

Nachteile:

- Verwendet ein modifiziertes Linux-Image, das LOGI-Image.
- Der eMMC (Embedded Multi Media Card) Speicher des Beaglebone kann nicht gleichzeitig mit dem GPMC verwendet werden.
- Die Verfügbarkeit vom Cape ist nicht garantiert.
- Nur ein FPGA und Prozessor erhältlich.

Eine modulare Bauweise ist grundsätzlich sehr flexibel. Leider sind auf dem Markt nur sehr wenige verschiedene Module zu finden. So ein kleines Angebot disqualifiziert diese Bauweise.

² Direkter Link: <https://www.element14.com/community/docs/DOC-69215/1/fpga-development-board-cape-for-the-beaglebone>
 Archivierter Link: <https://web.archive.org/save/https://www.element14.com/community/docs/DOC-69215/1/fpga-development-board-cape-for-the-beaglebone>

2.4.2 FPGA auf dem gleichen Modul wie der Prozessor (System On Module) - Bauweise "SOM"

Bei einem SOM (System On Module) ist die CPU und auch der FPGA auf dem gleichen PCB-Modul verbaut. Dadurch kann der Hersteller auf dem Modul ein Bus mit kontrollierter Impedanz implementieren. Dies ermöglicht eine sehr hohe Bandbreite bei der Kommunikation zwischen der CPU und dem FPGA. Das Modul benötigt ein zusätzliches PCB, ein Basisboard, in dem es eingebettet werden kann. Oft existieren Experimentierboards mit einer grossen Zahl an unterschiedlichen I/O-Möglichkeiten, die gebrauchsfertig gekauft werden können. Für eine spezifische Anwendung muss ein solches Basisboard für das SOM selbst designed werden, weil ein Experimentierboard oft zu gross ist, oder nicht die benötigte Peripherie enthält. Da neben dem FPGA auch High-Speed-Peripherie, wie z.B. RAM auf dem Modul verbaut ist, kann beim Basisboard oft auf die aufwändige Entwicklung von High-Speed-PCB-Traces verzichtet werden.

Es hat sich gezeigt, dass es nur zwei Anbieter SOM mit FPGA produzieren. Nur die beiden Anbieter *solectrix*³ und *OposSom*⁴ scheinen solche Module zu verkaufen.

Weil die Auswahl für SOMs sehr klein ist wurde diese Bauform nicht mehr weiter verfolgt.

2.4.3 FPGA im gleichen Gehäuse wie der Prozessor (System On Chip) - Bauweise "SOC"

Seit einigen Jahren werden Produkte verkauft, die eine programmierbare Logik (FPGA) und auch eine dedizierte CPU in einem Chip-Gehäuse verbaut haben. Da der FPGA und auch die CPU im selben Gehäuse verbaut sind, ist eine sehr schnelle, integrierte Kommunikation zwischen CPU und FPGA möglich.

Die beiden grossen FPGA-Hersteller Altera und auch Xilinx bieten beide mehrere Produkte als eine SOC Lösung an. Die Produkte von Altera sind aber deutlich teurer als die Chips von Xilinx. Besonders die Evaluierungsboards von Altera sind sehr teuer.

Bei der Produktfamilie Zynq von Xilinx gibt es ein breites Angebot von SOC's und auch von Experimentierboards. Das Experimentierboard "Zybo" wird sogar schon im Unterricht der NTB für die Entwicklung von VHDL genutzt.

2.4.4 ARM als Softcore in FPGA - Bauweise "FPGA"

In FPGAs können Prozessoren als sogenannte *Softcores* implementiert werden. Dabei wird ein Teil der FPGA-Gates so konfiguriert, dass sie wie ein Mikroprozessor verwendet werden können.

Es existieren aber nur Designs für einfachere Mikroprozessoren, da komplexe Prozessoren viel zu viele Gates benötigen, um ökonomisch sinnvoll zu sein. ARM-Prozessoren der Cortex-A-Familie sind sehr komplex und nicht als FPGA-Softcores erhältlich. Von der ARM-Cortex-Familie sind nur Cortex-M0 und Cortex-M1 erhältlich. Diese Cores sind aber kostenpflichtig und nicht Open Source.

Weil keine Cortex-A-Cores erhältlich sind und alle anderen ARM-Cores kostenpflichtig sind, wird diese Bauweise nicht mehr weiter verfolgt.

³ Direkter Link: <https://www.solectrix.de/de/sxom-module>

Archivierter Link: <https://web.archive.org/save/https://www.solectrix.de/de/sxom-module>

⁴ Direkter Link: http://www.oposom.com/english/products-processor_boards-apf6_sp.html

Archivierter Link: https://web.archive.org/save/http://www.oposom.com/english/products-processor_boards-apf6_sp.html

Bauweise	Vorteile	Nachteile
Modular	* Günstig wenn nur Prozessor verwendet wird * Unterschiedliche FPGAs können verwendet werden	* Datenbus evt. nicht Memory mapped
SOB	* Sauberes, abgeschlossenes System	* FPGA ist fix
SOC	* Potenziell sehr schnelle Datenverbindung zwischen FPGA und Prozessor * Sauberes, abgeschlossenes System	* FPGA ist fix * Relativ teuer
FPGA	* Flexibel	* Sehr teuer

Tabelle 2.2: Übersicht Bauformen

2.4.5 Fazit über die Wahl der Bauweise

Es hat sich gezeigt, dass es nicht sehr viele Produkte gibt, die einen Cortex-A-Prozessor in Kombination mit einem FPGA bieten. Einige Produkte zielen mehr auf den Hobby-Bereich, wie zum Beispiel das *"FPGA Development Board CAPE for the BEAGLEBONE"*. Für professionellere Lösungen scheinen selbstentwickelte PCBs der Standard zu sein. Alle anderen Ansätze sind oft nur Nischenprodukte für spezielle Anwendungen oder mit geringer Verfügbarkeit.

Seit einigen Jahren ist aber eine signifikante Auswahl von SOC's auf dem Markt. Diese werden aber nur von den beiden Herstellern Altera und Xilinx angeboten. Beide Hersteller bieten aber ein sehr umfangreiches Angebot.

2.5 Fazit - Auswahl der Hardware

Da die Wahl bereits auf einen Cortex-A in einem SOC eingeschränkt wurde, ist das verbleibende Angebot sehr begrenzt. Die Entscheidung zwischen Zynq von Xilinx und den SOC's von Altera fällt auf Zynq, da die Altera Experimentierboards mehrere tausend Franken kosten.

Das Zybo-Experimentierboard ist eine sehr naheliegende Wahl, da es bereits für den Unterricht in der NTB genutzt wird. Der Preis des Boards ist auch tief genug, dass eine ganze Klasse für den Unterricht damit ausgerüstet werden kann. Eine grosszügige Auswahl an I/Os bieten eine sehr hohe Flexibilität zum experimentieren und auch für den Unterricht.

Das Zybo ist mit einem Zynq-7000 bestückt. Der Zynq-7000 ist ein Modell mit einem Dual-Core-Cortex-A9-Prozessor mit 667 MHz. Es existieren aber auch noch günstigere Zynqs mit weniger Leistung und sehr viel teurere Varianten mit einem leistungsstärkeren Prozessor und grösseren FPGA. Zusätzlich sind die Zynqs als standalone Chip oder als Modul inklusive RAM erhältlich.

All diese Eigenschaften machen das Zybo (Abbildung 2.2) mit dem Zynq-7000 zum klaren Favorit.

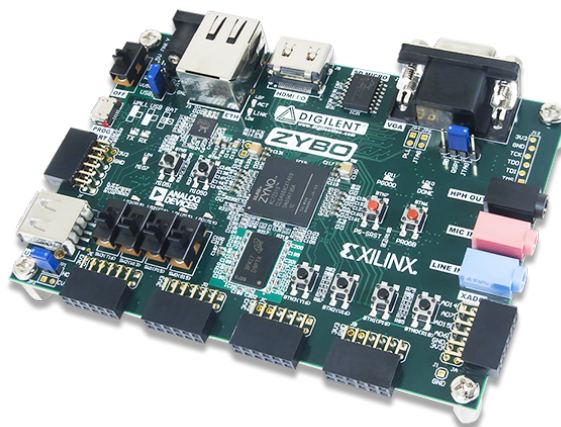


Abbildung 2.2: Das Zybo mit dem Zynq-7000 SOC

3 Überblick über das ganze System

Dieses Kapitel bietet eine grobe Übersicht über das ganze System, um die Zusammenhänge zwischen einzelnen Komponenten aufzuzeigen. Auf einzelne Komponenten und Toolchains wird in den folgenden Kapiteln genauer eingegangen.

3.1 Schematische Übersicht

In Abbildung 3.1 ist das ganze System abgebildet. Das *Zybo* beinhaltet neben dem FT2232-Chip auch noch diverse I/O-Peripherien, die in einer *deep*-Applikation genutzt werden können. Der FT2232 auf dem *Zybo* übernimmt zwei verschiedene Funktionen. Einerseits wird er als USB zu UART Brücke (schwarzer Pfeil) verwendet, damit der Windows PC einfach eine serielle Verbindung mit dem Prozessor aufbauen kann, andererseits fungiert er als Brücke zum blauen JTAG-Bus. Das bedeutet, er erhält Befehle von der OpenOCD-Software über USB und übersetzt diese elektrisch und auch logisch für das JTAG Interface. OpenOCD ist eine Software-Zwischenschicht die für den Debugger benötigt wird.

Auf dem *Windows PC* wird die *deep*-Applikation in Eclipse geschrieben, kompiliert und debuggt. Plugins erweitern Eclipse um die notwendigen Funktionen, die für die Entwicklung von *deep*-Applikationen notwendig sind. In dieser Übersicht sind beide Debug Toolchains, die "klassische" Abatron-Toolchain und die neue OpenOCD-Toolchain, abgebildet.

Bei der *Abatron-Toolchain* wird das *Abatron BDI3000* mit dem *abatronInterface*-Plugin über die rote TCP/IP-Verbindung angesprochen. Das BDI kommuniziert dann über die blaue JTAG-Verbindung direkt mit dem Zynq-Chip.

Die grünen Pfeile zeigen den Kommunikationsweg für die neuen OpenOCD-Toolchains. OpenOCD bildet zusammen mit der richtigen Hardware, hier ist es der FT2232-Chip, einen kompletten Debugger und ist somit eine Alternative zum BDI3000. Die OpenOCD-Software stellt einen *gdb*-Server und auch ein CLI (*Command Line Interface*) zur Verfügung. Das neue Eclipse-Plugin "*OpenOCDInterface*" verwendet das CLI über den TCP/IP-Port 4444 (grüner Pfeil) und bildet so die *CLI-OpenOCD-Toolchain*. OpenOCD verwendet dann den *WinUSB*-Treiber um mit dem FT2232-Chip über USB zu kommunizieren. Der FT2232-Chip verwendet denselben blauen JTAG-Bus wie das BDI3000 zur Kommunikation mit dem Zynq.

Die *gdb-OpenOCD-Toolchain* kann mit einem allein lauffähigen *gdb* verwendet werden (orange, gestrichelter Pfeil), wie in Kapitel 7 beschrieben. Eine weitere Möglichkeit wäre ein *gdb*-Plugin für Eclipse, damit der *gdb* direkt aus Eclipse heraus verwendet werden kann. Beide Varianten kommunizieren mit dem *gdb*-Server von OpenOCD über den TCP/IP-Port 3333 (oranger Pfeil).

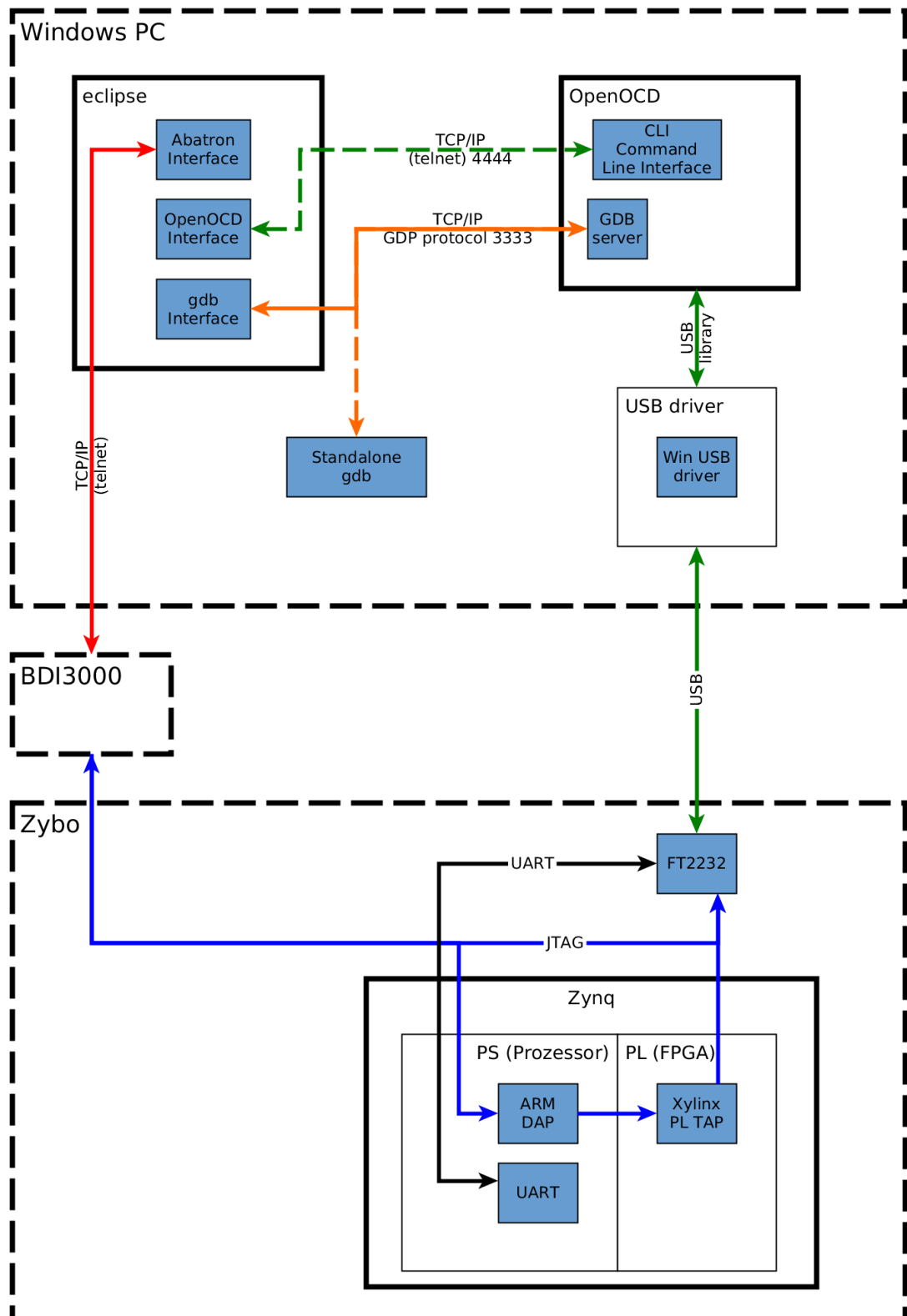


Abbildung 3.1: Systemübersicht Debugger Toolchain

3.2 Debugger Toolchains

Im Folgenden werden die drei verschiedenen Toolchains genauer erklärt.

3.2.1 Abatron-Toolchain

Die *Abatron-Toolchain* (Abbildung 3.2) benötigt weder die OpenOCD-Software noch den FT2232-Chip, dafür aber den teuren BDI3000-Debugger. Diese "klassische" Toolchain nutzt das bestehende *deep*-Plugin *abatronInterface* und wird für die Entwicklung von *deep* für den PowerPC verwendet. In dieser Arbeit wird die *Abatron-Toolchain* nicht verwendet.

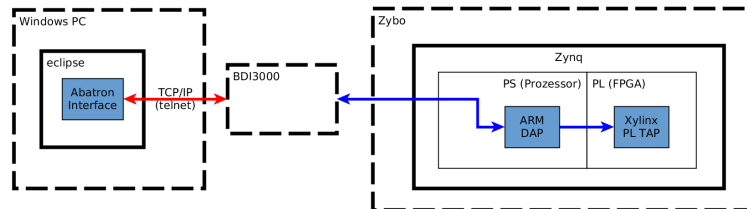


Abbildung 3.2: Abatron-Toolchain

3.2.2 CLI-OpenOCD-Toolchain

Wie in Abbildung 3.3 zu sehen ist, wird das teure BDI für diese Toolchain nicht benötigt. Da das CLI (Command Line Interface) von OpenOCD aber sehr ähnlich ist wie das CLI des BDI, ist eine Portierung der bestehenden *Abatron-Toolchain* in die neue *CLI-OpenOCD-Toolchain* relativ einfach. Die *CLI-OpenOCD-Toolchain* lehnt sich deshalb sehr stark an die bestehende *Abatron-Toolchain* an.

Mit dieser Toolchain ist *Sourcecode-Debugging* aber nicht möglich. Das bedeutet, es ist nicht möglich im Sourcecode Breakpoints zu setzen oder durch einzelne Zeilen im Sourcecode zu steppen. Bestehende Möglichkeiten aus der alten *Abatron-Toolchain*, wie "TargetOperations", bleiben aber erhalten.

Im Kapitel 5.4 wird die Implementation dieser Toolchain genauer beschrieben.

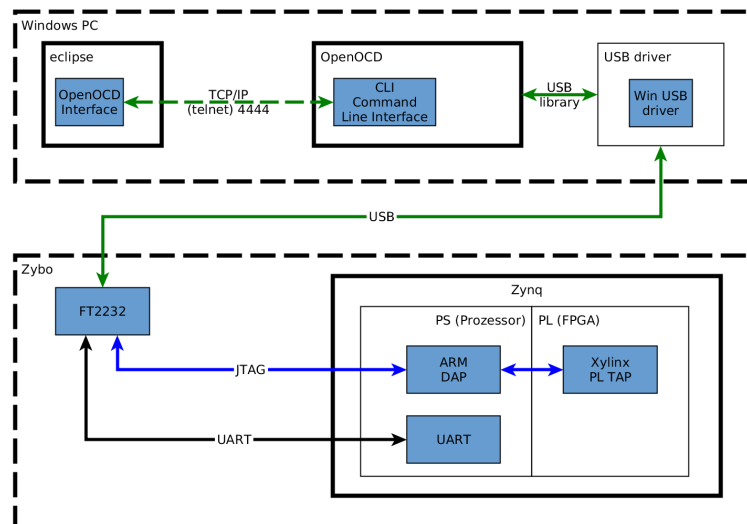


Abbildung 3.3: CLI-OpenOCD-Toolchain

3.2.3 *gdb*-OpenOCD-Toolchain

In der *gdb-OpenOCD-Toolchain* wird, wie bei der obigen Toolchain, ebenfalls die OpenOCD-Software und der FT2232-Chip verwendet. Es wird aber nicht mehr ein Interface bestehend auf der "klassischen" Abatron Toolchain verwendet, sondern es wird direkt das CLI des *gdb*-Debugger genutzt. In der schematischen Übersicht der Toolchain in Abbildung 3.4 wird deutlich, dass sie fast die gleichen Komponenten nutzt wie die *CLI-OpenOCD-Toolchain*. Mit dem *gdb* können auch erweiterte Debugging-Features wie *Sourcecode-Lookup* und *Breakpoints* verwendet werden.

In dieser Arbeit wird nur die vereinfachte Toolchain mit dem Standalone-*gdb*-Debugger implementiert. Mit der vereinfachten Toolchain kann das CLI des *gdb* in Kombination mit der *OpenOCD-Toolchain* für zum Debuggen genutzt werden.

Die komplette *gdb-OpenOCD-Toolchain* kann auf dieser Toolchain aufbauen. Bei der kompletten *gdb-OpenOCD-Toolchain* soll der *gdb* im Eclipse integriert werden. Dadurch kann in Eclipse die Applikation entwickelt und auch debuggt werden.

Im Kapitel 7 wird diese Toolchain detailliert beschrieben.

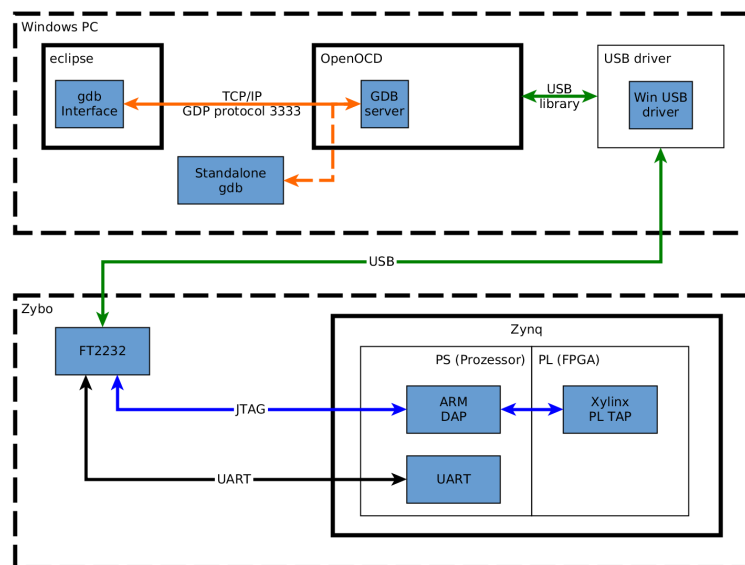


Abbildung 3.4: *gdb*-OpenOCD-Toolchain

4 Das SOC Zynq-7000

Der Zynq-7000 ist ein SoC (System on Chip), das einen 667 MHz Dual-Core ARM Cortex-A9 Prozessor und eine programmierbare Logik enthält, die einem Artix-7 FPGA entspricht. Der Prozessor und dessen Peripherie befindet sich im *Processing System* oder kurz PS. Der FPGA-Teil des Zynq wird oft PL oder *Programmable Logic* genannt. Über den internen AMBA-Bus kann der Prozessor und auch die PL auf die Peripherie, wie z.B. SPI, GPIO, Ethernet oder auch DDR3, zugreifen. Das Block Diagramm in der Abbildung 4.1 gibt einen guten Überblick über das ganze SoC. Das restliche Kapitel beschreibt relevante Komponenten und Eigenarten des Zynq.

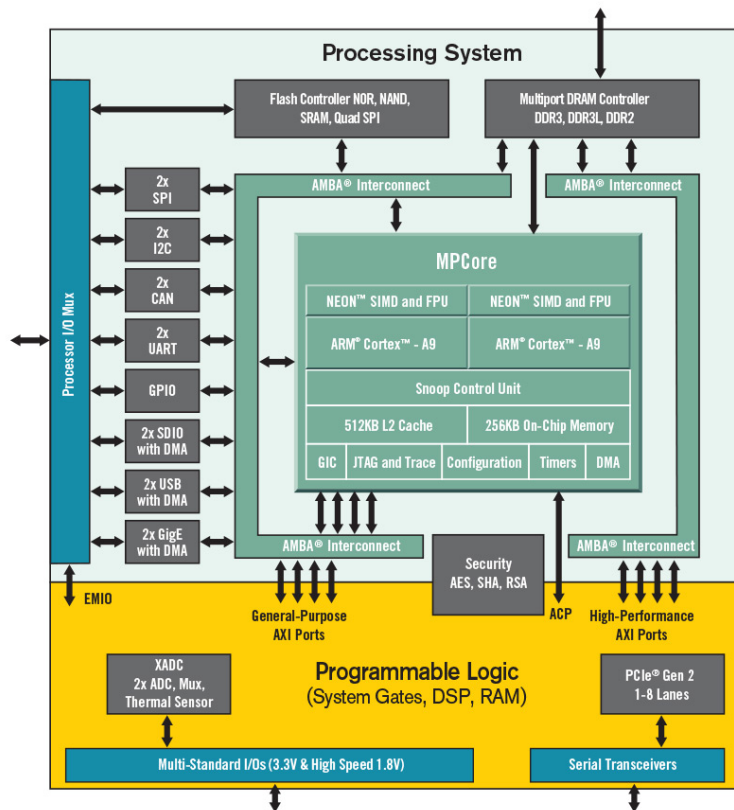


Abbildung 4.1: Block Diagramm Zynq7000¹

4.1 MIO und EMIO

MIOs sind *Multiplexed Input Output Pins*, welche direkt vom Prozessor angesprochen werden können, ohne dass die PL programmiert werden muss. Die EMIOs sind *Extended Multiplexed Input Output Pins*, welche nur über die PL angesprochen werden können. Aus diesem Grund können die EMIOs nur verwendet werden, wenn die PL entsprechend programmiert wurde. Diese Arbeit beschränkt sich nur auf die MIOs und das PS. Im TRM (*Technical Reference Manual*) des Zynq[1] im Kapitel "2.5.4 MIO-at-a-Glance Table" ist eine sehr gute Übersicht über alle möglichen Funktionen der MIOs gegeben.

¹ Direkter Link: <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>
 Archivierter Link: <https://web.archive.org/save/https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>

4.2 Standard Zybo Workflow

Im *Getting Started with Zynq*² Tutorial von Digilent ist beschrieben, wie ein einfaches Design für die PL und ein einfaches Programm für das PS erstellt werden kann. Das Tutorial deckt den ganzen Workflow ab. Dabei werden, z.B. für LED1, LED2 und LED3 auch die EMIOs verwendet. In Schritt 1 bis 7 wird mit Vivado das Design für die PL erstellt und exportiert.

Hinweis1: Die Zybo-Toolchain benötigt den standard USB-Treiber. Im Kapitel 5.1.2 ist beschrieben, wie der standard USB-Treiber wieder installiert werden kann.

Hinweis2: Vivado und die Xilinx SDK müssen für dieses Tutorial installiert sein.

Ab Schritt 8 wird beschrieben, wie im XSDK (*Xilinx Standard Development Kit*) ein einfaches "Hello World" Programm in C für den Prozessor geschrieben werden kann.

Das XSDK verwendet im Hintergrund das XSCT³ (*Xilinx Software Command-Line Tool*). Das XSDK kann interaktiv, oder mit Scripts verwendet werden. Wie Jim-TCL basiert auch die verwendete Scriptsprache auf der Sprache TCL. Wird das "Hello World" Programm im XSDK gestartet, erscheint im *SDK Log*-Fenster ein detailliertes Log des ausgeführten Scripts. In diesem Log kann nachvollzogen werden, was das Script beim Download und Start des Programms alles ausgeführt hat.

Im Anhang A.1 ist eine Kopie eines solchen Logs zu finden.

Das Script *ps7_init.tcl* definiert unter anderem die fünf Initialisierungs-Methoden, welche im Kapitel 4.2.2 genauer beschrieben werden:

- *ps7_mio_init_data_3_0*
- *ps7_pll_init_data_3_0*
- *ps7_clock_init_data_3_0*
- *ps7_dds_init_data_3_0*
- *ps7_peripherals_init_data_3_0*

Die Initialisierungs-Methoden werden in der Methode *ps7_init* aufgerufen. *ps7_init* wiederum wird in Zeile 8 des *...elf_on_local.tcl* Scripts aufgerufen, welches beim Start des "Hello World" Programms im XSDK ausgeführt wird. *...elf_on_local.tcl* "sourced" zu Beginn *ps7_init.tcl*, damit dessen Methoden zur Verfügung stehen. In Zeile 9 vom *...elf_on_local.tcl* wird dann die Methode *ps7_post_config* von *ps7_init.tcl* aufgerufen, welche im Anschluss *ps7_post_config_3_0* aufruft.

Auszug aus "system_debugger_using_debug_01_gettingstarted.elf_on_local.tcl" (Komplettes Programm im Anhang B.6):

```

1 connect -url tcp:127.0.0.1:3121
2 source D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
   design_1_wrapper_hw_platform_0/ps7_init.tcl
3 ...
4 stop
5 ps7_init
6 ps7_post_config
7 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
   Digilent Zybo 21027957373A"} -index 0
8 ...

```

Auszug aus "ps7_init.tcl" (Komplettes Programm im Anhang A.4):

² Direkter Link: <https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-getting-started-with-zynq/start?redirect=1>
 Archivierter Link: <https://web.archive.org/save/https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-getting-started-with-zynq/start?redirect=1>

³ Direkter Link: https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/xsct/intro/xsct_introduction.html
 Archivierter Link: <https://web.archive.org/save/http://www.xilinx.com/products/design-tools/embedded-software/sdk.html>

```

1  ...
2  proc ps7_post_config {} {
3  ...
4      ps7_post_config_3_0
5  }
6  ...
7  proc ps7_post_config_3_0 {} {
8  ...

```

Alle Konfigurationsregister sind im Anhang B vom *Zynq TRM* (bib: *ZynqTechnicalReferenceManual*) beschrieben. Bevor die Register aber verändert werden können, müssen sie "unlocked" werden, indem der Wert *0x0000DF0D* in die Adresse *0xF8000008* geschrieben wird.

4.2.1 Grundlegende Methoden

Alle Methoden des *ps7_init.tcl*-Scripts sind auf den folgenden vier Grundbefehlen aufgebaut:

mwr -force <address> <value>:

Schreibt den Wert <value> in die Adresse <address>.

mask_write <address> <mask> <value>:

Schreibt die Bits der Maske <mask> von <value> in die Adresse <address>.

mask_poll <address> <mask>:

Wartet, bis die maskierten Bits <mask> des Speicherinhalts von der Speicheradresse <address> gleich 0 sind.

mask_delay <address> <value>:

Wartet <value> Millisekunden.

