Embedded Systems

Aufgabenblatt 5, Assembler

Andre Matutat, Jonas Posselt

1. Juni 2018

Aufgabe 1

Compiler Optionen

-O0 Standarteinstellung. Der Compiler führt keine Optimierung durch und verringert so die Kompilierzeit und erlaubt sicheres debuggen.

-O2 Der Compiler führt fast alle Optimierungen durch bei denen kein Kompromiss zwischen Speicherplatz und Geschwindigkeit besteht. Erhöht die Performanz des erzeugten Codes aber auch die Kompilierzeit

-Os Verringert die Größe des Programms. Enthält alle -O2 Optimierungen, die den Code nicht vergrößern und führt weitere Optimierungen zur Reduzierung der Größe durch.

Version

Die Ausgabe von `arm-none-eabi-size blink.elf` ist

```
arm-none-eabi-gcc (GNU Tools for ARM Embedded Processors 6-2017-q2-update)
6.3.1 20170620 (release) [ARM/embedded-6-branch revision 249437]
```

Die Version des Compilers ist also `6.3.1`.

Größe

Der Aufruf von `arm-none-eabi-size blink.elf` liefert

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

Dabei gibt `text` die Größe des Programmcodes im Flash-Speicher an während `data` und `bss` (Block Started by Symbol) die Größe von initialisierten und uninitialisierten Daten. Die Spalten `dec` und `hex` sind die Gesamtgröße (`text`+`data`+`bss`) in dezimaler und hexadezimaler Schreibweise. Alle Angaben sind in Einheiten von Bytes.

Disassembly Spalten

1. Relative Position der aktuellen Zeile im Code als hexadezimaler Adressen-Offset (zum Anfang des Codes).
2. Opcode des auszuführenden Befehls (hexadezimalkodiert)
3. Auszuführender Befehl als Assembler-Mnemonic + Operanden
4. Kommentar (optional)

Disassembly

Die ersten 6 Zeilen (Listing 1) entsprechen den Aweisungen

```
GPIO_PORTF_DIR_R = 0x08;
GPIO_PORTF_DEN_R = 0x08;
```

im C-Quelltext. Dazu wird jeweils mit dem Befehl `ldr` (load register) ein Wert aus dem Speicher in Register 3 geladen. Die Speicheradresse des zu ladenden Werts ist dabei relativ zum Programmzähler (pc + Offset) gegeben. Die hier verwendeten Adressen befinden sich am Ende der Main-Routine und die dort hinterlegten Werte stellen für das Programm Konstanten dar.

Im nächsten Schritt wird mit `movs` (move) der Wert 8 in Register 2 kopiert und basierend auf dem Ergebnis die N und Z flags des Statusregisters aktualisiert. Zuletzt wird mit `str` (store register) der Wert in Register 2 zurück an die in Register 3 enthaltene Speicheradresse geschrieben (Offset ist 0, also wird direkt die Adresse verwendet).

```
1 2ae:   4b0d        ldr     r3, [pc, #52]   ; (2e4 <main+0x48>)
2 2b0:   2208        movs    r2, #8
3 2b2:   601a        str     r2, [r3, #0]
4 2b4:   4b0c        ldr     r3, [pc, #48]   ; (2e8 <main+0x4c>)
5 2b6:   2208        movs    r2, #8
6 2b8:   601a        str     r2, [r3, #0]
```

Listing 1: GPIO Konfiguration

Die nächsten Zeilen (Listing 2) entsprechen der Anweisung

```
GPIO_PORTF_DATA_R |= 0x08;
delay();
```

also dem Anschalten der LED und Aufruf der delay-Funktion. Die ersten beide `ldr`-Befehle laden den selben Wert in die Register 2 und 3 (wieder relative Adressierung pc + Offset). Der dritte Befehl lädt einen Wert aus dem Speicher in Register 3 wobei die zuvor in Register 3 gespeicherte Adresse als Quelle dient. Der Befehl `orr.w` (bit set in 32-bit Kodierung durch die .w-Option) führt eine bitweise Oder-Verknüpfung des Werts in Register 3 und dem Literal-Wert 8 aus und schreibt das Ergebnis wieder in Register 3. Der neue Wert in Register 3 wird dann mit `ldr` zurück an die in Register 2 enthaltene Speicheradresse geschrieben. Zuletzt wir mit `bl` (branch with link) ein bedingungsloser Sprung zum Offset `26c` an dem sich der Start der delay-Subroutine befindet.

```
 7 || 2ba:    4a0c            ldr     r2, [pc, #48]    ; (2ec <main+0x50>)
 8 || 2bc:    4b0b            ldr     r3, [pc, #44]    ; (2ec <main+0x50>)
 9 || 2be:    681b            ldr     r3, [r3, #0]
10 || 2c0:    f043 0308       orr.w   r3, r3, #8
11 || 2c4:    6013            str     r3, [r2, #0]
12 || 2c6:    f7ff ffd1       bl      26c <delay>
```

Listing 2: LED an + Warten

Die Befehlsfolge in Listing 3 ist weitgehend analog zu Listing 2 und entspricht

```
GPIO_PORTF_DATA_R &= ~(0x08);
delay();
```

also dem Abschalten der LED und Aufruf der delay-Funktion. Einziger Unterschied ist, dass mit `bic.w` (Bit Clear mit 32-bit Kodierung) eine bitweise Und-Verknüpfung von Register 3 und dem Komplement des Literals 8 erfolgt.

```
13 || 2ca:    4a08            ldr     r2, [pc, #32]    ; (2ec <main+0x50>)
14 || 2cc:    4b07            ldr     r3, [pc, #28]    ; (2ec <main+0x50>)
15 || 2ce:    681b            ldr     r3, [r3, #0]
16 || 2d0:    f023 0308       bic.w   r3, r3, #8
17 || 2d4:    6013            str     r3, [r2, #0]
18 || 2d6:    f7ff ffc9       bl      26c <delay>
```

