Inhaltsverzeichnis

1 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.

1.1 Nützliche Tools

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

1.2 Grundlegender Aufbau

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

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

[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/

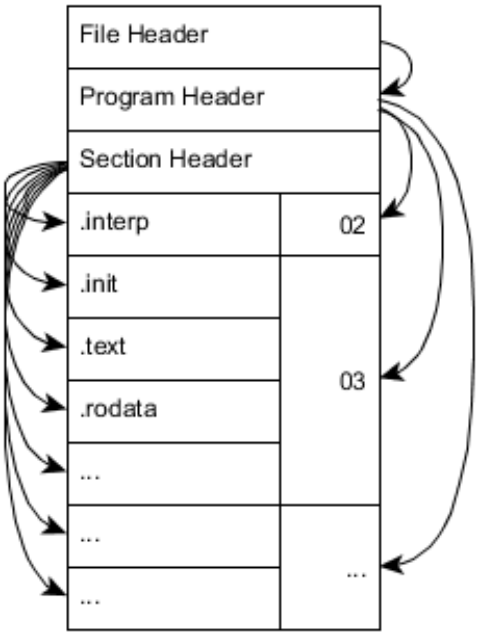

Abbildung 1.1: Der Aufbau von einer ELF Datei[3]

ein *Segmentation Fault* ausgelöst. Der *gdb* nutzt Informationen aus diesem Header um zu bestimmen, welche binären Daten mit dem Befehl ''`load`'' an welchen Speicherort kopiert werden sollen. Ein Segment beinhaltet ein oder mehrere *Sections*.

Im *Section Header* sind alle *Sections* beschrieben. Mit ''`readelf filename -WS`'' kann man sehen, dass jede *Section* unter anderem einen Namen, einen Typ, eine Adresse (absolut) und einen Offset (relativ, innerhalb der ELF-Datei) enthält. Jede *Section* beinhaltet einen anderen Teil des Programms. Die folgende Liste gibt eine 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 1.3
- `.stabstr` Enthält die STABS Debuginformationen. Mehr dazu im Kapitel 1.3

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

1.2.1 Informationen für den Debugger

Zusätzliche Informationen für den Debugger werden ebenfalls im ELF Format gespeichert. Moderne Compiler verwenden hauptsächlich das DWARF-Format und nicht das veraltete STABS-Format. Trotzdem wird von aktuellen Compilern und auch Debuggern das veraltete STABS-Format immer noch unterstützt.

DWARF ist flexibler und hat einen besseren funktionalen Umfang als das STABS-Format, aber die manuelle Implementation ist aufwändiger.

1.3 STABS

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

1.3.1 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.

1.3.2 Aufbau des STABS Formats

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

[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

Kessler als Erfinder genannt.

Der Aufbau dieses Formats 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.

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 hat jede Direktive einen bestimmten Typ. Der Typ definiert, was die einzelnen Direktiven genau beschreiben. Um die Leserlichkeit zu verbessern sind alle Typen in der Datei *"stabs.include"* (Siehe Anhang **??**) definiert. Im Kapitel 12 der Dokumentation der *"University of Utha"* sind die einzelnen Typen genau 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
```

1.3.3 DWARF

1.4 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.

[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

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.

1.4.1 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 disassembliert Programm, 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 Sourcecode des Java-Programms in die Reset-Methode des *deep*-Kernel kopieren.

b) Den modifizierten Kernel mit *deep* übersetzen.

c) **loopMachineCode.txt**: Enthält den Maschinen-Code aus der *ClassTreeView* von *deep*.

d) **loop.S**: Der aus *"loopMachineCode.txt"* abgeleitete Assembler-Code.

e) **loopWithSTABS.S**: Der Assembler-Code inklusive den manuell ergänzten STABS.

f) **loopWithSTABS.o**: Kompiliertes Objekt aus dem Assembler-Code.

g) **loopWithSTABS**: Gelinktes Objekt aus dem kompilierten Objekt.

h) **loopWithSTABS.Sd**: Das disassemblierte Programm, um die STABS in einer leserlichen Form zu erhalten.

1.4.2 Java Demoprogramm

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

loop.java:

```
1  static void reset() {
2
3
4
5    US.PUTGPR(SP, stackBase + stackSize - 4); // set stack pointer
6
7    int x00 = 0;
8    int x01 = 1;
9    int x02 = 2;
10
11   x00++;
12   x01++;
```

