Inhaltsverzeichnis

1 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 1.1 gibt einen guten Überblick über das ganze SoC.

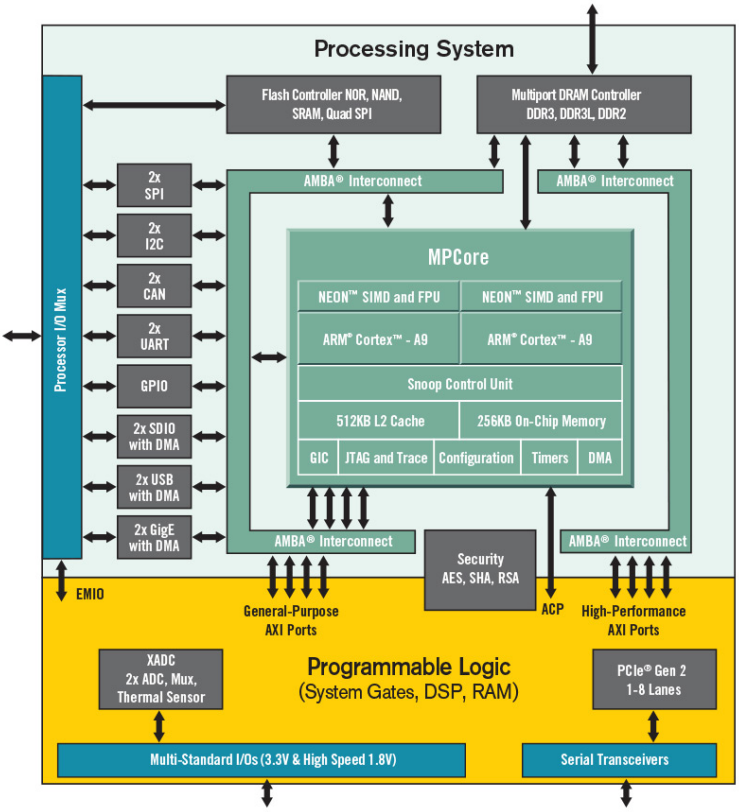

Abbildung 1.1: Block Diagramm Zynq7000[2]

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

1.1 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 **??** 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 **??** 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*[**?**] beschrieben. Bevor die Register aber verändert werden können, müssen sie *"unlocked"* werden, in dem der Wert *0x0000DF0D* in die Adresse *0xF8000008* geschrieben wird.

1.1.1 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

1.1.2 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();
}
...
```

1.1.3 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))]
}
```

1.2 Memory

1.2.1 Address Mapping

Im Kapitel 4.1 des *Zynq TRM*[**?**] ist der Aufbau des Speichers beschrieben. Die Abbildung 1.2 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[**?**] 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 1.2: Address Map des Zynq

2 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.

2.1 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.

2.1.1 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[**?**] 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
```

2.1.2 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

2.1.3 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[**?**] (*Architecture Reference Manual*) ist beschrieben, wie die unterstützten Features aus den Register gelesen werden können.

Der Zynq 7000 des Zybo unterstützt:

-

3 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[**?**] ist der Aufbau des Formates im Detail erklärt.

3.1 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.

3.2 Grundlegender Aufbau

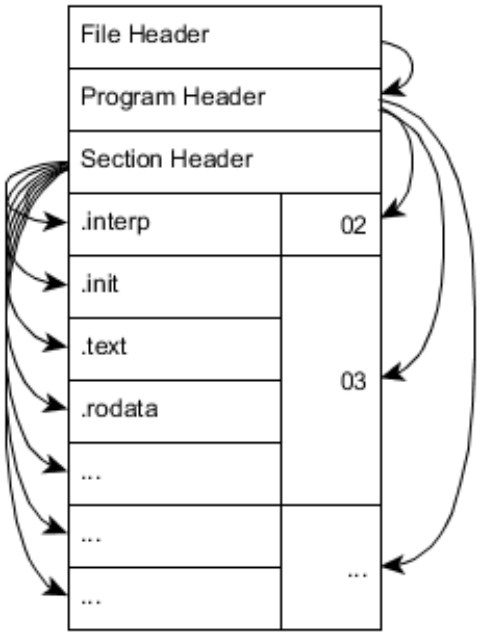

Abbildung 3.1: Der Aufbau von einer ELF Datei[3]

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 3.3
- `.stabstr` Enthält die Stabs Debuginformationen. Mehr dazu im Kapitel 3.3

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

3.2.1 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.

3.3 Stabs

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

3.3.1 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.

3.3.2 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 **??**) 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
```

