# Inhaltsverzeichnis

# 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 man sich der Entwickler den Eigenarten von C und C++ Programmen auseinandersetzen muss.

*deep* unterstützt einige grundlegende Debuging-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 *Breakpoints* oder *Sourcecode-Navigation*, wie man es aus bekannten Debuggern wie dem *gdb*[2] kennt.

## 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 kostspielig zu ersetzen.

Aus diesem Grund wird *deep* 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. Mit grosser Flexibilität bei der Auswahl von ARM-Prozessoren können sehr günstige oder auch sehr leistungsstarke Prozessoren verwendet werden.

[1] http://www.deepjava.org/start
[2] https://www.gnu.org/software/gdb/

*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. Bei Java muss man sich nicht so detailliert um Ressourcen, wie Speicher und Hardwareschnittstellen kümmern, wie in C-orientierten Sprachen. Dieser Aspekt 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 grundlegendes 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, dass 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.

# 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 auch für den Robotik-Unterricht. Zusätzlich sollte der Prozessor auch leistungsstark und auch 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[2], von Abatron unterstützen.

[1] https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=2ahUKEwjag87kpNbcAhWCM-wKHeEiCkUQFjAAegQIABAC&url=https%3A%2F%2Fwww.arm.com%2F-%2Fmedia%2Farm-com%2Fnews%2FARM-media-fact-sheet-2016.pdf

[2] http://www.abatron.ch/fileadmin/user_upload/news/BDI3000-Brochure.pdf

  - 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.
  - Netzwerkschnitstelle: RJ-45 inklusive MAC[3] und *Magnetics*.
  - 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.

[3] Media Access Control

### 2.1.2 Soll-Kriterien

- Systemebene
  - Einfach einbettbar: Der Prozessor ist als Prozessormodul erhältlich, so dass das Design von einem selbst entwickelten PCB einfacher wird.
  - Günstiger Hardwaredebugger: Der Hardwaredebugger kann auch für 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 Hardware Debugger

Der Begriff *Hardware Debugger* ist nicht eindeutig definiert. Im einfachsten Fall kann ein Hardware Debugger nur ein *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 so einem Scan kann in der Produktion bei der 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 grundlegende 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

### 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 Festplattenkontrolle 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 Familien 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 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.

Tabelle 2.1: Übersicht ARM Mikroarchitekturen

| | Vorteile | Nachteile |
|---|---|---|
| **A** | * Sehr leistungsstark<br>* Support für vollwertige Betriebssysteme<br>* Grosse Variation erhältlich (energiesparend / sehr leistungsstark)<br>* Reichhaltiger Funktionsumfang<br>* NEON und FPU Unterstützung | * Langsamer Context-Switch<br>* Relativ hoher Stromverbrauch<br>* Relativ teuer<br>* Mit GPU erhältlich<br>* Keine DSP Unterstützung<br>* Keine HW-Division |
| **R** | * Sehr gut geeignet für Echtzeitanwendungen<br>* Sehr schneller Context-Switch<br>* DSP Unterstützung | * Kleiner Funktionsumfang<br>* Nicht so leistungstark wie Cortex A<br>* Keine Linux-Unterstützung |
| **M** | * Sehr schneller Context-Switch<br>* Sehr energiesparend<br>* DSP-Unterstützung | * Geringe Rechenleistung<br>* Keine Linux-Unterstützung<br>* Unterstützt nur Thumb-Instruktionen |

## 2.4 Anbindung des FPGAs

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 **??** zusammengefasst. Bild 2.1 gibt eine schematische Übersicht über die verschiedenen Bauarten.