4.2.2 Initialisierungsmethoden

Im Folgenden werden alle Methoden beschrieben, welche zur Initialisierung des Zynq auf dem Zybo verwendet werden.

ps7_mio_init_data_3_0:

Diese Methode initialisiert die MIOs. Der Multiplexer für die IO Pins wird konfiguriert. Dadurch wird definiert, welcher Pin von welcher Peripherie, wie UART und auch RAM, verwendet wird. Zusätzlich werden auch, falls vorhanden, folgende elektrischen Charakteristiken definiert:

- **Pullup:** Pullup Widerstand aktivieren / deaktivieren.
- **IO_Type:** Buffer Type: LVCMOS 1.8V, LVCMOS 2.5V, LVCMOS 3.3V, oder HSTL.
- **Speed:** Slow / fast CMOS edge.
- **Tristate:** Enable / disable Tristate.

ps7_pll_init_data_3_0

Initialisiert die drei PLLs (*Phase Locked Loop*) ARM, DDR und IO. Bei jeder PLL-Initialisierung wird darauf gewartet, bis der PLL betriebsbereit (locked) ist. Die Dauer dieser Wartezeit ist unbekannt.

ps7_clock_init_data_3_0

Konfiguriert diverse Clocks, die im Prozessor gebraucht werden.

ps7_ddr_init_data_3_0

Konfiguriert den DDR Bus. Für die Konfiguration werden insgesamt 79 verschiedene Register geschrieben und die DCI (*Digital Controlled Impedance*) kalibriert. Nachdem diese Methode ausgeführt wurde, kann der DDR genutzt werden. Vorher ist nur der OCM (On Chip Memory) nutzbar. Mehr dazu im Kapitel 4.3.

ps7_peripherals_init_data_3_0

Konfiguriert folgende Peripherien:

- UART1

- QSPI (für Flash Speicher auf Zybo)
- POR timer
- High-Low-Wait(1msec)-High Sequenz für MIO46 (USB-OTG Ping)

Die oben genannten Initialisierungsfunktionen werden vom Xilinx Debugger jedes Mal ausgeführt, wenn die Applikation im XSDK mit *"Launch on Hardware (System Debugger)"* gestartet wird. Es ist aber auch möglich, die Initialisierung direkt mit der C-Applikation und nicht mit dem Debugger durchzuführen. Wird die Initialisierung in der Applikation durchgeführt, und die Applikation auf dem Flash Speicher des Zynq gespeichert, dann initialisiert sich der Zynq bei jedem Start selber. Im Beispielprogramm *"helloworld.c"* ist die Methode *"init_platform()"* enthalten, welche in *"platform.c"* deklariert ist. Standardmässig ist die darin enthaltene Methode *"ps7_init()"* aber auskommentiert. *"platform.c"* befindet sich im *"design_wrapper_hw_platform"*, welcher in Vivado erzeugt wurde. Wird *"ps7_init()"* mit *ps7_init.tcl* verglichen, dann wird schnell deutlich, dass das Script und auch die C-Methode genau die gleichen Register schreiben und lesen.

"psu_init()" ist für ein *"Zynq UltraScale+™ MPSoC"* Chip, welcher auf dem Zybo nicht verwendet wird.

Auszug aus *"helloworld.c"* (Komplettes Programm im Anhang A.2):

```

1  ...
2  #include "platform.h"
3  ...
4  int main ()
5  {
6  ...
7  init_platform();
8
9  while(1){
10 ...

```

Auszug aus *"platform.c"* (Komplettes Programm im Anhang A.3):

```

1  ...
2  /*#include "ps7_init.h"*/
3  /*#include "psu_init.h"*/
4  ...
5  void
6  init_platform()
7  {
8      /*
9       * If you want to run this example outside of SDK,
10      * uncomment one of the following two lines and also #include "ps7_init
11      * .h"
12      * or #include "ps7_init.h" at the top, depending on the target.
13      * Make sure that the ps7/psu_init.c and ps7/psu_init.h files are
14      * included
15      * along with this example source files for compilation.
16      */
17      /* ps7_init();*/
18      /* psu_init();*/
19      enable_caches();
20      init_uart();
21  }
22  ...

```

4.2.3 ps7_init.tcl Script für OpenOCD anpassen

Da das *ps7_init.tcl* Script ebenfalls auf der TCL-Sprache basiert, kann es gut für OpenOCD angepasst werden. Einige Methoden werden aber nur vom XSCT unterstützt und nicht von OpenOCD. Mit folgenden Änderungen ist das Script mit OpenOCD kompatibel:

1. Untenstehende Methoden wurden dem Script hinzugefügt.

Auszug aus *"ps7_init_modified.tcl"* (Komplettes Script im Anhang A.4):

```

1  proc unlock_SLCR {} {
2      mww 0xF8000008 0x0000DF0D
3  }
4
5  proc map_OCM_low {} {
6      unlock_SLCR
7      mww 0xF8000910 0x00000010
8  }
9
10 proc memread32 {ADDR} {
11     set foo(0) 0
12     if ![ catch { mem2array foo 32 $ADDR 1 } msg ] {
13         return $foo(0)
14     } else {
15         error "memread32: $msg"
16     }
17 }
18
19 proc mask_write { addr mask val } {
20     set curval [memread32 $addr]
21     set maskinv [expr {0xffffffff ~ $mask}]
22     set maskedcur [expr {$maskinv & $curval}]
23     set maskedval [expr {$mask & $val}]
24     set newval [expr $maskedcur | $maskedval]
25     mww $addr $newval
26 }
27
28 proc initPS {} {
29     ps7_init
30     ps7_post_config
31 }

```

2. Jeder "mwr -force <address> <value>" Befehl wurde mit "mww <address> <value>" ersetzt.
3. Folgende Methoden wurden mit den untenstehenden Implementationen ersetzt.

Auszug aus "ps7_init_modified.tcl" (Komplettes Script im Anhang A.4):

```

1  proc mask_poll { addr mask } {
2      set count 1
3      % set curval [memread32 $addr]
4      (*@ \textcolor{blue}{ set curval [memread32 $addr] } @*)
5      set maskedval [expr {$curval & $mask}] # & = bitwise AND
6      while { $maskedval == 0 } {
7          set curval [memread32 $addr]
8          set maskedval [expr {$curval & $mask}]
9          set count [ expr { $count + 1 } ]
10         if { $count == 100000000 } {
11             puts "Timeout Reached. Mask poll failed at ADDRESS: $addr
12                 MASK: $mask"
13             break
14         }
15     }
16 }
17
18 proc mask_delay { addr val } {
19     set delay [ get_number_of_cycles_for_delay $val ]
20     perf_reset_and_start_timer
21     set curval [memread32 $addr]
22     set maskedval [expr {$curval < $delay}]
23     while { $maskedval == 1 } {
24         set curval [memread32 $addr]
25         set maskedval [expr {$curval < $delay}]
26     }
27     perf_reset_clock
28 }
29
30 proc ps7_post_config {} {
31     ps7_post_config_3_0
32 }
33
34 proc ps7_init {} {
35     halt
36 }

```

```

35     ps7_mio_init_data_3_0
36     ps7_pll_init_data_3_0
37     ps7_clock_init_data_3_0
38     ps7_ddr_init_data_3_0
39     ps7_peripherals_init_data_3_0
40     puts "PCW Silicon Version : 3.0"
41 }
42
43 proc get_number_of_cycles_for_delay { delay } {
44     # GTC is always clocked at 1/2 of the CPU frequency (CPU_3x2x)
45     set APU_FREQ 650000000
46     return [ expr ($delay * $APU_FREQ / (2 * 1000)) ]
47 }

```

4.3 Memory Mapping

Im Kapitel 4.1 des *Zynq TRM*[1] ist der Aufbau des Speichers beschrieben. Die Abbildung 4.2 zeigt einen guten Überblick über die ganzen 4 GB des Adressraumes. Bei der Map fällt auf, dass nur ca. 1 GB für den DDR RAM verwendet werden kann.

Der OCM (*On Chip Memory*) ist ein kleiner Speicher im Zynq der ohne Initialisierung verwendet werden kann. Das ist ideal für einen Bootloader. Für den OCM stehen ganz am Anfang des Speicherbereichs (*0x0000_0000*) und ganz am Ende (*0xFFFC_0000*) 256 kB zur Verfügung. Der OCM besteht aus 4 x 64 kB grossen Teilbereichen, die dem Register *0xF8000910* wahlweise im oberen oder im unteren Bereich zugewiesen werden können. Beim Bootvorgang werden die ersten drei Teile in den unteren Bereich (*0x0000_0000 - 0x0002_FFFF*) und der vierte Teil in den obersten Bereich (*0xFFFF_0000 - 0xFFFF_FFFF*) gemapt. Das geschieht noch bevor die erste Instruktion aus dem User-Code ausgeführt wird, also auch vor dem selbstgeschriebenen Bootloader. Der oben beschriebene Bootvorgang kann nicht geändert werden. Mit Pull-Up-Widerständen kann aber beeinflusst werden, ob der ARM im *Secure-Mode* oder im *Non-Secure-Mode* booten soll und wo der Bootloader gesucht werden soll. Mehr dazu im *Zynq TRM*[1] im Kapitel "Kapitel 4.4: Boot and Configuration".

Der Speicherbereich für den RAM ist erst nutzbar, wenn der RAM initialisiert wurde. Die Initialisierungsmethode wird im Kapitel 4.2.2.

4.4 Floating Point Unit

FPU (*Floating Point Unit*) können je nach Implementation unterschiedliche Funktionen unterstützen. In den Registern MVFR0 und MVFR (*Media and VFP Feature Register*) lässt sich auslesen welche Funktionen in der Hardware implementiert wurden und genutzt werden können. Diese Register können aber nicht mit einer einfachen *Memory read* gelesen werden. Um diese Register oder die anderen speziellen FPU-Register, wie FPSID, FPSCR und PFEXC, lesen zu können, muss die ARM-Instruktion "VMRS" verwendet werden.

4.4.1 FPU initialisieren

Damit auf die FPU zugegriffen werden kann, muss der Co-Prozessor 15 erst so konfiguriert werden, dass das System im *secure* und im *non-secure mode* Zugriff auf die FPU hat. Der CP15 ist ein "System control coprocessor", der neben der FPU auch den Cache und die MPU (Memory Protection Unit) konfiguriert. Um in ein Register des Co-Prozessors schreiben zu können, muss eine spezielle Instruktion "MCR" verwendet werden, die ein ARM-Register in ein Co-Prozessor-Register speichert. Da OpenOCD diese Instruktion unterstützt, können die *Access Control Register* direkt mit dem Debugger gesetzt werden.

Das NSACR (*Non-secure Access Control Register*) kontrolliert, ob die FPU auch im *non-secure mode* genutzt werden kann. Das CPACR (*Coprocessor Access Control Register*) kontrolliert den Zugang zu allen Coprozessoren (CP10 und CP11 sind die FPU).

Zusätzlich muss auch noch das FPEXC_EN-Bit im FPEXC-Register (*Floating-Point Status and Control Register*) gesetzt werden. Das FPEXC-Register kann aber nicht mit dem Debugger direkt gesetzt werden,

Address Range	CPUs and ACP	AXI_HP	Other Bus Masters ⁽¹⁾	Notes
0000_0000 to 0003_FFFF ⁽²⁾	OCM	OCM	OCM	Address not filtered by SCU and OCM is mapped low
	DDR	OCM	OCM	Address filtered by SCU and OCM is mapped low
	DDR			Address filtered by SCU and OCM is not mapped low
				Address not filtered by SCU and OCM is not mapped low
0004_0000 to 0007_FFFF	DDR			Address filtered by SCU
				Address not filtered by SCU
0008_0000 to 000F_FFFF	DDR	DDR	DDR	Address filtered by SCU
		DDR	DDR	Address not filtered by SCU ⁽³⁾
0010_0000 to 3FFF_FFFF	DDR	DDR	DDR	Accessible to all interconnect masters
4000_0000 to 7FFF_FFFF	PL		PL	General Purpose Port #0 to the PL, M_AXI_GP0
8000_0000 to BFFF_FFFF	PL		PL	General Purpose Port #1 to the PL, M_AXI_GP1
E000_0000 to E02F_FFFF	IOP		IOP	I/O Peripheral registers, see Table 4-6
E100_0000 to E5FF_FFFF	SMC		SMC	SMC Memories, see Table 4-5
F800_0000 to F800_0BFF	SLCR		SLCR	SLCR registers, see Table 4-3
F800_1000 to F880_FFFF	PS		PS	PS System registers, see Table 4-7
F890_0000 to F8F0_2FFF	CPU			CPU Private registers, see Table 4-4
FC00_0000 to FDFE_FFFF ⁽⁴⁾	Quad-SPI		Quad-SPI	Quad-SPI linear address for linear mode
FFFC_0000 to FFFF_FFFF ⁽²⁾	OCM	OCM	OCM	OCM is mapped high
				OCM is not mapped high

Abbildung 4.2: Address Map des Zynq

da eine spezielle ARM-Instruktion dafür verwendet werden muss. Im Kapitel "2.4.2 Accessing the FPU registers" des *FPU-TRM*[3] ist detailliert beschrieben, welche Register genau gesetzt werden müssen.

Mit dem folgenden ARM-Code kann die FPU z.B. beim Booten des Kernels initialisiert werden:

```

1  ; Set bits [11:10] of the NSACR for access to CP10 and CP11 from both
    Secure and Non-secure states:
2  MRC p15, 0, r0, c1, c1, 2
3  ORR r0, r0, #2_11<<10 ; enable fpu/neon
4  MCR p15, 0, r0, c1, c1, 2
5  ; Set the CPACR for access to CP10 and CP11:
6  LDR r0, =(0xF << 20)
7  MCR p15, 0, r0, c1, c0, 2
8  ; Set the FPEXC EN bit to enable the FPU:
9  MOV r3, #0x40000000
10 VMSR FPEXC, r3

```

4.4.2 MVFR lesen mit OpenOCD

OpenOCD kann zwar direkt die Register der generischen Co-Prozessoren lesen und schreiben, nicht aber die Register der FPU. Der folgende Ablauf ermöglicht es aber trotzdem, diese Register auszulesen:

1. OpenOCD starten und für das CLI eine Telnetverbindung zu Port 4444 aufbauen
2. `reset init` // Reset und Initialisierung des ganzen Systems.
3. `arm mcr 15 0 1 1 2 0x0c00` // Non-secure access für FPU (NSACR Register).
4. `arm mcr 15 0 1 0 2 0x00f00000` // Genereller Zugang für FPU erlauben (CPACR Register).
5. `mw 0x0 0xEE70A10` // Speichert die Instruktion "VMRS R0, MVFR0" in den OCM.
6. `mw 0x4 0xEE61A10` // Speichert die Instruktion "VMRS R1, MVFR1" in den OCM.
7. `bp 0x8 1 hw` // Breakpoint nach der Instruktion (32 Bit Instruktion = 4 Byte)
8. `resume 0x0` // Führt die Instruktion bei der Adresse 0 aus
9. `reg 0` // Liest das Register 0 aus, welches eine Kopie des MVFR0 enthält.
10. `reg 1` // Liest das Register 1 aus, welches eine Kopie des MVFR1 enthält.

Die Inhalte der Register sind:

- MVFR0: 0x1011_0222
- MVFR1: 0x0111_1111

4.4.3 Unterstützte Features der FPU

Die Register MVFR0 und MVFR1 enthalten Informationen über die unterstützten Features der FPU. Auf der Seite B5-36 des ARMv7-A ARM[2] (*Architecture Reference Manual*) ist beschrieben, wie die unterstützten Features aus den Registern gelesen werden können.

Der Zynq des Zybo unterstützt:

- All rounding modes
- VFP square root operations
- VFP divide operations
- Full VFP double-precision v3 (VFPv3)
- VFPv3 single-precision
- Advanced SIMD register bank: 32 x 64-bit registers
- All VFP instructions (LDC, STC, MCR, and MRC)
- Half-precision floating-point conversion operations (VFP and advanced SIMD)

- Single-precision floating-point operations (advanced SIMD)
- Integer operations (advanced SIMD)
- Load/store operations (advanced SIMD)
- Propagation of NaN values

Nicht unterstützt wird:

- VFP short vectors
- VFP exception trapping

5 Die OpenOCD Software

OpenOCD¹ bildet den Software-Teil eines Debuggers. Zusammen mit einem Hardware-Adapter bildet OpenOCD einen vollständigen Debugger und kann als Ersatz für einen teuren Debugger, wie beispielsweise dem BDI3000 von Abatron, verwendet werden.

Der Adapter bildet dabei das elektrische Interface zum Prozessor und muss auch auf den Prozessor abgestimmt sein. Relevant sind dabei unter anderem der Transport Layer (JTAG/SWD), das elektrische Potential und natürlich auch der physikalische Stecker. In vielen Fällen basieren solche Adapter, wenn sie zusammen mit OpenOCD verwendet werden, auf dem FT2232-Chip von FTDI. Ein solcher generischer Adapter ist in Abbildung 5.1 zu sehen.

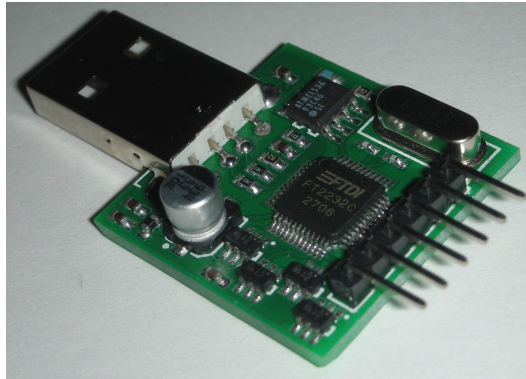


Abbildung 5.1: Generischer JTAG Adapter mit einem FTDI FT2232²

Bei Experimentierboards ist der FT2232 oft auch direkt auf das Board aufgelötet. So kann eine einfache USB-Verbindung genutzt werden, um den Prozessor zu debuggen. Beim Zybo wurde ebenfalls dieser Ansatz verfolgt. Aus diesem Grund reicht ein einfaches USB-Kabel um den Prozessor des Zybos auf einer Hardwareebene debuggen zu können.

5.1 Softwareinstallation der OpenOCD-Toolchain

Um OpenOCD nutzen zu können, muss auch der richtige USB-Treiber installiert sein. In den folgenden Kapiteln wird erklärt, wie der Treiber und auch OpenOCD-Software installiert werden kann.

5.1.1 Softwareinstallation - OpenOCD

OpenOCD kann direkt aus dem Sourcecode kompiliert werden³ oder es können vorkompilierte Binaries verwendet werden. Für diese Arbeit wurde das vorkompilierte Windows Binaries⁴ für ARM-Cores mit der Version 0.10.0 verwendet.

Das eigentliche Binary befindet sich im Ordner:
`/openocd-0.10.0/bin-x64/`

Das Open OCD User Manual[5] befindet sich im Ordner:
`/openocd-0.10.0/`

¹<http://openocd.org/about/>

²<https://www.ebay.com/itm/FPU1-FTDI-FT2232-USB-JTAG-XILINX-FPGA-CPLD-programmer-cable-/181635528314> Seite 5

³<http://sourceforge.net/p/openocd/code/>

⁴<http://www.freddiechopin.info/en/download/category/4-openocd?download=154%3Aopenocd-0.10.0>

5.1.2 Softwareinstallation - USB-Driver WinUSB

Damit OpenOCD mit dem FT2232-Chip kommunizieren kann, werden die richtigen USB-Treiber benötigt. Die Installation der Treiber ist am einfachsten mit dem *USB Driver Tool*⁵.

Das Zybo muss per USB mit dem PC verbunden sein, damit der Treiber installiert werden kann. Wenn der Jumper 'J15' auf USB gesetzt ist, wird keine zusätzliche Stromversorgung für das Zybo benötigt.

Wird das *USB Driver Tool* geöffnet, dann werden alle USB Devices aufgelistet. Das Device mit der *Vendor ID=0403*, der *Device ID=6010* und dem *Interface 0* ist das JTAG Interface des FT2232. Mit einem Rechtsklick kann *Install WinUSB* ausgewählt und der Treiber installiert werden. Abbildung 5.2 zeigt die Liste mit allen USB Devices und das Kontextmenü für die Installation des richtigen Treibers. Um den Standardtreiber wieder zu installieren, kann einfach "Restore default driver" ausgewählt werden. Nachdem das Zybo einmal aus- und wieder eingeschaltet worden ist, ist der Treiber einsatzbereit.

Das Device mit der *Vendor ID=0403*, *Device ID=6010* und *Interface 1* ist die UART-Verbindung zum Prozessor. Dieser Treiber darf **nicht** ersetzt werden.

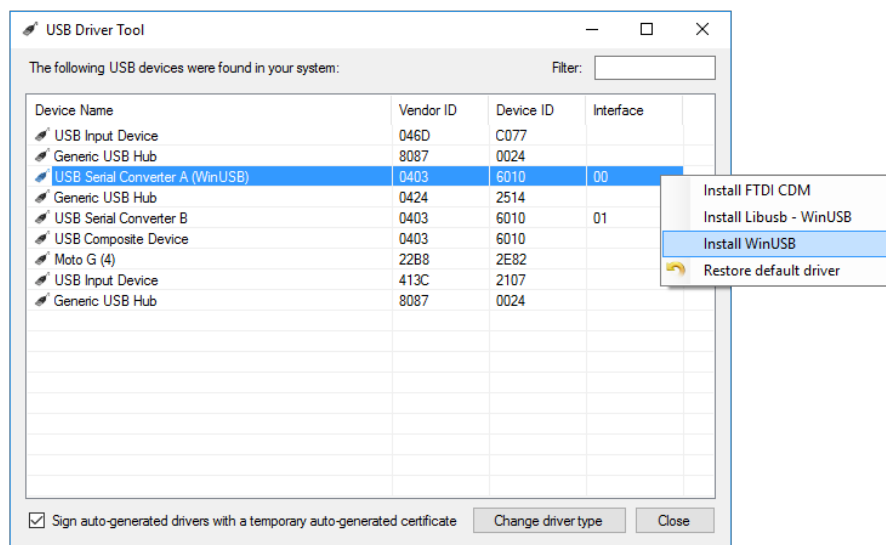


Abbildung 5.2: Installation des WinUSB Treibers mit dem USB Driver Tool

5.2 OpenOCD CLI - Command Line Interface

Das CLI (*Command Line Interface*) ist eine einfache Methode um mit dem Debugger zu kommunizieren. Sobald OpenOCD gestartet wurde, kann über den Port 4444, z.B. mit *Putty*, auf dem *Localhost* eine Telnet-Verbindung aufgebaut werden. Der Befehl "help" listet alle zulässigen Befehle auf.

In den folgenden Kapiteln wird folgende Notation verwendet, um einen CLI-Befehl zu beschreiben: (CLI: Befehl)

5.3 OpenOCD Konfiguration

OpenOCD unterstützt eine Vielzahl von Adaptern und Targets (Prozessoren). Beim Start muss die Software für die verwendete Hardware konfiguriert werden. Die Konfiguration erfolgt mit Konfigurationsscripts (*.cfg) in der Scriptsprache *Jim-Tcl*⁶. *Jim-Tcl* ist eine abgespeckte Version von *Tcl*⁷.

⁵<http://visualgdb.com/UsbDriverTool/>

⁶<http://jim.tcl.tk/index.html/doc/www/www/index.html>

⁷<http://www.tcl.tk>

Normalerweise werden die Scripts in die drei Gruppen *interface*, *board* und *target* aufgeteilt. So kann einfach ein Script ausgewechselt werden, wenn der gleiche Adapter mit einem anderen Prozessor verwendet will. Im Pfad `openocd-0.10.0/scripts` befindet sich eine Sammlung von Konfigurationsscripts für Standardhardware.

Mit folgendem Befehl kann OpenOCD mit der passenden Konfiguration für das Zybo gestartet werden: `openocd -f zybo-ftdi.cfg -f zybo.cfg` Die beiden Scripts werden in den folgenden Kapitel genauer beschrieben.

5.3.1 OpenOCD Konfiguration - Interface

Die Interfacekonfiguration beschreibt hauptsächlich den verwendeten Adapter. Da beim Zybo kein Adapter verwendet wird, sondern der aufgelötete FT2232, wird mit diesem Script der FTDI-Chip und dessen Anbindung an den Zynq konfiguriert.

Da ein FTDI-Chip als Interface verwendet wird, sollte ein passendes Script unter `openocd-0.10.0/scripts/interface/ftdi/` zu finden sein. Keiner der Scripts passte vom Namen her auf Zybo oder FT2232. Eine Google Suche nach einem passenden Script war erfolgreicher. Ein Github User mit dem Namen *emard* hat folgendes Script in einem seiner Repositories⁸ gespeichert:

zybo-ftdi.ocd:

```

1  #
2  # ZYBO ft2232hq usbserial jtag
3  #
4
5  interface ftdi
6  ftdi_device_desc "Digilent Adept USB Device"
7  ftdi_vid_pid 0x0403 0x6010
8
9  ftdi_layout_init 0x3088 0x1f8b
10 #ftdi_layout_signal nTRST -data 0x1000 -oe 0x1000
11 # 0x2000 is reset
12 ftdi_layout_signal nSRST -data 0x3000 -oe 0x1000
13 # green MI07 LED
14 ftdi_layout_signal LED -data 0x0010
15 #ftdi_layout_signal LED -data 0x1000
16
17 reset_config srst_pulls_trst

```

Zeile 5 bis 7 konfigurieren das Interface als ein Standard-FTDI-Interface. Von OpenOCD werden neben dem FT2232 auch noch andere Chips unterstützt. Zeile 7 definiert die *Vendor* und *Device-ID* des USB Devices.

Resetverhalten

Wird aus einer unerlaubten Speicheradresse (CLI: `mdw 0x40000000`) gelesen, dann hängt sich die Debug-Peripherie des Zynq auf. Nach einem unerlaubten Speicherzugriff können auch keine erlaubten Speicherstellen mehr gelesen werden. Beim Versuch erscheint die Fehlermeldung:

Timeout waiting for cortex_a_exec_optcode.

Wahrscheinlich ist die *CoreSight* Debug-Peripherie abgestürzt oder in einem undefinierten Zustand. Aus diesem Grund bekommt OpenOCD keine Antwort vom Zynq, wenn versucht wird, eine Speicheradresse zu lesen. Mit einem manuellen Powercycle des Zybos kann die Hardware wieder zurückgesetzt werden.

Im Supportbereich der Xilinx Homepage⁹ ist eine mögliche Erklärung für dieses Verhalten zu finden. In diesem Artikel wird beschrieben, dass die Fehlermeldung *"Invalid address - it can hang PS interconnect"* erscheint, wenn mit dem XSDB (*Xilinx System Debugger*) auf bestimmte Adressbereiche zugegriffen wird. Die Vermutung liegt nahe, dass der XSDB merkt, wenn auf eine *"Invalid address"* zugegriffen werden soll. Dieser Befehl wird abgefangen und stattdessen wird die Fehlermeldung angezeigt, so dass der *"PS interconnect"*, also der Bus innerhalb des Zynq, nicht abstürzen kann. OpenOCD fängt einen solchen

⁸ https://github.com/f32c/f32c/blob/master/rtl/proj/xilinx/zybo/xram_bram_hdmi_ise/zybo.ocd

⁹ Direkter Link: <https://www.xilinx.com/support/answers/63871.html>

Archivierter Link: <https://web.archive.org/save/https://www.xilinx.com/support/answers/63871.html>

invaliden Zugriff nicht ab, was dann zum Absturz des *”PS interconnect”* führt. Da auch die Peripherie für den Debugger im Zynq von diesem *Interconnect* abhängig ist, stürzt auch die Debug-Peripherie ab, sobald auf einen ungültigen Adressbereich zugegriffen wird.

Mit OpenOCD ist es grundsätzlich möglich, einen Reset automatisch durchzuführen. Dabei wird zwischen einem SRST (*System Reset*) und dem TRST (*TAP Reset*) unterschieden. Der SRST führt einen Powercycle vom ganzen System durch, der TRST setzt mit einem JTAG-Befehl nur den TAP (*Test Access Port*) zurück.

Beim obigen Script ist aber das Resetverhalten nicht sauber definiert. Mit dem Befehl `”CLI: reset halt”` sollte der FT2232 einen Reset des ganzen Zynq durchführen. Der Befehl führt aber zur Fehlermeldung:

```
zynq.cpu0: how to reset?
```

Im OpenOCD User Manual[5] in *”Kapitel 9: Reset Configuration”* ist beschrieben, wie das Resetverhalten konfiguriert werden kann. Mit dem Script-Befehl `”reset_config srst_only”` wird der TAP Reset ignoriert. Da jetzt nur noch der SRST und nicht mehr der TRST verwendet wird, kann das Problem auf den SRST begrenzt werden.

Wenn OpenOCD mit der neuen Konfiguration neu gestartet wird, scheint der Befehl `”CLI: reset halt”` zu funktionieren. Wird vorher aber wieder auf eine ungültige Speicherstelle zugegriffen, dann erscheint beim Reset die Fehlermeldung:

```
Timeout waiting for dpm prepare
```

Das erneute Timeout legt die Vermutung nahe, dass der Zynq nicht ordentlich zurückgesetzt wurde.

Zeile 12 `”ftdi_layout_signal nSRST -data 0x3000 -oe 0x1000”` konfiguriert die I/O Pins des FT2232, welche für den System Reset verwendet werden. Im elektrischen Schema des Zybo (siehe Anhang B.1) könnte überprüft werden, welche I/Os des FT2232 tatsächlich für den Reset verwendet werden. Die Seite mit dem Schema für den FT2232, Seite 7, ist aber als einzige Seite im Schema nicht veröffentlicht worden. Die korrekten I/O Pins lassen sich also nicht mit dem Schema ermitteln. Direkt aus dem PCB sind die Verbindungen auch nicht eindeutig ablesbar, da es sich beim Zybo um ein relativ dichtes PCD mit mehreren Lagen handelt.

Im OpenOCD User Manual[5] wird der `”ftdi_layout_signal nSRST` genauer beschrieben. Der Switch `-data 0x3000` definiert alle relevanten Pins für den SRST und `-oe 0x1000` konfiguriert alle Ausgänge. In einem Versuch wurden diverse Kombinationen für die beiden Switches ausprobiert. Keine Kombination mit nur einem Pin (z.B. `-data 0x2000` mit `-oe 0x2000`) hat funktioniert. Es hat sich dann aber herausgestellt, dass die Kombination `-data 0x3000` mit `-oe 0x3000` tatsächlich einen System Reset ermöglicht.

Weil der Debugger direkt nach dem SRST versucht mit dem Zynq zu kommunizieren, tritt folgende Fehlermeldung auf:

```
...
Invalid ACK (7) in DAP response
JTAG-DP STICKY ERROR
...
```

Mit dem Kommando `”adapter_nsrst_delay 40”` wartet der Debugger nach dem SRST zusätzliche 40 Millisekunden. Diese Wartezeit genügt, damit die FTDI-Interface des Zynq wieder betriebsbereit ist, wenn der Debugger versucht zu kommunizieren.

5.3.2 OpenOCD Konfiguration - Board

Da beim Zybo der Adapter direkt auf dem Board ist, ist die Bordkonfiguration bereits im Konfigurations-script für das Interface enthalten.

5.3.3 OpenOCD Konfiguration - Target

Für das Target, in diesem Fall der Zynq 7000 SOC, ist bereits ein Script unter *openocd-0.10.0/scripts/target/zynq_7000.cfg* enthalten. In diesem Script werden nicht nur beide Kerne des Prozessors definiert, sondern auch ein TAP für das FPGA. Es ist also auch möglich, den FPGA mit dieser Toolchain zu laden.

5.4 CLI-OpenOCD-Toolchain

Das Kernelement der *CLI-OpenOCD-Toolchain* ist das *deep*-Plugin "*OpenOCDInterface*". Es basiert auf dem bestehenden Plugin "*AbatronInterface*" und erfüllt die gleichen Funktionen.

Das Plugin kann von folgendem Repository geklont werden:

<https://github.com/MarcelGehrig/openOCDInterface.git>

5.4.1 Aufbau des OpenOCDInterface

Wie das "*AbatronInterface*" besteht dieses Interface auch nur aus einer Java-Datei. Es besteht aus einer einzigen Klasse (*ch.ntb.inf.openOCDInterface.OpenOCD*), welche die abstrakte Klasse *TargetConnection* von *deep* erweitert.

