# Praktikum 05 - Assembler

Malte Riechmann, André Kirsch

### Aufgabe 1

|  |  |
| --- | --- |
| Option | Erklärung |
| -o0 | Standard Option. Reduziert Kompilierungszeit und ermöglicht Debugging (liefert dabei das erwartete Ergebnis). |
| -o2 | Führt fast alle möglichen Optimierungen durch. Verlängert dabei aber die Kompilationszeit, aber erhöht die Performance des generierten Codes. |
| -os | Führt die gleichen Optimierungen wie -o2 durch, bis auf solche, die normalerweise die Code Größe erhöhen. Außerdem werden weitere Optimierungen durchgeführt, um die Code Größe zu reduzieren. |

Um auf die TivaC-Toolchain über die Kommandozeile zugreifen zu können, haben wir dessen Installationspfad in die Windows Umgebungsvariable PATH eingetragen. Die ermittelte Version ist 6.3.1 20170620.

Nachdem wir die Schritte ausgeführt haben, haben wir heraus gefunden, dass die Größe des Programmcodes 888 Bytes groß ist, die uninitialisierten Variablen belegen 260 Byte. Das ergibt eine Gesamtgröße von 1148 Byte.

text data bss dec hex filename  
 888 0 260 1148 47c blink.elf

Die Spalten des Disassembly haben folgende Bedeutung

|  |  |  |  |
| --- | --- | --- | --- |
| Addresse des Befehls in Hexadezimalem Format | Der Befehl in Maschinensprache in Hexadezimalem Format | Der Befehl in Assemblersprache (Lesbare Repräsentation der Maschinensprache) | Optionaler Kommentar |
| 2ae: | 4b0d | ldr r3 [pc, #52] | ; (2e4 <main+0x48>) |

Im Folgenden werden die Anweisungen von 2ae bis 2da erläutert

2ae: 4b0d ldr r3, [pc, #52] ; (2e4 <main+0x48>)  
 2b0: 2208 movs r2, #8  
 2b2: 601a str r2, [r3, #0]

Die ersten drei Befehle entsprechen der Anweisung GPIO\_PORTF\_DIR\_R = 0x08; . Die erste Zeile lädt über den Befehl ldr einen Wert aus dem Speicher in das Register r3. Der Wert, der in das Register geladen wird ist relativ zum Wert, der im Program-Counter steht angegeben. Die so berechnete Adresse befindet sich am Ende der Main-Methode und stellt quasi einen konstanten Wert dar, der in diesem fall als später als Speicheradresse interpretiert wird.

In der zweiten Zeile wird über movs der Wert 8 in das Register r2 geschrieben.

Anschließend wird in der dritten Zeile der Wert aus Register r2 (8) in den Speicher geschrieben. Dies geschieht über den str Befehl. Dieser Berechnet aus dem Wert, der in Register r3 steht (in Schritt 1 aus dem Speicher geladen) und dem Offset, in diesem Fall 0, die Zieladresse und schreibt den Wert aus r2 in diesen Spiecher.

2b4: 4b0c ldr r3, [pc, #48] ; (2e8 <main+0x4c>)
  
 2b6: 2208 movs r2, #8
  
 2b8: 601a str r2, [r3, #0]

Der zweite Abschnitt entspricht der Anweisung GPIO\_PORTF\_DEN\_R = 0x08;. Dieser Abschnitt läuft analog zu dem eben beschrieben Ablauf ab, mit dem Unterschied, dass bei ldr eine andere Adresse zum lesen angeben ist und somit bei str in eine andere Adresse geschrieben wird.

2ba: 4a0c ldr r2, [pc, #48] ; (2ec <main+0x50>)
  
 2bc: 4b0b ldr r3, [pc, #44] ; (2ec <main+0x50>)
  
 2be: 681b ldr r3, [r3, #0]
  
 2c0: f043 0308 orr.w r3, r3, #8
  
 2c4: 6013 str r3, [r2, #0]

Dieser Abnschnit entsricht der GPIO\_PORTF\_DATA\_R |= 0x08; Anweisung. Zunächst wird hierbei zwei mal der gleiche Wert über ldr aus dem Speicher geladen und in die Register r2 und r3 geschrieben. Im nächsten Schritt wird dieser in r3 geladene Wert wieder als Speicheradresse interpretiert und über ldr wird dessen Wert in das Register r3 geladen.

Der folgende Befehl orr.w führt nun eine bitweise Oder-Verknüpfung mit diesem Wert aus r3 und der Zahl 8 aus und schreibt das Ergebnis in r3. Das .w bedeutet, dass es sich um eine 32-bit codierte Anweisung handelt.

Danach wird der nun in r3 stehtende Wert über str an die in r2 stehende Adresse geschrieben.