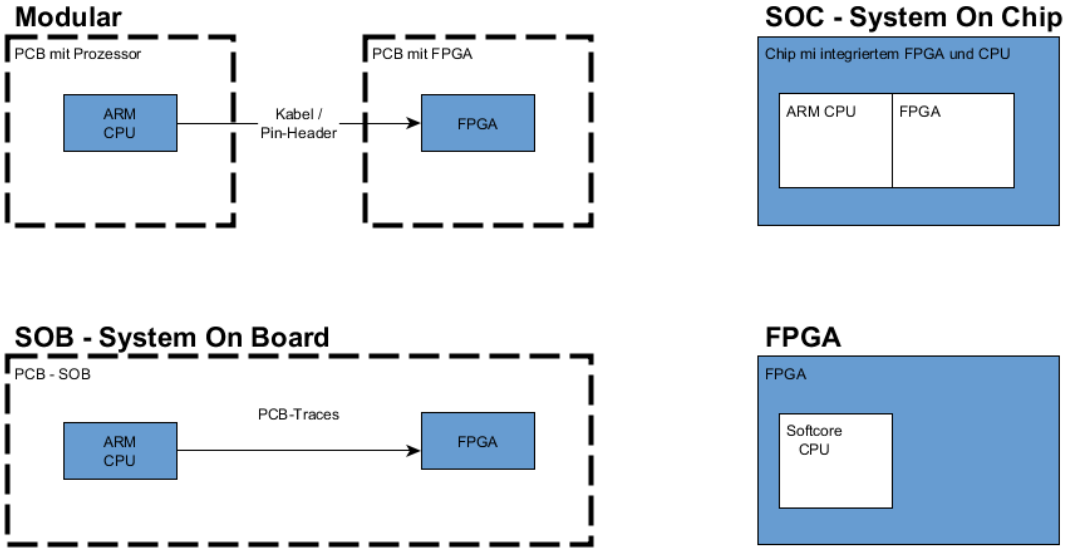


Abbildung 2.1: Mögliche Anbindungen des FPGA an die CPU

### 2.4.1 FPGA als Zusatzplatine zum Prozessorboard - Bauweise "Modular"

Das *"FPGA Development Board CAPE for the BEAGLEBONE"*[4] 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 (General-Purpose Memory Controller) Interface (bis zu 70 MB/s) 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.

*Fazit - Bauweise "Modular"*

[4] https://www.element14.com/community/docs/DOC-69215/l/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 verbaut. Dadurch kann der Hersteller auf dem Modul ein Bus mit kontrollierter Impedanz implementieren. Dies ermöglicht sehr hohe Bandbreite bei der Kommunikation zwischen der CPU und dem FGPA möglich. 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 so ein 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 nur eine Firma ein SOM mit FPGA produziert. Die Firma XXX verkauft ein Module mit einem XXX Prozessor und einem XXX FPGA.

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. x

Tabelle 2.2: Übersicht Bauformen

| Bauweise | Vorteile | Nachteile |
|---|---|---|
| **Modular** | * Günstig wenn nur Prozessor verwendet wird<br>* 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<br>* Sauberes, abgeschlossenes System | * FPGA ist fix<br>* Relativ teuer |
| **FPGA** | * Flexibel | * Sehr teuer |

### 2.4.4 ARM als Softcore in FPGA - Bauweise "FPGA"

#### STM23

STM

# 3 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 man mit dem 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 notwendige Funktionen, die für die Entwicklung von *deep*-Applikationen notwendig sind. Es sind beide Debug Toolchains, die ”klassische” Abatron-Toolchain und die neue OpenOCD-Toolchain, in dieser Übersicht 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 den selben, 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).