Da das BDI3000 ein sehr ähnliches CLI wie OpenOCD verwendet, musste oft nur die Syntax von einigen Befehlen angepasst werden. Die Kommunikation mit Telnet konnte übernommen werden. Ein sehr einfaches Beispiel für solch einen ähnlichen Befehl ist die *wirteWord()*-Methode:

```
1 // OpenOCDInterface:
2 out.write(("mw 0x" + Integer.toHexString(address) + " 0x" + Integer.
   toHexString(data) + "\r\n").getBytes()
3 // AbatronInterface:
4 out.write(("mm 0x" + Integer.toHexString(address) + " 0x" + Integer.
   toHexString(data) + "\r\n").getBytes()
```

Etwas aufwändiger waren Methoden wie etwa *readWord()*. Bei OpenOCD wird nicht nur der Wert der Speicherstelle zurückgesendet, sondern auch nochmals die Adresse. Eine Antwort wird in folgender Form zurückgegeben:

0x00000100: e41010004

Deshalb musste für einige Methoden die Antwort geparsed werden.

Alle Debugging-Views sind bereits im *deep*-Plugin selbst implementiert und müssen nicht erneut implementiert werden.

5.4.2 Anpassungen des *deep-Runtime-Library*

Die *deep-Runtime-Library* muss noch ergänzt werden, so dass das "*OpenOCDInterface*" in *deep* integriert werden kann. Die Datei "*openOCD.deep*" unter *config/programmers* hinzufügen. Der Inhalt der Datei ist im Anhang B.7 angehängt.

6 Das ELF-Dateiformat mit STABS Debuginformationen

ELF (*Executable and Linking Format*) ist das Standard-Binärformat von vielen UNIX-ähnlichen Betriebssystemen. Es wird für ausführbare Dateien und auch für Libraries verwendet. Es können auch notwendige Informationen für den Debugger in dieses Format gepackt werden.

Das ELF-Format wird auch für Embedded-Anwendungen verwendet. Das Cross-Kompilierte Programm kann zusammen mit Debuginformationen in eine ELF-Datei gepackt werden. Der *gdb* kann dann genutzt werden, um die Applikation auf das Target zu laden. Im Anschluss kann der *gdb* gleich als Debugger für die Applikation genutzt werden, da alle Notwendigen Informationen in der ELF-Datei vorhanden sind.

In diesem Kapitel wird der grundlegende Aufbau des Formats erklärt. Zusätzlich wird auf einige Details genauer eingegangen, die für einen Debugger relevant sind.

Einen sehr guten Einstieg bietet auch der Artikel "*Understanding the ELF*"¹ von James Fisher. In der Spezifikation für das ELF-Format[4] ist der Aufbau des Formats im Detail erklärt.

6.1 Nützliche Tools im Umgang mit ELF-Dateien

readelf ist ein nützliches Linux-Tool um Informationen einer ELF-Datei anzeigen zu lassen. Unter Windows kann diese Software ebenfalls in der Shell verwendet werden, wenn die "*GNU Embedded Toolchain*" installiert wurde. Im Kapitel 7.1 wird beschrieben, wie die Toolchain installiert werden kann.

6.2 Grundlegender Aufbau

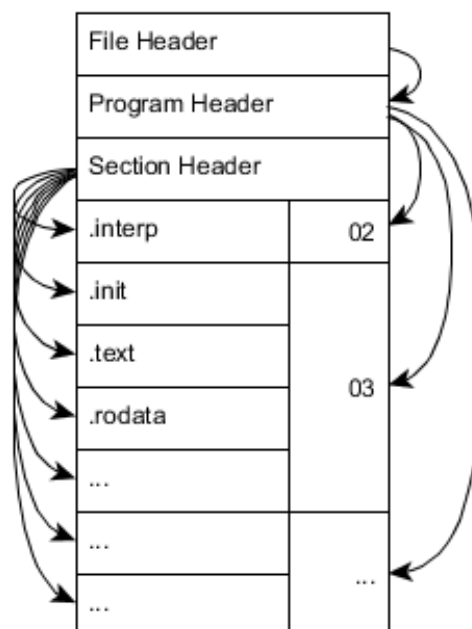


Abbildung 6.1: Der Aufbau einer ELF Datei²

¹ Direkter Link: <https://medium.com/@MrJamesFisher/understanding-the-elf-4bd60daac571>

Archivierter Link: <https://web.archive.org/web/20180705122234/https://medium.com/@MrJamesFisher/understanding-the-elf-4bd60daac571>

Der *File Header* beinhaltet Metainformationen über die Datei selbst. Mit `readelf filename -Wh` lässt sich der *File Header* einer Datei anzeigen.

Der *Program Header* kann mit `readelf filename -Wl` ausgegeben werden. Darin ist enthalten, welchen Offset die einzelnen Segmente innerhalb der Datei haben. Zusätzlich ist auch definiert, zu welcher Speicheradresse (im RAM) die Segmente kopiert werden, wenn das Programm gestartet wird und was für Rechte (ausführbar, lesen und schreiben) jedes Speichersegment hat. Wird, z.B. wegen eines nicht initialisierten Pointers, in einer Speicherstelle im Memory gelesen, die kein *read flag* hat, wird ein *Segmentation Fault* ausgelöst. Der *gdb* nutzt Informationen aus diesem Header um zu bestimmen, welche binären Daten mit dem Befehl `load` an welchen Speicherort die verschiedenen Segmente kopiert werden sollen. Ein Segment beinhaltet ein oder mehrere *Sections*.

Im *Section Header* sind alle *Sections* beschrieben. Mit `readelf filename -WS` wird deutlich, dass jede *Section* unter anderem einen Namen, einen Typ, eine Adresse (absolut) und einen Offset (relativ, innerhalb der ELF-Datei) enthält. Jede *Section* beinhaltet einen anderen Teil des Programms. Die folgende Liste gibt eine nicht vollständige Übersicht über die einzelnen *Sections*:

- `.text` Der ausführbare Teil des Programms.
- `.data` Enthält die globalen Variablen.
- `.rodata` Enthält alle Strings.
- `.stab` Enthält die STABS Debuginformationen. Mehr dazu im Kapitel 6.3
- `.stabstr` Enthält die STABS Debuginformationen. Mehr dazu im Kapitel 6.3

Der Compiler nutzt die *Sections*, um das Programm in logische Einheiten zu unterteilen.

6.2.1 Informationen für den Debugger

Zusätzliche Informationen für den Debugger werden ebenfalls im ELF-Format gespeichert. Moderne Compiler verwenden hauptsächlich das DWARF-Format und nicht das veraltete STABS-Format. Trotzdem wird von aktuellen Compilern und auch Debuggern das veraltete STABS-Format immer noch unterstützt.

DWARF ist flexibler und hat einen besseren funktionalen Umfang als das STABS-Format, aber die manuelle Implementation ist aufwändiger.

6.3 STABS Debuginformationen

STABS ist ein Datenformat für Debuginformationen. Die Informationen sind als Strings in *Symbol Table Strings* gespeichert.

6.3.1 Zielsetzung

Es soll getestet werden, ob es möglich ist, eine *deep*-Applikation mit dem *gdb* zu debuggen. Dazu benötigt der *gdb*, neben dem ausführbaren Maschinencode, zusätzliche Debuginformationen in der Form von STABS oder im DWARF-Format. In beiden Fällen werden die Informationen im ELF-Format eingebettet.

In dieser Arbeit wird ein Demoprogramm mit STABS implementiert, da STABS-Informationen einfacher manuell zu implementieren sind als DWARF-Informationen.

² Direkter Link: <https://slideplayer.com/slide/6444592/>
Archivierter Link: <https://web.archive.org/web/20180808115752/https://slideplayer.com/slide/6444592/>

6.3.2 Aufbau des STABS-Format

Eine einheitliche Dokumentation für STABS gibt es nicht. Es ist nicht einmal sicher bekannt, wer der ursprüngliche Erfinder dieses Formats ist. In der Dokumentation von *Sourceware*³ wird aber Peter Kessler als Erfinder genannt.

Der Aufbau dieses Formats wird in der oben genannten Dokumentation von *Sourceware* und in der Dokumentation der *"University of Utha"*⁴ beschrieben. Obwohl diese Dokumentationen zum Teil sehr detailliert sind, sind sie nicht lückenlos. Im Folgenden wird nur auf die Grundlagen eingegangen, die für die Demo-Applikation relevant sind.

STABS-Informationen sind in einzelne Informations-Elemente, sogenannte *directives*, unterteilt. Jede Direktive ist entweder ein *".stabs"* (String), ein *".stabn"* (Integer) oder ein *".stabd"* (Dot). Zusätzlich hat jede Direktive einen bestimmten Typ. Der Typ definiert, was die einzelnen Direktiven genau beschreiben. Um die Leserlichkeit zu verbessern sind alle Typen in der Datei *".stabs.include"* (Siehe Anhang C.2) definiert. Im Kapitel 12 der Dokumentation der *"University of Utha"* sind die einzelnen Typen genau beschrieben.

Die STABS werden mit folgender Syntax im Assembler-Code definiert:

```
1 .stabs 'string',type,other,desc,value
2 .stabn type,other,desc,value
3 .stabd type,other,desc
```

6.3.3 Das DWARF-Format

DWARF ist ein sehr weit verbreitetes und standardisiertes Dateiformat für Debugging-Informationen. Es hat das veraltete STABS-Format in den meisten modernen Anwendungen abgelöst. Im Gegensatz zum STABS-Format wird das DWARF-Format immer noch weiterentwickelt. Der Blog-Eintrag⁵ von Adarsh Thampan und Suchitra Venugopal zeigt sehr gut die Unterschiede zwischen den beiden Formaten auf.

6.4 Demoprogramm mit STABS

In diesem Kapitel wird beschrieben wie ein Demoprogramm mit STABS-Informationen erstellt werden kann. Das Demoprogramm soll dann mit dem *gdb* direkt auf den Zynq geladen werden. Zusätzlich sollen folgende *gdb*-Features getestet werden:

1. **Breakpoint:** Das Programm stoppt bei einer gewünschten Zeile im Java-Sourcecode.
2. **Sourcecode-Lookup:** Wenn das Programm gestoppt wird, kann die entsprechende Zeile im Java-Sourcecode angezeigt werden.
3. **Single-Stepping:** Nur eine Zeile im Java-Sourcecode ausführen und dann pausieren.
4. **Variable auslesen:** Eine Java-Variable, z.B. ein Integer, auslesen.
5. **Variable manipulieren:** Eine Java-Variable verändern.
6. **Prozessor-Register auslesen:** Ein Register der CPU auslesen.

³ Direkter Link: <https://www.sourceware.org/gdb/onlinedocs/stabs.html>
Archivierter Link: <https://web.archive.org/web/20180717131349/https://www.sourceware.org/gdb/onlinedocs/stabs.html>

⁴ Direkter Link: http://www.math.utah.edu/docs/info/stabs_toc.html
Archivierter Link: https://web.archive.org/web/20180717132825/http://www.math.utah.edu/docs/info/stabs_toc.html

⁵ Direkter Link: <https://www.ibm.com/developerworks/library/os-debugging/>
Archivierter Link: <https://web.archive.org/web/20180808115504/https://www.ibm.com/developerworks/library/os-debugging/>

6.4.1 Vorgehen

Um ein Demoprogramm zu erstellen, werden die untenstehenden Schritte durchgeführt. Alle Schritte werden weiter unten im Detail erklärt. Das Programm *"loop"*, beziehungsweise *"loopWithSTABS"*, soll für den *gdb*-Test verwendet werden. *"loopExample"* ist ein Hilfsprogramm, das vom *gdb* automatisch generierte STABS enthält. Es dient als Vorlage, um die korrekten STABS im Programm *"loop"* hinzufügen zu können.

1. **loop.java**: Demoprogramm als Java-Code Schreiben.
2. Beispiel-Programm mit automatisch generierten STABS erstellen:
 - a) **loopExample.c**: Das Java-Programm manuell in C-Code übersetzen.
 - b) **loopExample.o**: Das Programm mit STABS-Informationen kompilieren.
 - c) **loopExample.Sd**: Das disassemblierte Programm mit STABS in einer leserlichen Form.
 - d) **loopExample.host.c**: Leicht abgeändertes *"loopExample.c"*, um ein ausführbares Programm für den Host-PC zu erhalten.
 - e) **loopExample.host.a**: Ausführbares Programm für den Host-PC.
3. Lauffähiges Demoprogramm für den Zynq mit manuell ergänzten STABS erstellen:
 - a) **Reset.Java**: Den Sourcecode des Java-Programms in die Reset-Methode des *deep*-Kernel kopieren.
 - b) Den modifizierten Kernel mit *deep* übersetzen.
 - c) **loopMachineCode.txt**: Enthält den Maschinen-Code aus der *ClassTreeView* von *deep*.
 - d) **loop.S**: Der aus *"loopMachineCode.txt"* abgeleitete Assembler-Code.
 - e) **loopWithSTABS.S**: Der Assembler-Code inklusive den manuell ergänzten STABS.
 - f) **loopWithSTABS.o**: Kompiliertes Objekt aus dem Assembler-Code.
 - g) **loopWithSTABS**: Gelinktes Objekt aus dem kompilierten Objekt.
 - h) **loopWithSTABS.Sd**: Das disassemblierte Programm mit STABS in einer leserlichen Form.

6.4.2 Java Demoprogramm

Das untenstehende Programm ist das Testprogramm (*loop.java*), dass von *deep* in Maschinen-Code übersetzt werden soll und anschliessend manuell mit STABS ergänzt werden soll.

loop.java:

```

1  static void reset() {
2
3
4
5      US.PUTGPR(SP, stackBase + stackSize - 4); // set stack pointer
6
7      int x00 = 0;
8      int x01 = 1;
9      int x02 = 2;
10
11     x00++;
12     x01++;
13     x02++;
14
15     int x100 = 100;
16     for(int i=0; i<10; i++){
17         x100 += 10;
18     }
19
20     x100++;
21     x100++;
22     x100++;
23     x100++;

```

```

24     x100++;
25
26     US.PUTGPR("b -8"); // stop here
27 }

```

In diesem Beispiel wird die `reset()`-Methode genutzt, da sie bei *deep* als erstes beim Booten ausgeführt wird. „US.PUTGPR“ in Zeile 5 ist natürlich keine Java-Methode. Da Low-Level-Operationen, wie die Initialisierung des Stackpointers, mit Java normalerweise nicht möglich sind, wird hier die entsprechende *deep*-Instruktion verwendet.

6.4.3 Beispiel-Programm "loopExample"

Der Code in „*loopExample.c*“ im Anhang C.3 ist fast identisch mit dem Code des Java-Demoprogramms. Es wurden nur einige Änderungen vorgenommen, damit der Code als C-Programm kompiliert werden kann. `c_entry()` ist der Eintrittspunkt des Programms und erfüllt im embedded Bereich eine ähnliche Aufgabe wie die `main()`-Methode in einem generischen C-Programm.

Mit dem PowerShell-Script „*make_loopExample.ps1*“ im Anhang C.4 kann das C-Programm kompiliert werden. Es erzeugt das Object-File „*loopExample.o*“ inklusive Debuginformationen im STABS-Format. Das disassemblierte Object-File wird als „*loopExample.Sd*“ gespeichert. Im disassemblierten Object-File sind alle STABS-Informationen und auch der ausführbare Code als Assembler enthalten. Der Assembler-Code und auch die STABS-Informationen können direkt „*human readable*“ gelesen werden, aber sie können nicht direkt in einem kompilierbaren Programm verwendet werden, da die Syntax nicht übereinstimmt.

Beispiel mit disassemblierter Syntax:

```

1  ...
2  2      LSYM    0      0      00000000 44      int:t(0,1)=r(0,1)
      ; -2147483648;2147483647;
3  ...
4  00000000 <c_entry>:
5      0: e92d0810 push {r4, fp}

```

Kompilierbare Assembler Syntax:

```

1  ...
2  .stabs "int:t(0,1)=r(0,1);-2147483648;2147483647;",N_LSYM,0,0,0
3  ...
4  c_entry:
5  push {r4, fp}

```

6.4.4 Analyse der disassemblierten STABS

Die untenstehenden Direktiven sind ein Auszug aus der Datei „*loopExample.Sd*“ im Anhang C.5. Die Tabelle 6.1 beschreibt die Direktive 0 im Detail.

	Symnum	n_type	n_othr	n_desc	n_value	n_strx	String
1	...						
2	...						
3	0	S0	0	2	00000000	15	loopExample.c
4	1	OPT	0	0	00000000	29	gcc2_compiled.
5	2	LSYM	0	0	00000000	44	int:t(0,1)=r(0,1)
							; -2147483648;2147483647;
6	...						
7	51	GSYM	0	0	00000000	1919	global:G(0,1)
8	52	FUN	0	0	00000000	1933	c_entry:F(0,1)
9	53	SLINE	0	4	00000000	0	
10	54	SLINE	0	5	00000000	c 0	
11	...						
12	72	LSYM	0	0	ffffff0	1948	x00:(0,1)
13	73	LSYM	0	0	fffffec	1958	x01:(0,1)
14	74	LSYM	0	0	fffffe8	1968	x02:(0,1)
15	75	RSYM	0	0	00000004	1978	s:r(0,1)
16	76	LSYM	0	0	fffffe4	1987	float0:(0,14)
17	77	LSYM	0	0	ffffff8	2001	int0:(0,1)
18	78	LBRAC	0	0	00000000	0	
19	79	LSYM	0	0	ffffff4	2012	i:(0,1)

```

20 80    LBRAC 0      0      00000060 0
21 81    RBRAC 0      0      00000090 0
22 82    RBRAC 0      0      000000c4 0
23 83    SO    0      0      000000c4 0

```

Tabelle 6.1: Disassemblierte STAB-Direktive

<i>Symnum</i>	0	Eindeutige Identifikation der STAB-Direktive
<i>n_type</i>	S0	Typ der STAB-Direktive. Die SO-Direktive beschreibt das Source-File welches die "main()" -Methode enthält.
<i>n_othr</i>	0	Das <i>other</i> -Feld wird normalerweise nicht genutzt und auf "0" gesetzt.
<i>n_desc</i>	2	"the starting text address of the compilation." ⁶
<i>n_value</i>	00000000	Dieser Integer wird hauptsächlich für <i>.stabn</i> -Direktive genutzt.
<i>n_strx</i>	15	Start des Strings der nächste Direktive
<i>String</i>	loopExample.c	Der String, der die eigentliche Information enthält. In diesem Fall ist es das Source-File mit der "main()" -Methode.

Die Direktiven 2 bis 50 beschreiben alle Variablentypen. Für das Testprogramm "loop" können diese einfach kopiert werden.

Die GSYM-Direktive deklariert eine globale Variable. Direktive Nummer 52, vom Typ FUN, definiert eine Methode.

Die Direktiven 53 bis 71 sind vom Typ SLINE. Sie werden für die *Sourcecode-Lookup*-Funktion verwendet. *n_desc* beschreibt die Zeile im Sourcecode und *n_value* die entsprechende Adresse im Maschinencode. Es fällt auf, dass die Sourcecode-Adresse von der Direktive 53 auf 54 nur um eine Zeile steigt, die Maschinencode-Adresse aber von 00000000 auf 0000000c. Im Gegensatz zur Zeilennummer, wird die Adresse im Maschinencode im hexadezimalen System angegeben. Da es sich um 32-Bit lange Maschinen-Instruktionen (also 4 Byte) handelt, steigt die Adresse um 4 nach jeder Instruktion. Es werden also drei Maschinen-Instruktionen ausgeführt, bevor die erste Zeile in der Methode "c_entry()" ausgeführt wird. Im disassemblierten Maschinencode sind folgende Instruktionen zu sehen:

```

1      0: e92d0810  push {r4, fp}
2      4: e28db004   add fp, sp, #4
3      8: e24dd018   sub sp, sp, #24
4      c: e3a03000  mov r3, #0
5     10: e50b3010   str r3, [fp, #-16]

```

Wie es aussieht, wird der Stackpointer mit den ersten drei Instruktionen initialisiert, bevor die erste Zeile, oder genauer gesagt Zeile 5, in "loopExample.c", ausgeführt wird.

Die LSYM-Direktiven ab Nr. 72 definieren Variablen, welche auf dem Stack gespeichert sind. Mit *n_value* wird die Adresse der Variable im Speicher definiert. Der *String* definiert den Variablennamen "x00" und den Typ "(0,1)". Der Typ "(0,1)" wurde mit der Direktive 2 als Integer definiert.

Die Direktive 75 definiert eine Variable, die nicht auf dem Stack gespeichert wird. Dieser Typ wird verwendet, wenn die Variable nur in einem Prozessor-Register gespeichert und nicht auf dem Stack abgelegt wird. Der gcc speichert grundsätzlich alle Variablen direkt auf dem Stack, wenn sie erzeugt oder verändert werden und lädt sie jedes Mal neu vom Stack, wenn sie wieder gelesen werden. Wird beim Kompilieren eine Code-Optimierung verwendet, kann dieses Verhalten ändern. Mit der Zeile "register int s=1;" im C-Code wird der Compiler gezwungen, die Variable in den Registern zu behalten und nicht auf dem Stack abzulegen. Aus diesem Grund wird für die Variable "s" eine Direktive des Typs RSYM verwendet, die nur den Namen der Variable und die Registernummer beschreibt, in der die Variable gespeichert wird.

Mit STABS können auch lexikalische Blöcke abgegrenzt werden, ähnlich wie mit geschweiften Klammern {} in C-Code. Zusätzlich wird so auch die Lebensdauer von Variablen begrenzt. Die Direktiven 78 und 80 (LBRAC) markieren einen Start und die Direktiven 81 und 82 (RBRAC) markieren jeweils das Ende eines solchen Blocks.

6.4.5 Assemblerprogramm mit *deep* erzeugen

Um das Java-Programm möglichst einfach mit *deep* übersetzen zu können, wird die `reset()`-Methode des Objekts `Reset.java` aus dem Package `zynq7000` überschrieben. Diese Methode wird beim Starten einer *deep*-Applikation immer als erstes ausgeführt und ist somit mit einem Debugger gleich ab der ersten Instruktion der Applikation kontrollierbar.

Die untenstehenden Zeilen entsprechen den Zeilen 39-42 von `Reset.java` aus dem Anhang C.6. In diesen Zeilen wird die Position des Stacks ausgerechnet und im Stackpointer gespeichert:

```
1  int stackOffset = US.GET4(sysTabBaseAddr + stStackOffset);
2  int stackBase = US.GET4(sysTabBaseAddr + stackOffset + 4);
3  int stackSize = US.GET4(sysTabBaseAddr + stackOffset + 8);
4  US.PUTGPR(SP, stackBase + stackSize - 4); // set stack pointer
```

Wird ein Dummy-Programm mit dem *deep*-Compiler und dem modifiziertem Kernel kompiliert, dann wird auch der Kernel kompiliert. Mit der *ClassTreeView* (siehe Abbildung 6.2) von *deep* kann der Assemblercode der `reset()`-Methode kopiert werden, welcher im Anhang C.7 angehängt ist.

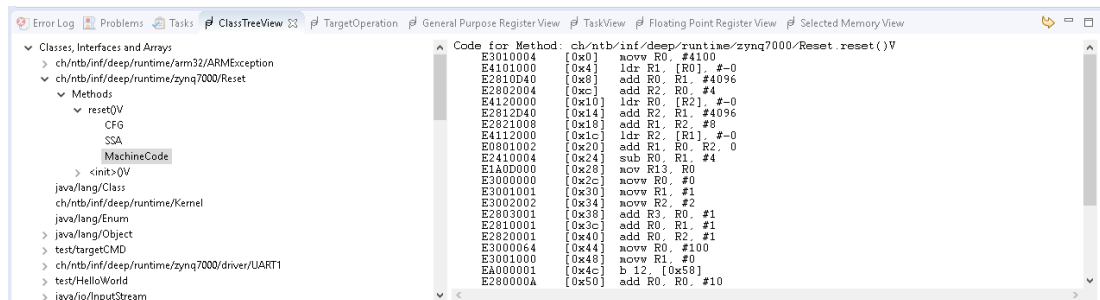


Abbildung 6.2: ClassTreeView mit Maschinencode der Reset-Methode in *deep*

loop.S:

```
1  .global _start
2
3  .org 0x000000
4  .text
5  Ltext0:
6
7  _start:
8  _reset:
9  c_entry:
10 movw R13, #1024
11
12 movw R0, #0
13 movw R1, #10
14 movw R2, #20
15 add R3, R0, #1
16 add R0, R1, #1
17 add R0, R2, #1
18 movw R0, #100
19 movw R1, #0
20 b CHECK_LOOP_EXIT
21 START_LOOP_BODY:
22 add R0, R0, #10
23 add R1, R1, #1
24 CHECK_LOOP_EXIT:
25 cmp R1, #10
26 blt START_LOOP_BODY
27 add R1, R0, #1
28 add R0, R1, #1
29 add R1, R0, #1
30 add R0, R1, #1
31 add R1, R0, #1
32 END:
33 b END
```

”loop.S” im Anhang C.8 enthält den bereinigten Assemblercode. Der Code wurde mit zusätzlichen Assembler-Direktiven ergänzt. ”c_entry” beschreibt den Start des Programms. ”START_LOOP_BODY” und ”CHECK_LOOP_EXIT” sind Punkte, welche für die *For-Loop* benötigt werden.

In Zeile 10 wird der Stackpointer direkt mit einer Konstante gesetzt und nicht mehr mit *deep*-Konstanten ausgerechnet. Zusätzlich kann so auch sichergestellt werden, dass der Stack in einem erlaubten Speicherbereich im OCM angelegt wird.

Die beiden Branch-Instruktionen wurden mit der korrekten Syntax ersetzt. Als Ziel für diese Instruktionen wurden die beiden Assembler-Direktiven ”START_LOOP_BODY” und ”CHECK_LOOP_EXIT” verwendet.

6.4.6 STABS in das Assemblerprogramm einfügen

Um das Assemblerprogramm mit STABS zu ergänzen wurden drei verschiedene Quellen genutzt. Das fertige Assemblerprogramm mit STABS ist im Anhang C.9 angehängt.

Direkter Link: [https://wiki.ntb.ch/infoportal/software/gdb/start?s\[\]=stabs](https://wiki.ntb.ch/infoportal/software/gdb/start?s[]=stabs)

Archivierter Link: [https://web.archive.org/save/https://wiki.ntb.ch/infoportal/software/gdb/start?s\[\]=stabs](https://web.archive.org/save/https://wiki.ntb.ch/infoportal/software/gdb/start?s[]=stabs)

Die NTB-Wiki-Dokumentation⁷ wurde als Ausgangslage genutzt. Die Datei ”stabs.include” (siehe Anhang C.2) konnte direkt genutzt werden. Die Definition des Sourcecods (N_SO) und die Definitionen der Zeilennummern (N_SLIN) konnten ebenfalls übernommen werden.

Da die Definitionen der Variablen-Typen in der NTB-Wiki-Dokumentation nicht vollständig waren, konnten sie leider nicht verwendet werden. Alle Variablendefinitionen, Zeilen 6-75, wurden aus dem disassemblierten Demoprogramm kopiert. Bei der *For-Loop* ist die Definition der Sourcecode-Zeile ebenfalls etwas speziell, da sie auch bei der Überprüfung der Exit-Condition stimmen muss. Die genaue Implementation für die *For-Loop* wurde ebenfalls aus dem disassemblierten Demoprogramm übernommen.

Sofern noch genügend Register frei sind, scheint der *deep*-Compiler die Variablen nicht auf dem Stack zu sichern. Zusätzlich werden die Variablen in den Registern überschrieben, wenn diese im späteren Programmverlauf nicht mehr verwendet werden. Eine Register-Variable wird mit einer Direktive des Typs ’N_RSYM’ definiert, die auf ein bestimmtes Register zeigt. So werden beispielsweise die Register-Variablen x00, x01 und x02 in den Zeilen 84, 89 und 94 definiert.

```
84  .stabs "x00:r(0,1)",N_RSYM,0,4,0
89  .stabs "x01:r(0,1)",N_RSYM,0,4,1
94  .stabs "x02:r(0,1)",N_RSYM,0,4,2
```

Auf der Sourcecode-Zeile 11 wird die Variable x00 um 1 inkrementiert. Im Assemblercode sieht man, dass die Variable neu im Register 3 abgespeichert wird. Aus diesem Grund muss die Register-Variable neu definiert werden.

```
98  # x00++;
99  .stabn N_SLIN, 0, 11, LM11
100 .stabn N_LBRAC, 0, 0, LM11
101 .stabs "x00:r(0,1)",N_RSYM,0,4,3
102 LM11:
103 add R3, R0, #1
```

6.4.7 Demoprogramm mit STABS kompilieren

Das Assemblerprogramm enthält nun alle notwendigen Informationen für den Maschinencode in Form von Assemblerinstruktionen. Die STABS ergänzen das Programm mit allen Informationen, welche der Debugger benötigt.

Mit dem Script ”make_loop.ps1” im Anhang C.10 kann das Programm assembliert werden. Die ELF-Datei ”loopWithSTABS” kann dann mit dem *gdb* geladen werden.

⁷ Direkter Link: [https://wiki.ntb.ch/infoportal/software/gdb/start?s\[\]=stabs](https://wiki.ntb.ch/infoportal/software/gdb/start?s[]=stabs)

Archivierter Link: [https://web.archive.org/save/https://wiki.ntb.ch/infoportal/software/gdb/start?s\[\]=stabs](https://web.archive.org/save/https://wiki.ntb.ch/infoportal/software/gdb/start?s[]=stabs)

7 Der gdb-Debugger

Es gibt diverse Debugger auf dem Markt. Diese Arbeit beschränkt sich aber auf den *gdb* (GNU-Debugger), da dieser unter der GNU GPL (General Public License) Lizenz steht und somit eine Open Source Software ist.

In diesem Kapitel wird beschrieben, wie der *gdb* installiert und genutzt werden kann, um das Demoprogramm aus dem Kapitel 6.4 auf den Zynq zu laden. Anschliessend wird auch gezeigt, wie die Demo-Applikation mit dem *gdb* debuggt werden kann.

7.1 Installation der *gdb*-Software mit der "GNU Embedded Toolchain"

ARM stellt eine komplette "GNU Embedded Toolchain" für ARM Prozessoren zur Verfügung. Sie enthält neben dem GCC-Compiler und dem *gdb* auch noch diverse Hilfsprogramme wie "*readelf*" und "*objdump*". Für diese Arbeit wird die zur Zeit aktuellste "GNU Arm Embedded Toolchain: 7-2018-q2-update" Toolchain verwendet. Sie kann von der ARM-Webseite¹ heruntergeladen werden. Sobald das Archiv auf der lokalen Festplatte entpackt wird, ist die Toolchain einsatzbereit. Bei den Build-Scripten in dieser Arbeit muss jeweils die "*PATH*"-Variable mit dem Pfad zur Toolchain ergänzt werden, damit die Toolchain vom Script gefunden wird.

