Inhaltsverzeichnis

1 STUFF TO SORT

1.1 Zybo

1.2 Was ist ein Debugger

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. Das *deep* Projekt ermöglicht es 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. Auch werden keine *Breakpoints* oder *Source Code Navigation* unterstützt, so wie man es aus bekannten Debuggern wie dem *gdb*[1] 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 ARM-Prozessoren 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 kann wahlweise 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-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.
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] 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 zu finden, 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

Um die richtige Hardware im riesigen Angebotsdschungel zu finden, sind Soll- und Muss-Kriterien ermittelt worden.

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 Modul oder direkt als Prozessor in eigenem Package.
  - Noch lange erhältlich.
- Prozessorebene
  - ARMv7: Der Prozessor muss auf der ARMv7 ISA[3] basieren.
  - ARM Instruktionen: Der Prozessor muss ARM Instruktionen unterstützen. *Thumb* Instruktionen sind nicht ausreichend.
  - FPU: Für Gleitzahlenarithmetik.
  - Netzwerkschnitstelle: RJ-45 inklusive MAC[4] und *Magnetics*.
  - USB: USB Schnittstelle als Host und als Slave.
  - Flash: Mehr als 50kByte Flash.
  - RAM: Mehr als 100kByte RAM.

Soll-Kriterien

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

- 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 bei der Produktion des Bestückten PCBs überprüft werden, ob alle Lötstellen Kontakt herstellen und 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 laufen 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**
- **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 installierte Software auf dem Prozessor benötigt, kann er auch gut verwendet werden, um die grundlegendsten Funktionen 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. Die 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 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 nicht die standard-ARM Instruktionen.

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.

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		* FPGA ist fix
SOC	* Potenziell sehr schnelle Datenverbindung zwischen FPGA und Prozessor	* FPGA ist fix * Relativ teuer
FPGA	* Flexibel	* Sehr teuer

3.5.2 FPGA als Zusatzplatine zum Prozessorboard - Bauweise "Modular"

Das *FPGA Development Board CAPE for the BEAGLEBONE*[5] 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[6] Interface (bis zu 70 MB/s) zwischen Prozessor und FPGA.

Nachteile:

- Verwendet ein modifiziertes Linux-Image, das LOGI-Image.
- Der eMMC[7] 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 Board wie der Prozessor (System On Board) - Bauweise "SOB"

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

3.5.5 ARM als Softcore in FPGA - Bauweise "FPGA"

STM23

STM

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

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 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 *Command Line Interface* (CLI) 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

