Inhaltsverzeichnis

1 STUFF TO SORT

2 Einleitung

2.1 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 eine Java Programm zu schreiben, welches direkt auf einem Prozessor läuft und Echtzeit-Fähigkeiten hat. Zur Zeit wird dieses Projekt in der NTB für die Ausbildung Von Systemtechnik Studenten verwendet. Es erlaubt einfach und schnell Robotersteuerungen und Regelungen zu implementieren, ohne dass man sich mit den Eigenarten von C und C++ Programmen auseinandersetzen muss.

deep unterstützt einige grundlegende Debuging-Funktionalitäten. 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 *Source Code Navigation*, wie man es aus bekannten Debuggern wie dem *gdb*[2] kennt.

2.2 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 teuer und veraltet.

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 grosse Flexibilität bei der Auswahl von ARM-Prozessoren können sehr günstige oder auch sehr leistungsstarke Prozessoren verwendet werden.

deep ist ein Open-Source-Projekt welches auch für den Unterricht verwendet wird. Damit nicht für jeden Student 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 Speicher Auslesen sollen auch Variablen gelesen und geschrieben werden können. Eine native *Source Code Navigation* in Eclipse vereinfacht die Entwicklung einer *deep*x-Applikation sehr.

2.3 Zielsetzung

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

1. Passende Hardware (Experimentierboard) finden, welche auch im Unterricht verwendet werden kann.
2. Grundlegendes Debug-Interfeace, 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 Plug-In 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/

3 Auswahl der Hardware

3.1 Einleitung

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 von den verschiedenen Prozessoren, aber sie zeigt recht gut wie enorm weit ARM Prozessoren verbreitet sind.

In diesem Kapitel soll in 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 in anspruchsvollen Robotikprojekten verwenden zu können.

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

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

3.2.1 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 JTAG Interface BDI3000[2] 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: Eigenes PCB entwickeln und herstellen ist keine Option.
  - Einbettbar: Der Prozessor muss auch bei einem selbst entwickelten PCB verwendet werden können. Wahlweise als SOM (*System On Module*) oder direkt als Prozessor in eigenem Package.
  - Noch lange erhältlich.
- 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.
  - 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.

[1] Elektronischer/Anhang/ARM-media-fact-sheet-2016.pdf
[2] http://www.abatron.ch/fileadmin/user_upload/news/BDI3000-Brochure.pdf
[3] Media Access Control

3.2.2 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.

3.3 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 separate Peripherie im Prozessor. Über das JTAG Interface kann der Scan ausgeführt werden, ohne dass eine Software 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 von 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**
- **Hardwaer 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 der Bootvorgang, vom *deep* Laufzeit System zu entwickeln.

3.4 Übersicht über die ARM Mikroarchitekturen

3.4.1 Cortex-A

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

Tabelle 3.1: Übersicht ARM Mikroarchitekturen

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

3.4.2 Cortex-R

Cortex-R 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.

3.4.3 Cortex-M

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

3.4.4 ARM Prozessoren ausserhalb der Cortex Reihe

Seit 2004 werden die meisten Kerne in eine der Cortex Gruppen 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.

3.4.5 Fazit über die ARM Mikroarchitekturen

Prozessoren die auf der Cortex-A Mikroarchitektur basieren bieten die grösste Flexibilität. Zusätzlich ist auch 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 aufs der Cortex-A Reihe begrenzt.

3.5 Anbindung des FPGAs

3.5.1 Einleitung

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 im Bild 3.1 schematisch zusammengefasst.

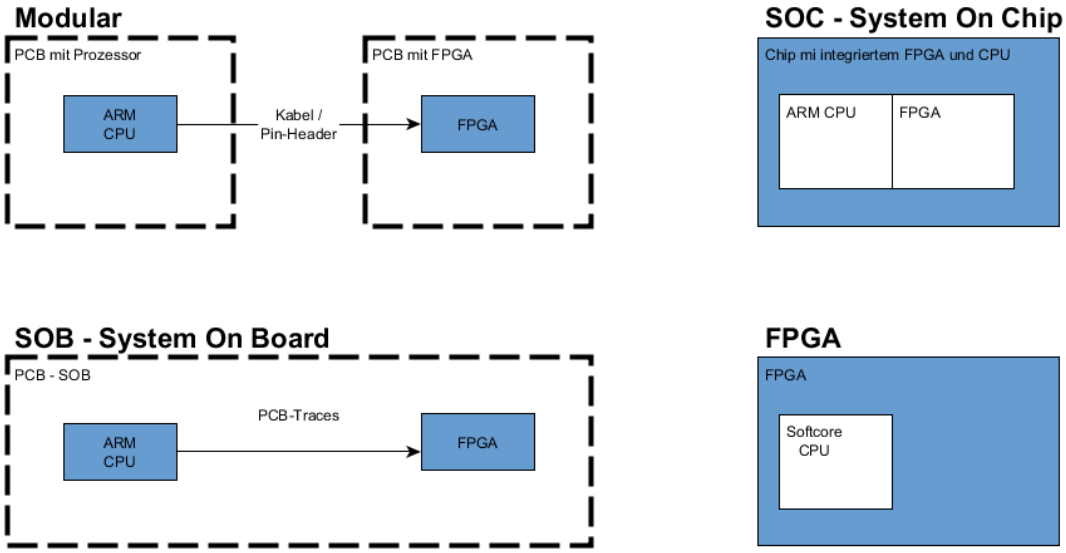

Abbildung 3.1: Mögliche Anbindungen des FPGA an die CPU

3.5.2 FPGA als Zusatzplatine zum Prozessorboard - Bauweise "Modular"

Das *FPGA Development Board CAPE for the BEAGLEBONE*[4] ist eine Aufsteckplatine für den *Beaglebone Black*. Wenn es auf den *Beaglebone Black* aufgesteckt wird, erweitert es den ARM basierten Linux PC um *Spatran 6 LX9* FPGA inklusive einiger I/O-Peripherie und SDRAM.