7.2 *gdb*-Anwendungsbeispiel: "loopWithSTABS" auf das Zybo laden

Mit folgenden Schritten kann das kompilierte Programm "loopWithSTABS" aus dem Kapitel 6.4 auf den Zynq geladen und debuggt werden:

1. Die notwendige Software, wie im Kapitel 5.1 beschrieben, installieren.
2. Das Zybo per USB-Kabel mit dem PC verbinden.
3. OpenOCD in der Shell mit dem Befehl "*openocd -f zybo-ftdi.cfg -f zybo.cfg*" starten. Dazu müssen sich die beiden Konfigurationsdateien "*zybo-ftdi.cfg*" und "*zybo.cfg*" (siehe Anhang B.3 und Anhang B.5) im gleichen Ordner wie das "*openocd*"-Binary befinden.
4. In einer zweiten Shell *gdb* starten. Dazu kann das Shell-Script "*startGdb.ps1*" aus dem Anhang D.1 genutzt werden. Die Pfade im Script müssen angepasst werden. Die Konfigurationsdatei "*gdbInit.txt*" (siehe Anhang D.2) muss im aktiven Ordner vorhanden sein. Alle Pfade in der Konfigurationsdatei müssen ebenfalls angepasst werden.
5. Im "*gdbInit.txt*" wird die ELF-Datei "loopWithSTABS" mit der Instruktion "*file M:/MA/stabs/loopWithSTABS*" automatisch vom *gdb* geladen. Die Instruktion "*load*" lädt dann das Segment ".text" mit dem ausführbaren Code direkt in den Speicher des Zynq.
6. Die Applikation kann jetzt mit dem *gdb* auf dem Zybo debuggt werden.

7.3 Test der *gdb*-Funktionen

In diesem Kapitel werden alle aus dem Kapitel 6.4 geforderten Funktionen getestet. Als Ausgangspunkt dient das Anwendungsbeispiel aus dem Kapitel 7.2. *gdb* kann mit dem Befehl "*q*" beendet und dann neu gestartet werden, damit die Ausgangslage bei jedem Test identisch ist.

Für die bessere Übersicht wird hier nochmals der Java-Code des Demoprogramms "*loop.java*" aufgelistet:

¹<https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads>

```

1  static void reset() {
2
3
4
5      US.PUTGPR(SP, stackBase + stackSize - 4); // set stack pointer
6
7      int x00 = 0;
8      int x01 = 1;
9      int x02 = 2;
10
11     x00++;
12     x01++;
13     x02++;
14
15     int x100 = 100;
16     for(int i=0; i<10; i++){
17         x100 += 10;
18     }
19
20     x100++;
21     x100++;
22     x100++;
23     x100++;
24     x100++;
25
26     US.ASM("b -8"); // stop here
27 }

```

7.3.1 Durchführung des *gdb*-Tests

Mit `list` kann der Sourcecode des Programmes angezeigt werden. `list 10` zeigt den Sourcecode ab der 10. Zeile an. Ein Hardware-Breakpoint auf Zeile 11 kann mit `hbreak 11` erstellt werden. Wird das Programm mit `c` gestartet, dann wird die Ausführung gestoppt, sobald die 11. Zeile des Sourcecodes erreicht wurde. *gdb* zeigt dann an, dass die nächste Zeile `x00++;` sein wird.

Mit `p x00` wird der Inhalt der Variable `x00` angezeigt. Wird mit `s` ein einzelner Step ausgeführt, dann wird eine Zeile Sourcecode ausgeführt und der Wert der Variable `x00` erhöht sich um 1. Das kann mit `p x00` wieder überprüft werden.

Ein weiterer Hardware-Breakpoint auf Zeile 17 (`hbreak 17`) stoppt das Programm innerhalb der For-Loop. Die Variable `i` zeigt zu diesem Zeitpunkt wie erwartet `0`. Wird das Programm fortgesetzt, dann stoppt das Programm wieder auf der Zeile 17 und `i` zeigt `1`. Die Variable `i` kann mit `set var i=9` gesetzt werden. Da mit `i=9` die Abbruchbedingung der For-Loop erfüllt ist, wird der Breakpoint nicht mehr erreicht, wenn das Programm weiter ausgeführt wird. Das Programm hängt jetzt auf der letzten Zeile des Programms fest, und kann mit der Tastenkombination `CTRL + C` gestoppt werden.

Das Schlüsselwort `monitor` kann genutzt werden, um OpenOCD aus dem *gdb* heraus direkt einen Befehl zu erteilen. So kann mit `monitor reg` der OpenOCD-Befehl `reg` genutzt werden, um alle Register anzuzeigen.

Hinweis: Seit der *gdb*-Version 8 funktionieren Software-Breakpoints (z.B. `break 12`) nicht mehr. Bei einem Software-Breakpoint wird eine Instruktion mit einer speziellen Instruktion ersetzt, die dann das Programm stoppt und den Debugger triggert. Das funktioniert bei allen *gdb*-Versionen. Ab der *gdb*-Version 8 wird diese Instruktion aber nicht mehr mit der alten, gültigen Instruktion ersetzt. Aus diesem Grund kann dann das Programm nach einem Software-Breakpoint nicht mehr weiter ausgeführt werden. Die Hardware-Breakpoints funktionieren bei allen Versionen.

7.3.2 Fazit des *gdb*-Tests

Alle geforderten Funktionen des Debuggers können grundsätzlich genutzt werden.

Bei *gdb*-Versionen die neuer als Version 8 sind, können aber nur die Hardware-Breakpoints verwendet werden. Software-Breakpoints könnten aber auch verwendet werden, wenn die ersetzte Instruktion manuell wiederhergestellt wird.

8 Kundennutzen, Fazit und Ausblick

8.1 Rückblick und Kundennutzen

Für das Projekt *deep* und den Robotik-Unterricht steht mit dem Zybo ein kostengünstiges Experimentierboard mit einem leistungsstarken Prozessor und FPGA zur Verfügung.

Das "*OpenOCDInterface*" kann mit dem Zybo gleich eingesetzt werden, wie das bestehende "*Abatron-Interface*" für PowerPCs. Damit kann die Entwicklung des *deep*-Kernel mit dem neuen ARM-Prozessor gleich weitergeführt werden wie bisher.

Die in dieser Arbeit entwickelte Toolchain ermöglicht es, ohne zusätzliche Hardware, ein von *deep* kompiliertes Programm auf den Zynq des Zybos zu laden.

Mit dieser Arbeit konnte auch aufgezeigt werden, dass der *gdb* grundsätzlich genutzt werden kann, um eine *deep*-Applikation direkt auf dem Prozessor zu debuggen. Dabei können auch *gdb*-Features, wie Sourcecode-Lookup verwendet werden, obwohl der *gdb*-Debugger Java nicht mehr direkt unterstützt. Für diese Funktion mussten aber STABS-Informationen manuell zu einem Programm hinzugefügt werden. Der *gdb* kann bis jetzt nur mit dem Command Line Interface genutzt werden.

Die Toolchain kann ebenfalls als Hardware-Debugger verwendet werden. Es ist möglich Prozessorregister und Speichersegmente zu lesen und zu schreiben und es können Hardware-Breakpoints gesetzt werden.

8.2 Ausblick

Die entwickelte Toolchain bietet eine gute Ausgangslage für die weitere Entwicklung vom Projekt *deep*. Damit der Debugger aber effizient genutzt werden kann, fehlen noch zwei Kernelemente:

1. Die Debuginformationen müssen automatisch beim kompilieren der *deep*-Applikation erzeugt werden. Dabei sollten nicht die veralteten STABS verwendet werden, wie in dieser Arbeit, sondern das modernere DWARF-Format.
2. Ein *gdb*-Plugin für *Eclipse* muss noch entwickelt werden. Das "*GNU MCU Eclipse plug-ins for ARM & RISC-V C/C++ developers*"¹ ist ein Plugin für *Eclipse*, das genau dieses Problem löst. Allerdings muss es noch so angepasst werden, damit es in die OpenOCD-Toolchain dieser Arbeit integrierte werden kann.

8.3 Fazit

Die meisten Ziele konnten in dieser Arbeit sehr gut erfüllt werden. Diese Arbeit kann gut als Grundlage für die weiter Entwicklung der *gdb*-Unterstützung verwendet werden.

Mit dem Experimentierboard Zybo Wurde eine sehr geeignete Hardware für Weiterentwicklung von *deep* selber als auch für den zukünftigen Unterricht gefunden. Der FT2232-Chip ermöglicht in Kombination mit OpenOCD eine sehr günstige Toolchain um eine *deep*-Applikation auf auf die Hardware zu laden. Trotz dem geringen Preis ist es aber trotzdem möglich, den ARM-Prozessor auf der Hardwareebene mit dem bestehenden Debuggerinterface von *deep* zu debuggen.

Es konnte auch erfolgreich gezeigt werden, dass es möglich ist, mit *gdb*-Debugger eine *deep*-Applikation zu debuggen.

¹<https://github.com/gnu-mcu-eclipse/eclipse-plugins>

9 Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich während der Masterarbeit fachlich und persönlich unterstützt haben.

Danke auch an Prof. Dr. Urs Graf der mir mit seinem fachlichen Rat stets zur Seite stand.

Danken möchte ich auch der Hirschmann Stiftung für das Stipendium.

Besonders möchte ich mich bei meiner Familie bedanken, welche mir viel Verständnis und Unterstützung während der ganzen Dauer des Studiums entgegengebracht haben.

10 Ehrenwörtliche Versicherung

Der unterzeichnende Autor dieser Arbeit erklärt hiermit, dass er die Arbeit selbst erstellt hat, dass die Literaturangaben vollständig sind und der tatsächlich verwendeten Literatur entsprechen.

St. Gallen, 10. August 2018

Marcel Gehrig

11 Quellenverzeichnis

- [1] Xilinx: *Zynq-7000 - Technical Reference Manual v1.12*, 20 Oktober 2017, <https://www.xilinx.com>
- [2] ARM: *ARM Architecture Reference Manual - ARMv7-A and ARMv7R edition Errata markup*, 2011 Q2, <http://www.arm.com>
- [3] ARM: *Cortex-A9 Floating-Point Unit - Technical Reference Manual r4p1*, 2012, <http://www.arm.com>
- [4] TIS Committee: *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification v1.2* Mai 1995, <http://refspecs.linuxbase.org/elf/elf.pdf>
- [5] Sreekishnan Venkateswaran: *Essential Linux Device Drivers*, 15 Januar 2017, Open On-Chip Debugger: OpenOCD User's Guide

Anhang

A Zynq

A.1 Xilinx SDK Log:

```

1  14:26:34 INFO : Disconnected from the channel tcfchan#2.
2  14:26:36 INFO : 'targets -set -filter {jtag_cable_name =~ "Digilent Zybo
    210279573773A" && level==0} -index 1' command is executed.
3  14:26:36 INFO : 'fpga -state' command is executed.
4  14:26:36 INFO : Connected to target on host '127.0.0.1' and port '3121'.
5  14:26:36 INFO : Jtag cable 'Digilent Zybo 210279573773A' is selected.
6  14:26:36 INFO : 'jtag frequency' command is executed.
7  14:26:36 INFO : Sourcing of 'D:/Vivado/01_gettingStarted/01_gettingStarted.
    sdk/design_1_wrapper_hw_platform_0/ps7_init.tcl' is done.
8  14:26:36 INFO : Context for 'APU' is selected.
9  14:26:38 INFO : Hardware design information is loaded from 'D:/Vivado/01
    _gettingStarted/01_gettingStarted.sdk/design_1_wrapper_hw_platform_0/
    system.hdf'.
10 14:26:38 INFO : 'configparams force-mem-access 1' command is executed.
11 14:26:38 INFO : Context for 'APU' is selected.
12 14:26:38 INFO : 'stop' command is executed.
13 14:26:38 INFO : 'ps7_init' command is executed.
14 14:26:38 INFO : 'ps7_post_config' command is executed.
15 14:26:38 INFO : Context for processor 'ps7_cortexa9_0' is selected.
16 14:26:38 INFO : Processor reset is completed for 'ps7_cortexa9_0'.
17 14:26:38 INFO : Context for processor 'ps7_cortexa9_0' is selected.
18 14:26:39 INFO : The application 'D:/Vivado/01_gettingStarted/01
    _gettingStarted.sdk/01_gettingStarted_ApplicationProject/Debug/01
    _gettingStarted_ApplicationProject.elf' is downloaded to processor '
    ps7_cortexa9_0'.
19 14:26:39 INFO : 'configparams force-mem-access 0' command is executed.
20 14:26:39 INFO : -----XSDB Script-----
21 connect -url tcp:127.0.0.1:3121
22 source D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
    design_1_wrapper_hw_platform_0/ps7_init.tcl
23 targets -set -nocase -filter {name =~ "APU*" && jtag_cable_name =~ "Digilent
    Zybo 210279573773A"} -index 0
24 loadhw -hw D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
    design_1_wrapper_hw_platform_0/system.hdf -mem-ranges [list {0x40000000
    0xbfffffff}]
25 configparams force-mem-access 1
26 targets -set -nocase -filter {name =~ "APU*" && jtag_cable_name =~ "Digilent
    Zybo 210279573773A"} -index 0
27 stop
28 ps7_init
29 ps7_post_config
30 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
    Digilent Zybo 210279573773A"} -index 0
31 rst -processor
32 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
    Digilent Zybo 210279573773A"} -index 0
33 dow D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/01
    _gettingStarted_ApplicationProject/Debug/01
    _gettingStarted_ApplicationProject.elf
34 configparams force-mem-access 0
35 -----End of Script-----
36
37 14:26:39 INFO : Memory regions updated for context APU
38 14:26:39 INFO : Context for processor 'ps7_cortexa9_0' is selected.
39 14:26:39 INFO : 'con' command is executed.
40 14:26:39 INFO : -----XSDB Script (After Launch)-----
41 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
    Digilent Zybo 210279573773A"} -index 0
42 con
43 -----End of Script-----
44
45 14:26:39 INFO : Launch script is exported to file 'D:\Vivado\01
    _gettingStarted\01_gettingStarted.sdk\.sdk\launch_scripts\xilinx_c-c++
    _application_(system_debugger)\

```

```
system_debugger_using_debug_01_gettingstarted_applicationproject.  
elf_on_local.tcl'
```

A.2 Xilinx SDK; helloworld.c:

```
1  /*****  
2  Getting Started Guide for Zybo  
3  
4  This demo displays the status of the switches on the  
5  LEDs and prints a message to the serial communication  
6  when a button is pressed.  
7  
8  Terminal Settings:  
9      -Baud: 115200  
10     -Data bits: 8  
11     -Parity: no  
12     -Stop bits: 1  
13  
14  1/6/14: Created by MarshallW  
15  *****/  
16  
17  #include <stdio.h>  
18  #include "platform.h"  
19  #include <xgpio.h>  
20  #include "xparameters.h"  
21  #include "sleep.h"  
22  
23  int main()  
24  {  
25      XGpio input, output;  
26      int button_data = 0;  
27      int switch_data = 0;  
28  
29      XGpio_Initialize(&input, XPAR_AXI_GPIO_0_DEVICE_ID); //initialize input  
30      XGpio_Initialize(&output, XPAR_AXI_GPIO_1_DEVICE_ID); //initialize  
31      XGpio variable  
32      XGpio_Initialize(&output, XPAR_AXI_GPIO_1_DEVICE_ID); //initialize  
33      output XGpio variable  
34  
35      XGpio_SetDataDirection(&input, 1, 0xF); //set first channel  
36      tristate buffer to input  
37      XGpio_SetDataDirection(&input, 2, 0xF); //set second channel  
38      tristate buffer to input  
39  
40      XGpio_SetDataDirection(&output, 1, 0x0); //set first channel tristate  
41      buffer to output  
42  
43      init_platform();  
44  
45      while(1){  
46          float f = 3.3 * 2.2;  
47          switch_data = XGpio_DiscreteRead(&input, 2); //get switch data  
48  
49          XGpio_DiscreteWrite(&output, 1, switch_data); //write switch data to  
50          the LEDs  
51  
52          button_data = XGpio_DiscreteRead(&input, 1); //get button data  
53  
54          //print message dependent on whether one or more buttons are pressed  
55          if(button_data == 0b0000){} //do nothing  
56  
57          else if(button_data == 0b0001)  
58              xil_printf("button 0 pressed%f\n\r", f);  
59  
60          else if(button_data == 0b0010)  
61              xil_printf("button 1 pressed\n\r");  
62  
63          else if(button_data == 0b0100)  
64              xil_printf("button 2 pressed\n\r");  
65  
66          else if(button_data == 0b1000)  
67              xil_printf("button 3 pressed\n\r");  
68  
69          else  
70              xil_printf("multiple buttons pressed\n\r");  
71      }  
72  }
```

```

64
65         usleep(200000);          //delay
66
67     }
68     cleanup_platform();
69     return 0;
70 }

```

A.3 Xilinx SDK; platform.c:

```

1  /*
2      *****
3
4      *
5      * Copyright (C) 2010 - 2015 Xilinx, Inc. All rights reserved.
6      *
7      * Permission is hereby granted, free of charge, to any person obtaining a
8      * copy
9      * of this software and associated documentation files (the "Software"), to
10     * deal
11     * in the Software without restriction, including without limitation the
12     * rights
13     * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
14     * copies of the Software, and to permit persons to whom the Software is
15     * furnished to do so, subject to the following conditions:
16     *
17     * The above copyright notice and this permission notice shall be included
18     * in
19     * all copies or substantial portions of the Software.
20     *
21     * Use of the Software is limited solely to applications:
22     * (a) running on a Xilinx device, or
23     * (b) that interact with a Xilinx device through a bus or interconnect.
24     *
25     * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
26     * OR
27     * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
28     * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
29     * XILINX BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
30     * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF
31     * OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
32     * SOFTWARE.
33     *
34     * Except as contained in this notice, the name of the Xilinx shall not be
35     * used
36     * in advertising or otherwise to promote the sale, use or other dealings in
37     * this Software without prior written authorization from Xilinx.
38     *
39     *****
40     */
41
42     #include "xparameters.h"
43     #include "xil_cache.h"
44
45     #include "platform_config.h"
46
47     /*
48     * Uncomment one of the following two lines, depending on the target,
49     * if ps7/psu init source files are added in the source directory for
50     * compiling example outside of SDK.
51     */
52     /*#include "ps7_init.h"
53     #include "psu_init.h"
54
55     #ifdef STDOUT_IS_16550
56     #include "xuartns550_1.h"
57
58     #define UART_BAUD 9600
59     #endif
60
61     void
62     enable_caches()
63     {
64     #ifdef __PPC__

```

```

56     Xil_ICacheEnableRegion(CACHEABLE_REGION_MASK);
57     Xil_DCacheEnableRegion(CACHEABLE_REGION_MASK);
58 #elif __MICROBLAZE__
59 #ifdef XPAR_MICROBLAZE_USE_ICACHE
60     Xil_ICacheEnable();
61 #endif
62 #ifdef XPAR_MICROBLAZE_USE_DCACHE
63     Xil_DCacheEnable();
64 #endif
65 #endif
66 }
67
68 void
69 disable_caches()
70 {
71     Xil_DCacheDisable();
72     Xil_ICacheDisable();
73 }
74
75 void
76 init_uart()
77 {
78 #ifdef STDOUT_IS_16550
79     XUartNs550_SetBaud(STDOUT_BASEADDR, XPAR_XUARTNS550_CLOCK_HZ, UART_BAUD
80 );
81     XUartNs550_SetLineControlReg(STDOUT_BASEADDR, XUN_LCR_8_DATA_BITS);
82 #endif
83     /* Bootrom/BSP configures PS7/PSU UART to 115200 bps */
84 }
85
86 void
87 init_platform()
88 {
89     /*
90      * If you want to run this example outside of SDK,
91      * uncomment one of the following two lines and also #include "ps7_init
92      * .h"
93      * or #include "ps7_init.h" at the top, depending on the target.
94      * Make sure that the ps7/psu_init.c and ps7/psu_init.h files are
95      * included
96      * along with this example source files for compilation.
97      */
98     /* ps7_init();*/
99     /* psu_init();*/
100     enable_caches();
101     init_uart();
102 }
103
104 void
105 cleanup_platform()
106 {
107     disable_caches();
108 }

```

A.4 ps7_init_modified.tcl

```

1  #
2  # Based on D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
3  #   design_1_wrapper_hw_platform_0/ps7_init.tcl
4
5  proc unlock_SLCR {} {
6      mww 0xF8000008 0x0000DF0D
7  }
8
9  proc map_OCM_low {} {
10     unlock_SLCR
11     mww 0xF8000910 0x00000010
12 }
13
14
15
16
17

```



```

18  proc memread32 {ADDR} {
19      set foo(0) 0
20      if ![ catch { mem2array foo 32 $ADDR 1 } msg ] {
21          return $foo(0)
22      } else {
23          error "memread32: $msg"
24      }
25  }
26
27  proc mask_write { addr mask val } {
28      set curval [memread32 $addr]
29      set maskinv [expr {0xffffffff ~ $mask}]
30      set maskedcur [expr {$maskinv & $curval}]
31      set maskedval [expr {$mask & $val}]
32      set newval [expr $maskedcur | $maskedval]
33      mww $addr $newval
34  }
35
36  proc ps7_pll_init_data_3_0 {} {
37      mww 0XF8000008 0x0000DF0D
38      mask_write 0XF8000110 0x003FFFFF 0x001772C0
39      mask_write 0XF8000100 0x0007F000 0x0001A000
40      mask_write 0XF8000100 0x00000010 0x00000010
41      mask_write 0XF8000100 0x00000001 0x00000001
42      mask_write 0XF8000100 0x00000001 0x00000000
43      mask_poll 0XF800010C 0x00000001
44      mask_write 0XF8000100 0x00000010 0x00000000
45      mask_write 0XF8000120 0x1F003F30 0x1F000200
46      mask_write 0XF8000114 0x003FFFFF 0x001DB2C0
47      mask_write 0XF8000104 0x0007F000 0x00015000
48      mask_write 0XF8000104 0x00000010 0x00000010
49      mask_write 0XF8000104 0x00000001 0x00000001
50      mask_write 0XF8000104 0x00000001 0x00000000
51      mask_poll 0XF800010C 0x00000002
52      mask_write 0XF8000104 0x00000010 0x00000000
53      mask_write 0XF8000124 0xFFF00003 0x0C200003
54      mask_write 0XF8000118 0x003FFFFF 0x001F42C0
55      mask_write 0XF8000108 0x0007F000 0x00014000
56      mask_write 0XF8000108 0x00000010 0x00000010
57      mask_write 0XF8000108 0x00000001 0x00000001
58      mask_write 0XF8000108 0x00000001 0x00000000
59      mask_poll 0XF800010C 0x00000004
60      mask_write 0XF8000108 0x00000010 0x00000000
61      mww 0XF8000004 0x0000767B
62  }
63
64  proc ps7_clock_init_data_3_0 {} {
65      mww 0XF8000008 0x0000DF0D
66      mask_write 0XF8000128 0x03F03F01 0x00203401
67      mask_write 0XF8000138 0x00000011 0x00000001
68      mask_write 0XF8000140 0x03F03F71 0x00100801
69      mask_write 0XF800014C 0x00003F31 0x00000501
70      mask_write 0XF8000150 0x00003F33 0x00001401
71      mask_write 0XF8000154 0x00003F33 0x00000A02
72      mask_write 0XF8000168 0x00003F31 0x00000501
73      mask_write 0XF8000170 0x03F03F30 0x00200500
74      mask_write 0XF80001C4 0x00000001 0x00000001
75      mask_write 0XF800012C 0x01FFCCCD 0x01EC044D
76      mww 0XF8000004 0x0000767B
77  }
78
79  proc ps7_ddr_init_data_3_0 {} {
80      mask_write 0XF8006000 0x0001FFFF 0x00000080
81      mask_write 0XF8006004 0x0007FFFF 0x0000107F
82      mask_write 0XF8006008 0x03FFFFFF 0x03C0780F
83      mask_write 0XF800600C 0x03FFFFFF 0x02001001
84      mask_write 0XF8006010 0x03FFFFFF 0x00014001
85      mask_write 0XF8006014 0x001FFFFF 0x0004151A
86      mask_write 0XF8006018 0xF7FFFFFF 0x44E354D2
87      mask_write 0XF800601C 0xFFFFFFFF 0x720238E5
88      mask_write 0XF8006020 0x7FDFFFC 0x270872D0
89      mask_write 0XF8006024 0x0FFFFFC3 0x00000000
90      mask_write 0XF8006028 0x00003FFF 0x00002007
91      mask_write 0XF800602C 0xFFFFFFFF 0x00000008
92      mask_write 0XF8006030 0xFFFFFFFF 0x00040930
93      mask_write 0XF8006034 0x13FF3FFF 0x00011674
94      mask_write 0XF8006038 0x00000003 0x00000000

```

```

93     mask_write 0XF800603C 0x000FFFFF 0x00000777
94     mask_write 0XF8006040 0xFFFFFFFF 0xFFFF0000
95     mask_write 0XF8006044 0x0FFFFFFF 0x0FF66666
96     mask_write 0XF8006048 0x0003F03F 0x0003C008
97     mask_write 0XF8006050 0xFF0F8FFF 0x77010800
98     mask_write 0XF8006058 0x00010000 0x00000000
99     mask_write 0XF800605C 0x0000FFFF 0x00005003
100    mask_write 0XF8006060 0x000017FF 0x0000003E
101    mask_write 0XF8006064 0x00021FE0 0x00020000
102    mask_write 0XF8006068 0x03FFFFFF 0x00284141
103    mask_write 0XF800606C 0x0000FFFF 0x00001610
104    mask_write 0XF8006078 0x03FFFFFF 0x00466111
105    mask_write 0XF800607C 0x0000FFFF 0x00032222
106    mask_write 0XF80060A4 0xFFFFFFFF 0x10200802
107    mask_write 0XF80060A8 0x0FFFFFFF 0x0670C845
108    mask_write 0XF80060AC 0x000001FF 0x000001FE
109    mask_write 0XF80060B0 0x1FFFFFFF 0x1CFFFFFF
110    mask_write 0XF80060B4 0x00000200 0x00000200
111    mask_write 0XF80060B8 0x01FFFFFF 0x00200066
112    mask_write 0XF80060C4 0x00000003 0x00000000
113    mask_write 0XF80060C8 0x000000FF 0x00000000
114    mask_write 0XF80060DC 0x00000001 0x00000000
115    mask_write 0XF80060F0 0x0000FFFF 0x00000000
116    mask_write 0XF80060F4 0x0000000F 0x00000008
117    mask_write 0XF8006114 0x000000FF 0x00000000
118    mask_write 0XF8006118 0x7FFFFFFC 0x40000001
119    mask_write 0XF800611C 0x7FFFFFFC 0x40000001
120    mask_write 0XF8006120 0x7FFFFFFC 0x40000001
121    mask_write 0XF8006124 0x7FFFFFFC 0x40000001
122    mask_write 0XF800612C 0x0000FFFF 0x00023C00
123    mask_write 0XF8006130 0x0000FFFF 0x00022800
124    mask_write 0XF8006134 0x0000FFFF 0x00022C00
125    mask_write 0XF8006138 0x0000FFFF 0x00024800
126    mask_write 0XF8006140 0x0000FFFF 0x00000035
127    mask_write 0XF8006144 0x0000FFFF 0x00000035
128    mask_write 0XF8006148 0x0000FFFF 0x00000035
129    mask_write 0XF800614C 0x0000FFFF 0x00000035
130    mask_write 0XF8006154 0x0000FFFF 0x00000077
131    mask_write 0XF8006158 0x0000FFFF 0x0000007C
132    mask_write 0XF800615C 0x0000FFFF 0x0000007C
133    mask_write 0XF8006160 0x0000FFFF 0x00000075
134    mask_write 0XF8006168 0x0010FFFF 0x000000E4
135    mask_write 0XF800616C 0x0010FFFF 0x000000DF
136    mask_write 0XF8006170 0x0010FFFF 0x000000E0
137    mask_write 0XF8006174 0x0010FFFF 0x000000E7
138    mask_write 0XF800617C 0x0000FFFF 0x000000B7
139    mask_write 0XF8006180 0x0000FFFF 0x000000BC
140    mask_write 0XF8006184 0x0000FFFF 0x000000BC
141    mask_write 0XF8006188 0x0000FFFF 0x000000B5
142    mask_write 0XF8006190 0x6FFFFFFE 0x00040080
143    mask_write 0XF8006194 0x0000FFFF 0x0001FC82
144    mask_write 0XF8006204 0xFFFFFFFF 0x00000000
145    mask_write 0XF8006208 0x000703FF 0x000003FF
146    mask_write 0XF800620C 0x000703FF 0x000003FF
147    mask_write 0XF8006210 0x000703FF 0x000003FF
148    mask_write 0XF8006214 0x000703FF 0x000003FF
149    mask_write 0XF8006218 0x000F03FF 0x000003FF
150    mask_write 0XF800621C 0x000F03FF 0x000003FF
151    mask_write 0XF8006220 0x000F03FF 0x000003FF
152    mask_write 0XF8006224 0x000F03FF 0x000003FF
153    mask_write 0XF80062A8 0x00000FF5 0x00000000
154    mask_write 0XF80062AC 0xFFFFFFFF 0x00000000
155    mask_write 0XF80062B0 0x003FFFFFF 0x00005125
156    mask_write 0XF80062B4 0x0003FFFF 0x000012A6
157    mask_poll 0XF8000B74 0x00002000
158    mask_write 0XF8006000 0x0001FFFF 0x00000081
159    mask_poll 0XF8006054 0x00000007
160 }
161
162 proc ps7_mio {} {
163     mww 0XF8000008 0x0000DF0D
164     mask_write 0XF8000B00 0x00000071 0x00000001
165     mask_write 0XF8000B40 0x00000FFF 0x00000600
166     mask_write 0XF8000B44 0x00000FFF 0x00000600
167     mask_write 0XF8000B48 0x00000FFF 0x00000672