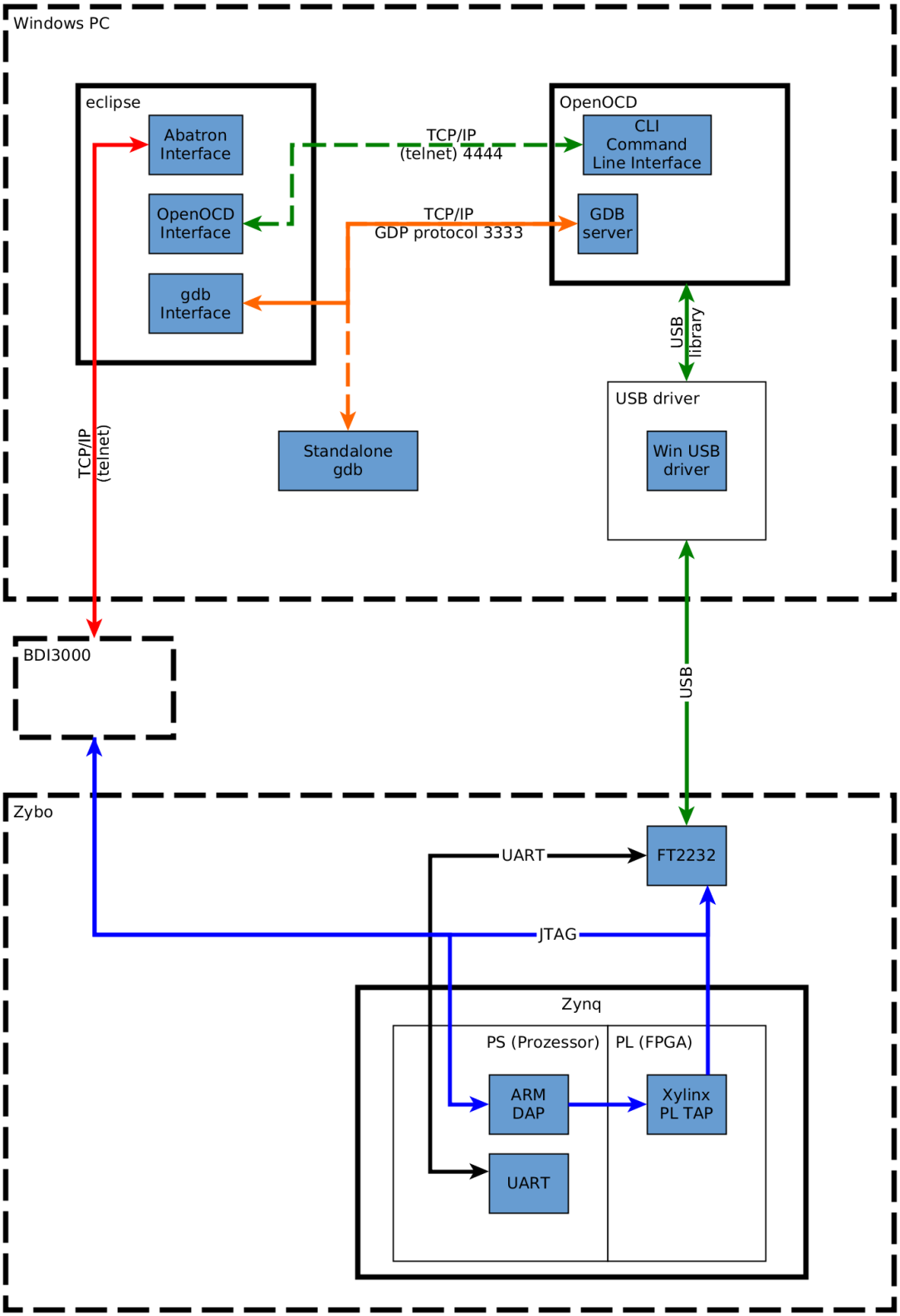


Abbildung 3.1: Systemübersicht Debugger Toolchain

## 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 3.2 nicht verwendet.

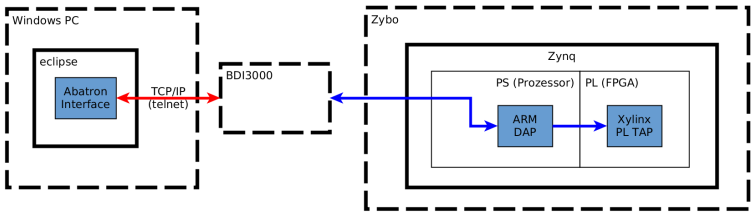


Abbildung 3.2: Abatron-Toolchain

### 3.2.2 CLI-OpenOCD-Toolchain

Wie in der Abbildung 3.3 zu sehen ist, wird das teure BDI wird 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 setzten oder durch einzelne Zeilen im Sourcecode zu steppen wie man es von Debuggern, wie dem *gdb*, gewohnt ist. Bestehende Möglichkeiten aus der alten *Abatron-Toolchain*, wie *Target Commands*, bleiben aber erhalten.

Im Kapitel 5.4 wird die Implementation dieser Toolchain genauer beschrieben.