Vorteile:

- Relativ günstig.
- Funktioniert "Out of the Box"
- Schnelles GPMC[5] Interface (bis zu 70 MB/s) zwischen Prozessor und FPGA.

Nachteile:

- Verwendet ein modifiziertes Linux-Image, das LOGI-Image.
- Der eMMC[6] 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"

3.5.3 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 ist eine 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, da ein Experimentierboard oft zu gross ist, oder nicht die benötigte Peripherie enthält. Da neben dem FPGA auch High-Speed-Peripherie wie z.B. RAM

[4] https://www.element14.com/community/docs/DOC-69215/l/fpga-development-board-cape-for-the-beaglebone
[5] General-Purpose Memory Controller
[6] Embedded Multi Media Card

Tabelle 3.2: Übersicht Bauformen

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

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.

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

3.5.4 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

3.5.5 ARM als Softcore in FPGA - Bauweise "FPGA"

STM23

STM

4 System

4.1 Einleitung

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

4.2 Schematische Übersicht

In der Abbildung 4.1 ist das ganze System abgebildet. Auf dem *Windows PC* wird die *deep*-Applikation in Eclipse geschrieben, kompiliert und debuggt. Plug-Ins erweitern Eclipse um die notwendige Funktionalitäten, 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 BDI 3000* über die rote TCP/IP Verbindung angesprochen. Das BDI kommuniziert dann über eine 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 BDI. OpenOCD stellt einen GDB Server und auch ein CLI (*Command Line Interface*) zur Verfügung. Das Eclipse Plugin *openOCDInterface* verwendet das CLI über den TCP/IP Port 4444 (dunkelgrüner Pfeil). Der GDB Client kommuniziert mit dem GDB Server mit dem GDB Protokoll über den TCP/IP Port 3333 (hellgrüner Pfeil). OpenOCD verwendet dann den *WinUSB* Treiber um mit dem FT2232 Chip zu kommunizieren. Der FT2232 verwendet den selben JTAG Bus wie das BDI 3000 als Verbindung mit dem Zynq.

Das *Zybo* beinhaltet neben dem FT2232 auch noch diverse I/O Peripherie die in einer *deep*-Applikation genutzt werden kann. Der FT2232 Chip übernimmt zwei verschiedene Funktionen. Zum einen wird er als USB zu UART Brücke verwendet, damit man mit dem Windows PC einfach eine serielle Verbindung mit dem Prozessor aufbauen kann. Zusätzlich fungiert er ebenfalls als Brücke zum JTAG Bus. Das bedeutet, er erhält Befehle von OpenOCD über USB und übersetzt diese elektrisch und auch logisch für das JTAG Interface.

4.3 Debugger Toolchains

4.3.1 Abatron-Toolchain

Die Abatron-Toolchain benötigt weder OpenOCD noch den FT2232, dafür aber das teure BDI 3000. Diese ”klassische” Toolchain wird für die Entwicklung von Deep Applikationen für den PowerPC verwendet. In dieser Arbeit wird sie aber nicht direkt verwendet.

4.3.2 CLI-OpenOCD-Toolchain

Das teure BDI wird für diese Toolchain nicht mehr benötigt. Da das CLI[1] von OpenOCD ist aber sehr ähnlich wie das CLI des BDI. Eine Portierung ist somit relativ einfach. Die CLI-OpenOCD-Toolchain lehnt sich deshalb sehr stark an die bestehende Abatron Toolchain an.

Mit dieser Toolchain ist *Source Code Debugging* aber nicht möglich. Das bedeutet, es ist nicht möglich im Source Code Breakpoints zu setzten, oder durch einzelne Zeilen im Source Code 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.