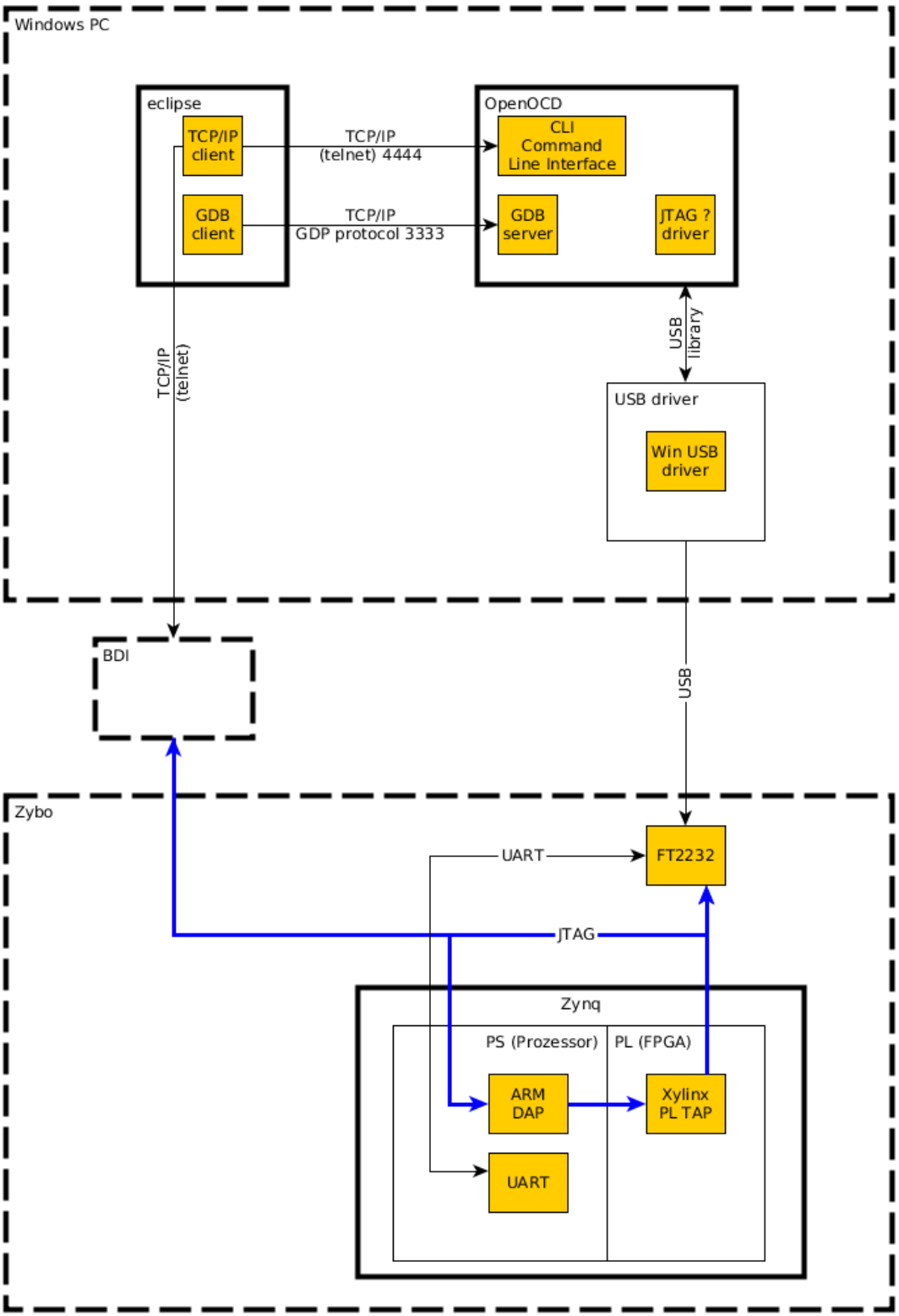

Abbildung 4.1: Systemübersicht Debugger Toolchain

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 Zybo

5.1 Zynq

5.1.1 Übersicht

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 den ganzen SoC.

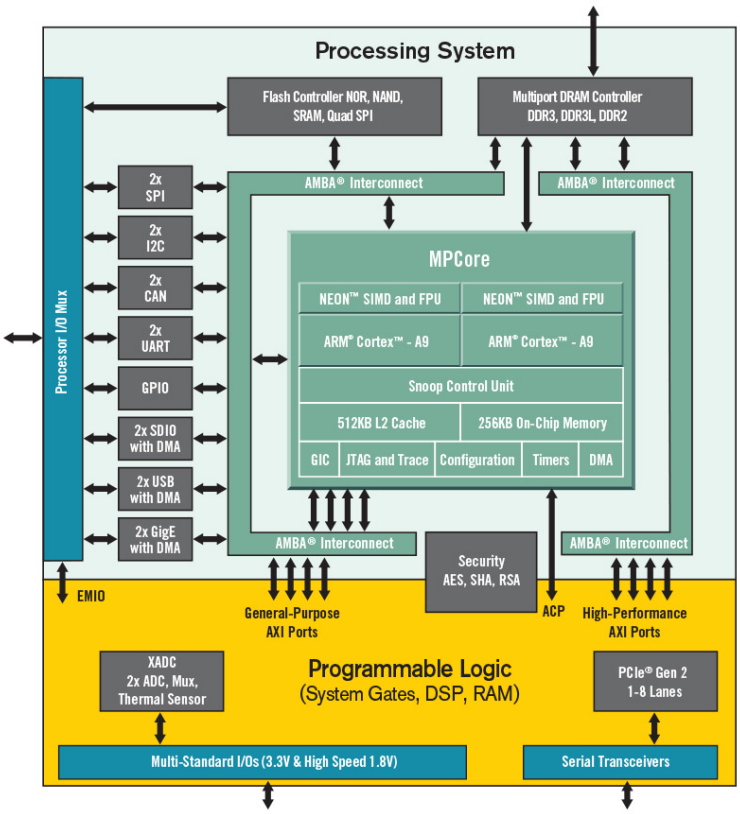

Abbildung 5.1: Block Diagramm Zynq7000[2]

5.1.2 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**??** im Kapitel *"2.5.4 MIO-at-a-Glance Table"* ist eine sehr gute Übersicht über alle möglichen Funktionen der MIOs gegeben.