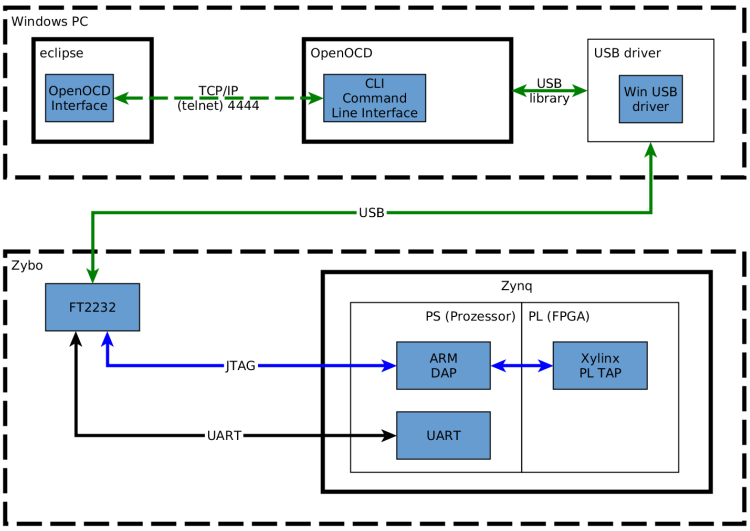


Abbildung 3.3: CLI-OpenOCD-Toolchain

### 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 der *gdb*-Debugger. In der schematische 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-Featurers 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 der *gdb* im Zusammenhang mit der *OpenOCD-Toolchain*. Die komplette *gdb-OpenOCD-Toolchain* kann auf dieser Toolchain aufbauen.

Im Kapitel 7 wird diese Toolchain detailliert beschrieben.

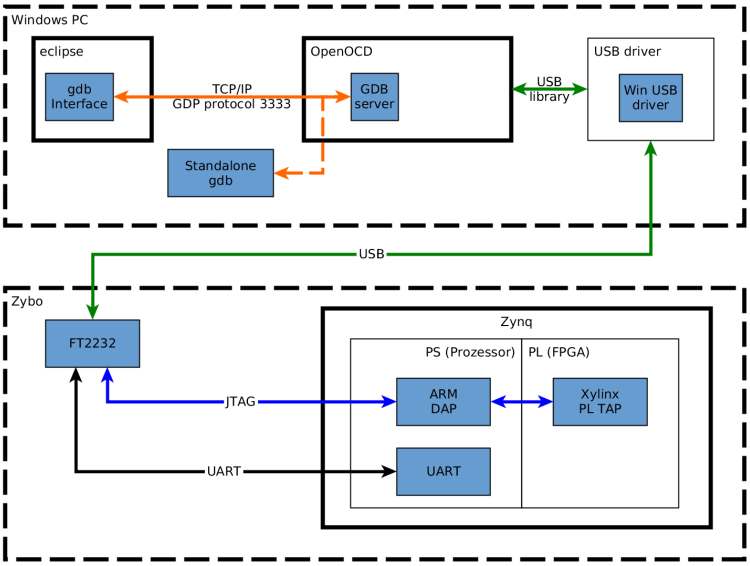


Abbildung 3.4: *gdb*-OpenOCD-Toolchain

# 4 Zynq

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 Komponente und Eigenarten des Zynq.

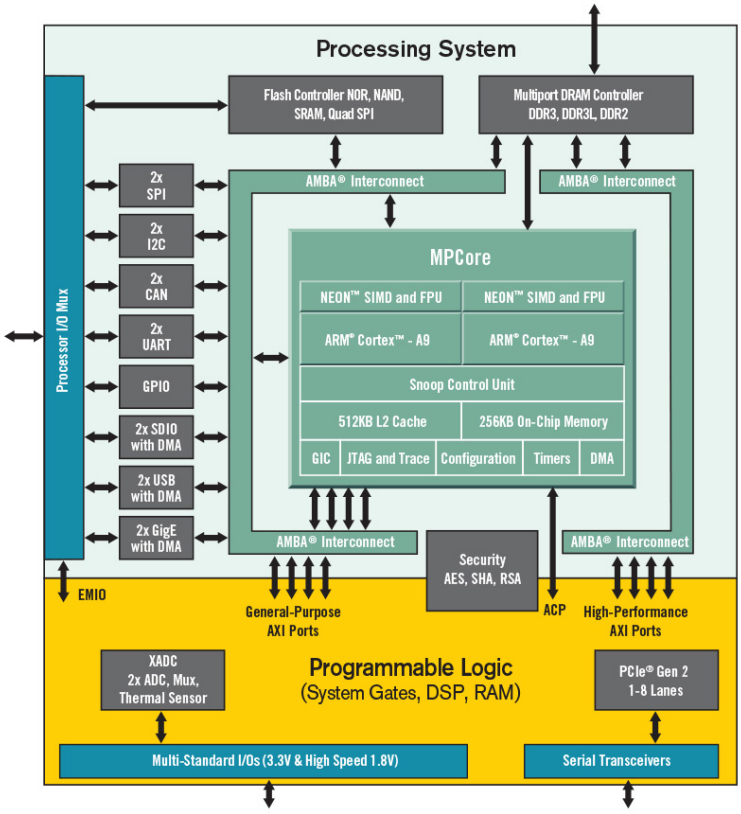


Abbildung 4.1: Block Diagramm Zynq7000[1]

[1] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html

## 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[2] des Zynq[**?**] im Kapitel *"2.5.4 MIO-at-a-Glance Table"* ist eine sehr gute Übersicht über alle möglichen Funktionen der MIOs gegeben.