```

```

168     mask_write 0XF8000B4C 0x00000FFF 0x00000672
169     mask_write 0XF8000B50 0x00000FFF 0x00000674
170     mask_write 0XF8000B54 0x00000FFF 0x00000674
171     mask_write 0XF8000B58 0x00000FFF 0x00000600
172     mask_write 0XF8000B5C 0xFFFFFFFF 0x0018C61C
173     mask_write 0XF8000B60 0xFFFFFFFF 0x00F9861C
174     mask_write 0XF8000B64 0xFFFFFFFF 0x00F9861C
175     mask_write 0XF8000B68 0xFFFFFFFF 0x00F9861C
176     mask_write 0XF8000B6C 0x00007FFF 0x00000260
177     mask_write 0XF8000B70 0x00000001 0x00000001
178     mask_write 0XF8000B70 0x00000021 0x00000020
179     mask_write 0XF8000B70 0x07FFFFFF 0x00000823
180     mask_write 0XF8000700 0x00003FFF 0x00001600
181     mask_write 0XF8000704 0x00003FFF 0x00000702
182     mask_write 0XF8000708 0x00003FFF 0x00000702
183     mask_write 0XF800070C 0x00003FFF 0x00000702
184     mask_write 0XF8000710 0x00003FFF 0x00000702
185     mask_write 0XF8000714 0x00003FFF 0x00000702
186     mask_write 0XF8000718 0x00003FFF 0x00000702
187     mask_write 0XF800071C 0x00003FFF 0x00000600
188     mask_write 0XF8000720 0x00003FFF 0x00000702
189     mask_write 0XF8000724 0x00003FFF 0x00001600
190     mask_write 0XF8000728 0x00003FFF 0x00001600
191     mask_write 0XF800072C 0x00003FFF 0x00001600
192     mask_write 0XF8000730 0x00003FFF 0x00001600
193     mask_write 0XF8000734 0x00003FFF 0x00001600
194     mask_write 0XF8000738 0x00003FFF 0x00001600
195     mask_write 0XF800073C 0x00003FFF 0x00001600
196     mask_write 0XF8000740 0x00003FFF 0x00002902
197     mask_write 0XF8000744 0x00003FFF 0x00002902
198 }
199
200 proc ps7_mio_2 {} {
201     mww 0XF8000008 0x0000DF0D
202     mask_write 0XF8000B00 0x00000071 0x00000001
203     mask_write 0XF8000B40 0x00000FFF 0x00000600
204     mask_write 0XF8000B44 0x00000FFF 0x00000600
205     mask_write 0XF8000B48 0x00000FFF 0x00000672
206     mask_write 0XF8000B4C 0x00000FFF 0x00000672
207     mask_write 0XF8000B50 0x00000FFF 0x00000674
208     mask_write 0XF8000B54 0x00000FFF 0x00000674
209     mask_write 0XF8000B58 0x00000FFF 0x00000600
210     mask_write 0XF8000B5C 0xFFFFFFFF 0x0018C61C
211     mask_write 0XF8000B60 0xFFFFFFFF 0x00F9861C
212     mask_write 0XF8000B64 0xFFFFFFFF 0x00F9861C
213     mask_write 0XF8000B68 0xFFFFFFFF 0x00F9861C
214     mask_write 0XF8000B6C 0x00007FFF 0x00000260
215     mask_write 0XF8000B70 0x00000001 0x00000001
216     mask_write 0XF8000B70 0x00000021 0x00000020
217 }
218 proc ps7_mio_init_data_3_0 {} {
219     mww 0XF8000008 0x0000DF0D
220     mask_write 0XF8000B00 0x00000071 0x00000001
221     mask_write 0XF8000B40 0x00000FFF 0x00000600
222     mask_write 0XF8000B44 0x00000FFF 0x00000600
223     mask_write 0XF8000B48 0x00000FFF 0x00000672
224     mask_write 0XF8000B4C 0x00000FFF 0x00000672
225     mask_write 0XF8000B50 0x00000FFF 0x00000674
226     mask_write 0XF8000B54 0x00000FFF 0x00000674
227     mask_write 0XF8000B58 0x00000FFF 0x00000600
228     mask_write 0XF8000B5C 0xFFFFFFFF 0x0018C61C
229     mask_write 0XF8000B60 0xFFFFFFFF 0x00F9861C
230     mask_write 0XF8000B64 0xFFFFFFFF 0x00F9861C
231     mask_write 0XF8000B68 0xFFFFFFFF 0x00F9861C
232     mask_write 0XF8000B6C 0x00007FFF 0x00000260
233     mask_write 0XF8000B70 0x00000001 0x00000001
234     mask_write 0XF8000B70 0x00000021 0x00000020
235     mask_write 0XF8000B70 0x07FFFFFF 0x00000823
236     mask_write 0XF8000700 0x00003FFF 0x00001600
237     mask_write 0XF8000704 0x00003FFF 0x00000702
238     mask_write 0XF8000708 0x00003FFF 0x00000702
239     mask_write 0XF800070C 0x00003FFF 0x00000702
240     mask_write 0XF8000710 0x00003FFF 0x00000702
241     mask_write 0XF8000714 0x00003FFF 0x00000702
242     mask_write 0XF8000718 0x00003FFF 0x00000702

```

```

243     mask_write 0XF800071C 0x00003FFF 0x00000600
244     mask_write 0XF8000720 0x00003FFF 0x00000702
245     mask_write 0XF8000724 0x00003FFF 0x00001600
246     mask_write 0XF8000728 0x00003FFF 0x00001600
247     mask_write 0XF800072C 0x00003FFF 0x00001600
248     mask_write 0XF8000730 0x00003FFF 0x00001600
249     mask_write 0XF8000734 0x00003FFF 0x00001600
250     mask_write 0XF8000738 0x00003FFF 0x00001600
251     mask_write 0XF800073C 0x00003FFF 0x00001600
252     mask_write 0XF8000740 0x00003FFF 0x00002902
253     mask_write 0XF8000744 0x00003FFF 0x00002902
254
255     mask_write 0XF8000748 0x00003FFF 0x00002902
256     mask_write 0XF800074C 0x00003FFF 0x00002902
257     mask_write 0XF8000750 0x00003FFF 0x00002902
258     mask_write 0XF8000754 0x00003FFF 0x00002902
259     mask_write 0XF8000758 0x00003FFF 0x00000903
260     mask_write 0XF800075C 0x00003FFF 0x00000903
261     mask_write 0XF8000760 0x00003FFF 0x00000903
262     mask_write 0XF8000764 0x00003FFF 0x00000903
263     mask_write 0XF8000768 0x00003FFF 0x00000903
264     mask_write 0XF800076C 0x00003FFF 0x00000903
265     mask_write 0XF8000770 0x00003FFF 0x00000304
266     mask_write 0XF8000774 0x00003FFF 0x00000305
267     mask_write 0XF8000778 0x00003FFF 0x00000304
268     mask_write 0XF800077C 0x00003FFF 0x00000305
269     mask_write 0XF8000780 0x00003FFF 0x00000304
270     mask_write 0XF8000784 0x00003FFF 0x00000304
271     mask_write 0XF8000788 0x00003FFF 0x00000304
272     mask_write 0XF800078C 0x00003FFF 0x00000304
273     mask_write 0XF8000790 0x00003FFF 0x00000305
274     mask_write 0XF8000794 0x00003FFF 0x00000304
275     mask_write 0XF8000798 0x00003FFF 0x00000304
276     mask_write 0XF800079C 0x00003FFF 0x00000304
277     mask_write 0XF80007A0 0x00003FFF 0x00000380
278     mask_write 0XF80007A4 0x00003FFF 0x00000380
279     mask_write 0XF80007A8 0x00003FFF 0x00000380
280     mask_write 0XF80007AC 0x00003FFF 0x00000380
281     mask_write 0XF80007B0 0x00003FFF 0x00000380
282     mask_write 0XF80007B4 0x00003FFF 0x00000380
283     mask_write 0XF80007B8 0x00003FFF 0x00001200
284     mask_write 0XF80007BC 0x00003F01 0x00000201
285     mask_write 0XF80007C0 0x00003FFF 0x000002E0
286     mask_write 0XF80007C4 0x00003FFF 0x000002E1
287     mask_write 0XF80007C8 0x00003FFF 0x00000200
288     mask_write 0XF80007CC 0x00003FFF 0x00000200
289     mask_write 0XF80007D0 0x00003FFF 0x00000200
290     mask_write 0XF80007D4 0x00003FFF 0x00000200
291     mask_write 0XF8000830 0x003F003F 0x002F0037
292     mww 0XF8000004 0x0000767B
293 }
294 proc ps7_peripherals_init_data_3_0 {} {
295     mww 0XF8000008 0x0000DF0D
296     mask_write 0XF8000B48 0x00000180 0x00000180
297     mask_write 0XF8000B4C 0x00000180 0x00000180
298     mask_write 0XF8000B50 0x00000180 0x00000180
299     mask_write 0XF8000B54 0x00000180 0x00000180
300     mww 0XF8000004 0x0000767B
301     mask_write 0XE0001034 0x000000FF 0x00000006
302     mask_write 0XE0001018 0x0000FFFF 0x0000007C
303     mask_write 0XE0001000 0x000001FF 0x00000017
304     mask_write 0XE0001004 0x000003FF 0x00000020
305     mask_write 0XE000D000 0x00080000 0x00080000
306     mask_write 0XF8007000 0x20000000 0x00000000
307     mask_write 0XE000A244 0x003FFFFFFF 0x00004000
308     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF4000
309     mask_write 0XE000A248 0x003FFFFFFF 0x00004000
310     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF0000
311     mask_delay 0XF8F00200 1
312     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF4000
313     mask_delay 0XF8F00200 1
314     mask_delay 0XF8F00200 1
315     mask_delay 0XF8F00200 1
316     mask_delay 0XF8F00200 1
317     mask_delay 0XF8F00200 1

```

```

318 }
319 proc ps7_post_config_3_0 {} {
320     mww 0XF8000008 0x0000DF0D
321     mask_write 0XF8000900 0x0000000F 0x0000000F
322     mask_write 0XF8000240 0xFFFFFFFF 0x00000000
323     mww 0XF8000004 0x0000767B
324 }
325 proc ps7_debug_3_0 {} {
326     mww 0XF8898FB0 0xC5ACCE55
327     mww 0XF8899FB0 0xC5ACCE55
328     mww 0XF8809FB0 0xC5ACCE55
329 }
330 proc ps7_pll_init_data_2_0 {} {
331     mww 0XF8000008 0x0000DF0D
332     mask_write 0XF8000110 0x003FFFF0 0x001772C0
333     mask_write 0XF8000100 0x0007F000 0x0001A000
334     mask_write 0XF8000100 0x00000010 0x00000010
335     mask_write 0XF8000100 0x00000001 0x00000001
336     mask_write 0XF8000100 0x00000001 0x00000000
337     mask_poll 0XF800010C 0x00000001
338     mask_write 0XF8000100 0x00000010 0x00000000
339     mask_write 0XF8000120 0x1F003F30 0x1F000200
340     mask_write 0XF8000114 0x003FFFF0 0x001DB2C0
341     mask_write 0XF8000104 0x0007F000 0x00015000
342     mask_write 0XF8000104 0x00000010 0x00000010
343     mask_write 0XF8000104 0x00000001 0x00000001
344     mask_write 0XF8000104 0x00000001 0x00000000
345     mask_poll 0XF800010C 0x00000002
346     mask_write 0XF8000104 0x00000010 0x00000000
347     mask_write 0XF8000124 0xFFFF0003 0x0C200003
348     mask_write 0XF8000118 0x003FFFF0 0x001F42C0
349     mask_write 0XF8000108 0x0007F000 0x00014000
350     mask_write 0XF8000108 0x00000010 0x00000010
351     mask_write 0XF8000108 0x00000001 0x00000001
352     mask_write 0XF8000108 0x00000001 0x00000000
353     mask_poll 0XF800010C 0x00000004
354     mask_write 0XF8000108 0x00000010 0x00000000
355     mww 0XF8000004 0x0000767B
356 }
357 proc ps7_clock_init_data_2_0 {} {
358     mww 0XF8000008 0x0000DF0D
359     mask_write 0XF8000128 0x03F03F01 0x00203401
360     mask_write 0XF8000138 0x00000011 0x00000001
361     mask_write 0XF8000140 0x03F03F71 0x00100801
362     mask_write 0XF800014C 0x00003F31 0x00000501
363     mask_write 0XF8000150 0x00003F33 0x00001401
364     mask_write 0XF8000154 0x00003F33 0x00000A02
365     mask_write 0XF8000168 0x00003F31 0x00000501
366     mask_write 0XF8000170 0x03F03F30 0x00200500
367     mask_write 0XF80001C4 0x00000001 0x00000001
368     mask_write 0XF800012C 0x01FFCCCD 0x01EC044D
369     mww 0XF8000004 0x0000767B
370 }
371 proc ps7_ddr_init_data_2_0 {} {
372     mask_write 0XF8006000 0x0001FFFF 0x00000080
373     mask_write 0XF8006004 0x1FFFFFFF 0x0008107F
374     mask_write 0XF8006008 0x03FFFFFF 0x03C0780F
375     mask_write 0XF800600C 0x03FFFFFF 0x02001001
376     mask_write 0XF8006010 0x03FFFFFF 0x00014001
377     mask_write 0XF8006014 0x001FFFFFF 0x0004151A
378     mask_write 0XF8006018 0xF7FFFFFF 0x44E354D2
379     mask_write 0XF800601C 0xFFFFFFFF 0x720238E5
380     mask_write 0XF8006020 0xFFFFFFFF 0x272872D0
381     mask_write 0XF8006024 0x0FFFFFFF 0x0000003C
382     mask_write 0XF8006028 0x00003FFF 0x00002007
383     mask_write 0XF800602C 0xFFFFFFFF 0x00000008
384     mask_write 0XF8006030 0xFFFFFFFF 0x00040930
385     mask_write 0XF8006034 0x13FF3FFF 0x00011674
386     mask_write 0XF8006038 0x00001FC3 0x00000000
387     mask_write 0XF800603C 0x000FFFFF 0x00000777
388     mask_write 0XF8006040 0xFFFFFFFF 0xFFF00000
389     mask_write 0XF8006044 0x0FFFFFFF 0xFF666666
390     mask_write 0XF8006048 0x3FFFFFFF 0x0003C248
391     mask_write 0XF8006050 0xFF0F8FFF 0x77010800
392     mask_write 0XF8006058 0x0001FFFF 0x00000101

```

```

393     mask_write 0XF800605C 0x0000FFFF 0x00005003
394     mask_write 0XF8006060 0x000017FF 0x0000003E
395     mask_write 0XF8006064 0x00021FE0 0x00020000
396     mask_write 0XF8006068 0x03FFFFFF 0x00284141
397     mask_write 0XF800606C 0x0000FFFF 0x00001610
398     mask_write 0XF8006078 0x03FFFFFF 0x00466111
399     mask_write 0XF800607C 0x0000FFFF 0x00032222
400     mask_write 0XF80060A0 0x00FFFFFF 0x00008000
401     mask_write 0XF80060A4 0xFFFFFFFF 0x10200802
402     mask_write 0XF80060A8 0x0FFFFFFF 0x0670C845
403     mask_write 0XF80060AC 0x000001FF 0x000001FE
404     mask_write 0XF80060B0 0x1FFFFFFF 0x1CFFFFFF
405     mask_write 0XF80060B4 0x000007FF 0x00000200
406     mask_write 0XF80060B8 0x01FFFFFF 0x00200066
407     mask_write 0XF80060C4 0x00000003 0x00000000
408     mask_write 0XF80060C8 0x000000FF 0x00000000
409     mask_write 0XF80060DC 0x00000001 0x00000000
410     mask_write 0XF80060F0 0x0000FFFF 0x00000000
411     mask_write 0XF80060F4 0x0000000F 0x00000008
412     mask_write 0XF8006114 0x000000FF 0x00000000
413     mask_write 0XF8006118 0x7FFFFFFF 0x40000001
414     mask_write 0XF800611C 0x7FFFFFFF 0x40000001
415     mask_write 0XF8006120 0x7FFFFFFF 0x40000001
416     mask_write 0XF8006124 0x7FFFFFFF 0x40000001
417     mask_write 0XF800612C 0x0000FFFF 0x00023C00
418     mask_write 0XF8006130 0x0000FFFF 0x00022800
419     mask_write 0XF8006134 0x0000FFFF 0x00022C00
420     mask_write 0XF8006138 0x0000FFFF 0x00024800
421     mask_write 0XF8006140 0x0000FFFF 0x00000035
422     mask_write 0XF8006144 0x0000FFFF 0x00000035
423     mask_write 0XF8006148 0x0000FFFF 0x00000035
424     mask_write 0XF800614C 0x0000FFFF 0x00000035
425     mask_write 0XF8006154 0x0000FFFF 0x00000077
426     mask_write 0XF8006158 0x0000FFFF 0x0000007C
427     mask_write 0XF800615C 0x0000FFFF 0x0000007C
428     mask_write 0XF8006160 0x0000FFFF 0x00000075
429     mask_write 0XF8006168 0x001FFFFF 0x000000E4
430     mask_write 0XF800616C 0x001FFFFF 0x000000DF
431     mask_write 0XF8006170 0x001FFFFF 0x000000E0
432     mask_write 0XF8006174 0x001FFFFF 0x000000E7
433     mask_write 0XF800617C 0x0000FFFF 0x000000B7
434     mask_write 0XF8006180 0x0000FFFF 0x000000BC
435     mask_write 0XF8006184 0x0000FFFF 0x000000BC
436     mask_write 0XF8006188 0x0000FFFF 0x000000B5
437     mask_write 0XF8006190 0xFFFFFFFF 0x10040080
438     mask_write 0XF8006194 0x0000FFFF 0x0001FC82
439     mask_write 0XF8006204 0xFFFFFFFF 0x00000000
440     mask_write 0XF8006208 0x000F03FF 0x000803FF
441     mask_write 0XF800620C 0x000F03FF 0x000803FF
442     mask_write 0XF8006210 0x000F03FF 0x000803FF
443     mask_write 0XF8006214 0x000F03FF 0x000803FF
444     mask_write 0XF8006218 0x000F03FF 0x000003FF
445     mask_write 0XF800621C 0x000F03FF 0x000003FF
446     mask_write 0XF8006220 0x000F03FF 0x000003FF
447     mask_write 0XF8006224 0x000F03FF 0x000003FF
448     mask_write 0XF80062A8 0x00000FF7 0x00000000
449     mask_write 0XF80062AC 0xFFFFFFFF 0x00000000
450     mask_write 0XF80062B0 0x003FFFFFF 0x00005125
451     mask_write 0XF80062B4 0x0003FFFF 0x000012A6
452     mask_poll 0XF8000B74 0x00002000
453     mask_write 0XF8006000 0x0001FFFF 0x00000081
454     mask_poll 0XF8006054 0x00000007
455 }
456 proc ps7_mio_init_data_2_0 {} {
457     mww 0XF8000008 0x0000DF0D
458     mask_write 0XF8000B00 0x00000303 0x00000001
459     mask_write 0XF8000B40 0x00000FFF 0x00000600
460     mask_write 0XF8000B44 0x00000FFF 0x00000600
461     mask_write 0XF8000B48 0x00000FFF 0x00000672
462     mask_write 0XF8000B4C 0x00000FFF 0x00000672
463     mask_write 0XF8000B50 0x00000FFF 0x00000674
464     mask_write 0XF8000B54 0x00000FFF 0x00000674
465     mask_write 0XF8000B58 0x00000FFF 0x00000600
466     mask_write 0XF8000B5C 0xFFFFFFFF 0x0018C61C
467     mask_write 0XF8000B60 0xFFFFFFFF 0x00F9861C

```

```

468     mask_write 0XF8000B64 0xFFFFFFFF 0x00F9861C
469     mask_write 0XF8000B68 0xFFFFFFFF 0x00F9861C
470     mask_write 0XF8000B6C 0x00007FFF 0x00000260
471     mask_write 0XF8000B70 0x00000021 0x00000021
472     mask_write 0XF8000B70 0x00000021 0x00000020
473     mask_write 0XF8000B70 0x07FFFFFF 0x00000823
474     mask_write 0XF8000700 0x00003FFF 0x00001600
475     mask_write 0XF8000704 0x00003FFF 0x00000702
476     mask_write 0XF8000708 0x00003FFF 0x00000702
477     mask_write 0XF800070C 0x00003FFF 0x00000702
478     mask_write 0XF8000710 0x00003FFF 0x00000702
479     mask_write 0XF8000714 0x00003FFF 0x00000702
480     mask_write 0XF8000718 0x00003FFF 0x00000702
481     mask_write 0XF800071C 0x00003FFF 0x00000600
482     mask_write 0XF8000720 0x00003FFF 0x00000702
483     mask_write 0XF8000724 0x00003FFF 0x00001600
484     mask_write 0XF8000728 0x00003FFF 0x00001600
485     mask_write 0XF800072C 0x00003FFF 0x00001600
486     mask_write 0XF8000730 0x00003FFF 0x00001600
487     mask_write 0XF8000734 0x00003FFF 0x00001600
488     mask_write 0XF8000738 0x00003FFF 0x00001600
489     mask_write 0XF800073C 0x00003FFF 0x00001600
490     mask_write 0XF8000740 0x00003FFF 0x00002902
491     mask_write 0XF8000744 0x00003FFF 0x00002902
492     mask_write 0XF8000748 0x00003FFF 0x00002902
493     mask_write 0XF800074C 0x00003FFF 0x00002902
494     mask_write 0XF8000750 0x00003FFF 0x00002902
495     mask_write 0XF8000754 0x00003FFF 0x00002902
496     mask_write 0XF8000758 0x00003FFF 0x00000903
497     mask_write 0XF800075C 0x00003FFF 0x00000903
498     mask_write 0XF8000760 0x00003FFF 0x00000903
499     mask_write 0XF8000764 0x00003FFF 0x00000903
500     mask_write 0XF8000768 0x00003FFF 0x00000903
501     mask_write 0XF800076C 0x00003FFF 0x00000903
502     mask_write 0XF8000770 0x00003FFF 0x00000304
503     mask_write 0XF8000774 0x00003FFF 0x00000305
504     mask_write 0XF8000778 0x00003FFF 0x00000304
505     mask_write 0XF800077C 0x00003FFF 0x00000305
506     mask_write 0XF8000780 0x00003FFF 0x00000304
507     mask_write 0XF8000784 0x00003FFF 0x00000304
508     mask_write 0XF8000788 0x00003FFF 0x00000304
509     mask_write 0XF800078C 0x00003FFF 0x00000304
510     mask_write 0XF8000790 0x00003FFF 0x00000305
511     mask_write 0XF8000794 0x00003FFF 0x00000304
512     mask_write 0XF8000798 0x00003FFF 0x00000304
513     mask_write 0XF800079C 0x00003FFF 0x00000304
514     mask_write 0XF80007A0 0x00003FFF 0x00000380
515     mask_write 0XF80007A4 0x00003FFF 0x00000380
516     mask_write 0XF80007A8 0x00003FFF 0x00000380
517     mask_write 0XF80007AC 0x00003FFF 0x00000380
518     mask_write 0XF80007B0 0x00003FFF 0x00000380
519     mask_write 0XF80007B4 0x00003FFF 0x00000380
520     mask_write 0XF80007B8 0x00003FFF 0x00001200
521     mask_write 0XF80007BC 0x00003F01 0x00000201
522     mask_write 0XF80007C0 0x00003FFF 0x000002E0
523     mask_write 0XF80007C4 0x00003FFF 0x000002E1
524     mask_write 0XF80007C8 0x00003FFF 0x00000200
525     mask_write 0XF80007CC 0x00003FFF 0x00000200
526     mask_write 0XF80007D0 0x00003FFF 0x00000200
527     mask_write 0XF80007D4 0x00003FFF 0x00000200
528     mask_write 0XF8000830 0x003F003F 0x002F0037
529     mww 0XF8000004 0x0000767B
530 }
531 proc ps7_peripherals_init_data_2_0 {} {
532     mww 0XF8000008 0x0000DF0D
533     mask_write 0XF8000B48 0x00000180 0x00000180
534     mask_write 0XF8000B4C 0x00000180 0x00000180
535     mask_write 0XF8000B50 0x00000180 0x00000180
536     mask_write 0XF8000B54 0x00000180 0x00000180
537     mww 0XF8000004 0x0000767B
538     mask_write 0XE0001034 0x000000FF 0x00000006
539     mask_write 0XE0001018 0x0000FFFF 0x0000007C
540     mask_write 0XE0001000 0x000001FF 0x00000017
541     mask_write 0XE0001004 0x00000FFF 0x00000020
542     mask_write 0XE000D000 0x00080000 0x00080000

```

```

543     mask_write 0XF8007000 0x20000000 0x00000000
544     mask_write 0XE000A244 0x003FFFFFFF 0x00004000
545     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF4000
546     mask_write 0XE000A248 0x003FFFFFFF 0x00004000
547     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF0000
548     mask_delay 0XF8F00200 1
549     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF4000
550     mask_delay 0XF8F00200 1
551     mask_delay 0XF8F00200 1
552     mask_delay 0XF8F00200 1
553     mask_delay 0XF8F00200 1
554     mask_delay 0XF8F00200 1
555 }
556 proc ps7_post_config_2_0 {} {
557     mww 0XF8000008 0x0000DF0D
558     mask_write 0XF8000900 0x0000000F 0x0000000F
559     mask_write 0XF8000240 0xFFFFFFFF 0x00000000
560     mww 0XF8000004 0x0000767B
561 }
562 proc ps7_debug_2_0 {} {
563     mww 0XF8898FB0 0xC5ACCE55
564     mww 0XF8899FB0 0xC5ACCE55
565     mww 0XF8809FB0 0xC5ACCE55
566 }
567 proc ps7_pll_init_data_1_0 {} {
568     mww 0XF8000008 0x0000DF0D
569     mask_write 0XF8000110 0x003FFFFFFF 0x001772C0
570     mask_write 0XF8000100 0x0007F000 0x0001A000
571     mask_write 0XF8000100 0x00000010 0x00000010
572     mask_write 0XF8000100 0x00000001 0x00000001
573     mask_write 0XF8000100 0x00000001 0x00000000
574     mask_poll 0XF800010C 0x00000001
575     mask_write 0XF8000100 0x00000010 0x00000000
576     mask_write 0XF8000120 0x1F003F30 0x1F000200
577     mask_write 0XF8000114 0x003FFFFFFF 0x001DB2C0
578     mask_write 0XF8000104 0x0007F000 0x00015000
579     mask_write 0XF8000104 0x00000010 0x00000010
580     mask_write 0XF8000104 0x00000001 0x00000001
581     mask_write 0XF8000104 0x00000001 0x00000000
582     mask_poll 0XF800010C 0x00000002
583     mask_write 0XF8000104 0x00000010 0x00000000
584     mask_write 0XF8000124 0xFFFF0003 0x0C200003
585     mask_write 0XF8000118 0x003FFFFFFF 0x001F42C0
586     mask_write 0XF8000108 0x0007F000 0x00014000
587     mask_write 0XF8000108 0x00000010 0x00000010
588     mask_write 0XF8000108 0x00000001 0x00000001
589     mask_write 0XF8000108 0x00000001 0x00000000
590     mask_poll 0XF800010C 0x00000004
591     mask_write 0XF8000108 0x00000010 0x00000000
592     mww 0XF8000004 0x0000767B
593 }
594 proc ps7_clock_init_data_1_0 {} {
595     mww 0XF8000008 0x0000DF0D
596     mask_write 0XF8000128 0x03F03F01 0x00203401
597     mask_write 0XF8000138 0x00000011 0x00000001
598     mask_write 0XF8000140 0x03F03F71 0x00100801
599     mask_write 0XF800014C 0x00003F31 0x00000501
600     mask_write 0XF8000150 0x00003F33 0x00001401
601     mask_write 0XF8000154 0x00003F33 0x00000A02
602     mask_write 0XF8000168 0x00003F31 0x00000501
603     mask_write 0XF8000170 0x03F03F30 0x00200500
604     mask_write 0XF80001C4 0x00000001 0x00000001
605     mask_write 0XF800012C 0x01FFCCCD 0x01EC044D
606     mww 0XF8000004 0x0000767B
607 }
608 proc ps7_ddr_init_data_1_0 {} {
609     mask_write 0XF8006000 0x0001FFFF 0x00000080
610     mask_write 0XF8006004 0x1FFFFFFF 0x0008107F
611     mask_write 0XF8006008 0x03FFFFFF 0x03C0780F
612     mask_write 0XF800600C 0x03FFFFFF 0x02001001
613     mask_write 0XF8006010 0x03FFFFFF 0x00014001
614     mask_write 0XF8006014 0x001FFFFF 0x0004151A
615     mask_write 0XF8006018 0xF7FFFFFF 0x44E354D2
616     mask_write 0XF800601C 0xFFFFFFFF 0x720238E5
617     mask_write 0XF8006020 0xFFFFFFFF 0x272872D0

```



```

618     mask_write 0XF8006024 0x0FFFFFFF 0x0000003C
619     mask_write 0XF8006028 0x00003FFF 0x00002007
620     mask_write 0XF800602C 0xFFFFFFFF 0x00000008
621     mask_write 0XF8006030 0xFFFFFFFF 0x00040930
622     mask_write 0XF8006034 0x13FF3FFF 0x00011674
623     mask_write 0XF8006038 0x00001FC3 0x00000000
624     mask_write 0XF800603C 0x000FFFFF 0x00000777
625     mask_write 0XF8006040 0xFFFFFFFF 0xFFF00000
626     mask_write 0XF8006044 0x0FFFFFFF 0xFF666666
627     mask_write 0XF8006048 0x3FFFFFFF 0x0003C248
628     mask_write 0XF8006050 0xFF0F8FFF 0x77010800
629     mask_write 0XF8006058 0x0001FFFF 0x00000101
630     mask_write 0XF800605C 0x0000FFFF 0x00005003
631     mask_write 0XF8006060 0x000017FF 0x0000003E
632     mask_write 0XF8006064 0x00021FE0 0x00020000
633     mask_write 0XF8006068 0x03FFFFFF 0x00284141
634     mask_write 0XF800606C 0x0000FFFF 0x00001610
635     mask_write 0XF80060A0 0x0000FFFF 0x00008000
636     mask_write 0XF80060A4 0xFFFFFFFF 0x10200802
637     mask_write 0XF80060A8 0x0FFFFFFF 0x0670C845
638     mask_write 0XF80060AC 0x000001FF 0x000001FE
639     mask_write 0XF80060B0 0x1FFFFFFF 0x1CFFFFFF
640     mask_write 0XF80060B4 0x000007FF 0x00000200
641     mask_write 0XF80060B8 0x01FFFFFF 0x00200066
642     mask_write 0XF80060C4 0x00000003 0x00000000
643     mask_write 0XF80060C8 0x000000FF 0x00000000
644     mask_write 0XF80060DC 0x00000001 0x00000000
645     mask_write 0XF80060F0 0x0000FFFF 0x00000000
646     mask_write 0XF80060F4 0x0000000F 0x00000008
647     mask_write 0XF8006114 0x000000FF 0x00000000
648     mask_write 0XF8006118 0x7FFFFFFF 0x40000001
649     mask_write 0XF800611C 0x7FFFFFFF 0x40000001
650     mask_write 0XF8006120 0x7FFFFFFF 0x40000001
651     mask_write 0XF8006124 0x7FFFFFFF 0x40000001
652     mask_write 0XF800612C 0x000FFFFF 0x00023C00
653     mask_write 0XF8006130 0x000FFFFF 0x00022800
654     mask_write 0XF8006134 0x000FFFFF 0x00022C00
655     mask_write 0XF8006138 0x000FFFFF 0x00024800
656     mask_write 0XF8006140 0x000FFFFF 0x00000035
657     mask_write 0XF8006144 0x000FFFFF 0x00000035
658     mask_write 0XF8006148 0x000FFFFF 0x00000035
659     mask_write 0XF800614C 0x000FFFFF 0x00000035
660     mask_write 0XF8006154 0x000FFFFF 0x00000077
661     mask_write 0XF8006158 0x000FFFFF 0x0000007C
662     mask_write 0XF800615C 0x000FFFFF 0x0000007C
663     mask_write 0XF8006160 0x000FFFFF 0x00000075
664     mask_write 0XF8006168 0x001FFFFF 0x000000E4
665     mask_write 0XF800616C 0x001FFFFF 0x000000DF
666     mask_write 0XF8006170 0x001FFFFF 0x000000E0
667     mask_write 0XF8006174 0x001FFFFF 0x000000E7
668     mask_write 0XF800617C 0x000FFFFF 0x000000B7
669     mask_write 0XF8006180 0x000FFFFF 0x000000BC
670     mask_write 0XF8006184 0x000FFFFF 0x000000BC
671     mask_write 0XF8006188 0x000FFFFF 0x000000B5
672     mask_write 0XF8006190 0xFFFFFFFF 0x10040080
673     mask_write 0XF8006194 0x000FFFFF 0x0001FC82
674     mask_write 0XF8006204 0xFFFFFFFF 0x00000000
675     mask_write 0XF8006208 0x000F03FF 0x000803FF
676     mask_write 0XF800620C 0x000F03FF 0x000803FF
677     mask_write 0XF8006210 0x000F03FF 0x000803FF
678     mask_write 0XF8006214 0x000F03FF 0x000803FF
679     mask_write 0XF8006218 0x000F03FF 0x000003FF
680     mask_write 0XF800621C 0x000F03FF 0x000003FF
681     mask_write 0XF8006220 0x000F03FF 0x000003FF
682     mask_write 0XF8006224 0x000F03FF 0x000003FF
683     mask_write 0XF80062A8 0x00000FF7 0x00000000
684     mask_write 0XF80062AC 0xFFFFFFFF 0x00000000
685     mask_write 0XF80062B0 0x003FFFFF 0x00005125
686     mask_write 0XF80062B4 0x0003FFFF 0x000012A6
687     mask_poll 0XF8000B74 0x00002000
688     mask_write 0XF8006000 0x0001FFFF 0x00000081
689     mask_poll 0XF8006054 0x00000007
690 }
691 proc ps7_mio_init_data_1_0 {} {
692     mww 0XF8000008 0x0000DF0D