[1] Command Line Interface

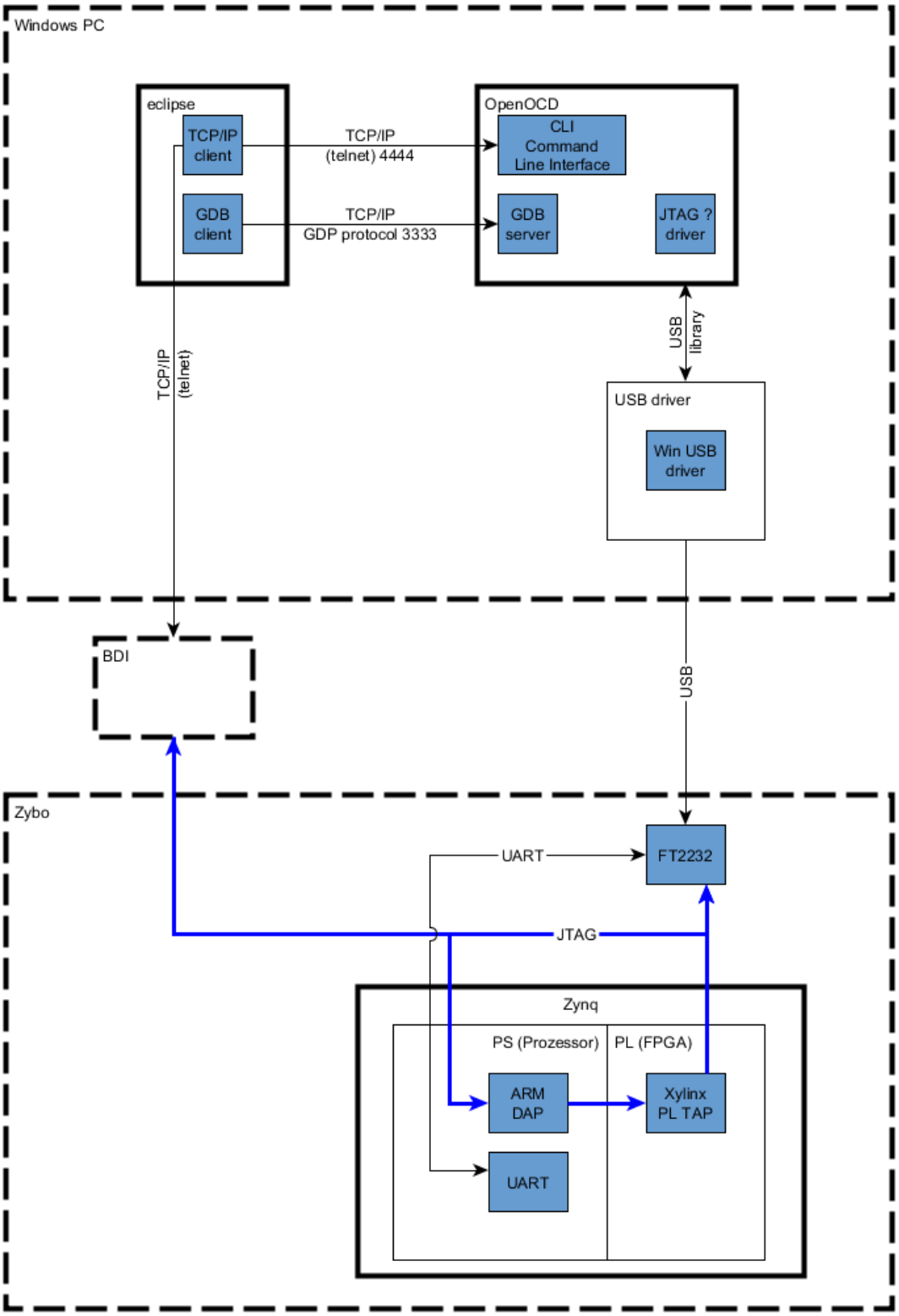

Abbildung 4.1: Systemübersicht Debugger Toolchain

4.3.3 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 bekannte GDB in Eclipse verwendet. Dadurch kann *Source Code Debugging* direkt in Eclipse eingesetzt werden.

5 Zynq

Der Zynq-7000 ist ein SoC[1] der einen 667 MHz Dual-Core ARM Cortex-A9 Prozessor und einem 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 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 5.1 gibt einen guten Überblick über das ganze SoC.

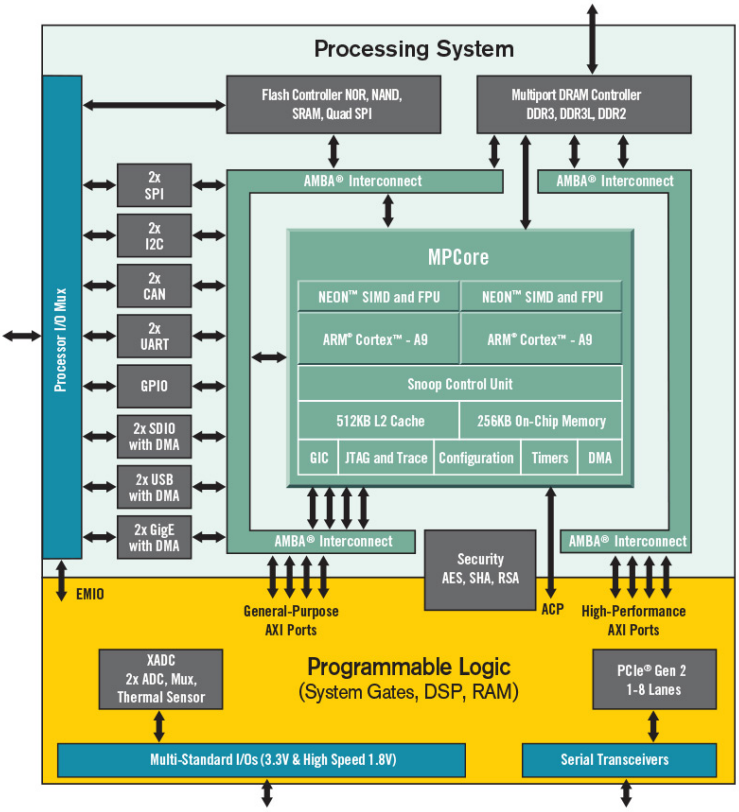

Abbildung 5.1: Block Diagramm Zynq7000[2]

5.0.1 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 direkt an die PL angeschlossen sind. Aus diesem Grund können die EMIOs nur verwendet werden, wen die PL entsprechend programmiert wurde. Diese Arbeit beschränkt sich nur auf die MIOs und das PS. Im TRM[3] 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.