## 4.2 Standard Zybo Workflow

Im *Getting Started with Zynq*[3] Tutorial von Digilent ist beschrieben, wie man ein einfaches Design für die PL und ein einfaches Programm für das PS erstellt. 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 XSCT[4] (*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.

[2] Technical Reference Manual
[3] https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-getting-started-with-zynq/start?redirect=1
[4] https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/xsct/intro/xsct_introduction.html

Im Anhang **??** ist eine Kopie eines solchen Logs zu finden. *D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/.sdk/launch_s*

*c++_application_(system_debugger)/system_debugger_using_debug_01_gettingstarted_applicationproject.elf_on_local.tc*

Das Script *ps7_init.tcl* definiert unter anderem die fünf Initialisierungs-Methoden:

- *ps7_mio_init_data_3_0*
- *ps7_pll_init_data_3_0*
- *ps7_clock_init_data_3_0*
- *ps7_ddr_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. In Zeile 9 vom *...elf_on_local.tcl* wird auch die Methode *ps7_post_config* von *ps7_init.tcl* aufgerufen, welche im Anschluss *ps7_post_config_3_0* aufruft.

Alle Konfigurationsregister sind im Anhang B vom *Zynq TRM*[**?**] beschrieben. Bevor die Register aber verändert werden können, müssen sie *”unlocked”* werden, in dem 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 Addresse <address>.

**mask_poll <address> <mask>:**

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

**mask_dellay <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 oder fast CMOS edge.
- **Tristate:** Enalbe / disable Tristate.

**ps7_pll_init_data_3_0**

Initialisiert die drei PLLs[5] 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.

[5] Phase Locked Loop

**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.

**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 jedesmal ausgeführt, wenn die Applikation im XSDK mit *"Launch on Hardware (System Debuger)"* 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. Vergleicht man *"ps7_init()"* mit *ps7_init.tcl*, dann sieht man schnell, 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.

*helloworld.c:*

```
1 ...
2 #include "platform.h"
3 ..
4 int main ()
5 {
```

```
...
init_platform();

while(1){
...
```

*platform.c:*

```
...
/*#include "ps7_init.h"*/
/*#include "psu_init.h"*/
...
void
init_platform()
{
    /*
     * If you want to run this example outside of SDK,
     * uncomment one of the following two lines and also #include "ps7_init
         .h"
     * or #include "ps7_init.h" at the top, depending on the target.
     * Make sure that the ps7/psu_init.c and ps7/psu_init.h files are
         included
     * along with this example source files for compilation.
     */
    /* ps7_init();*/
    /* psu_init();*/
    enable_caches();
    init_uart();
}
...
```

### 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.

*ps7_init_modified.tcl:*

```
proc unlock_SLCR {} {
  mww 0xF8000008 0x0000DF0D
}

proc map_OCM_low {} {
  unlock_SLCR
  mww 0xF8000910 0x00000010
}

proc memread32 {ADDR} {
    set foo(0) 0
    if ![ catch { mem2array foo 32 $ADDR 1  } msg ] {
  return $foo(0)
    } else {
  error "memread32: $msg"
    }
}

proc mask_write { addr mask val } {
  set curval [memread32 $addr]
  set maskinv [expr {0xffffffff ^ $mask}]
    set maskedcur [expr {$maskinv & $curval}]
  set maskedval [expr {$mask  & $val}]
    set newval [expr $maskedcur | $maskedval]
  mww $addr $newval
}

proc initPS {} {
  ps7_init
  ps7_post_config
}
```