```

```

693     mask_write 0XF8000B00 0x00000303 0x00000001
694     mask_write 0XF8000B40 0x00000FFF 0x00000600
695     mask_write 0XF8000B44 0x00000FFF 0x00000600
696     mask_write 0XF8000B48 0x00000FFF 0x00000672
697     mask_write 0XF8000B4C 0x00000FFF 0x00000672
698     mask_write 0XF8000B50 0x00000FFF 0x00000674
699     mask_write 0XF8000B54 0x00000FFF 0x00000674
700     mask_write 0XF8000B58 0x00000FFF 0x00000600
701     mask_write 0XF8000B5C 0xFFFFFFFF 0x0018C61C
702     mask_write 0XF8000B60 0xFFFFFFFF 0x00F9861C
703     mask_write 0XF8000B64 0xFFFFFFFF 0x00F9861C
704     mask_write 0XF8000B68 0xFFFFFFFF 0x00F9861C
705     mask_write 0XF8000B6C 0x000073FF 0x00000260
706     mask_write 0XF8000B70 0x00000021 0x00000021
707     mask_write 0XF8000B70 0x00000021 0x00000020
708     mask_write 0XF8000B70 0x07FFFFFF 0x00000823
709     mask_write 0XF8000700 0x00003FFF 0x00001600
710     mask_write 0XF8000704 0x00003FFF 0x00000702
711     mask_write 0XF8000708 0x00003FFF 0x00000702
712     mask_write 0XF800070C 0x00003FFF 0x00000702
713     mask_write 0XF8000710 0x00003FFF 0x00000702
714     mask_write 0XF8000714 0x00003FFF 0x00000702
715     mask_write 0XF8000718 0x00003FFF 0x00000702
716     mask_write 0XF800071C 0x00003FFF 0x00000600
717     mask_write 0XF8000720 0x00003FFF 0x00000702
718     mask_write 0XF8000724 0x00003FFF 0x00001600
719     mask_write 0XF8000728 0x00003FFF 0x00001600
720     mask_write 0XF800072C 0x00003FFF 0x00001600
721     mask_write 0XF8000730 0x00003FFF 0x00001600
722     mask_write 0XF8000734 0x00003FFF 0x00001600
723     mask_write 0XF8000738 0x00003FFF 0x00001600
724     mask_write 0XF800073C 0x00003FFF 0x00001600
725     mask_write 0XF8000740 0x00003FFF 0x00002902
726     mask_write 0XF8000744 0x00003FFF 0x00002902
727     mask_write 0XF8000748 0x00003FFF 0x00002902
728     mask_write 0XF800074C 0x00003FFF 0x00002902
729     mask_write 0XF8000750 0x00003FFF 0x00002902
730     mask_write 0XF8000754 0x00003FFF 0x00002902
731     mask_write 0XF8000758 0x00003FFF 0x00000903
732     mask_write 0XF800075C 0x00003FFF 0x00000903
733     mask_write 0XF8000760 0x00003FFF 0x00000903
734     mask_write 0XF8000764 0x00003FFF 0x00000903
735     mask_write 0XF8000768 0x00003FFF 0x00000903
736     mask_write 0XF800076C 0x00003FFF 0x00000903
737     mask_write 0XF8000770 0x00003FFF 0x00000304
738     mask_write 0XF8000774 0x00003FFF 0x00000305
739     mask_write 0XF8000778 0x00003FFF 0x00000304
740     mask_write 0XF800077C 0x00003FFF 0x00000305
741     mask_write 0XF8000780 0x00003FFF 0x00000304
742     mask_write 0XF8000784 0x00003FFF 0x00000304
743     mask_write 0XF8000788 0x00003FFF 0x00000304
744     mask_write 0XF800078C 0x00003FFF 0x00000304
745     mask_write 0XF8000790 0x00003FFF 0x00000305
746     mask_write 0XF8000794 0x00003FFF 0x00000304
747     mask_write 0XF8000798 0x00003FFF 0x00000304
748     mask_write 0XF800079C 0x00003FFF 0x00000304
749     mask_write 0XF80007A0 0x00003FFF 0x00000380
750     mask_write 0XF80007A4 0x00003FFF 0x00000380
751     mask_write 0XF80007A8 0x00003FFF 0x00000380
752     mask_write 0XF80007AC 0x00003FFF 0x00000380
753     mask_write 0XF80007B0 0x00003FFF 0x00000380
754     mask_write 0XF80007B4 0x00003FFF 0x00000380
755     mask_write 0XF80007B8 0x00003FFF 0x00001200
756     mask_write 0XF80007BC 0x00003F01 0x00000201
757     mask_write 0XF80007C0 0x00003FFF 0x000002E0
758     mask_write 0XF80007C4 0x00003FFF 0x000002E1
759     mask_write 0XF80007C8 0x00003FFF 0x00000200
760     mask_write 0XF80007CC 0x00003FFF 0x00000200
761     mask_write 0XF80007D0 0x00003FFF 0x00000200
762     mask_write 0XF80007D4 0x00003FFF 0x00000200
763     mask_write 0XF8000830 0x003F003F 0x002F0037
764     mww 0XF8000004 0x0000767B
765 }
766 proc ps7_peripherals_init_data_1_0 {} {
767     mww 0XF8000008 0x0000DF0D

```

```

768     mask_write 0XF8000B48 0x00000180 0x00000180
769     mask_write 0XF8000B4C 0x00000180 0x00000180
770     mask_write 0XF8000B50 0x00000180 0x00000180
771     mask_write 0XF8000B54 0x00000180 0x00000180
772     mww 0XF8000004 0x0000767B
773     mask_write 0XE0001034 0x000000FF 0x00000006
774     mask_write 0XE0001018 0x0000FFFF 0x0000007C
775     mask_write 0XE0001000 0x000001FF 0x00000017
776     mask_write 0XE0001004 0x00000FFF 0x00000020
777     mask_write 0XE000D000 0x00080000 0x00080000
778     mask_write 0XF8007000 0x20000000 0x00000000
779     mask_write 0XE000A244 0x003FFFFFFF 0x00004000
780     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF4000
781     mask_write 0XE000A248 0x003FFFFFFF 0x00004000
782     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF0000
783     mask_delay 0XF8F00200 1
784     mask_write 0XE000A008 0xFFFFFFFF 0xBFFF4000
785     mask_delay 0XF8F00200 1
786     mask_delay 0XF8F00200 1
787     mask_delay 0XF8F00200 1
788     mask_delay 0XF8F00200 1
789     mask_delay 0XF8F00200 1
790 }
791 proc ps7_post_config_1_0 {} {
792     mww 0XF8000008 0x0000DF0D
793     mask_write 0XF8000900 0x0000000F 0x0000000F
794     mask_write 0XF8000240 0xFFFFFFFF 0x00000000
795     mww 0XF8000004 0x0000767B
796 }
797 proc ps7_debug_1_0 {} {
798     mww 0XF8898FB0 0xC5ACCE55
799     mww 0XF8899FB0 0xC5ACCE55
800     mww 0XF8809FB0 0xC5ACCE55
801 }
802 set PCW_SILICON_VER_1_0 "0x0"
803 set PCW_SILICON_VER_2_0 "0x1"
804 set PCW_SILICON_VER_3_0 "0x2"
805 set APU_FREQ 650000000
806
807
808 proc mask_poll { addr mask } {
809     set count 1
810     set curval [memread32 $addr]
811     # & = bitwise AND
812     set maskedval [expr {$curval & $mask}]
813     while { $maskedval == 0 } {
814         set curval [memread32 $addr]
815         set maskedval [expr {$curval & $mask}]
816         set count [ expr { $count + 1 } ]
817         if { $count == 100000000 } {
818             puts "Timeout Reached. Mask poll failed at ADDRESS: $addr MASK:
819                 $mask"
820             break
821         }
822     }
823 }
824
825 proc mask_delay { addr val } {
826     set delay [ get_number_of_cycles_for_delay $val ]
827     perf_reset_and_start_timer
828     set curval [memread32 $addr]
829     set maskedval [expr {$curval < $delay}]
830     while { $maskedval == 1 } {
831         set curval [memread32 $addr]
832         set maskedval [expr {$curval < $delay}]
833     }
834     perf_reset_clock
835 }
836
837 proc ps_version { } {
838     set si_ver "0x[string range [mrd 0XF8007080] end-8 end]"
839     set mask_si_ver "0x[expr {$si_ver >> 28}]"
840     return $mask_si_ver;
841 }

```

```

842
843 proc ps7_post_config {} {
844     # set saved_mode [configparams force-mem-accesses]
845     # configparams force-mem-accesses 1
846
847     # variable PCW_SILICON_VER_1_0
848     # variable PCW_SILICON_VER_2_0
849     # variable PCW_SILICON_VER_3_0
850     # set sil_ver [ps_version]
851     #
852     # if { $sil_ver == $PCW_SILICON_VER_1_0 } {
853     #     ps7_post_config_1_0
854     # } elseif { $sil_ver == $PCW_SILICON_VER_2_0 } {
855     #     ps7_post_config_2_0
856     # } else {
857         ps7_post_config_3_0
858     # }
859     # configparams force-mem-accesses $saved_mode
860 }
861
862 proc ps7_debug {} {
863     variable PCW_SILICON_VER_1_0
864     variable PCW_SILICON_VER_2_0
865     variable PCW_SILICON_VER_3_0
866     set sil_ver [ps_version]
867
868     if { $sil_ver == $PCW_SILICON_VER_1_0 } {
869         ps7_debug_1_0
870     } elseif { $sil_ver == $PCW_SILICON_VER_2_0 } {
871         ps7_debug_2_0
872     } else {
873         ps7_debug_3_0
874     }
875 }
876
877 proc ps7_init {} {
878     halt
879     ps7_mio_init_data_3_0
880     ps7_pll_init_data_3_0
881     ps7_clock_init_data_3_0
882     ps7_ddr_init_data_3_0
883     ps7_peripherals_init_data_3_0
884     puts "PCW Silicon Version : 3.0"
885 }
886
887 # For delay calculation using global timer
888
889 # start timer
890 proc perf_start_clock { } {
891
892     #writing SCU_GLOBAL_TIMER_CONTROL register
893
894     mask_write 0xF8F00208 0x00000109 0x00000009
895 }
896
897 # stop timer and reset timer count regs
898 proc perf_reset_clock { } {
899     perf_disable_clock
900     mask_write 0xF8F00200 0xFFFFFFFF 0x00000000
901     mask_write 0xF8F00204 0xFFFFFFFF 0x00000000
902 }
903
904 # Compute mask for given delay in milliseconds
905 proc get_number_of_cycles_for_delay { delay } {
906
907     # GTC is always clocked at 1/2 of the CPU frequency (CPU_3x2x)
908     # variable APU_FREQ
909     set APU_FREQ 650000000
910     return [ expr ($delay * $APU_FREQ / (2 * 1000)) ]
911 }
912
913
914 # stop timer
915 proc perf_disable_clock {} {
916     mask_write 0xF8F00208 0xFFFFFFFF 0x00000000

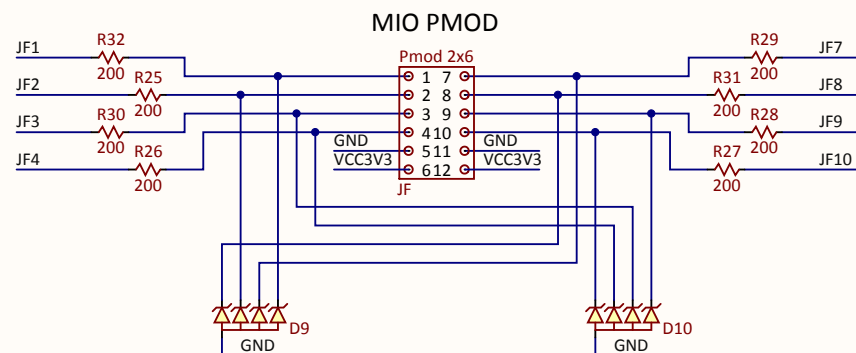
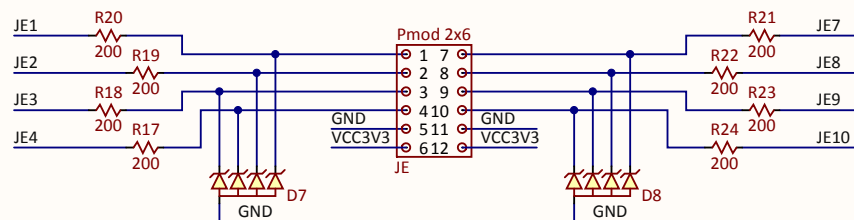
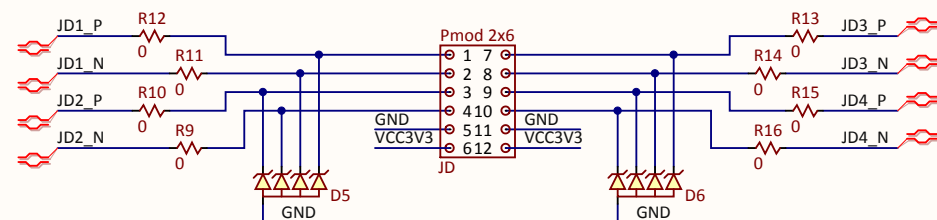
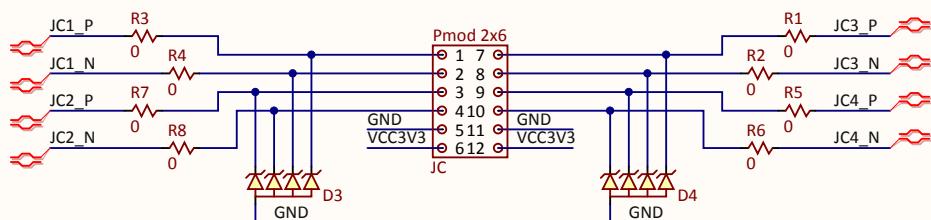
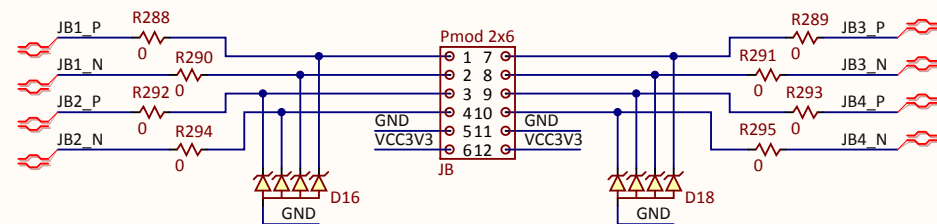
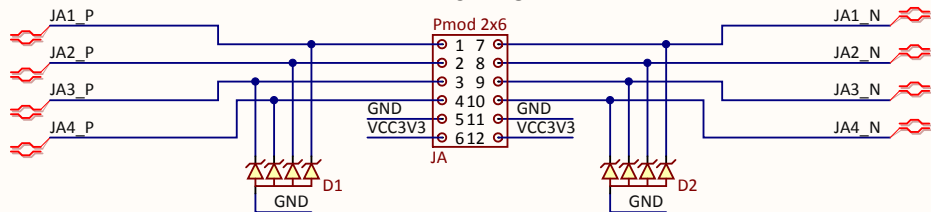
```

```
917 }
918
919 proc perf_reset_and_start_timer {} {
920     perf_reset_clock
921     perf_start_clock
922 }
923
924
925 proc initPS {} {
926     ps7_init
927     initFPU
928     ps7_post_config
929 }
930
931
932 proc initFPU {} {
933     arm mcr 15 0 1 1 2 0x0c00
934     arm mcr 15 0 1 0 2 0x00f00000
935 }
```

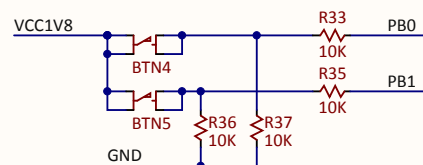
B OpenOCD

B.1 Schema Zybo

XADC PMOD



MIO BUTTON



MIO LED



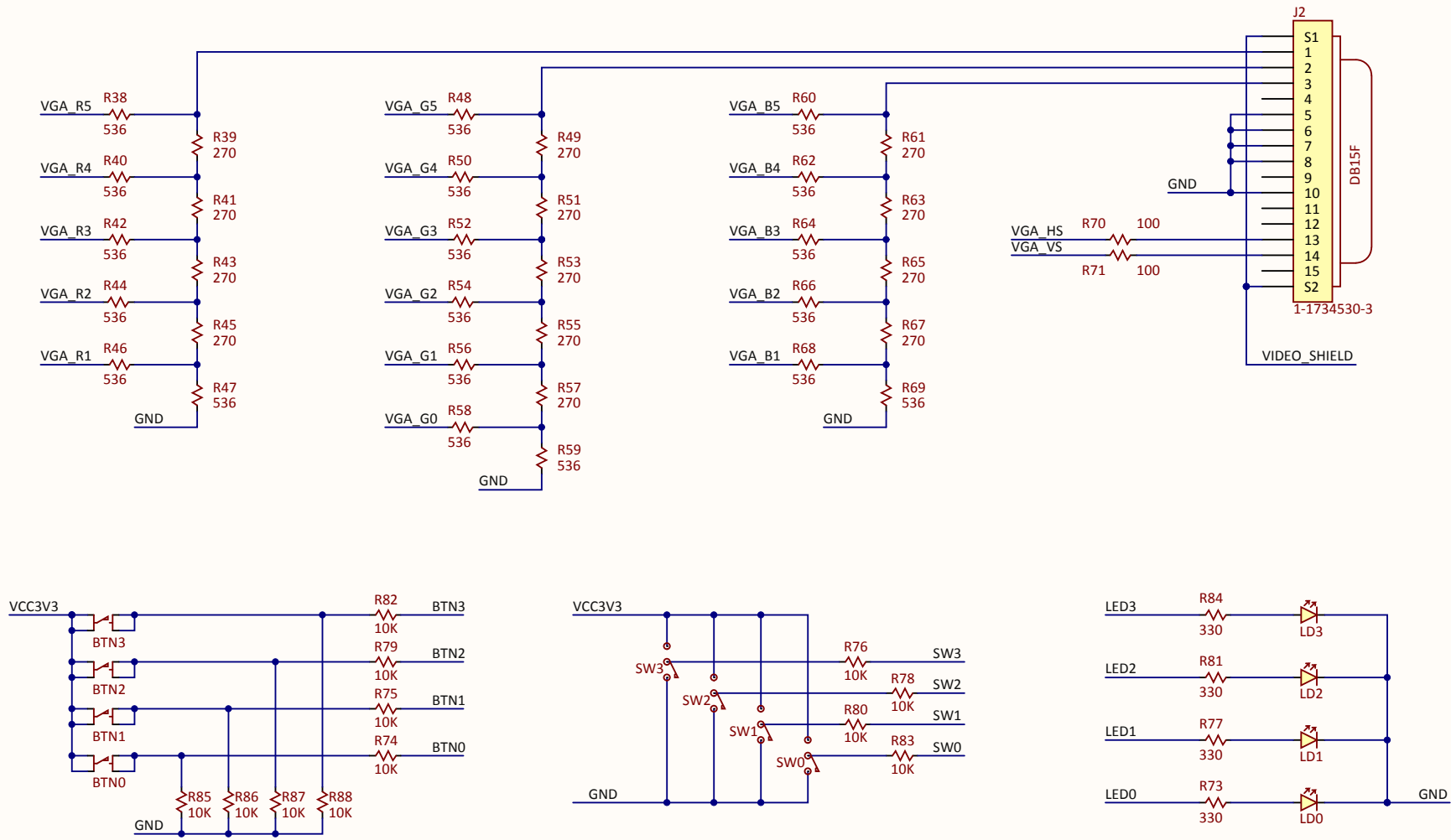
TP
1x1
1
J1
No Load

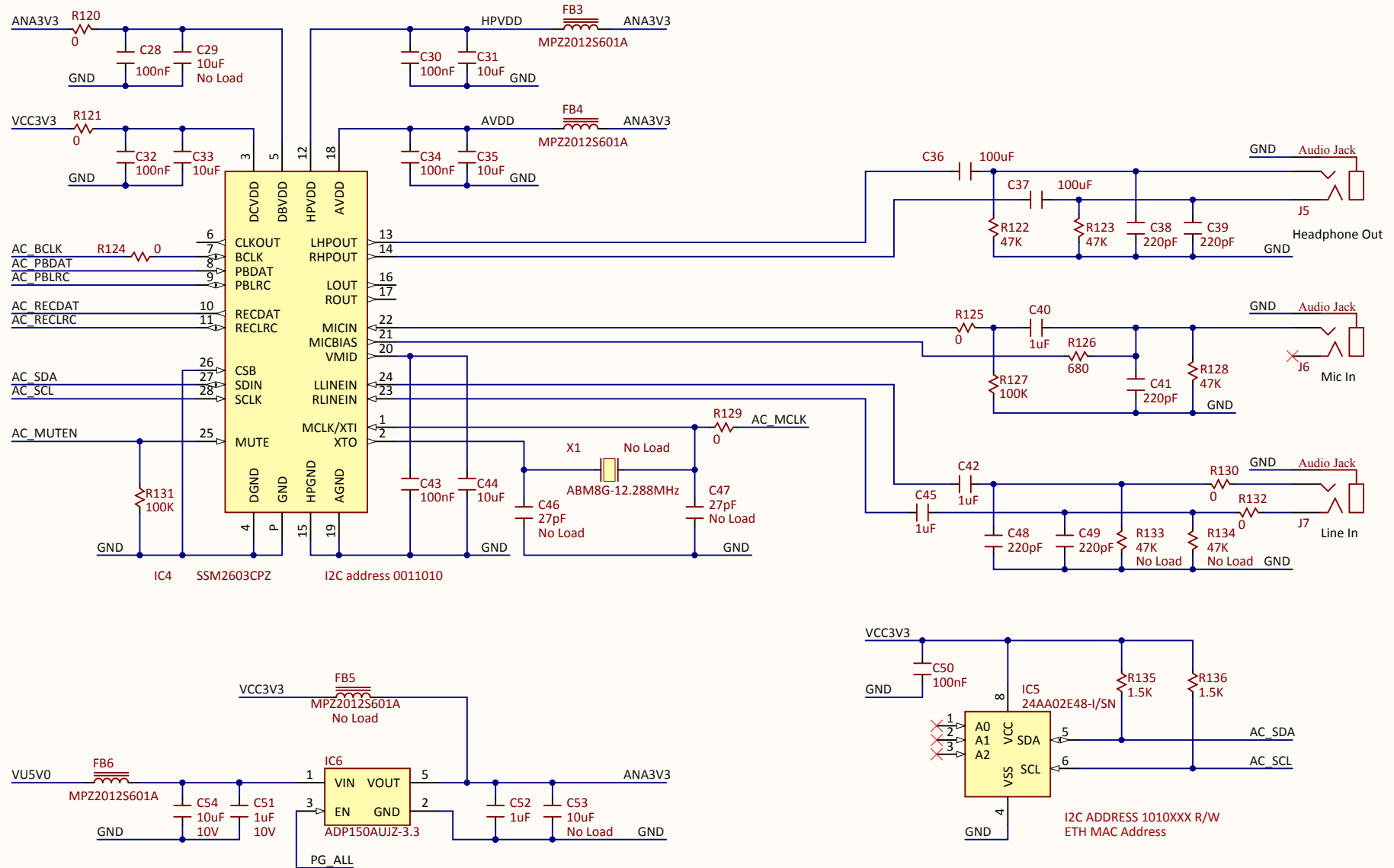
ZYBO Board | Digilent Inc. | Xilinx | Analog Devices | CE | ROHS | Chinese ROHS

Foot
F1
Foot
F2
Foot
F3
Foot
F4

Title ZYBO		Rev B.3 Copyright 2015
Circuit PMODs, MIO		
Doc# 500-279		
Engineer EG		
Author DL		
Date 5/7/2015		
Sheet# 1 out of 13		







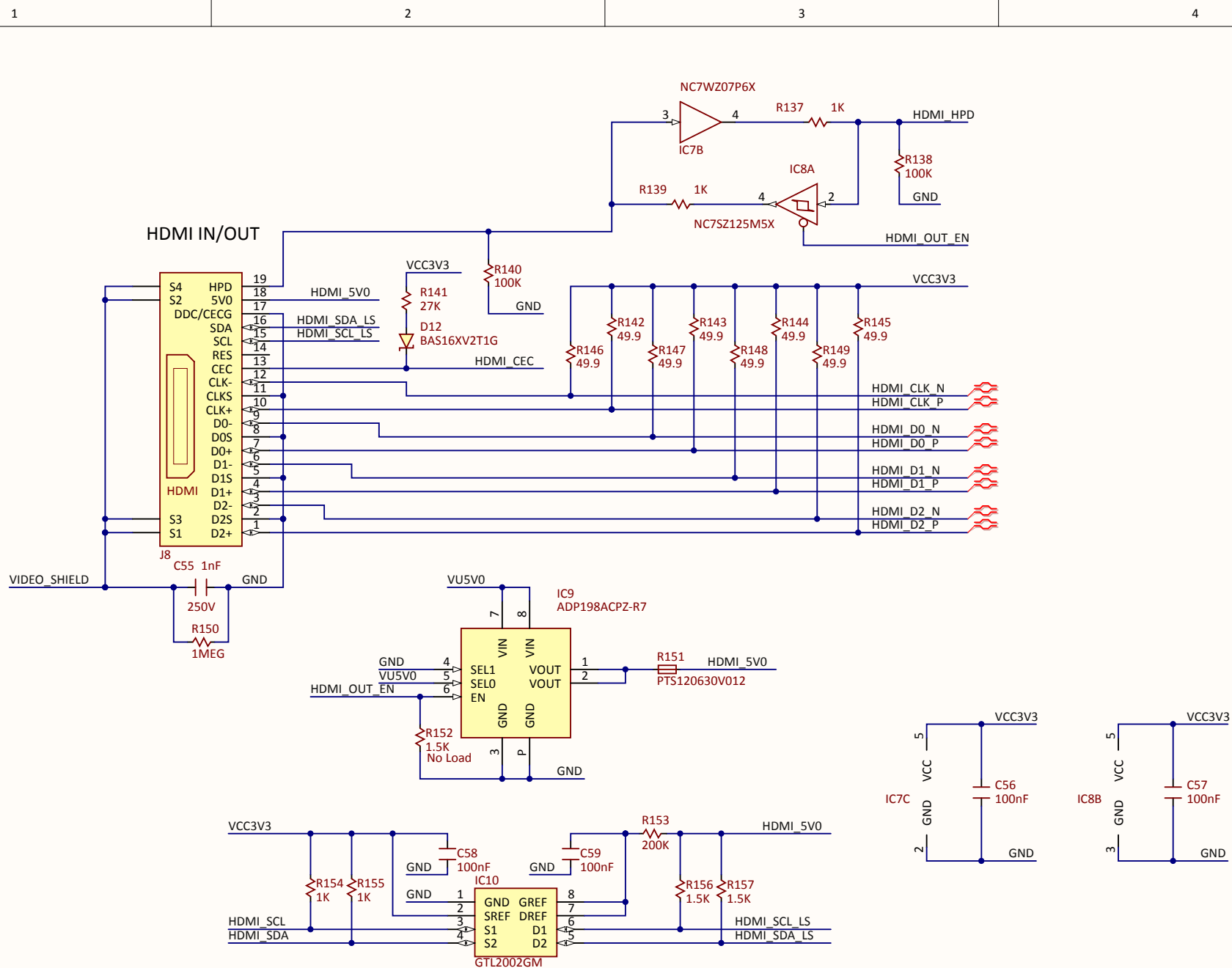
For more information on the parts used in this design, please refer to:

www.analog.com/ssm2603 (Low Power Audio Codec)

www.analog.com/adp150 (Ultra Low Noise, 150 mA CMOS Linear Regulator)


Title		Rev
ZYBO		B.3
Circuit		Audio Codec, EUI EEPROM
Doc#		500-279
Engineer		EG
Author		DL
Date		5/7/2015
Sheet#		4 out of 13