5.1 Standard Zybo Workflow

Im *Getting Started with Zynq*[4] 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.

[1] System on Chip
[2] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html
[3] Technical Reference Manual
[4] https://reference.digilentinc.com/learn/programmable-logic/tutorials/zybo-getting-started-with-zynq/start?redirect=1

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 7.1.2 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 XDSK verwendet im Hintergrund das XSCT[5] (*Xilinx Software Command-Line Tool*). Das XDSK kann interaktiv, oder mit Scripts verwendet werden. Wie auch Jim-TCL basiert die verwendete Scriptsprache auf der Sprache TCL. Wird das "Hello World" Programm im XSDK gestartet, erhält man im *SDK Log* Fenster ein detailliertes Log des ausgeführten Script. In diesem Log kann nachvollzogen werden, was das Script beim Download und Start des Programms alles ausgeführt.

Im Anhang C ist eine Kopie eines solchen Logs zu finden. *D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/.sdk/launch_c++_application_(system_debugger)/system_debugger_using_debug_01_gettingstarted_applicationproject.elf_on_local*

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" Programm im XSDK ausgeführt wird. In Zeile 9 vom *...elf_on_local.tcl* wird auch noch die Methode *ps7_post_config* von *ps7_init.tcl* auf, welche im Anschluss *ps7_post_config_3_0* aufruft.

Alle Konfigurationsregister sind im Anhang B vom *Zynq TRM*[1] beschrieben. Bevor die Register aber verändert werden können, müssen sie *"unlocked"* werden, in dem der Wert *0x0000DF0D* in die Adresse *0xF8000008* geschrieben wird.

5.1.1 Grundlegende Methoden

Alle Methoden 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 Speicherinhalt von der Speicheradresse <address> gleich 0 sind.

mask_dellay <address> <value>:
Wartet <value> Millisekunden.

[5] https://www.xilinx.com/html_docs/xilinx2018_1/SDK_Doc/xsct/intro/xsct_introduction.html

5.1.2 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. Es wird der Multiplexer für die IO Pins 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[6] 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.

ps7_peripherals_init_data_3_0
Konfiguriert folgende Peripherie:

- 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 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.

helloworld.c:

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

platform.c:

[6] Phase Locked Loop

```
...
/*#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();
}
...
```

5.1.3 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. Unten stehende 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 unten stehenden 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
  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
  puts "PCW Silicon Version : 3.0"
}

proc get_number_of_cycles_for_delay { delay } {
  # GTC is always clocked at 1/2 of the CPU frequency (CPU_3x2x)
  set APU_FREQ  650000000
  return [ expr ($delay * $APU_FREQ /(2 * 1000))]
}
```

5.2 Memory

5.2.1 Address Mapping

Im Kapitel 4.1 des *Zynq TRM*[1] ist der Aufbau des Speichers beschrieben. Die Abbildung 5.2 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 direkt 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 mit 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, also

auch vor dem selbst geschriebenen Bootloader, ausgeführt wird. 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"*.

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 5.2: Address Map des Zynq

6 Zybo

Das Zybo ist ein Experimentierboard für den Zynq-7000. Es beinhaltet die notwendigen Hardware wie Signaltransformatoren und Buchsen für Ethernet, USB, HDMI und VGA. Neben der Stromversorgung wird liefert es auch ein JTAG-Interface um den Zynq zu debuggen.

6.1 Floating Point Unit

FPUs (*Floating Point Unit*) können je nach Implementation unterschiedliche Funktionen unterstützen. In den Register MVFR0 und MVFR (*Media and VFP Feature Register*) lässt sich auslesen, welche Funktionen effektiv 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 der Assembler Befehl *"VMRS"* verwendet werden.

6.1.1 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 kann.

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) abgesehen von CP14 und CP15.

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[3] 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
```

6.1.2 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

6.1.3 Unterstützte Features der FPU

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

Der Zynq 7000 des Zybo unterstützt:

-

7 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 BDI 3000 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 Physikalischer Stecker. In den 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 7.1 zu sehen.

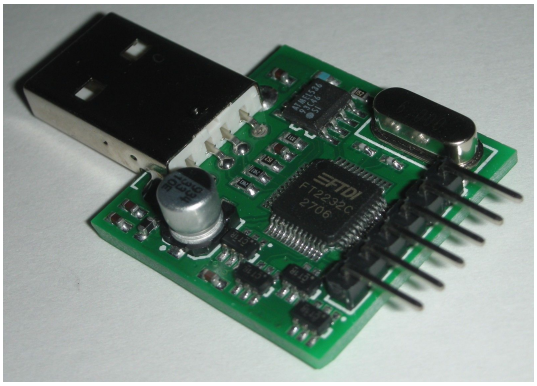

Abbildung 7.1: Generischer JTAG Adapter mit einem FTDI FT2232[2]

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.