2. Jeder ”`mwr -force <address> <value>`” Befehl wurde mit ”`mww <address> <value>`” ersetzt.

3. Folgende Methoden wurden mit den untenstehenden Implementationen ersetzt:

*ps7_init_modified.tcl:*

```
proc mask_poll { addr mask } {
    set count 1
    % set curval [memread32 $addr]
    (*@  \textcolor{blue}{ set curval [memread32 $addr] }  @*)
    set maskedval [expr {$curval & $mask}] # & = bitwise AND
    while { $maskedval == 0 } {
    set curval [memread32 $addr]
        set maskedval [expr {$curval & $mask}]
        set count [ expr { $count + 1 } ]
        if { $count == 100000000 } {
          puts "Timeout Reached. Mask poll failed at ADDRESS: $addr
              MASK: $mask"
          break
        }
    }
}

proc mask_delay { addr val } {
    set delay  [ get_number_of_cycles_for_delay $val ]
    perf_reset_and_start_timer
    set curval [memread32 $addr]
    set maskedval [expr {$curval < $delay}]
    while { $maskedval == 1 } {
        set curval [memread32 $addr]
        set maskedval [expr {$curval < $delay}]
    }
    perf_reset_clock
}

proc ps7_post_config {} {
        ps7_post_config_3_0
}

proc ps7_init {} {
  halt
```

```
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*[**?**] 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 DDR RAM verwendet werden kann.

Der OCM (*On Chip Memory*) ist ein kleiner Speicher im Zynq der ohne Initialisierung verwendet werden kann. Ideal für ein 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[**?**] im Kapitel *"Kapitel 4.4: Boot and Configuration"*.

| 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 FDFF_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

## 4.4 Floating Point Unit

FPUs (*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 Satus and Control Register*) gesetzt werden. Das FPEXC Register kann aber nicht mit dem Debugger direkt gesetzt werden, da eine spezielle ARM Instruktion dafür verwendet werden muss. Im Kapitel *"2.4.2 Accessing the FPU registers* des FPU-TRM[**?**] sind die Details 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. `mww 0x0 0xEEF70A10` // Speichert die Instruktion ''`VMRS R0, MVFR0`'' in den OCM.
6. `mww 0x4 0xEEF61A10` // 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 dass Register 0 aus, welches eine Kopie des MVFR0 enthält.
10. `reg 1` // Liest dass 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[**?**] (*Architecture Reference Manual*) ist beschrieben, wie die unterstützten Features aus den Registern gelesen werden können.

Der Zynq des Zybo unterstützt:

- All rounding modes
- VFP squarde root operations
- VFP divide operations
- Full VFP douple-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 OpenOCD

OpenOCD[1] 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. Solch ein generischer Adapter ist in der Abbildung 5.1 zu sehen.

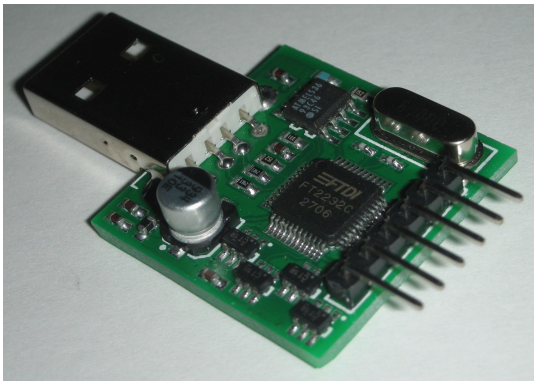

Abbildung 5.1: Generischer JTAG Adapter mit einem FTDI FT2232[2]

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.

[1] http://openocd.org/about/
[2] https://www.ebay.com/itm/FPU1-FTDI-FT2232-USB-JTAG-XILINX-FPGA-CPLD-programmer-cable-/181635528314 Seite 5

## 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 werden[3] oder es können vorkompilierte Binaries verwendet werden. Für diese Arbeit wurde das vorkompilierte Windows Binaries[4] 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[**?**] befindet sich im Ordner:

`/openocd-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.

[3] http://sourceforge.net/p/openocd/code/
[4] http://www.freddiechopin.info/en/download/category/4-openocd?download=154%3Aopenocd-0.10.0
[5] http://visualgdb.com/UsbDriverTool/

Um den Standardtreiber wieder zu installieren, kann einfach *"Restore default driver"* ausgewählt werden. Nachdem das Zybo einmal aus- und wieder einschaltet wird, 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.

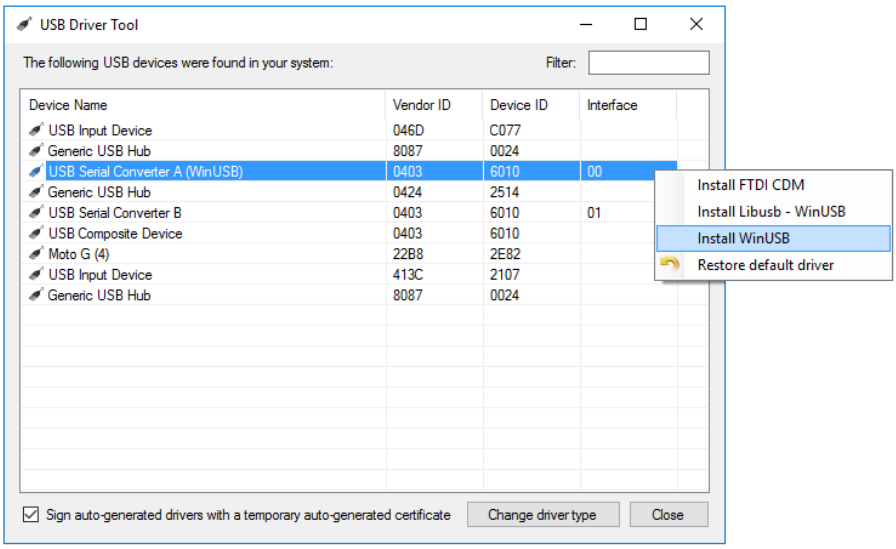


Abbildung 5.2: Installation des *WinUSB* Treibers mit dem *USB Driver Tool*

## 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].

Normalerweise werden die Scripts in die drei Gruppen *interface, board* und *target* aufgeteilt. So kann einfach ein Script ausgewechselt werden, wenn man den gleichen Adapter aber einen anderen Prozessor verwenden 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
```