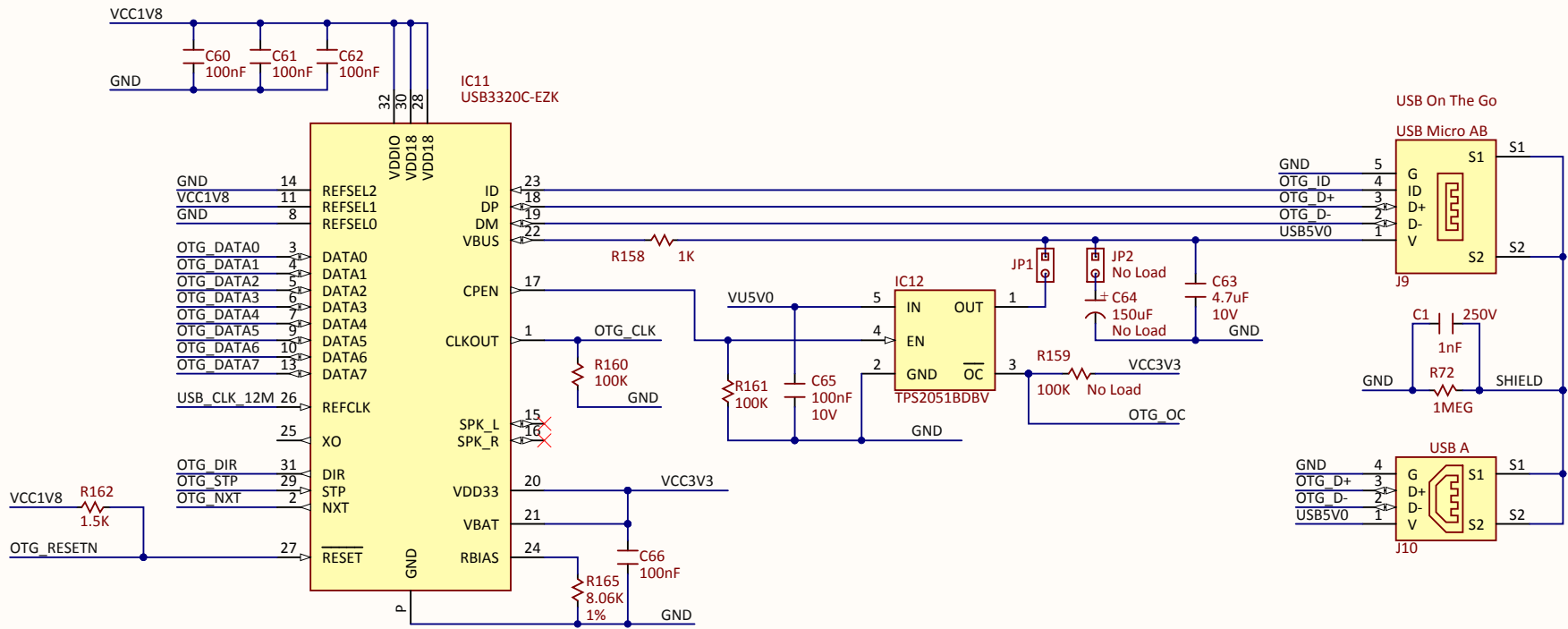




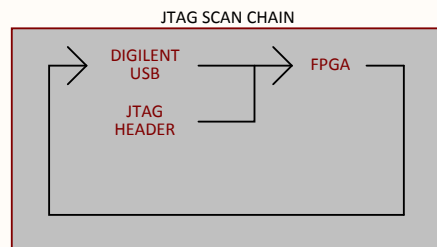
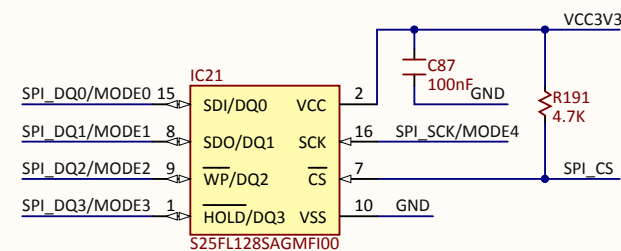
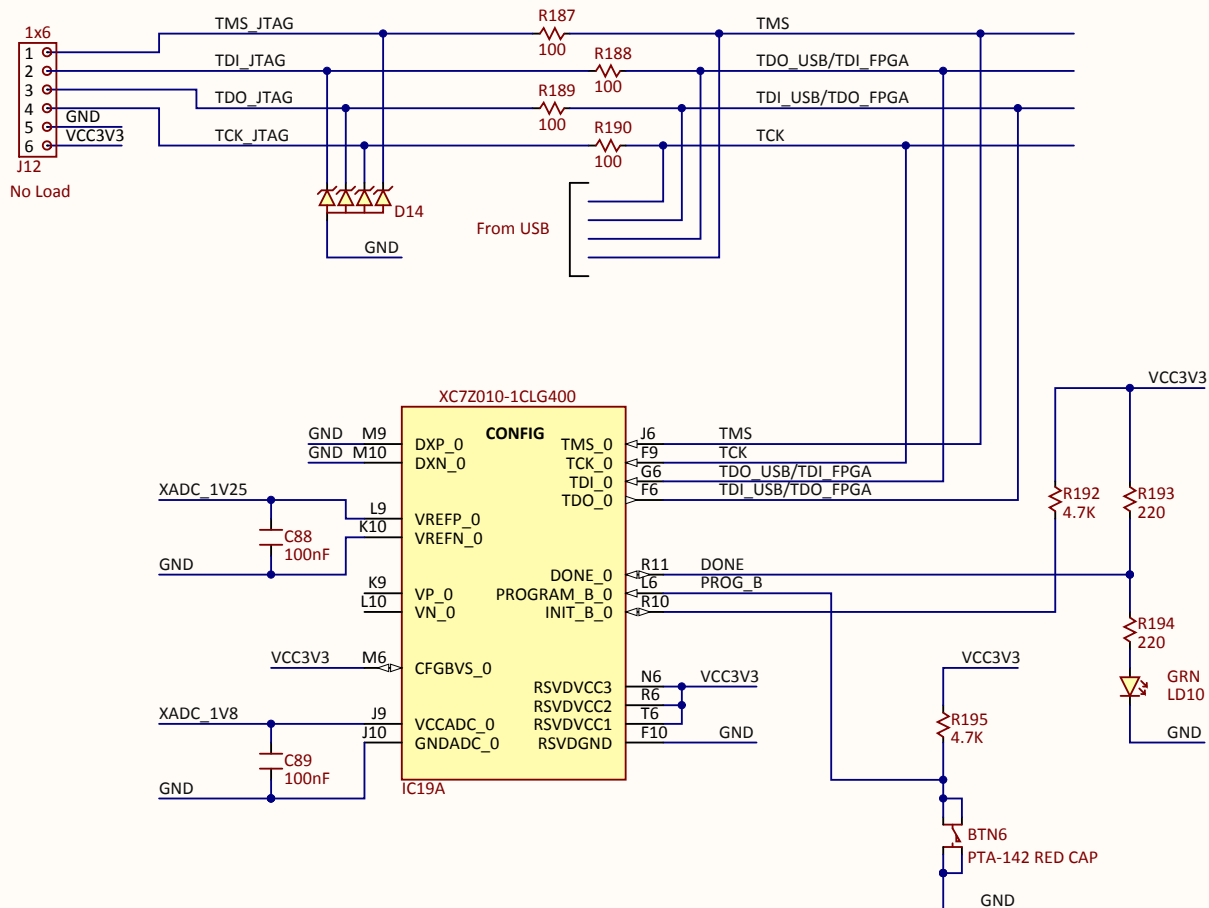
For more information on the parts used in this design, please refer to:
www.analog.com/adp198 (Logic Controlled, 1 A, High-Side Load Switch with Reverse Current Blocking)

Title ZYBO		Rev B.3
Circuit	HDMI	
Doc#	500-279	
Engineer	EG	
Author	DL	
Date	5/7/2015	
Sheet#	5 out of 13	


DIGILENT
 BEYOND THEORY

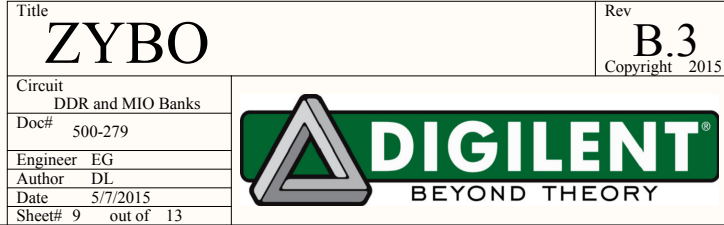


This page intentionally left blank.



Title		Rev
ZYBO		B.3
Circuit		Copyright 2015
FPGA Configuration, Flash		
Doc#		500-279
Engineer		EG
Author		DL
Date		5/7/2015
Sheet#		8 out of 13





VCC3V3

BANK 34

IO_0_34 R19 VGA VS
IO_L1P_T0_34 T11 JC2 P
IO_L1N_T0_34 T10 JC2 N
IO_L2P_T0_34 T12 JC4 P
IO_L2N_T0_34 U12 JC4 N
IO_L3P_T0_DQS_PUDC_B_34 U13 OTG_OC
IO_L3N_T0_DQS_34 V13 JE7
IO_L4P_T0_34 V12 JE1
IO_L4N_T0_34 W13 SW2
IO_L5P_T0_34 T14 JD1 P
IO_L5N_T0_34 T15 JD1 N
IO_L6P_T0_34 P14 JD2 P
IO_L6N_T0_VREF_34 R14 JD2 N
IO_L7P_T1_34 Y16 BTN3
IO_L7N_T1_34 Y17 JE10
IO_L8P_T1_34 W14 JC3 P
IO_L8N_T1_34 Y14 JC3 N
IO_L9P_T1_DQS_34 T16 SW3
IO_L9N_T1_DQS_34 U17 JE8
IO_L10P_T1_34 V15 JC1 P
IO_L10N_T1_34 W15 JC1 N
IO_L11P_T1_SRCC_34 U14 JD3 P
IO_L11N_T1_SRCC_34 U15 JD3 N
IO_L12P_T1_MRCC_34 U18
IO_L12N_T1_MRCC_34 U19
IO_L13P_T2_MRCC_34 N18 AC_SCL
IO_L13N_T2_MRCC_34 P19 VGA_HS
IO_L14P_T2_SRCC_34 N20 VGA_G1
IO_L14N_T2_SRCC_34 P20 VGA_B1
IO_L15P_T2_DQS_34 T20 JB1 P
IO_L15N_T2_DQS_34 U20 JB1 N
IO_L16P_T2_34 V20 JB2 P
IO_L16N_T2_34 W20 JB2 N
IO_L17P_T2_34 Y18 JB3 P
IO_L17N_T2_34 Y19 JB3 N
IO_L18P_T2_34 V16 BTN2
IO_L18N_T2_34 W16 JE2
IO_L19P_T3_34 R16
IO_L19N_T3_VREF_34 R17
IO_L20P_T3_34 T17 JE9
IO_L20N_T3_34 R18 BTNO
IO_L21P_T3_DQS_34 V17 JD4 P
IO_L21N_T3_DQS_34 V18 JD4 N
IO_L22P_T3_34 W18 JB4 P
IO_L22N_T3_34 W19 JB4 N
IO_L23P_T3_34 N17 AC_SDA
IO_L23N_T3_34 P18 AC_MUTEN
IO_L24P_T3_34 P15 SW1
IO_L24N_T3_34 P16 BTN1
IO_25_34 T19 AC_MCLK

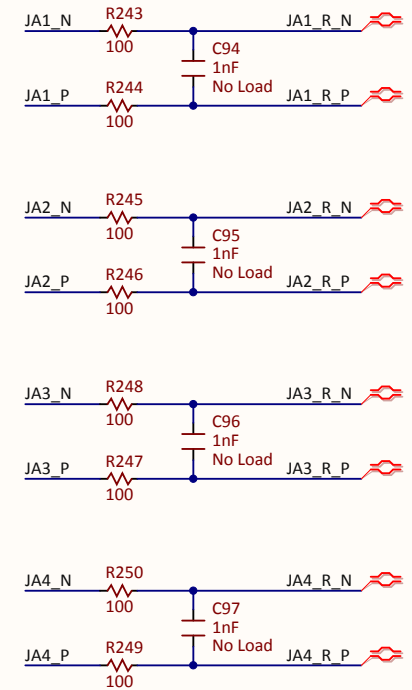
IC19B

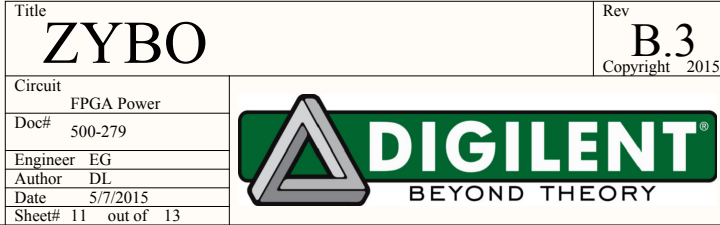
VCC3V3

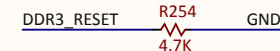
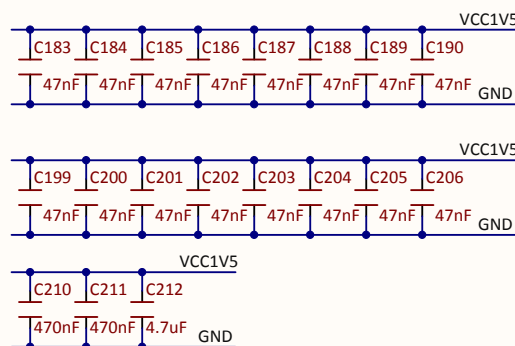
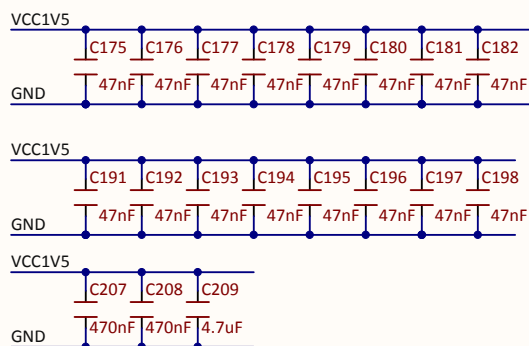
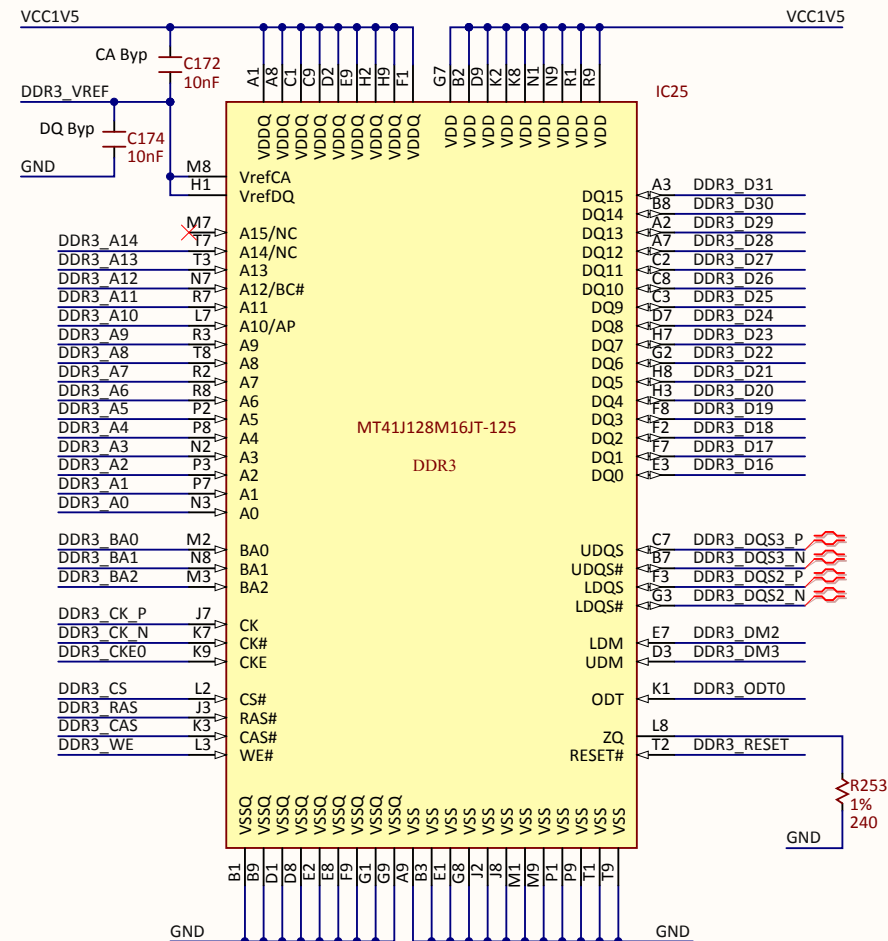
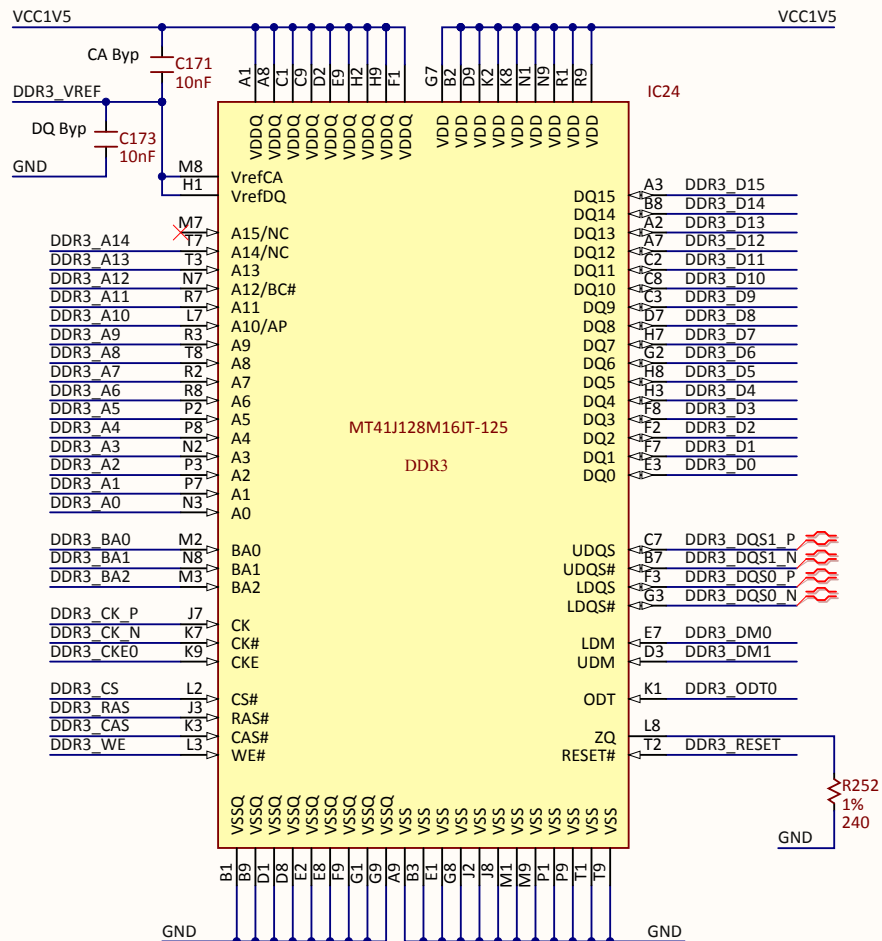
BANK 35

IO_0_35 G14 LED2
IO_L1P_T0_AD0P_35 C20 HDMI_D1_P
IO_L1N_T0_AD0N_35 B20 HDMI_D1_N
IO_L2P_T0_AD8P_35 B19 HDMI_D2_P
IO_L2N_T0_AD8N_35 A20 HDMI_D2_N
IO_L3P_T0_DQS_AD1P_35 E17 ETH_RST_B
IO_L3N_T0_DQS_AD1N_35 D18 LED3
IO_L4P_T0_35 D19 HDMI_D0_P
IO_L4N_T0_35 D20 HDMI_D0_N
IO_L5P_T0_AD9P_35 E18 HDMI_HPD
IO_L5N_T0_AD9N_35 E19 HDMI_CEC
IO_L6P_T0_35 F16 ETH_INT_B
IO_L6N_T0_VREF_35 F17 HDMI_OUT_EN
IO_L7P_T1_AD2P_35 M19 VGA_R1
IO_L7N_T1_AD2N_35 M20 VGA_B2
IO_L8P_T1_AD10P_35 M17 AC_PBDAT
IO_L8N_T1_AD10N_35 M18 AC_RECLRC
IO_L9P_T1_DQS_AD3P_35 L19 VGA_G2
IO_L9N_T1_DQS_AD3N_35 L20 VGA_R2
IO_L10P_T1_AD11P_35 K19 VGA_B3
IO_L10N_T1_AD11N_35 J19 VGA_G3
IO_L11P_T1_SRCC_35 L16 SYSCLK
IO_L11N_T1_SRCC_35 L17 AC_PBLRC
IO_L12P_T1_MRCC_35 K17 AC_RECDAT
IO_L12N_T1_MRCC_35 K18 AC_BCLK
IO_L13P_T2_MRCC_35 H16 HDMI_CLK_P
IO_L13N_T2_MRCC_35 H17 HDMI_CLK_N
IO_L14P_T2_AD4P_SRCC_35 J18 VGA_B4
IO_L14N_T2_AD4N_SRCC_35 H18 VGA_G0
IO_L15P_T2_DQS_AD12P_35 F19 VGA_R5
IO_L15N_T2_DQS_AD12N_35 F20 VGA_G5
IO_L16P_T2_35 G17 HDMI_SCL
IO_L16N_T2_35 G18 HDMI_SDA
IO_L17P_T2_AD5P_35 J20 VGA_R3
IO_L17N_T2_AD5N_35 H20 VGA_G4
IO_L18P_T2_AD13P_35 G19 VGA_B5
IO_L18N_T2_AD13N_35 G20 VGA_R4
IO_L19P_T3_35 H15 JE4
IO_L19N_T3_VREF_35 G15 SW0
IO_L20P_T3_AD6P_35 K14 JA4_R_P
IO_L20N_T3_AD6N_35 J14 JA4_R_N
IO_L21P_T3_DQS_AD14P_35 N15 JA1_R_P
IO_L21N_T3_DQS_AD14N_35 N16 JA1_R_N
IO_L22P_T3_AD7P_35 L14 JA2_R_P
IO_L22N_T3_AD7N_35 L15 JA2_R_N
IO_L23P_T3_35 M14 LED0
IO_L23N_T3_35 M15 LED1
IO_L24P_T3_AD15P_35 K16 JA3_R_P
IO_L24N_T3_AD15N_35 J16 JA3_R_N
IO_25_35 J15 JE3

IC19C







Title		Rev
ZYBO		B.3
Circuit		DDR3 Memory
Doc#		500-279
Engineer		EG
Author		DL
Date		5/7/2015
Sheet#		12 out of 13



B.2 zybo-ftdi.cfg original:

```
1  #
2  # ZYBO ft2232hq usbserial jtag
3  #
4
5  interface ftdi
6  ftdi_device_desc "Digilent Adept USB Device"
7  ftdi_vid_pid 0x0403 0x6010
8
9  ftdi_layout_init 0x3088 0x1f8b
10 #ftdi_layout_signal nTRST -data 0x1000 -oe 0x1000
11 # 0x2000 is reset
12 ftdi_layout_signal nSRST -data 0x3000 -oe 0x1000
13 # green MI07 LED
14 ftdi_layout_signal LED -data 0x0010
15 #ftdi_layout_signal LED -data 0x1000
16
17 reset_config srst_pulls_trst
```

B.3 zybo-ftdi.cfg angepasst:

```
1  #
2  # FTDI2232 on Zybo
3  #
4  # https://github.com/f32c/f32c/blob/master/rtl/proj/xilinx/zybo/ram\_bram\_hdmi\_ise/ftdi-zybo.o cd
5
6  interface ftdi
7  ftdi_device_desc "Digilent Adept USB Device"
8  ftdi_vid_pid 0x0403 0x6010
9
10 #ftdi_layout_init data direction
11 ftdi_layout_init 0x3088 0x1f8b
12
13 ftdi_layout_signal nSRST -data 0x3000 -oe 0x3000
14
15 # green MI07 LED
16 ftdi_layout_signal LED -data 0x0010
17
18 reset_config srst_only
19 adapter_nsrst_delay 40
```

B.4 zynq_7000.cfg:

```
1  #
2  # Xilinx Zynq-7000 All Programmable SoC
3  #
4  # http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/index.htm
5  #
6
7  set _CHIPNAME zynq
8  set _TARGETNAME $_CHIPNAME.cpu
9
10 jtag newtap zynq_pl bs -irlen 6 -ircapture 0x1 -irmask 0x03 \
11     -expected-id 0x23727093 \
12     -expected-id 0x13722093 \
13     -expected-id 0x03727093
14
15 jtag newtap $_CHIPNAME dap -irlen 4 -ircapture 0x1 -irmask 0xf -expected-id
16     0x4ba00477
17
18 target create ${_TARGETNAME}0 cortex_a -chain-position $_CHIPNAME.dap \
19     -coreid 0 -dbgbase 0x80090000
20 target create ${_TARGETNAME}1 cortex_a -chain-position $_CHIPNAME.dap \
21     -coreid 1 -dbgbase 0x80092000
22 target smp ${_TARGETNAME}0 ${_TARGETNAME}1
23
24 adapter_khz 1000
25
26 ${_TARGETNAME}0 configure -event reset-assert-post "cortex_a dbginit"
27 ${_TARGETNAME}1 configure -event reset-assert-post "cortex_a dbginit"
```

B.5 zybo.cfg:

```

1  #
2  # ZYBO board
3  #
4  # https://github.com/emard/wifi\_jtag/blob/master/openocd/scripts/board/zybo.cfg
5
6  set _CHIPNAME zynq
7  set _TARGETNAME $_CHIPNAME.cpu
8
9  jtag newtap chip tap -irlen 6 -ircapture 0x1 -irmask 0x03 \
10     -expected-id 0x23727093 \
11     -expected-id 0x03727093 \
12     -expected-id 0x13722093
13
14  jtag newtap $_CHIPNAME dap -irlen 4 -ircapture 0x1 -irmask 0xf -expected-id
15     0x4ba00477
16
17  target create ${_TARGETNAME}0 cortex_a -chain-position $_CHIPNAME.dap \
18     -coreid 0 -dbgbase 0x80090000
19  target create ${_TARGETNAME}1 cortex_a -chain-position $_CHIPNAME.dap \
20     -coreid 1 -dbgbase 0x80092000
21  target smp ${_TARGETNAME}0 ${_TARGETNAME}1
22
23  adapter_khz 1000
24
25  ${_TARGETNAME}0 configure -event reset-assert-post "cortex_a dbginit"
26  ${_TARGETNAME}1 configure -event reset-assert-post "cortex_a dbginit"
27
28  script ps7_init_modified.tcl
29
30  # http://openocd.org/doc/html/CPU-Configuration.html#targetevents
31  ${_TARGETNAME}0 configure -event reset-init {
32      echo "Running reset init script for Zybo"
33      # Reset script for AT91EB40a
34      # map_OCM_low
35      initPS
36  }
37
38  init
39  scan_chain
40  halt
41
42  #map_OCM_low
43  #initPS

```

B.6 system_debugger_using_debug_01_gettingstarted
_applicationproject.elf_on_local.tcl:

```

1  connect -url tcp:127.0.0.1:3121
2  source D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
3  design_1_wrapper_hw_platform_0/ps7_init.tcl
4  targets -set -nocase -filter {name =~ "APU*" && jtag_cable_name =~ "Diligent
5  Zybo 210279573773A"} -index 0
6  loadhw -hw D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
7  design_1_wrapper_hw_platform_0/system.hdf -mem-ranges [list {0x40000000
8  0xbfffffff}]
9  configparams force-mem-access 1
10 targets -set -nocase -filter {name =~ "APU*" && jtag_cable_name =~ "Diligent
11 Zybo 210279573773A"} -index 0
12 stop
13 ps7_init
14 ps7_post_config
15 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
16 Diligent Zybo 210279573773A"} -index 0
17 rst -processor
18 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
19 Diligent Zybo 210279573773A"} -index 0
20 dow D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/01
21 _gettingStarted_ApplicationProject/Debug/01
22 _gettingStarted_ApplicationProject.elf

```

```
14 configparams force-mem-access 0
15 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
    Digilent Zybo 210279573773A"} -index 0
16 con
```

B.7 CLI-OpenOCD-Toolchain

```
1 #deep-1
2
3 meta {
4     version = "2018-02-28";
5     description = "Programmer description file for use with OpenOCD";
6 }
7
8 programmer openOCD {
9     description = "OpenOCD";
10    pluginid = "ch.ntb.inf.openOCDInterface";
11    classname = "ch.ntb.inf.openOCDInterface.OpenOCD";
12 }
```

C Das ELF-Dateiformat

C.1 loop.java:

```
1 static void reset() {
2
3
4
5     US.PUTGPR(SP, stackBase + stackSize - 4); // set stack pointer
6
7     int x00 = 0;
8     int x01 = 1;
9     int x02 = 2;
10
11     x00++;
12     x01++;
13     x02++;
14
15     int x100 = 100;
16     for(int i=0; i<10; i++){
17         x100 += 10;
18     }
19
20     x100++;
21     x100++;
22     x100++;
23     x100++;
24     x100++;
25
26     US.ASM("b -8"); // stop here
27 }
```

C.2 stabs.include:

```
1 # non-stab symbol types
2 .set N_UNDF, 0x0
3 .set N_EXT, 0x1
4 .set N_ABS, 0x2
5 .set N_TEXT, 0x4
6 .set N_DATA, 0x6
7 .set N_BSS, 0x8
8 .set N_FN_SEQ, 0x0c
9 .set N_INDR, 0x0a
10 .set N_COMM, 0x12
11 .set N_SETA, 0x14
12 .set N_SETT, 0x16
13 .set N_SETD, 0x18
14 .set N_SETB, 0x1a
15 .set N_SETV, 0x1c
16 .set N_WARNING, 0x1e
```

```
17  .set N_FN,      0x1f
18
19  # stab symbol types
20  .set N_GSYM,     0x20
21  .set N_FNAME,    0x22
22  .set N_FUN,      0x24
23  .set N_STSYM,    0x26
24  .set N_LCSYM,    0x28
25  .set N_MAIN,     0x2a
26  .set N_ROSYM,    0x2c
27  .set N_PC,       0x30
28  .set N_NSYSMS,   0x32
29  .set N_NOMAP,    0x34
30  .set N_MAC_DEFINE, 0x36
31  .set N_OBJ,      0x38
32  .set N_MAC_UNDEF, 0x3a
33  .set N_OPT,      0x3c
34  .set N_RSYM,     0x40
35  .set N_M2C,      0x42
36  .set N_SLINE,    0x44
37  .set N_DSLINE,   0x46
38  .set N_BSLINE,   0x48
39  .set N_BROWS,    0x48
40  .set N_DEFD,     0x4a
41  .set N_FLINE,    0x4c
42  .set N_EHDECL,   0x50
43  .set N_MOD2,     0x50
44  .set N_CATCH,    0x54
45  .set N_SSYM,     0x60
46  .set N_ENDM,     0x62
47  #.set N_SO,      0x100
48  .set N_SO,       0x64
49  .set N_LSYM,     0x80
50  .set N_BINCL,    0x82
51  .set N_SOL,      0x84
52  .set N_PSYM,     0xa0
53  .set N_EINCL,    0xa2
54  .set N_ENTRY,    0xa4
55  .set N_LBRAC,    0xc0
56  .set N_EXCL,     0xc2
57  .set N_SCOPE,    0xc4
58  .set N_RBRAC,    0xe0
59  .set N_BCOMM,    0xe2
60  .set N_ECOMM,    0xe4
61  .set N_ECOML,    0xe8
62  .set N_WITH,     0xea
63  .set N_NBTEXT,   0xf0
64  .set N_NBDATA,   0xf2
65  .set N_NBBSS,    0xf4
66  .set N_NBSTS,    0xf6
67  .set N_NBLCS,    0xf8
```