7.1 Installation

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

7.1.1 Installation - OpenOCD

OpenOCD kann direkt aus dem Sourcecode kompiliert werden[3] oder es können vorkompillierte 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[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

7.1.2 Installation - 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 den *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, dann wird keine zusätzliche Stromversorgung für das Zybo benötigt.

Öffnet man das *USB Driver Tool* werden alle USB Devices aufgelistet. Das Device mit der *Vendor ID=0403, Device ID=6010* und *Interface 0* ist das JTAG Interface des FT2232. Mit einem Rechtsklick darauf kann man den *Install WinUSB* Treiber auswählen und installieren. Abbildung 7.2 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 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.

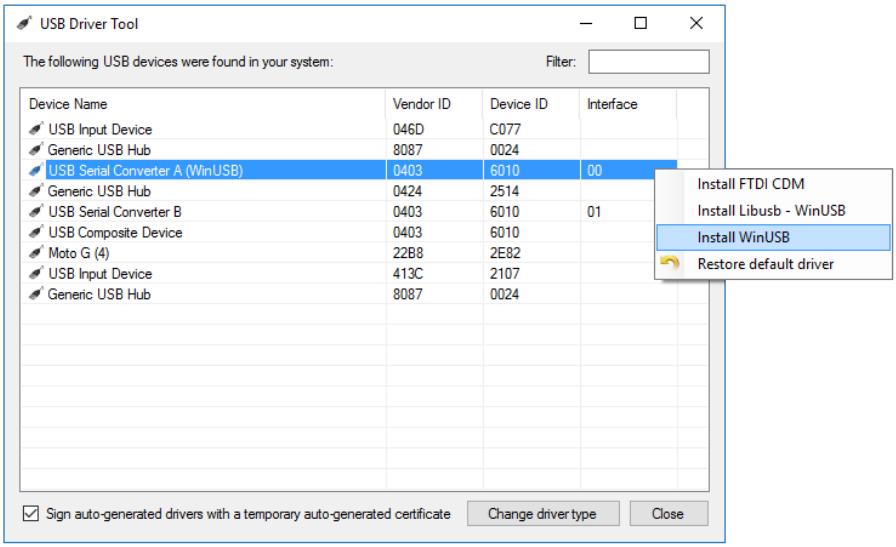

Abbildung 7.2: Installation des *WinUSB* Treibers mit dem *USB Driver Tool*

7.2 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 Kapitel wird folgende Notation verwendet, um einen CLI-Befehl zu beschreiben: (`CLI: Befehl`)

7.3 OpenOCD Konfiguration - Einleitung

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 man den gleichen Adapter aber einen anderen Prozessor verwenden will. Im Pfad `openocd-0.10.0/scripts` befinden 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`

7.4 OpenOCD Konfiguration - Interface

Die Interface Konfiguration 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 von 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
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.

7.4.1 Resetverhalten

Liest man aus einer unerlaubten Speicheradresse (`CLI: mdw 0x40000000`), dann hängt sich die Debug-Peripherie des Zynq auf. Nach so 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 vom Zybo 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

[8] https://github.com/f32c/f32c/blob/master/rtl/proj/xilinx/zybo/xram_bram_hdmi_ise/zybo.ocd
[9] https://www.xilinx.com/support/answers/63871.html

fängt so einen 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 einen SRST (*System Reset*) und dem TRST (*TAP Reset*) unterschieden. Der SRST führt dabei 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, dann scheint der Befehl ''`CLI: reset halt`'' zu funktionieren. Greift man vorher aber wieder auf eine ungültige Speicherstelle zu, dann erscheint beim Reset die Fehlermeldung:

```
...
Timeout waiting for dpm prepare
...
```

Der erneute Timeout legt die Vermutung nahe, dass der Zynq nicht ordentlich zurück gesetzt 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 A) könnte man überprüfen, welche I/Os des FT2232 effektiv 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.

Im OpenOCD User Manual[5] wird der für ''`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 herausgestellt, dass die Kombination *-data 0x3000* mit *-oe 0x3000* tatsächlich einen System Reset ermöglicht.

Weil der Debugger direkt nach dem SRST versuch 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 Debug Peripherie wieder betriebsbereit ist, wenn der Debugger zu kommunizieren versucht.

7.5 OpenOCD Konfiguration - Board

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

7.6 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.

8 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. In diesem Kapitel wird der grundlegende Aufbau des Formates 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 Formates im Detail erklärt.

8.1 Nützliche Tools

readelf ist ein nützliches Linux-Tool um Informationen einer ELF-Datei anzeigen zu lassen. Unter Windows kann dieses Software ebenfalls in der Shell verwendet werden, wenn *mingw*[2] installiert ist.

