Inhaltsverzeichnis

1 Einleitung

1.1 Stand der Technik

Das Projekt *deep* [1] ist eine Cross Development Plattform, die es erlaubt, ein Java Programm direkt auf einem Prozessor auszuführen. 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.

1.2 Motivation

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

1.3 Zielsetzung

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

1. Passende Hardware (Experimentierboard) finden, welche auch im Unterricht verwendet werden kann.
2. 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/

2 Auswahl der Hardware

2.1 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]

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

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

Muss-Kriterien

- Systemebene
  - FPGA: Der Prozessor muss mit einem FPGA kommunizieren können
  - Hardware Debugger: Der Prozessor muss für die Entwicklung von *deep* ein Hardware Debugger 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 oder Nischenprodukte sind nicht akzeptabel
  - Als fertiges Modul erhältlich: Eigenes PCB Entwickeln und Herstellen ist keine Option
  - Einbettbar: Der Prozessor muss auch bei einem eigenen PCB eingelötet werden können
  - Noch lange erhältlich
- Prozessorebene
  - ARMv7: Der Prozessor muss auf der ARMv7 ISA[2] basieren
  - ARM Instruktionen: Der Prozessor muss ARM Instruktionen unterstützen. *Thumb* Instruktionen sind nicht ausreichend
  - FPU: Für Gleitzahlenarithmetik
  - Netzwerkschnittstelle: RJ-45 inklusive MAC[3] und *Magnetics*
  - USB: USB Schnittstelle
  - Flash: Mehr als 50kByte Flash
  - RAM: Mehr als 100kByte RAM

Soll-Kriterien

- Systemebene
  - Einfach einbettbar: Der Prozessor ist als Prozessormodul erhältlich, so dass das Design von einem PCB einfacher wird
  - Günstiger Hardwaredebugger: Der Hardwaredebugger kann auch für Applikationsentwicklung mit *deep* eingesetzt werden
  - Möglichst schneller Download der Applikation

[1] Elektronischer/Anhang/ARM-media-fact-sheet-2016.pdf
[2] Instruction Set Architecture
[3] Media Access Control

- Prozessorebene
  - Memory Mapped Bus für FPGA Schnittstelle
  - FPU unterstützt *Double Precision*
  - Integerdivision
  - Prozessortakt über 500MHz

2.2.1 Hardware Debugger

2.2.2 Übersicht über die ARM Mikroarchitekturen

2.2.3 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.

2.2.4 Cortex-R

Cortex-R werden entwickelt für Echtzeitanwendungen und Sicherheitskritische Applikationen wie Festplattenkontroller 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.

2.2.5 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.

2.2.6 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.

2.2.7 Fazit Core-Designs

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.

Tabelle 2.1: Bauformen

	Vorteile	Nachteile
A	* Sehr Leistungsstartk * Support für vollwertige Betriebssysteme * Grosse Variation erhältich (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 geignet 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

Tabelle 2.2: My caption

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

2.3 Bauform der Hardware

2.3.1 Einleitung

2.3.2 Bauweise

2.3.2.1 FPGA als Zusatzplatine zum Prozessorboard - Bauweise "Modular"

2.3.2.2 FPGA auf dem gleichen Board wie der Prozessor (System On Board) - Bauweise "SOB"

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

2.3.4 ARM als Softcore in FPGA - Bauweise "FPGA"

2.3.4.1 Übersicht über die Bauweisen

2.3.4.2 STM23

STM

3 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.

3.1 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.

3.1.1 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.

3.1.2 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.

3.2 Unterschied zwischen einem Software- und Hardwaredebugger

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

3.3 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

3.4 Erstellen einer Dummy-Applikation mit Debug-Informationen

3.4.1 Vorgehen

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

4 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

Anhang