[1] System on Chip
[2] https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html
[3] Technical Reference Manual

5.2 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. Dabei werden, z.B. für LED1 bis 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 6.2.2 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 man nachvollziehen, was das Script beim Download und Start des Programmes alles ausgeführt.

Im Anhang B 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 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 <mask> gleich 0 sind.

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

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.

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

- **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 sie aber auskommentiert. *"platform.c"* befindet sich im *"design_wrapper_hw_platform"* welcher in Vivado erzeugt wurde. *"ps7_init()"*

helloworld.c:

```
...
#include "platform.h"
..
int main ()
{
...
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();
}
...
```

[6] Phase Locked Loop

6 OpenOCD

6.1 Einleitung

OpenOCD[3] bildet den Software Teil von einem Debugger. 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 6.1 zu sehen.

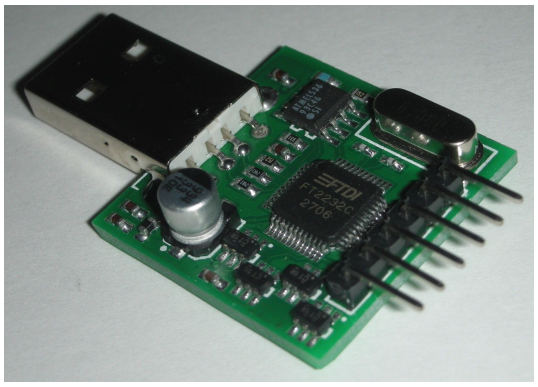

Abbildung 6.1: Generischer JTAG Adapter mit einem FTDI FT2232[4]

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 Hardware-Ebene debuggen zu können.

6.2 Installation

6.2.1 Installation - OpenOCD

OpenOCD kann direkt vom Sourcecode kompiliert werden[1] oder auch als vorkompiliertes Binary herunter geladen werden. Für diese Arbeit wurde das vorkompilierte Windows Binary für ARM Cores Version 0.10.0 von folgender URL verwendet:
http://www.freddiechopin.info/en/download/category/4-openocd?download=154%3Aopenocd-0.10.0

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://sourceforge.net/p/openocd/code/

6.2.2 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*, welches man unter folgender Adresse findet:
http://visualgdb.com/UsbDriverTool/

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 vom FT2232. Mit einem Rechtsklick darauf kann man den *Install WinUSB* Treiber auswählen und installieren. Abbildung 6.2 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.

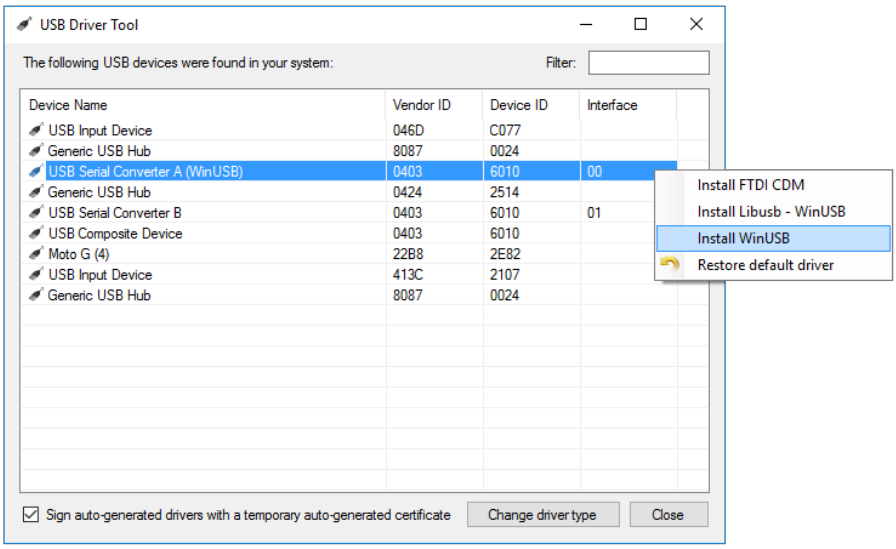

Abbildung 6.2: Installation des *WinUSB* Treibers mit dem *USB Driver Tool*

6.3 OpenOCD CLI - Command Line Interface

Das CLI[2] ist eine einfache Methode um mit dem Debugger zu kommunizieren. Über den Port 4444 kann, z.B. mit Putty, auf dem *Localhost* eine Telnet-Verbindung aufgebaut werden, sobald OpenOCD gestartet wurde. Der Befehl "`help`" listet alle zulässigen Befehle auf.