2c6: f7ff ffd1 bl 26c <delay>

Der nun folgende Abschnitt entspricht dem delay(); Aufruf. Über bl wird eine Subroutine *delay* gestartet mit einem Sprung an die Adresse 26c. Der aktulle Wert des Program-Counters wird in das Link Register geschrieben und in den Program-Counter wird der Wert 26c geladen. Am Ende der Subroutine wird der im Link Register gespeicherte Wert genutzt um wieder an die richtige Stelle im Programm zu springen. Der Sprung ist an keine Bedingung genküpft und wird somit immer ausgeführt.

2ca: 4a08 ldr r2, [pc, #32] ; (2ec <main+0x50>)
  
 2cc: 4b07 ldr r3, [pc, #28] ; (2ec <main+0x50>)
  
 2ce: 681b ldr r3, [r3, #0]
  
 2d0: f023 0308 bic.w r3, r3, #8
  
 2d4: 6013 str r3, [r2, #0]

Der Abschnitt entspricht der GPIO\_PORTF\_DATA\_R &= ~(0x08); Anweisung. Die ersten drei Schritte sind analog zu den von 2ba bis 2be. Es wird bei den ersten beiden ldr auch wie da der gleiche Wert geladen. Das der Offset ein anderer ist liegt daran, dass der Program-Counter, von dem aus die Adresse berechnet wird, größer geworden ist.

Über bic wird nun der wieder in r3 geladene Wert bitweise Und-Verknüft mit der Negation der Zahl 8 und das Ergebnis wird wieder in r3 geschrieben. Das .w zeigt wieder an, dass es sich um eine 32-bit codierte Anweisung handelt.

Anschließend wird wieder der eben errechnete Wert aus r3 in die, in r2 stehende, Adresse geschrieben.

2d6: f7ff ffc9 bl 26c <delay>

Dieser Abschnitt enstpricht wieder dem delay() Aufruf und läuft Analog zum letzten mal ab.

2da: e7ee b.n 2ba <main+0x1e>

Dieser Abnschnitt entspricht der while(1) Schleife. Über b.n wird ein bedingungsloser Sprung zu 2ba. Das .n bedeutet dabei, dass es sich um ein 16-bit encoding handel.

### Aufgabe 2

text data bss dec hex filename  
 844 0 260 1104 450 blink.elf

Bei den Größen lässt sich erkennen, dass der Teil des Programmcodes um 44 Byte geschrumpft ist. Hier konnte der Compiler also durch die Optimierung den Programmcode ein wenig komprimieren.

Sofort erkennbare Änderungen im Assembler Code sind zum einen die komplett fehlenden Delay Funktion. Diese ist mit in die Main Funktion gerutscht. Als Folge davon gibt es aber in der Main Funktion wesentlich mehr Sprünge. Desweiteren werden alle Speicherstellen, die abhängig vom Programmcounter sind, zu Beginn der Main Funktion in die Register geladen. Dadurch wurde die Anzahl der gesamten ldr Aufrufe verringert und es wird eine größere Anzahl an Registern verwendet.

|  |  |
| --- | --- |
| Codestelle | Beschreibung |
| ab 26c | Vorladen aller wichtigen Adressen |
| ab 278 | Anweisungen vor der while-Schleife |
| ab 288 | Beginn der while-Schleife und bitweise OR-Anweisung |
| ab 290 | Erste Delay-Funktion |
| ab 2a4 | Bitweise AND-Anweisung |
| ab 2ac | Zweite Delay-Funktion |
| ab 2c0 | Sprung zurück zu Beginn der while-Schleife |

### Aufgabe 3

Das Wort volatile bedeutet so viel wie: "Der Compiler darf an dieser Stelle nicht optimieren". Das bedeutet, dass jeder Programmcode, in den diese Variable genutzt wird, vom Compiler nicht optimiert wird.

Durch das fehlende volatile konnte der Compiler nun also auch die Stellen optimieren, die in Aufgabe 2 nicht optimiert werden durften. Dadurch hat er in diesem Fall den zweiten Aufruf der Funktion delay komplett aus dem Programmcode entfernt. Die erste Delay Funktion wurde verkürzt auf die zwei Befehle an Stelle 290 und 292. Hier wird nun der Wert in r3 bis auf 0 dekrementiert.

Durch das fehlen des zweiten Delays würde das Programm nicht mehr wie gewünscht arbeiten. Es würde so wirken, als wäre die LED dauerhaft an, da quasi direkt nach dem Ausschalten das Anschalten folgt.

Der restliche Programmcode wurde ähnlich compiliert, es unterscheidet sich lediglich in der Nutzung der verwendeten Register, wodurch teilweise aus anderen Registern geladen und in andere Register geschrieben wird.