3.3.3 DWARF

3.4 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.

3.4.1 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.

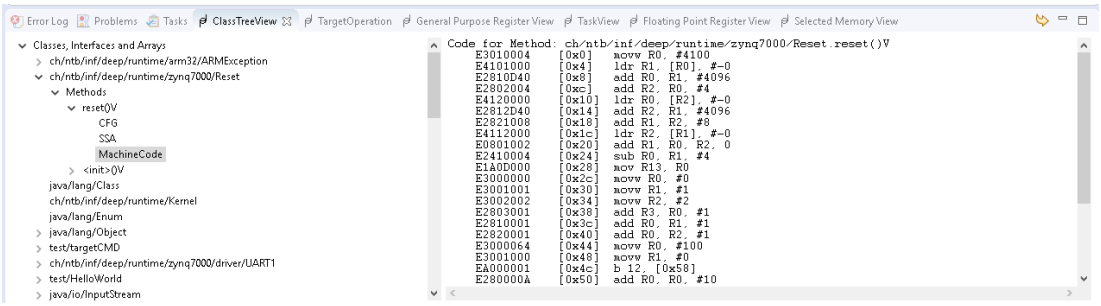

Abbildung 3.2: ClassTreeView mit Maschinencode der Reset-Methode in *deep*

3.4.2 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 Stack-Pointers, mit Java normalerweise nicht möglich sind, wird hier die entsprechende *deep*-Instruktion verwendet.

3.4.3 Beispiel-Programm "loopExample"

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

Beispiel mit disassemblierter Syntax:

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

Kompilierbare Assembler Syntax:

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

3.4.4 Analyse der disassemblierten STABS

Die unten stehenden Direktiven sind ein Auszug aus der Datei *"loopExample.Sd"* im Anhang **??**. Die Tabelle 3.1 beschreibt die Direktive 0 im Detail.

```
1  Symnum n_type n_othr n_desc n_value  n_strx String
2  ...
3  0       SO      0       2       00000000 15      loopExample.c
4  1       OPT     0       0       00000000 29      gcc2_compiled.
5  2       LSYM    0       0       00000000 44      int:t(0,1)=r(0,1)
       ;-2147483648;2147483647;
6  ...
7  51      GSYM    0       0       00000000 1919    global:G(0,1)
8  52      FUN     0       0       00000000 1933    c_entry:F(0,1)
9  53      SLINE   0       4       00000000 0
10 54      SLINE   0       5       0000000c 0
11 ...
12 72      LSYM    0       0       fffffff0 1948    x00:(0,1)
13 73      LSYM    0       0       fffffec 1958     x01:(0,1)
14 74      LSYM    0       0       ffffffe8 1968    x02:(0,1)
15 75      RSYM    0       0       00000004 1978    s:r(0,1)
16 76      LSYM    0       0       ffffffe4 1987    float0:(0,14)
17 77      LSYM    0       0       fffffff8 2001    int0:(0,1)
18 78      LBRAC   0       0       00000000 0
19 79      LSYM    0       0       fffffff4 2012    i:(0,1)
20 80      LBRAC   0       0       00000060 0
21 81      RBRAC   0       0       00000090 0
22 82      RBRAC   0       0       000000c4 0
23 83      SO      0       0       000000c4 0
```

Tabelle 3.1: 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.

Anhang