```
13      x02++;
14
15      int x100 = 100;
16      for(int i=0; i<10; i++){
17        x100 += 10;
18       }
19
20      x100++;
21      x100++;
22      x100++;
23      x100++;
24      x100++;
25
26      US.ASM("b -8"); // stop here
27  }
```

In diesem Beispiel wird die `reset()`-Methode genutzt, da sie bei *deep* als erstes beim Booten ausgeführt wird. ''`US.PUTGPR`'' in Zeile 5 ist natürlich keine Java Methode. Da Low-Level-Operationen, wie die Initialisierung des Stackpointers, mit Java normalerweise nicht möglich sind, wird hier die entsprechende *deep*-Instruktion verwendet.

1.4.3 Beispiel-Programm "loopExample"

Der Code in *"loopExample.c"* im Anhang **??** ist fast identisch mit dem Code des Java Demoprogramms. Es wurden nur einige Änderungen gemacht, damit der Code als C-Programm kompiliert werden kann. `c_entry()` ist der Eintrittspunkt des Programms und erfüllt im embedded Bereich eine ähnliche Aufgabe wie die `main()`-Methode in einem generischen C-Programm.

Mit dem PowerShell-Script *"make_loopExample.ps1"* im Anhang **??** kann das C-Programm kompiliert werden. Es erzeugt das Object-File *"loopExample.o"* inklusive Debuginformationen im STABS Format. Das disassemblierte Object-File wird als *"loopExample.Sd"* gespeichert. Im disassemblierten Object-File sind alle STABS-Informationen und auch der ausführbare Code als Assembler enthalten. Der Assembler-Code und auch die STABS-Informationen können direkt *"human readable"* gelesen

werden, aber sie können nicht direkt in einem kompilierbaren Programm verwendet werden, da die Syntax nicht übereinstimmt.

Beispiel mit disassemblierter Syntax:

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

Kompilierbare Assembler Syntax:

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

1.4.4 Analyse der disassemblierten STABS

Die untenstehenden Direktiven sind ein Auszug aus der Datei *"loopExample.Sd"* im Anhang **??**. Die Tabelle 1.1 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 1.1: 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.

1.4.5 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 Star-

ten 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 1.2) von *deep* kann der Assemblercode der "`reset()`"-Methode kopiert werden welcher im Anhang **??** angehängt ist.

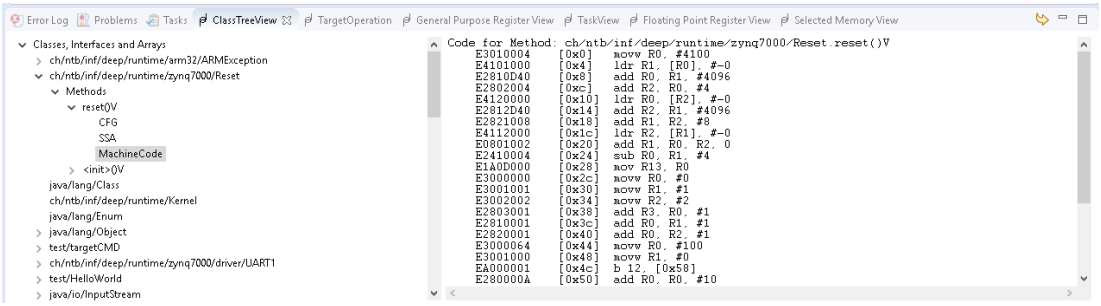

Abbildung 1.2: ClassTreeView mit Maschinencode der Reset-Methode in *deep*

loop.S:

```
1  .global _start
2
3  .org 0x000000
4  .text
5  Ltext0:
6
7  _start:
8  _reset:
9  c_entry:
10 movw R13, #1024
11
```

```
12  movw R0, #0
13  movw R1, #10
14  movw R2, #20
15  add R3, R0, #1
16  add R0, R1, #1
17  add R0, R2, #1
18  movw R0, #100
19  movw R1, #0
20  b CHECK_LOOP_EXIT
21  START_LOOP_BODY:
22  add R0, R0, #10
23  add R1, R1, #1
24  CHECK_LOOP_EXIT:
25  cmp R1, #10
26  blt START_LOOP_BODY
27  add R1, R0, #1
28  add R0, R1, #1
29  add R1, R0, #1
30  add R0, R1, #1
31  add R1, R0, #1
32  b 0
```

"loop.S" im Anhang **??** enthält den "aufgeräumten" Assemblercode. Der Code wurde mit zusätzlichen Assembler-Direktiven ergänzt. ''`c_entry`'' beschreibt den 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.

1.4.6 STABS in das Assemblerprogramm einfügen

Um das Assemblerprogramm mit STABS zu ergänzen wurden drei verschiedene Quellen genutzt. Das Fertige Assemblerprogramm mit STABS ist im Anhang **??** angehängt.

Die NTB-Wiki-Dokumentation[7] wurde als Ausgangslage genutzt. Die Datei *"stabs.include"* (siehe Anhang **??**) konnte direkt genutzt werde. Die Definition des Sourcecode (N_SO) und die Definitionen der Zeilennummern (N_SLINE) konnte ebenfalls übernommen werden.

Da die Definitionen der Variablen-Typen in der NTB-Wiki-Dokumentation nicht vollständig war, konnte sie leider nicht verwendet werden. Alle Variablendefinitionen, Zeile 6-75, wurde aus dem disassemblierten Demoprogramm kopiert. Bei der While-Loop ist die Definition der Sourcecode-Zeile ebenfalls etwas speziell, da sie auch bei der Überprüfung der Exit-Condition stimmen muss. Die genaue Implementation für die While-Loop wurde ebenfalls aus dem disassemblierten Demoprogramm übernommen.

Der *deep*-Compiler scheint die Variablen nicht auf dem Stack zu sichern, sofern noch genügend Register frei sind. Zusätzlich werden die Variablen in den Registern überschrieben, wenn diese im späteren Programmverlauf nicht mehr verwendet werden. Eine Register-Variable wird mit einer Direktive vom Typ 'N_RSYM' definiert, die auf ein bestimmtes Register zeigt. So werden beispielsweise die Register-Variablen `x00`, `x01` und `x02` in den Zeilen 84, 89 und 94 definiert.