### 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 passender Script unter *openocd-0.10.0/scripts/interface/ftdi/* zu finden sein. Keiner der Scripts passt vom Namen her auf *Zybo* oder *FT2232*. Eine Google Suche nach einem passenden Script war erfolgreicher. Ein Github User mit dem Namen *emard* hat folgenden Script in einem von seinen Repositories[8] 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
```

[6] http://jim.tcl.tk/index.html/doc/www/www/index.html
[7] http://www.tcl.tk
[8] https://github.com/f32c/f32c/blob/master/rtl/proj/xilinx/zybo/xram_bram_hdmi_ise/zybo.ocd

```
11 # 0x2000 is reset
12 ftdi_layout_signal nSRST -data 0x3000 -oe 0x1000
13 # green MIO7 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

Liest man aus einer unerlaubten Speicheradresse (`CLI: mdw 0x40000000`), 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 Homepage[9] 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 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.

[9] https://www.xilinx.com/support/answers/63871.html

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[**?**] 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 Zybos (siehe Anhang **??**) könnte man überprüfen, 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[**?**] 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 Konfigurationsscript 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 Repositorie 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 so einen ähnlichen Befehl ist die `wirteWord()`-Methode:

```
1 // OpenOCDInterface:
2 out.write(("mww 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 Methode 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 **??** angehängt.

# 6 Das ELF-Dateiformat

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 Debug-Informationen 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[**?**] 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

[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
[2] https://slideplayer.com/slide/6444592/

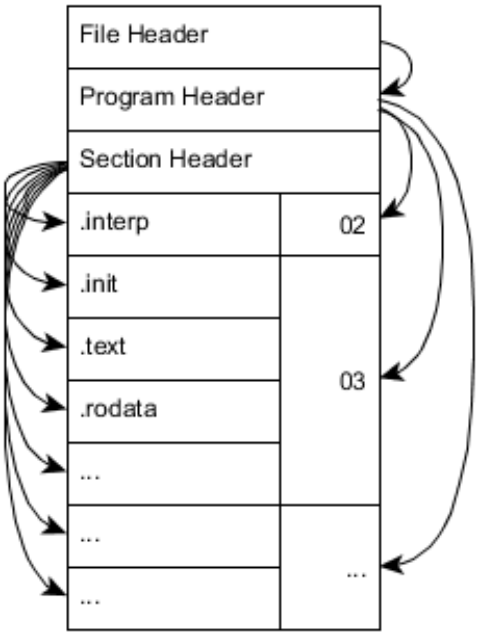


Abbildung 6.1: Der Aufbau von einer ELF Datei[2]

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 der Programmcode, die Variablen und die Konstanten kopiert werden sollen. Ein Segment beinhaltet ein oder mehrere *Sections*.

Im *Section Header* sind alle *Sections* beschrieben. Mit ''`readelf filename -WS`'' kann man sehen, 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 *Secitons*, 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

STABS ist ein Datenformat für Debug-Informationen. Die Informationen sind als Strings in ***S**ymbol **TA**ble **S**trings* 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 Debug-Informationen 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.

### 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 **??**) 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
```

[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

### 6.3.3 DWARF

## 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. **Source 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.