8.2 Grundlegender Aufbau

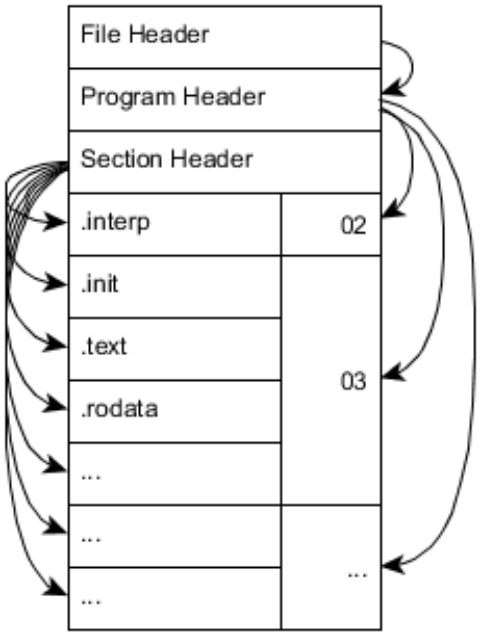

Abbildung 8.1: Der Aufbau von einer ELF Datei[3]

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

Der *Program Header* kann mit ''`readelf filename -Wl`'' ausgegeben werden. Darin ist enthalten, welcher Offset innerhalb der Datei die einzelnen Segmente haben. Zusätzlich wird auch definiert, zur welchen Speicheraddresse (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 einem nicht initialisierten Pointer, in einer Speicherstelle im Memory gelesen, das kein *"read flag"* hat,

[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] http://www.mingw.org/
[3] https://slideplayer.com/slide/6444592/

dann wird ein *Segmentation Fault* ausgelöst. Der *gdb* nutzt Informationen aus diesem Header um zu bestimmen, welche binäre Daten mit dem Befehl ''`load`'' an welchen Speicherort kopiert werden soll. 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 kleine, 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 8.3
- `.stabstr` Enthält die Stabs Debuginformationen. Mehr dazu im Kapitel 8.3

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

8.2.1 Informationen für den Debugger

Zusätzliche Informationen für den Debugger werden ebenfalls in dem 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 funktionaler Umfang wie das STABS-Format, aber die manuelle Implementation ist aufwändiger.

8.3 Stabs

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

8.3.1 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 Demo-Programm mit STABS implementiert, da STABS-Informationen einfacher manuell zu implementieren sind als DWARF-Informationen.

8.3.2 Aufbau des STABS Format

Eine einheitliche Dokumentation für STABS gibt es nicht. Es ist nicht einmal sicher bekannt, wer der ursprüngliche Erfinder von diesem Format ist. In der Dokumentation von *Sourceware*[4] wird aber Peter Kessler als Erfinder genannt.

Der Aufbau von diesem Format wird in der oben genannten Dokumentation von *Sourceware* und in der Dokumentation von der *"University of Utha"*[5] beschrieben. Obwohl diese Dokumentationen zum teil sehr detailliert sind, sind sie nicht lückenlos. Im Folgenden wird nur auf die Grundlagen eingegangen, die für das Beispielprogramm relevant sind.

[4] 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
[5] 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

STABS-Informationen sind in einzelne Informations-Elemente, so genannte *directives*, unterteilt. Jede Direktive ist entweder ein *".stabs"* (String), ein *".stabn"* (Integer) oder ein *".stabd"* (Dot). Zusätzlich ist jede Direktive von einem 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 E.1) definiert. Im Kapitel 12 der Dokumentation der *"University of Utha"* sind die einzelnen Typen genau beschreiben.

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
```

8.3.3 DWARF

8.4 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.

8.4.1 Vorgehen

Um ein Demoprogramm zu erstellen, werden untenstehende Schritte durchgeführt. Alle Schritte werden weiter unten im Detail erklärt. Das Programm *"loop"* soll für den *gdb*-Test verwendet werden. *"loopExample"* ist ein Hilfsprogramm, das vom *gdb* automatische generierte STABS enthält. Es dient als Vorlage, um die richtigen 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 kompilierte Programm disassembliert, um die STABS in einer leserlichen Form zu erhalten.
   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 Programm für den Zynq mit manuell ergänzten STABS erstellen:
   a) **Reset.Java**: Den Source-Code 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 Assembler-Code abgeleitet aus *"loopMachineCode.txt"*.

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 kompilierte Programm disassembliert, um die STABS in einer leserlichen Form zu erhalten.

8.4.2 Java Demoprogramm

Das unten stehende Programm ist das Testprogramm, dass von *deep* in Maschinen-Code übersetzt werden soll und anschliessend manuell mit STABS ergänzt werden soll.

```
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.

8.4.3 Beispiel-Programm "loopExample"

Der Code in *"loopExample.c"* im Anhang E ist fast identisch wie der Code des Java Demoprogramms. Es wurden nur einige Änderungen gemacht, damit es 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 E 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:

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

Kompilierbare Assembler Syntax:

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

8.4.4 Analyse der disassemblierten STABS

Die unten stehenden Direktiven sind ein Auszug aus der Datei *"loopExample.Sd"* im Anhang E. Die Tabelle 8.1 beschreibt die Direktive 0 im Detail.

```
Symnum n_type n_othr n_desc n_value  n_strx String
...
0       SO      0       2       00000000 15     loopExample.c
1       OPT     0       0       00000000 29     gcc2_compiled.
2       LSYM    0       0       00000000 44     int:t(0,1)=r(0,1)
    ;-2147483648;2147483647;
...
51      GSYM    0       0       00000000 1919   global:G(0,1)
52      FUN     0       0       00000000 1933   c_entry:F(0,1)
53      SLINE   0       4       00000000 0
54      SLINE   0       5       0000000c 0
...
72      LSYM    0       0       fffffff0 1948   x00:(0,1)
73      LSYM    0       0       fffffec 1958    x01:(0,1)
74      LSYM    0       0       ffffffe8 1968   x02:(0,1)
75      RSYM    0       0       00000004 1978   s:r(0,1)
76      LSYM    0       0       ffffffe4 1987   float0:(0,14)
77      LSYM    0       0       fffffff8 2001   int0:(0,1)
78      LBRAC   0       0       00000000 0
79      LSYM    0       0       fffffff4 2012   i:(0,1)
80      LBRAC   0       0       00000060 0
81      RBRAC   0       0       00000090 0
82      RBRAC   0       0       000000c4 0
83      SO      0       0       000000c4 0
```