In den folgenden Kapitel wird folgende Notation verwendet, um ein Befehl zu beschreiben, der das CLI verwendet:
(`CLI: Befehl`)

6.4 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 Konfigurationss-

[2]Command Line Interface

cripts (*.cfg) in der Scriptsprache *Jim-Tcl*. *Jim-Tcl* ist eine abgespeckte Version von *Tcl*[3].

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 Konfigurationsscripten nach Wahl gestartet werden:
`openocd -f zybo-ftdi.cfg -f zybo.cfg`

6.5 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 *Zynq, 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[4] 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.

6.5.1 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`.
Mit einem manuellen Powercycle vom Zybo kann die Hardware wieder zurückgesetzt werden.

Mit OpenOCD ist es grundsätzlich möglich, einen Reset automatisch durchzuführen. Dabei wird zwischen einen *System Reset* (SRST) und dem *TAP*[5] *Reset* (TRST) unterschieden. Der SRST führt dabei einen Powercycle vom ganzen System durch, der TRST setzt nur den TAP zurück

Beim obigen Script ist aber das Resetverhalten nicht sauber definiert. Mit dem Befehl ''`CLI: reset halt`'' im CLI sollte ein Reset vom FT2232 durchgeführt werden. Der Befehl führt aber zur Fehlermeldung:
`... zynq.cpu0: how to reset? ...`

[3] http://www.tcl.tk
[4] https://github.com/f32c/f32c/blob/master/rtl/proj/xilinx/zybo/xram_bram_hdmi_ise/zybo.ocd
[5] Test Access Port

Im OpenOCD User Manual[5] im 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. So kann das Problem auf den System Reset 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. Auf dem elektrischen Schema vom Zybo**??** könnte man überprüfen, welche I/Os vom 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 aufgestartet ist, wenn der Debugger versucht zu kommunizieren.

6.6 OpenOCD Konfiguration - Board

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

6.7 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.

7 Debugger

Es gibt diverse Debugger auf dem Markt. In dieser Arbeit beschränke ich mich aber auf den GNU-Debugger (*gdb*). Der *gdb* steht unter der PGL Lizenz und ist somit Open Source. Bei den meisten Linux Distributionen wird der *gdb* direkt mitgeliefert und kann sofort verwendet werden.

7.1 Anwendungsbeispiel gdb auf einem Linux-System

Mit diesem Beispiel will ich zum einen ein kurzes Tutorial bieten um den Umgang mit dem *gdb* zu lernen. Zum anderen will ich damit aber auch die verschiedenen Funktionen vom Debugger zeigen, damit die beschriebenen Probleme später im Kapitel besser in einen Kontext gestellt werden können.

Für dieses Tutorial verwende ich ein Linux Mint 18.1 (basierend auf einem Ubuntu 16.01). Solange *gdb* installiert ist, ist das verwendete Betriebssystem aber nicht relevant.

7.1.1 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.

7.1.2 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.

7.2 Unterschied zwischen einem Software- und Hardwaredebugger

Auf einem Linux-System ist es sehr einfach ein Debugger einzusetzen.

7.3 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

7.4 Erstellen einer Dummy-Applikation mit Debug-Informationen

7.4.1 Vorgehen

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

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

Quellenverzeichnis

[1] Urs Graf: *deep - a Cross Development Platform for Java*, Juni 2018, http://www.deepjava.org/start

[2] Xilinx: *Zynq-7000 - Technical Reference Manual v1.12*, 20 Oktober 2017, http://openocd.org/about/

[3] Urs Graf: *OpenOCD - Open source On-Chip Debugger*, Juni 2018, http://openocd.org/about/

[4] Urs Graf: *Ebay - FPU1 FTDI FT2232 USB JTAG XILINX FPGA CPLD programmer cable* , Juni 2018, https://www.ebay.com/itm/FPU1-FTDI-FT2232-USB-JTAG-XILINX-FPGA-CPLD-programmer-cable-/181635528314

[5] Sreekishnan Venkateswaran: *Essential Linux Device Drivers*, 15 Januar 2017, Open On-Chip Debugger: OpenOCD User's Guide

Anhang

A 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
```

B 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
ps7_init
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
       '
```

C system_debugger_using_debug_01_gettingstarted_applicationpr

```
1  connect -url tcp:127.0.0.1:3121
2  source D:/Vivado/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
```