Listing 3: LED aus + Warten

Als letzter Schritt (Listing 4) wird mit `b.n` (branch mit 16-Bit Kodierung durch .n-Option) ein bedingungsloser Sprung zum Offset `2ba` (siehe Listing 2) durchgeführt, was der Rückkehr zum Anfang der Endlosschleife im C-Code entspricht.

```
19 || 2da:    e7ee            b.n     2ba <main+0x1e>
```

Listing 4: Rücksprung Endlosschleife

Aufgabe 2

Größe

Für die Größe ergibt sich hier

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

also 44 Bytes weniger als zuvor.

Disassembly

Ein Unterschied im Vergleich zur Kompilierung mit `-O0` ist, dass gleich zu Beginn der main-Routine alle benötigten Konstanten in verschiedene Register geladen werden (Listing 5).

```
1 26c:   4a15        ldr   r2, [pc, #84]   ; (2c4 <main+0x58>)
2 26e:   4b16        ldr   r3, [pc, #88]   ; (2c8 <main+0x5c>)
3 270:   4c16        ldr   r4, [pc, #88]   ; (2cc <main+0x60>)
4 272:   4d17        ldr   r5, [pc, #92]   ; (2d0 <main+0x64>)
5 274:   4817        ldr   r0, [pc, #92]   ; (2d4 <main+0x68>)
6 276:   4918        ldr   r1, [pc, #96]   ; (2d8 <main+0x6c>)
```

Listing 5: Konstanten laden

Der zweite wesentlich Unterschied ist, dass keine delay-Subroutine mehr existiert und statt dessen die Warte-Schleifen direkt implementiert sind (Listing 6). Die Implementierung der Schleife schreibt zunächst mit `str` den Wert 0 (Inhalt von Register 4 an dieser Stelle) an die Speicher-Adressen in Register 3 und lädt diesen Wert dann mit `ldr` in Register 2. Der `cmp`-Befehl (compare) vergleicht den Wert in Register 2 (0) mit dem Wert in Register 1 (199999; als Konstante in Listing 5 geladen) und aktualisert dem Ergebnis entsprechend die Bits im Statusregister.

Die Anweisung `bhi.n` (branch higher mit 16-bit Kodierung) überspringt die restlichen Zeilen des obigen Listings wenn der Wert in Register 2 größer als der in Register 1 ist. Dies sollte bei normalem Programmablauf aber nie der Fall sein.

Die nächsten drei Zeilen (Offset `298` - `29c`) laden den Wert von der Adresse in Register 3, inkrementieren den Wert um 1 mit `adds` (Addition ohne Übertrag mit Update des Statusregisters) und schreiben das Ergebnis wieder an die ursprüngliche Adresse. Dieser Wert wird dann wieder geladen (Offset `29e`), und mit dem Wert in Register 1 verglichen (Offset `2a0`, Wert in Register 1 immer noch 199999). Zuletzt wird mit `bls.n` (branch less or same, 16-bit Kodierung) entschieden ob zum Offset 298 zurückgesprungen und der durch Register 3 referenzierte Wert nochmals erhöht wird (falls Wert $\leq$ 199999).

Insgesamt inkrementiert die Befehlsfolge also, wie erwartet, eine Zählvariable von 0 bis 200000 um so die Ausführung der nachfolgenden Anweisungen zu verzögern. Da diese Implementierung eine Optimierung der Warteschleife darstellt, wird diese zur Laufzeit auch schneller durchlaufen wodurch die LED mit höherer Frequenz blinkt.

```
1 290:   601c        str   r4, [r3, #0]
2 292:   681a        ldr   r2, [r3, #0]
3 294:   428a        cmp   r2, r1
4 296:   d805        bhi.n 2a4 <main+0x38>
5 298:   681a        ldr   r2, [r3, #0]
```

```
 6 ||  29a:    3201          adds    r2, #1
 7 ||  29c:    601a          str     r2, [r3, #0]
 8 ||  29e:    681a          ldr     r2, [r3, #0]
 9 ||  2a0:    428a          cmp     r2, r1
10 ||  2a2:    d9f9          bls.n   298 <main+0x2c>
```

Listing 6: Inline Warteschleife

Aufgabe 3

Die Deklaration einer Variablen als `volatile` verhindert, dass der Compiler beim Laden (und Speichern) der Variablen aus dem Hauptspeicher Optimierungen durchführt. Lässt beim gegebenen Code `volatile` weg erzeugt der Compiler nur eine Warteschleife (nach dem Anschalten der LED), die zudem durch das dekrementieren einer Zählvariablen realisiert wird und kürzer ist als die vorherige Variante. Die zweite Warteschleife (nach dem Abschalten der LED) wird durch die „Optimierung“ entfernt, da diese, aus der Sicht des Compilers, keine erkennbare Funktion ausführt. Das führt hier aber dazu, dass die Funktionalität des Programms beeinträchtigt ist. Die LED bleibt dauerhaft angeschaltet, da zwischen dem Abschalten und dem erneuten Anschalten keine relevante Verzögerung existiert.