Tabelle 8.1: 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."*[6]
n_value	00000000	Dieser Integer wird hauptsächlich für *.stabn*-Direktive genutzt.
n_strx	15	Start des Strings für die 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 alles verschiedene 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 sich 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 initialisiert, bevor die erste Zeile, oder genauer gesagt Zeile 5 in *"looopExample.c"*, C-Code 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 Variablenname "x00" und den Typ "(0,1)". Der Typ "(0,1)" wird 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, dann kann dieses Verhalten ändern. Mit der Zeile ''`register int s=1;`'' im C-Code wird der Compiler gezwungen, die Variable nur in den Registern zu behalten und nicht auf dem Stack abzulegen. Aus diesem Grund wird für die Variable *"s"* eine Direktive vom Typ 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 werden 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 von so einem Block.

8.4.5 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 unten stehenden 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 8.2) von *deep* kann der Assemblercode der ''`reset()`''-Methode kopiert werden welcher im Anhang E.3 angehängt ist.

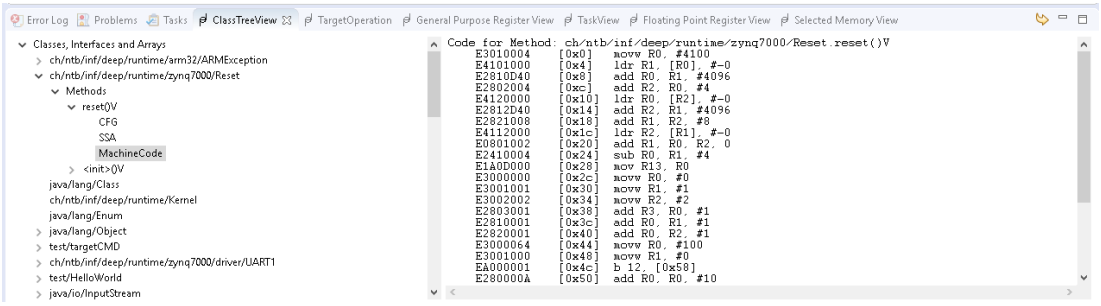

Abbildung 8.2: ClassTreeView mit Maschinencode der Reset-Methode in *deep*

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 E.4 enthält den "aufgeräumten" Assemblercode. Der Code wurde mit zusätzlichen Assembler-Direktiven ergänzt. "`c_entry`" beschreibt den Punkt, an dem das Programm anfängt. "`START_LOOP_BODY`" und "`CHECK_LOOP_EXIT`" sind Punkte, welche für die *While-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.

9 Debugger

Es gibt diverse Debugger auf dem Markt. Diese Arbeit beschränke sich aber auf den GNU-Debugger (*gdb*), da unter der PGL Lizenz steht und somit eine Open Source Software ist. Bei den meisten Linux Distributionen wird der *gdb* direkt mitgeliefert und kann sofort verwendet werden.

9.0.1 Grundlegende Funktionsweise

Auf Linux verwendet der *gdb* den System Call *ptrace* (Kurzform für "process trace"). Dieser System Call erlaubt dem *gdb* einen anderen Prozess zu inspizieren und zu manipulieren. Im Hardwaredebugger, den wir später bearbeiten, verwenden wir stattdessen JTAG in Verbindung mit der Debugginghardware im Prozessor.

9.0.2 Vorbereitung

Für dieses Tutorial verwenden wir folgendes Beispielprogramm:

```
1  #include <iostream>
2  using namespace std;
3
4  int divint(int, int);
5  int main()
6  {
7      int x = 5, y = 2;
8      cout << divint(x, y);
9
10     x =3; y = 0;
11     cout << divint(x, y);
12
13     return 0;
14 }
15
16 int divint(int a, int b)
17 {
18     return a / b;
19 }
```

Diese Applikation können wir jetzt Kompilieren und mit *gdb* starten:

```
# g++ gdbTest.cpp -o gdbTest
# gdb gdbTest
# run
```
// startet die Applikation im *gdb*

```
(gdb) run
Starting program: /home/mgehrig2/projects/gdbTest/gdbTest
```

Program received signal SIGFPE, Arithmetic exception.
0x00000000004007b5 in divint(int, int) ()

Obwohl die Applikation nicht mit Debug-Symbolen kompiliert wurde, wird nicht nur die Adresse des Ursprungs der Floating Point Exception angezeigt, sondern auch der Name der Methode.

9.1 Funktionen eines Debuggers

Ein Debugger kann verschiedene Funktionen besitzen. Die grundlegenden Funktionen sind sehr einfach und brauchen keine grosse Ïntelligenz"vom Debugger selber.