C.3 loopExample.c

```
1  int global = 111;
2
3
4  int c_entry() {
5      int x00 = 0;
6      int x01 = 10;
7      int x02 = 20;
8      x00++;
9      x01++;
10     x02++;
11     register int s=1;
12     float float0=1.1;
13     int int0 = 10;
14     for(int i=0; i<=2; i++) {
15         int0=int0+10;
16     }
17     int0 = int0 + s;
18     int0 = int0 + s;
19     int0 = int0 + s;
20     int0 = int0 + s;
```

```

21
22     while(1);
23     return 0;
24 }

```

C.4 make_loopExample.ps1

```

1  # Add the path for the GNU Arm Embedded Toolchain to the 'Env:Path'
   variable
2  $Env:Path += ";D:\GNUArmEmbeddedToolchain\7-2018-q2-update\bin"
3
4  # Change to directory containing the program
5  cd M:\MA\stabs\cExample
6
7
8  # Compile the C test program with automatic generated stabs
9  # * -c          compile and assemble, but do not link.
10 # * -O0         no optimization
11 # * -march=armv7-a compile for architecture armv7
12 # * -g          compile with debugsymbols
13 # * -gstabs     compile with stabs debug symbols
14 arm-none-eabi-gcc -c -march=armv7-a -O0 -g -gstabs loopExample.c -o
   loopExample.o
15
16 # Disassemble object file again
17 # * --disassemble : disassemble the executable code section
18 # * --disassemble : include all STABS informations
19 arm-none-eabi-objdump -d -G loopExample.o > loopExample.Sd
20
21
22
23 # Build for host
24 gcc -std=c99 -g loopExample.host.c -o loopExample.host.a

```

C.5 loopExample.Sd

```

1
2  loopExample.o:      file format elf32-littlearm
3
4  Contents of .stab section:
5
6  Symnum  n_type  n_othr  n_desc  n_value  n_strx  String
7
8  -1      HdrSym  0      84      000007e4  1
9  0       SO      0      2      00000000  15      loopExample.c
10 1       OPT      0      0      00000000  29      gcc2_compiled.
11 2       LSYM     0      0      00000000  44      int:t(0,1)=r(0,1)
12       ; -2147483648;2147483647;
13 3       LSYM     0      0      00000000  86      char:t(0,2)=r(0,2);0;255;
14 4       LSYM     0      0      00000000  112     long int:t(0,3)=r(0,3)
15       ; -2147483648;2147483647;
16 5       LSYM     0      0      00000000  159     unsigned int:t(0,4)=r(0,4)
17       ;0;4294967295;
18 6       LSYM     0      0      00000000  200     long unsigned int:t(0,5)=r(0,5)
19       ;0;4294967295;
20 7       LSYM     0      0      00000000  246     __int128:t(0,6)=r(0,6);0;-1;
21 8       LSYM     0      0      00000000  275     __int128 unsigned:t(0,7)=r(0,7)
22       ;0;-1;
23 9       LSYM     0      0      00000000  313     long long int:t(0,8)=r(0,8)
24       ; -0;4294967295;
25 10      LSYM     0      0      00000000  356     long long unsigned int:t(0,9)=r
26       (0,9);0;-1;
27 11      LSYM     0      0      00000000  399     short int:t(0,10)=r(0,10)
28       ; -32768;32767;
29 12      LSYM     0      0      00000000  439     short unsigned int:t(0,11)=r
30       (0,11);0;65535;
31 13      LSYM     0      0      00000000  483     signed char:t(0,12)=r(0,12)
32       ; -128;127;
33 14      LSYM     0      0      00000000  521     unsigned char:t(0,13)=r(0,13)
34       ;0;255;
35 15      LSYM     0      0      00000000  558     float:t(0,14)=r(0,1);4;0;
36 16      LSYM     0      0      00000000  584     double:t(0,15)=r(0,1);8;0;
37 17      LSYM     0      0      00000000  611     long double:t(0,16)=r(0,1);8;0;

```

```

27 18      LSYM    0      0      00000000 643      short _Fract:t(0,17)=r(0,1)
      ;1;0;
28 19      LSYM    0      0      00000000 676      _Fract:t(0,18)=r(0,1);2;0;
29 20      LSYM    0      0      00000000 703      long _Fract:t(0,19)=r(0,1);4;0;
30 21      LSYM    0      0      00000000 735      long long _Fract:t(0,20)=r(0,1)
      ;8;0;
31 22      LSYM    0      0      00000000 772      unsigned short _Fract:t(0,21)=r
      (0,1);1;0;
32 23      LSYM    0      0      00000000 814      unsigned _Fract:t(0,22)=r(0,1)
      ;2;0;
33 24      LSYM    0      0      00000000 850      unsigned long _Fract:t(0,23)=r
      (0,1);4;0;
34 25      LSYM    0      0      00000000 891      unsigned long long _Fract:t
      (0,24)=r(0,1);8;0;
35 26      LSYM    0      0      00000000 937      _Sat short _Fract:t(0,25)=r
      (0,1);1;0;
36 27      LSYM    0      0      00000000 975      _Sat _Fract:t(0,26)=r(0,1);2;0;
37 28      LSYM    0      0      00000000 1007     _Sat long _Fract:t(0,27)=r(0,1)
      ;4;0;
38 29      LSYM    0      0      00000000 1044     _Sat long long _Fract:t(0,28)=r
      (0,1);8;0;
39 30      LSYM    0      0      00000000 1086     _Sat unsigned short _Fract:t
      (0,29)=r(0,1);1;0;
40 31      LSYM    0      0      00000000 1133     _Sat unsigned _Fract:t(0,30)=r
      (0,1);2;0;
41 32      LSYM    0      0      00000000 1174     _Sat unsigned long _Fract:t
      (0,31)=r(0,1);4;0;
42 33      LSYM    0      0      00000000 1220     _Sat unsigned long long _Fract:
      t(0,32)=r(0,1);8;0;
43 34      LSYM    0      0      00000000 1271     short _Accum:t(0,33)=r(0,1)
      ;2;0;
44 35      LSYM    0      0      00000000 1304     _Accum:t(0,34)=r(0,1);4;0;
45 36      LSYM    0      0      00000000 1331     long _Accum:t(0,35)=r(0,1);8;0;
46 37      LSYM    0      0      00000000 1363     long long _Accum:t(0,36)=r(0,1)
      ;8;0;
47 38      LSYM    0      0      00000000 1400     unsigned short _Accum:t(0,37)=r
      (0,1);2;0;
48 39      LSYM    0      0      00000000 1442     unsigned _Accum:t(0,38)=r(0,1)
      ;4;0;
49 40      LSYM    0      0      00000000 1478     unsigned long _Accum:t(0,39)=r
      (0,1);8;0;
50 41      LSYM    0      0      00000000 1519     unsigned long long _Accum:t
      (0,40)=r(0,1);8;0;
51 42      LSYM    0      0      00000000 1565     _Sat short _Accum:t(0,41)=r
      (0,1);2;0;
52 43      LSYM    0      0      00000000 1603     _Sat _Accum:t(0,42)=r(0,1);4;0;
53 44      LSYM    0      0      00000000 1635     _Sat long _Accum:t(0,43)=r(0,1)
      ;8;0;
54 45      LSYM    0      0      00000000 1672     _Sat long long _Accum:t(0,44)=r
      (0,1);8;0;
55 46      LSYM    0      0      00000000 1714     _Sat unsigned short _Accum:t
      (0,45)=r(0,1);2;0;
56 47      LSYM    0      0      00000000 1761     _Sat unsigned _Accum:t(0,46)=r
      (0,1);4;0;
57 48      LSYM    0      0      00000000 1802     _Sat unsigned long _Accum:t
      (0,47)=r(0,1);8;0;
58 49      LSYM    0      0      00000000 1848     _Sat unsigned long long _Accum:
      t(0,48)=r(0,1);8;0;
59 50      LSYM    0      0      00000000 1899     void:t(0,49)=(0,49)
60 51      GSYM    0      0      00000000 1919     global:G(0,1)
61 52      FUN     0      0      00000000 1933     c_entry:F(0,1)
62 53      SLINE   0      4      00000000 0
63 54      SLINE   0      5      00000000 c 0
64 55      SLINE   0      6      00000014 0
65 56      SLINE   0      7      0000001c 0
66 57      SLINE   0      8      00000024 0
67 58      SLINE   0      9      00000030 0
68 59      SLINE   0      10     0000003c 0
69 60      SLINE   0      11     00000048 0
70 61      SLINE   0      12     0000004c 0
71 62      SLINE   0      13     00000058 0
72 63      SLINE   0      14     00000060 0
73 64      SLINE   0      15     0000006c 0
74 65      SLINE   0      14     00000078 0
75 66      SLINE   0      14     00000084 0

```



```

76 67 SLINE 0 17 00000090 0
77 68 SLINE 0 18 0000009c 0
78 69 SLINE 0 19 000000a8 0
79 70 SLINE 0 20 000000b4 0
80 71 SLINE 0 22 000000c0 0
81 72 LSYM 0 0 ffffffff0 1948 x00:(0,1)
82 73 LSYM 0 0 fffffffec 1958 x01:(0,1)
83 74 LSYM 0 0 fffffffe8 1968 x02:(0,1)
84 75 RSYM 0 0 00000004 1978 s:r(0,1)
85 76 LSYM 0 0 fffffffe4 1987 float0:(0,14)
86 77 LSYM 0 0 ffffffff8 2001 int0:(0,1)
87 78 LBRAC 0 0 00000000 0
88 79 LSYM 0 0 fffffff4 2012 i:(0,1)
89 80 LBRAC 0 0 00000060 0
90 81 RBRAC 0 0 00000090 0
91 82 RBRAC 0 0 000000c4 0
92 83 SO 0 0 000000c4 0

```

93

94

95 Disassembly of section .text:

96

97 00000000 <c_entry>:

```

98 0: e92d0810 push {r4, fp}
99 4: e28db004 add fp, sp, #4
100 8: e24dd018 sub sp, sp, #24
101 c: e3a03000 mov r3, #0
102 10: e50b3010 str r3, [fp, #-16]
103 14: e3a0300a mov r3, #10
104 18: e50b3014 str r3, [fp, #-20] ; 0xffffffffec
105 1c: e3a03014 mov r3, #20
106 20: e50b3018 str r3, [fp, #-24] ; 0xffffffffe8
107 24: e51b3010 ldr r3, [fp, #-16]
108 28: e2833001 add r3, r3, #1
109 2c: e50b3010 str r3, [fp, #-16]
110 30: e51b3014 ldr r3, [fp, #-20] ; 0xffffffffec
111 34: e2833001 add r3, r3, #1
112 38: e50b3014 str r3, [fp, #-20] ; 0xffffffffec
113 3c: e51b3018 ldr r3, [fp, #-24] ; 0xffffffffe8
114 40: e2833001 add r3, r3, #1
115 44: e50b3018 str r3, [fp, #-24] ; 0xffffffffe8
116 48: e3a04001 mov r4, #1
117 4c: e30c3ccd movw r3, #52429 ; 0xcccd
118 50: e3433f8c movt r3, #16268 ; 0x3f8c
119 54: e50b301c str r3, [fp, #-28] ; 0xffffffffe4
120 58: e3a0300a mov r3, #10
121 5c: e50b3008 str r3, [fp, #-8]
122 60: e3a03000 mov r3, #0
123 64: e50b300c str r3, [fp, #-12]
124 68: ea000005 b 84 <c_entry+0x84>
125 6c: e51b3008 ldr r3, [fp, #-8]
126 70: e283300a add r3, r3, #10
127 74: e50b3008 str r3, [fp, #-8]
128 78: e51b300c ldr r3, [fp, #-12]
129 7c: e2833001 add r3, r3, #1
130 80: e50b300c str r3, [fp, #-12]
131 84: e51b300c ldr r3, [fp, #-12]
132 88: e3530002 cmp r3, #2
133 8c: dafffff6 ble 6c <c_entry+0x6c>
134 90: e51b3008 ldr r3, [fp, #-8]
135 94: e0833004 add r3, r3, r4
136 98: e50b3008 str r3, [fp, #-8]
137 9c: e51b3008 ldr r3, [fp, #-8]
138 a0: e0833004 add r3, r3, r4
139 a4: e50b3008 str r3, [fp, #-8]
140 a8: e51b3008 ldr r3, [fp, #-8]
141 ac: e0833004 add r3, r3, r4
142 b0: e50b3008 str r3, [fp, #-8]
143 b4: e51b3008 ldr r3, [fp, #-8]
144 b8: e0833004 add r3, r3, r4
145 bc: e50b3008 str r3, [fp, #-8]
146 c0: eafffffe b c0 <c_entry+0xc0>

```

C.6 Reset.Java:

```

1  /*
2   * Copyright 2011 - 2013 NTB University of Applied Sciences in Technology
3   * Buchs, Switzerland, http://www.ntb.ch/inf
4   *
5   * Licensed under the Apache License, Version 2.0 (the "License");
6   * you may not use this file except in compliance with the License.
7   * You may obtain a copy of the License at
8   *
9   * http://www.apache.org/licenses/LICENSE-2.0
10  *
11  * Unless required by applicable law or agreed to in writing, software
12  * distributed under the License is distributed on an "AS IS" BASIS,
13  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14  * See the License for the specific language governing permissions and
15  * limitations under the License.
16  *
17  */
18
19 package ch.ntb.inf.deep.runtime.zynq7000;
20 import ch.ntb.inf.deep.runtime.IdeepCompilerConstants;
21 import ch.ntb.inf.deep.runtime.arm32.Iarm32;
22 import ch.ntb.inf.deep.runtime.arm32.ARMEException;
23 import ch.ntb.inf.deep.unsafe.US;
24
25 /* changes:
26  * 13.05.16 NTB/Urs Graf creation
27  */
28 /**
29  * The class for the ARM reset exception.
30  * The stack pointer will be initialized and the program counter will be
31  * set to the beginning of the class initializer of the kernel.
32  *
33  * @author Urs Graf
34  */
35 class Reset extends ARMEException implements Iarm32, Izybo7000,
36         IdeepCompilerConstants {
37     // static int g = 5555;
38
39     static void reset() {
40         int stackOffset = US.GET4(sysTabBaseAddr + stStackOffset);
41         int stackBase = US.GET4(sysTabBaseAddr + stackOffset + 4);
42         int stackSize = US.GET4(sysTabBaseAddr + stackOffset + 8);
43         US.PUTGPR(SP, stackBase + stackSize - 4); // set stack pointer
44
45         int x00 = 0;
46         int x01 = 10;
47         int x02 = 20;
48         x00++;
49         x01++;
50         x02++;
51
52         int x100 = 100;
53         for(int i=0; i<10; i++){
54             x100 += 10;
55         }
56         x100++;
57         x100++;
58         x100++;
59         x100++;
60         x100++;
61
62         US.ASM("b -8"); // stop here
63     }
64 }
65 }

```

C.7 loopMachineCode.txt:

```

1 Code for Method: ch/ntb/inf/deep/runtime/zynq7000/Reset.reset()V
2 E3010004 [0x0] movw R0, #4100
3 E4101000 [0x4] ldr R1, [R0], #-0
4 E2810D40 [0x8] add R0, R1, #4096
5 E2802004 [0xc] add R2, R0, #4

```

```
6      E4120000 [0x10] ldr R0, [R2], #-0
7      E2812D40 [0x14] add R2, R1, #4096
8      E2821008 [0x18] add R1, R2, #8
9      E4112000 [0x1c] ldr R2, [R1], #-0
10     E0801002 [0x20] add R1, R0, R2, 0
11     E2410004 [0x24] sub R0, R1, #4
12     E1A0D000 [0x28] mov R13, R0
13     E3000000 [0x2c] movw R0, #0
14     E300100A [0x30] movw R1, #10
15     E3002014 [0x34] movw R2, #20
16     E2803001 [0x38] add R3, R0, #1
17     E2810001 [0x3c] add R0, R1, #1
18     E2820001 [0x40] add R0, R2, #1
19     E3000064 [0x44] movw R0, #100
20     E3001000 [0x48] movw R1, #0
21     EA000001 [0x4c] b 12, [0x58]
22     E280000A [0x50] add R0, R0, #10
23     E2811001 [0x54] add R1, R1, #1
24     E351000A [0x58] cmp R1, #10
25     BAFFFFFFB [0x5c] b if less -12, [0x50]
26     E2801001 [0x60] add R1, R0, #1
27     E2810001 [0x64] add R0, R1, #1
28     E2801001 [0x68] add R1, R0, #1
29     E2810001 [0x6c] add R0, R1, #1
30     E2801001 [0x70] add R1, R0, #1
31     EAFFFFFFE [0x74] b 0, [0x74]
```

C.8 loop.S:

```
1      .global _start
2
3      .org 0x000000
4      .text
5      Ltext0:
6
7      _start:
8      _reset:
9      c_entry:
10     movw R13, #1024
11
12     movw R0, #0
13     movw R1, #10
14     movw R2, #20
15     add R3, R0, #1
16     add R0, R1, #1
17     add R0, R2, #1
18     movw R0, #100
19     movw R1, #0
20     b CHECK_LOOP_EXIT
21     START_LOOP_BODY:
22     add R0, R0, #10
23     add R1, R1, #1
24     CHECK_LOOP_EXIT:
25     cmp R1, #10
26     blt START_LOOP_BODY
27     add R1, R0, #1
28     add R0, R1, #1
29     add R1, R0, #1
30     add R0, R1, #1
31     add R1, R0, #1
32     END:
33     b END
```

C.9 loopWithAssembler.S:

```
1      .include "stabs.include"
2      .global _start
3      #.stabs "M:/MA/stabs/",N_S0,0,0,Ltext0
4      .stabs "loop.java",N_S0,0,0,Ltext0
5
6      .stabs "char:t(0,2)=r(0,2);0;255;",N_LSYM,0,0,0
7      .stabs "int:t(0,1)=r(0,1);-2147483648;2147483647;",N_LSYM,0,0,0
8      .stabs "long int:t(0,3)=r(0,3);-2147483648;2147483647;",N_LSYM,0,0,0
```

```

9  .stabs "unsigned int:t(0,4)=r(0,4);0;4294967295;",N_LSYM,0,0,0
10 .stabs "long unsigned int:t(0,5)=r(0,5);0;4294967295;",N_LSYM,0,0,0
11 .stabs "__int128:t(0,6)=r(0,6);0;-1;",N_LSYM,0,0,0
12 .stabs "__int128 unsigned:t(0,7)=r(0,7);0;-1;",N_LSYM,0,0,0
13 .stabs "long long int:t(0,8)=r(0,8);-0;4294967295;",N_LSYM,0,0,0
14 .stabs "long long unsigned int:t(0,9)=r(0,9);0;-1;",N_LSYM,0,0,0
15 .stabs "short int:t(0,10)=r(0,10);-32768;32767;",N_LSYM,0,0,0
16 .stabs "short unsigned int:t(0,11)=r(0,11);0;65535;",N_LSYM,0,0,0
17 .stabs "signed char:t(0,12)=r(0,12);-128;127;",N_LSYM,0,0,0
18 .stabs "unsigned char:t(0,13)=r(0,13);0;255;",N_LSYM,0,0,0
19 .stabs "float:t(0,14)=r(0,1);4;0;",N_LSYM,0,0,0
20 .stabs "double:t(0,15)=r(0,1);8;0;",N_LSYM,0,0,0
21 .stabs "long double:t(0,16)=r(0,1);8;0;",N_LSYM,0,0,0
22 .stabs "_Float32:t(0,17)=r(0,1);4;0;",N_LSYM,0,0,0
23 .stabs "_Float64:t(0,18)=r(0,1);8;0;",N_LSYM,0,0,0
24 .stabs "_Float32x:t(0,19)=r(0,1);8;0;",N_LSYM,0,0,0
25 .stabs "short _Fract:t(0,20)=r(0,1);1;0;",N_LSYM,0,0,0
26 .stabs "_Fract:t(0,21)=r(0,1);2;0;",N_LSYM,0,0,0
27 .stabs "long _Fract:t(0,22)=r(0,1);4;0;",N_LSYM,0,0,0
28 .stabs "long long _Fract:t(0,23)=r(0,1);8;0;",N_LSYM,0,0,0
29 .stabs "unsigned short _Fract:t(0,24)=r(0,1);1;0;",N_LSYM,0,0,0
30 .stabs "unsigned _Fract:t(0,25)=r(0,1);2;0;",N_LSYM,0,0,0
31 .stabs "unsigned long _Fract:t(0,26)=r(0,1);4;0;",N_LSYM,0,0,0
32 .stabs "unsigned long long _Fract:t(0,27)=r(0,1);8;0;",N_LSYM,0,0,0
33 .stabs "_Sat short _Fract:t(0,28)=r(0,1);1;0;",N_LSYM,0,0,0
34 .stabs "_Sat _Fract:t(0,29)=r(0,1);2;0;",N_LSYM,0,0,0
35 .stabs "_Sat long _Fract:t(0,30)=r(0,1);4;0;",N_LSYM,0,0,0
36 .stabs "_Sat long long _Fract:t(0,31)=r(0,1);8;0;",N_LSYM,0,0,0
37 .stabs "_Sat unsigned short _Fract:t(0,32)=r(0,1);1;0;",N_LSYM,0,0,0
38 .stabs "_Sat unsigned _Fract:t(0,33)=r(0,1);2;0;",N_LSYM,0,0,0
39 .stabs "_Sat unsigned long _Fract:t(0,34)=r(0,1);4;0;",N_LSYM,0,0,0
40 .stabs "_Sat unsigned long long _Fract:t(0,35)=r(0,1);8;0;",N_LSYM,0,0,0
41 .stabs "short _Accum:t(0,36)=r(0,1);2;0;",N_LSYM,0,0,0
42 .stabs "_Accum:t(0,37)=r(0,1);4;0;",N_LSYM,0,0,0
43 .stabs "long _Accum:t(0,38)=r(0,1);8;0;",N_LSYM,0,0,0
44 .stabs "long long _Accum:t(0,39)=r(0,1);8;0;",N_LSYM,0,0,0
45 .stabs "unsigned short _Accum:t(0,40)=r(0,1);2;0;",N_LSYM,0,0,0
46 .stabs "unsigned _Accum:t(0,41)=r(0,1);4;0;",N_LSYM,0,0,0
47 .stabs "unsigned long _Accum:t(0,42)=r(0,1);8;0;",N_LSYM,0,0,0
48 .stabs "unsigned long long _Accum:t(0,43)=r(0,1);8;0;",N_LSYM,0,0,0
49 .stabs "_Sat short _Accum:t(0,44)=r(0,1);2;0;",N_LSYM,0,0,0
50 .stabs "_Sat _Accum:t(0,45)=r(0,1);4;0;",N_LSYM,0,0,0
51 .stabs "_Sat long _Accum:t(0,46)=r(0,1);8;0;",N_LSYM,0,0,0
52 .stabs "_Sat long long _Accum:t(0,47)=r(0,1);8;0;",N_LSYM,0,0,0
53 .stabs "_Sat unsigned short _Accum:t(0,48)=r(0,1);2;0;",N_LSYM,0,0,0
54 .stabs "_Sat unsigned _Accum:t(0,49)=r(0,1);4;0;",N_LSYM,0,0,0
55 .stabs "_Sat unsigned long _Accum:t(0,50)=r(0,1);8;0;",N_LSYM,0,0,0
56 .stabs "_Sat unsigned long long _Accum:t(0,51)=r(0,1);8;0;",N_LSYM,0,0,0
57 .stabs "void:t(0,52)=(0,52)",N_LSYM,0,0,0
58
59 .stabs "_start:F(0,1)",N_FUN,0,0,_start
60 .stabs "c_entry:F(0,1)",N_FUN,0,0,c_entry
61 #.stabs "reset:F(0,1)",N_FUN,0,0,_reset
62
63
64 #.global reset
65
66 .stabs "int:t2=r2;-2147483648;2147483647;",N_LSYM,0,0,0
67
68
69 .org 0x000000
70 .text
71 Ltext0:
72
73 _start:
74 _reset:
75 c_entry:
76 # US.PUTGPR(SP, 1024); // set stack pointer
77 .stabn N_SLINE, 0, 5, LM5
78 LM5:
79 movw R13, #1024
80
81
82 # int x00 = 0;
83 .stabn N_SLINE, 0, 7, LM7

```

```

84  .stabs "x00:r(0,1)",N_RSYM,0,4,0
85  LM7:
86  movw R0, #0
87  # int x01 = 10;
88  .stabn N_SLINE, 0, 8, LM8
89  .stabs "x01:r(0,1)",N_RSYM,0,4,1
90  LM8:
91  movw R1, #10
92  # int x02 = 20;
93  .stabn N_SLINE, 0, 9, LM9
94  .stabs "x02:r(0,1)",N_RSYM,0,4,2
95  LM9:
96  movw R2, #20
97
98  # x00++;
99  .stabn N_SLINE, 0, 11, LM11
100 .stabn N_LBRAC, 0, 0, LM11
101 .stabs "x00:r(0,1)",N_RSYM,0,4,3
102 LM11:
103 add R3, R0, #1
104 # x01++;
105 .stabn N_SLINE, 0, 12, LM12
106 .stabs "x01:r(0,1)",N_RSYM,0,4,0
107 LM12:
108 add R0, R1, #1
109 # x02++;
110 .stabn N_SLINE, 0, 13, LM13
111 .stabs "x02:r(0,1)",N_RSYM,0,4,0
112 LM13:
113 add R0, R2, #1
114
115 # int x100 = 100;
116 .stabn N_SLINE, 0, 15, LM15
117 .stabs "x100:r(0,1)",N_RSYM,0,4,0
118 LM15:
119 movw R0, #100
120
121
122 .stabn N_RBRAC, 0, 0, LM27
123
124
125 # for(int i=0; i<10; i++){
126 .stabn N_SLINE, 0, 16, LM16
127 .stabs "i:r(0,1)",N_RSYM,0,4,1
128 LM16:
129 movw R1, #0
130 # jump to check loop exit condition
131 b CHECK_LOOP_EXIT
132
133 # x100 += 10;
134 .stabn N_SLINE, 0, 17, LM17
135 LM17:
136 START_LOOP_BODY:
137 add R0, R0, #10
138 # (i++)
139 .stabn N_SLINE, 0, 16, LM16_2
140 LM16_2:
141 add R1, R1, #1
142 # (i<10)
143 .stabn N_SLINE, 0, 16, LM16_3
144 LM16_3:
145 CHECK_LOOP_EXIT:
146 cmp R1, #10
147 # branch if less than 0 to relative position -12
148 blt START_LOOP_BODY
149
150
151 # x100++;
152 .stabn N_SLINE, 0, 20, LM20
153 LM20:
154 add R1, R0, #1
155 # x100++;
156 .stabn N_SLINE, 0, 21, LM21
157 LM21:
158 add R0, R1, #1

```

```
159 # x100++;
160 .stabsn N_SLINE, 0, 22, LM22
161 LM22:
162 add R1, R0, #1
163 # x100++;
164 .stabsn N_SLINE, 0, 23, LM23
165 LM23:
166 add R0, R1, #1
167 # x100++;
168 .stabsn N_SLINE, 0, 24, LM24
169 LM24:
170 add R1, R0, #1
171
172 # US.ASM("b -8"); // stop here
173 .stabsn N_SLINE, 0, 26, LM26
174 LM26:
175 b 0
176
177 LM27:
178
179 .stabs "x03:r(0,1)",N_RSYM,0,4,3
180
181 #.stabs "x00:(0,1)",N_LSYM,0,0,1024
182 #.stabs "x01:(0,1)",N_LSYM,0,0,1024+4
183 #.stabs "x02:(0,1)",N_LSYM,0,0,1024+8
184 .stabs "x100:(0,1)",N_LSYM,0,0,1024+12
185 .stabsn N_LBRAC, 0, 0, LM5
186 .stabsn N_LBRAC, 0, 0, LM16
187 .stabsn N_RBRAC, 0, 0, LM20
188 .stabsn N_RBRAC, 0, 0, LM27+4
189
190 .stabsn N_SO,0,0,LM27+4
```

C.10 make_loop.ps1

```
1 # Add the path for the GNU Arm Embedded Toolchain to the 'Env:Path'
  variable
2 $Env:Path += ";D:\GNUArmEmbeddedToolchain\7-2018-q2-update\bin"
3
4 # Change to directory containing the program
5 cd M:\MA\stabs
6
7
8 $FILENAME="loopWithSTABS"
9 $$FILENAME="loop"
10
11 #arm-none-eabi-as -gstabs -march=armv7-a "$FILENAME.S" -o "$FILENAME.o"
12
13 # Assembles object file
14 # * -march=armv7-a : assemble for ARMv7 architecture
15 arm-none-eabi-as -march=armv7-a "$FILENAME.S" -o "$FILENAME.o"
16
17 # Linking one single object file
18 # * -Ttext=0x0 : text section will be copied to address 0x0 (
  executable code)
19 # * -Tdata=0x1000 : data section will be copied to address 0x100
20 arm-none-eabi-ld -Ttext=0x0 -Tdata=0x100 "$FILENAME.o" -o "$FILENAME"
21
22 # Disassemble linked file again
23 # * --disassemble : disassemble the executable code section
24 # * --disassemble : include all STABS informations
25 arm-none-eabi-objdump --disassemble -G "$FILENAME.o" >"$FILENAME.Sd"
```

D Der gdb-Debugger

D.1 startGdb.ps1

```
1 $Env:Path += ";D:\GNUArmEmbeddedToolchain\6-2017-q2-update\bin"
2 $$Env:Path += ";D:\GNUArmEmbeddedToolchain\7-2018-q2-update\bin"
3
4 cd M:\MA\stabs
5 arm-none-eabi-gdb --command=gdbInit.txt
```

D.2 gdbInit.txt

```
1  set extension-language .java minimal
2  file M:/MA/stabs/loop
3  dir M:/MA/stabs/
4  target remote localhost:3333
5  monitor reset halt
6  #monitor halt
7  load
8  monitor reg pc 0
```