### 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.ASM("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 **??** 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 **??** 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 **??**. Die Tabelle 6.1 beschreibt die Direktive 0 im Detail.

```
1  Symnum n_type n_othr n_desc n_value  n_strx String
2  ...
3  0      SO     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      0000000c 0
11 ...
12 72     LSYM   0      0      fffffff0 1948   x00:(0,1)
13 73     LSYM   0      0      ffffffec 1958   x01:(0,1)
14 74     LSYM   0      0      ffffffe8 1968   x02:(0,1)
15 75     RSYM   0      0      00000004 1978   s:r(0,1)
16 76     LSYM   0      0      ffffffe4 1987   float0:(0,14)
17 77     LSYM   0      0      fffffff8 2001   int0:(0,1)
18 78     LBRAC  0      0      00000000 0
19 79     LSYM   0      0      fffffff4 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

| | | |
|---|---|---|
| *Symnum* | 0 | Eindeutige Identifikation der STAB-Direktive |
| *n_type* | S0 | Typ der STAB-Direktive. Die SO-Direktive beschreibt das Source-File welches die ”`main()`”-Methode enthält. |
| *n_othr* | 0 | Das *other*-Feld wird normalerweise nicht genutzt und auf ”0” gesetzt. |
| *n_desc* | 2 | *”the starting text address of the compilation.”*[5] |
| *n_value* | 00000000 | Dieser Integer wird hauptsächlich für *.stabn*-Direktive genutzt. |
| *n_strx* | 15 | Start des Strings der nächste Direktive |
| *String* | 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 *Source 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 sieht man folgende Instruktionen:

```
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 *"looopExample.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 jedesmal 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 geschwungenen 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. Das vollständige Programm ist im Anhang **??** angehängt.

Die untenstehenden Zeilen entsprechen den Zeilen 39-42 von *"Reset.java* aus dem Anhang **??**. 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 **??** angehängt ist.

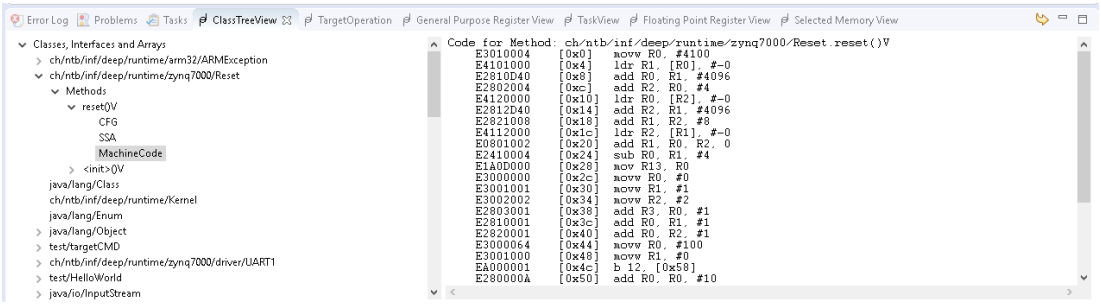


Abbildung 6.2: ClassTreeView mit Maschinencode der Reset-Methode in *deep*

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  b 0
```

*"loop.S"* im Anhang **??** enthält den "aufgeräumten" 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 **??** angehängt.

Die NTB-Wiki-Dokumentation[6] wurde als Ausgangslage genutzt. Die Datei *"stabs.include"* (siehe Anhang **??**) konnte direkt genutzt werden. Die Definition des Sourcecods (N_SO) und die Definitionen der Zeilennummern (N_SLINE) 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.

[6] https://wiki.ntb.ch/infoportal/software/gdb/start?s[]=stabs

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_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
```

### 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 Ahang **??** kann das Programm assembliert werden. Die ELF-Datei *"loopWithSTABS"* kann dann mit dem *gdb* geladen werden.

# 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 *"GNU Embedded Toolchain"* mit *gdb*

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-Webseite[1] heruntergeladen werden kann. 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.

[1] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

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 **??** und Anhang **??**) 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 **??** genutzt werden. Die Pfade im Script müssen angepasst werden. Die Konfigurationsdatei *"gdbInit.txt"* (siehe Anhang **??**) 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 Befehlt ''`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  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. Führt man mit ”`s`” einen einzelnen Step, also eine Zeile im Sourcecode aus, dann erhöht sich der Wert der Variable ”`x00`” 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 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 Eidesstattliche Erklärung

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

# Anhang