```
84  .stabs "x00:r(0,1)",N_RSYM,0,4,0

89  .stabs "x01:r(0,1)",N_RSYM,0,4,1

94  .stabs "x02:r(0,1)",N_RSYM,0,4,2
```

Auf der Sourcecode-Zeile 11 wird die Variable `x00` um 1 inkrementiert. Im Assemblercode sieht man, dass die Variable neu im Register 3 abgespeichert wird. Aus diesem Grund muss die Register-Variable neu definiert werden.

```
98  # x00++;
99  .stabn N_SLINE, 0, 11, LM11
100 .stabn N_LBRAC, 0, 0, LM11
101 .stabs "x00:r(0,1)",N_RSYM,0,4,3
```

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

```
102 LM11:
103 add R3, R0, #1
```

1.4.7 Demoprogramm mit STABS kompilieren

Das Assemblerprogramm enthält nun alle notwendigen Informationen für den Maschinencode in Form von Assemblerinstruktionen. Die STABS ergänzen das Programm mit allen Informationen, welche der Debugger benötigt.

Mit dem Script *"make_loop.ps1"* im Ahang **??** kann das Programm assembliert werden. Die ELF-Datei *"loopWithSTABS"* kann dann mit dem *gdb* geladen werden.

2 Der *gdb*-Debugger

Es gibt diverse Debugger auf dem Markt. Diese Arbeit beschränke sich aber auf den *gdb* (GNU-Debugger), da dieser unter der GNU GPL (General Public License) Lizenz steht und somit eine Open Source Software ist.

In diesem Kapitel wird beschrieben, wie der *gdb* installiert und genutzt werden kann, um das Demoprogramm aus dem Kapitel 1.4 auf den Zynq zu laden.

2.1 Installation des *gdb*

ARM stellt eine komplette *"GNU Embedded Toolchain"* für ARM Prozessoren zur Verfügung. Sie enthält neben dem GCC Compiler und dem *gdb* auch noch diverse Hilfsprogramme wie *"readelf"* und *"objdump"*. Für diese Arbeit wird die *"GNU Arm Embedded Toolchain: 7-2018-q2-update"* Toolchain verwendet, die von der ARM-Webseite[1] heruntergeladen werden kann. Sobald das Archiv auf der lokalen Festplatte entpackt wird, ist die Toolchain einsatzbereit. Bei den Build-Scripten in dieser Arbeit muss jeweils die *"PATH"*-Variable mit dem Pfad zur Toolchain ergänzt werden, damit die Toolchain vom Script gefunden wird.

2.2 *gdb*-Anwendungsbeispiel mit *"loopWithSTABS"* auf dem Zybo

Mit folgenden Schritten kann das kompilierte Programm *"loopWithSTABS"* aus dem Kapitel 1.4 auf den Zynq geladen und debuggt werden:

1. Die notwendige Software, wie im Kapitel **??** beschrieben, installieren.

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

2. Das Zybo per USB-Kabel mit dem PC verbinden.

3. OpenOCD in der Shell mit dem Befehl *"openocd -f zybo-ftdi.cfg -f zybo.cfg"* starten. Dazu müssen sich die beiden Konfigurationsdateien *"zybo-ftdi.cfg"* und *"zybo.cfg"* (siehe Anhang **??** und Anhang **??**) im gleichen Ordner wie das *"openocd"*-Binary befinden.

4. In einer zweiten Shell *gdb* starten. Dazu kann das Shell-Script *"startGdb.ps1"* aus dem Anhang **??** genutzt werden. Die Pfade im Script müssen angepasst werden. Die Konfigurationsdatei *"gdbInit.txt"* (siehe Anhang **??**) muss im aktiven Ordner vorhanden sein. Alle Pfade in der Konfigurationsdatei müssen ebenfalls angepasst werden.

5. Im *"gdbInit.txt"* wir die ELF-Datei *"loopWithSTABS"* mit der Instruktion ''`file M:/MA/stabs/loopWithSTABS`'' in den *gdb* geladen. Die Instruktion ''`load`'' lädt dann das Segment *".text"* mit dem ausführbaren Code direkt in den Speicher des Zynq.

6. Die Applikation kann jetzt mit dem *gdb* auf dem Zybo debuggt werden.

2.3 Test der *gdb*-Funktionen

In diesem Kapitel werden alle aus dem Kapitel 1.4 geforderten Funktionen getestet. Als Ausgangspunkt dient das Anwendungsbeispiel aus dem Kapitel 2.2. *gdb* kann mit dem Befehlt ''q'' beendet und dann neu gestartet werden, damit die Ausgangslage bei jedem Test identisch ist.

Für die bessere Übersicht wird hier nochmals der Java-Code des Demo-Programms *"loop.java"* aufgelistet:

```
1  static void reset() {
2
3
4
5    US.PUTGPR(SP, stackBase + stackSize - 4); // set stack pointer
6
7    int x00 = 0;
8    int x01 = 1;
```

```
9     int x02 = 2;
10
11    x00++;
12    x01++;
13    x02++;
14
15    int x100 = 100;
16    for(int i=0; i<10; i++){
17      x100 += 10;
18     }
19
20    x100++;
21    x100++;
22    x100++;
23    x100++;
24    x100++;
25
26    US.ASM("b -8"); // stop here
27  }
```

2.3.1 Durchführung des *gdb*-Tests

Mit ”`list`” kann der Sourcecode des Programmes angezeigt werden. ”`list 10`” zeigt den Sourcecode ab der 10. Zeile an. Ein Hardware-Breakpoint auf Zeile 11 kann mit ”`hbreak 11`” erstellt werden. Wird das Programm mit ”`c`” gestartet, dann wird die Ausführung gestoppt, sobald die 11. Zeile des Sourcecodes erreicht wurde. *gdb* zeigt dann an, dass die nächste Zeile ”`x00++;`” sein wird.

Mit ”`p x00`” wird der Inhalt der Variable ”`x00`” angezeigt. Führt man mit ”`s`” einen einzelnen Step, also eine Zeile im Sourcecode aus, dann erhöht sich der Wert der Variable ”`x00`” um 1. Das kann mit ”`p x00`” wieder überprüft werden.

Ein weiterer Hardware-Breakpoint auf Zeile 17 (”`break 17`) stoppt das Programm innerhalb der For-Loop. Die Variable ”`i`” zeigt zu diesem Zeitpunkt wie erwartet ”0”. Wird das Programm fortgesetzt, dann stoppt das Programm wieder auf der Zeile 17 und ”`i`” zeigt ”1”. Die Variable ”`i`” kann mit

''`set var i=9`'' gesetzt werden. Da mit "i=9" die Abbruchbedingung der For-Loop erfüllt ist, wird der Breakpoint nicht mehr erreicht, wenn das Programm weiter ausgeführt wird. Das Programm hängt jetzt auf der letzten Zeile des Programms fest, und kann mit der Tastenkombination *ĈTRL + C* gestoppt werden.

Das Schlüsselwort ''`monitor`'' kann genutzt werden, um OpenOCD aus dem *gdb* heraus direkt ein Befehl zu erteilen. So kann mit ''`monitor reg`'' der OpenOCD-Befehl ''`reg`'' genutzt werden, um alle Register anzuzeigen.

Hinweis: Seit der *gdb*-Version 8 funktionieren Software-Breakpoints (z.B ''`break 12`'') nicht mehr. Bei einem Software-Breakpoint wird eine Instruktion mit einer speziellen Instruktion ersetzt, die dann das Programm stoppt und den Debugger triggert. Das funktioniert bei allen *gdb*-Versionen. Ab der *gdb*-Version 8 wird diese Instruktion aber nicht mehr mit der alten, gültigen Instruktion ersetzt. Aus diesem Grund kann dann das Programm nicht mehr weiter ausgeführt werden. Die Hardware-Breakpoints funktionieren bei allen Versionen.

2.3.2 Fazit des *gdb*-Tests

Alle geforderten Funktionen des Debuggers können grundsätzlich genutzt werden.

Bei *gdb*-Versionen die neuer als Version 8 können aber nur die Hardware-Breakpoints verwendet werden. Software-Breakpoints könnten aber auch verwendet werden, wenn die ersetzte Instruktion manuell wieder hergestellt wird.