Erweiterte Funktionen erwarten vom

9.2 Erstellen einer Dummy-Applikation mit Debug-Informationen

9.2.1 Vorgehen

Das Ziel von diesem Kapitel ist es, eine Deep-Applikation zu erzeugen, die mit *gdb* und *OpenOCD*

9.3 ELF-File

ELF

10 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

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 Schema Zybo

B zybo-ftdi.ocd angepasst:

```
#
# FTDI2232 on Zybo
#
#  https://github.com/f32c/f32c/blob/master/rtl/proj/xilinx/zybo/
    xram_bram_hdmi_ise/ftdi-zybo.ocd

interface ftdi
ftdi_device_desc "Digilent Adept USB Device"
ftdi_vid_pid 0x0403 0x6010

#ftdi_layout_init data direction
ftdi_layout_init 0x3088 0x1f8b

ftdi_layout_signal nSRST -data 0x3000 -oe 0x3000

# green MIO7 LED
ftdi_layout_signal LED -data 0x0010

reset_config srst_only
adapter_nsrst_delay 40
```

C Xilinx SDK Log:

```
14:26:34 INFO : Disconnected from the channel tcfchan#2.
14:26:36 INFO : 'targets -set -filter {jtag_cable_name =~ "Digilent Zybo
    210279573773A" && level==0} -index 1' command is executed.
14:26:36 INFO : 'fpga -state' command is executed.
14:26:36 INFO : Connected to target on host '127.0.0.1' and port '3121'.
14:26:36 INFO : Jtag cable 'Digilent Zybo 210279573773A' is selected.
14:26:36 INFO : 'jtag frequency' command is executed.
14:26:36 INFO : Sourcing of 'D:/Vivado/01_gettingStarted/01_gettingStarted.
    sdk/design_1_wrapper_hw_platform_0/ps7_init.tcl' is done.
14:26:36 INFO : Context for 'APU' is selected.
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'.
14:26:38 INFO : 'configparams force-mem-access 1' command is executed.
14:26:38 INFO : Context for 'APU' is selected.
14:26:38 INFO : 'stop' command is executed.
14:26:38 INFO : 'ps7_init' command is executed.
14:26:38 INFO : 'ps7_post_config' command is executed.
14:26:38 INFO : Context for processor 'ps7_cortexa9_0' is selected.
14:26:38 INFO : Processor reset is completed for 'ps7_cortexa9_0'.
14:26:38 INFO : Context for processor 'ps7_cortexa9_0' is selected.
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'.
14:26:39 INFO : 'configparams force-mem-access 0' command is executed.
14:26:39 INFO : ----------------XSDB Script----------------
connect -url tcp:127.0.0.1:3121
source D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
    design_1_wrapper_hw_platform_0/ps7_init.tcl
targets -set -nocase -filter {name =~"APU*" && jtag_cable_name =~ "Digilent
     Zybo 210279573773A"} -index 0
loadhw -hw D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
    design_1_wrapper_hw_platform_0/system.hdf -mem-ranges [list {0x40000000
     0xbfffffff}]
configparams force-mem-access 1
targets -set -nocase -filter {name =~"APU*" && jtag_cable_name =~ "Digilent
     Zybo 210279573773A"} -index 0
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'
```

D system_debugger_using_debug_01_gettingstarted_applicationpr

```
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  targets -set -nocase -filter {name =~"APU*" && jtag_cable_name =~ "Digilent
        Zybo 210279573773A"} -index 0
4  loadhw -hw D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/
       design_1_wrapper_hw_platform_0/system.hdf -mem-ranges [list {0x40000000
        0xbfffffff}]
5  configparams force-mem-access 1
6  targets -set -nocase -filter {name =~"APU*" && jtag_cable_name =~ "Digilent
        Zybo 210279573773A"} -index 0
7  stop
8  ps7_init
9  ps7_post_config
10 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
       Digilent Zybo 210279573773A"} -index 0
11 rst -processor
12 targets -set -nocase -filter {name =~ "ARM*#0" && jtag_cable_name =~ "
       Digilent Zybo 210279573773A"} -index 0
13 dow D:/Vivado/01_gettingStarted/01_gettingStarted.sdk/01
       _gettingStarted_ApplicationProject/Debug/01
       _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
```

E STABS

E.1 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_NSYMS,    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
```

E.2 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.ARMException;
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 ARMException implements Iarm32, Izybo7000,
        IdeepCompilerConstants {
36 //  static int g = 5555;
37
38   static void reset() {
39     int stackOffset = US.GET4(sysTabBaseAddr + stStackOffset);
40     int stackBase = US.GET4(sysTabBaseAddr + stackOffset + 4);
41     int stackSize = US.GET4(sysTabBaseAddr + stackOffset + 8);
42     US.PUTGPR(SP, stackBase + stackSize - 4); // set stack pointer
43
44     int x00 = 0;
45     int x01 = 10;
46     int x02 = 20;
47     x00++;
48     x01++;
49     x02++;
50
51     int x100 = 100;
52     for(int i=0; i<10; i++){
53             x100 += 10;
54        }
55
56     x100++;
57     x100++;
58     x100++;
59     x100++;
60     x100++;
61
62     US.ASM("b -8"); // stop here
63
64   }
65 }
```

E.3 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    BAFFFFFB   [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    EAFFFFFE   [0x74]   b 0, [0x74]
```

E.4 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
```