Praktikum 5 – Assembler

Jan Augstein, Michael Nickel

Aufgabe 1

Optimierungsoptionen:

- Ohne Optimierungsoptionen versucht der Compiler die Kompilierungskosten zu reduzieren und den Code zum erwarteten Ergebnis zu führen
- Wenn man dann die Optimierungsoptionen hinzufügt, versucht der Compiler die Performance und/oder die Code Größe auf Kosten der Kompilierungszeit zu verbessern

Set the compiler's optimization level.

option	optimization level	execution time	code size	memory usage	compile time
-O0	optimization for compilation time (default)	+	+	-	-
-O1 or -O	optimization for code size and execution time	-	-	+	+
-O2	optimization more for code size and execution time	--		+	++
-O3	optimization more for code size and execution time	---		+	+++
-Os	optimization for code size		--		++
-Ofast	O3 with fast none accurate math calculations	---		+	+++

+increase ++increase more +++increase even more -reduce --reduce more ---reduce even more

1: Optimierungsoptionen

- Hier sieht man nun welche Auswirkungen die verschiedenen Optimierungslevel auf die Ausführungszeit, die Code Größe, die Speichernutzung und die Kompilierzeit hat

- Die erste Spalte zeigt den Speicherort innerhalb des Programmspeicherplatzes an, die zweite Spalte ist der Maschinencode und die letzte Spalte zeigt den dazugehörigen Assembler Code an.

Assemblerbefehle:

- ldr: Lädt einen Register mit einem Wert vom Speicher
- str: Speichert ein Registerwert im Speicher
- Offset addressing (ldr/str): der Offset Wert wird auf die Adresse addiert/subtrahiert. Das Ergebnis wird als Adresse für den Speicherzugang genutzt. Das Register Rn bleibt unverändert
- Bei dieser Aufgabe konzentrieren wir uns auf die ldr Befehle mit sofortigem Offset, da nur diese bei uns vorkommen

Syntax

```
op{type}{cond} Rt, [Rn {, #offset}]        ; immediate offset
```

where:

op Is one of:
- LDR Load Register.
- STR Store Register.

type Is one of:
- B unsigned byte, zero extend to 32 bits on loads.
- SB signed byte, sign extend to 32 bits (LDR only).
- H unsigned halfword, zero extend to 32 bits on loads.
- SH signed halfword, sign extend to 32 bits (LDR only).
- - omit, for word.

cond Is an optional condition code, see *Conditional execution* on page 3-18.

Rt Specifies the register to load or store.

Rn Specifies the register on which the memory address is based.

offset Specifies an offset from *Rn*. If *offset* is omitted, the address is the contents of *Rn*.

Rt2 Specifies the additional register to load or store for two-word operations.

2: ldr und str

- bl: siehe Punkt 12.
- b: siehe Punkt 19.

Branch instructions.

Syntax

```
B{cond} label
BL{cond} label
BX{cond} Rm
BLX{cond} Rm
```

where:

B Is branch (immediate).

BL Is branch with link (immediate).

Operation

All these instructions cause a branch to *label*, or to the address indicated in *Rm*. In addition:

- The BL and BLX instructions write the address of the next instruction to LR (the link register, R14).

3: b und bl

- movs: kopiert den Wert von Operand2 in Rd (siehe Abbildung 3), S ist ein optionaler suffix welcher die condition flags als Resultat der Operation updated falls S spezifiziert ist

Move and Move NOT.

Syntax

`MOV{S}{cond} Rd, Operand2`

where:

S	Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-18.
cond	Is an optional condition code. See *Conditional execution* on page 3-18.
Rd	Specifies the destination register.
Operand2	Is a flexible second operand, see *Flexible second operand* on page 3-12 for details of the options.

4: movs

- orr: Bitweises Oder, Rd ist das Zielregister, Operation wird auf den Werten in Rn und Operand2 ausgeführt
- bic: bit clear, Und Operation mit den bits in Rn mit dem Komplement der zugehörigen bits in Operand2

Logical AND, OR, Exclusive OR, Bit Clear, and OR NOT.

Syntax

`op{S}{cond} {Rd,} Rn, Operand2`

where:

op	Is one of: AND logical AND. ORR logical OR, or bit set. EOR logical Exclusive OR. BIC logical AND NOT, or bit clear. ORN logical OR NOT.
S	Is an optional suffix. If S is specified, the condition code flags are updated on the result of the operation, see *Conditional execution* on page 3-18.
cond	Is an optional condition code, see *Conditional execution* on page 3-18.
Rd	Specifies the destination register.
Rn	Specifies the register holding the first operand.
Operand2	Is a flexible second operand. See *Flexible second operand* on page 3-12 for details of the options.

5: orr und bic

- 1. Der Wert aus dem PC (r15) wird mit einem Offset von 52 Bytes in r2 geladen
- 2. r2 bekommt den Wert 8 (r2 = 8), außerdem werden die condition flags geupdatet da beim mov Befehl der optionale suffix s gesetzt ist
- 3. Der Wert Wert aus r3 wird wird in der Adresse von r2 gespeichert
- 4. Der Wert aus dem PC wird mit einem Offset von 48 Bytes in r3 geladen
- 5. r2 bekommt den Wert 8 (r2 = 8), außerdem werden die condition flags geupdatet da beim mov Befehl der optionale suffix s gesetzt ist (siehe Begriffserklärung)
- 6. Der Wert aus r2 wird in der Adresse von r3 gespeichert

- 7. Der Wert aus r15 wird mit einem Offset von 48 Bytes in r3 geladen
- 8. Der Wert aus r3 wird in r3 geladen
- 9. Der Wert aus r15 wird mit einem Offset von 44 Bytes in r2 geladen
- 10. Hier findet ein bitweises Oder statt, also r3 = r3 | 8
- 11. Der Wert aus r3 wird in der Adresse von r2 gespeichert
- 12. Der bl Befehl (branch with link) sorgt dafür, dass der PC auf eine neue Adresse zeigt, und von dort die Befehle weitergehen, zusätzlich wird eine return Adresse in r14 geschrieben. In diesem Fall zeigt der PC also auf die Adresse 26c der <delay> Methode
- 13. Der Wert aus dem PC wird mit einem Offset von 32 Bytes in r3 geladen
- 14. Der Wert aus r3 wird in r3 geladen
- 15. Der Wert aus dem PC wird mit einem Offset von 28 Bytes in r2 geladen
- 16. Hier kommt es zu einem bitweisen Und mit den bits in r3 und dem Komplement von r3, jedoch werden nur die entsprechenden bits vom Wert 8 (00001000) in Betracht gezogen
- 17. Der Wert aus r3 wird in der Adresse von r2 gespeichert
- 18. Siehe 13.
- 19. B ist ein ist ein Jump, welchen man in etwa mit jmp vergleichen kann, hier wird also wieder in den Punkt 2ba in der main gesprungen

Aufgabe 2

```
Micha2@micha-PC MINGW64 ~/AppData/Local/Energia15/packages/energia/hardware/tiva
c/1.0.3/system
$ arm-none-eabi-size blink.elf
   text    data     bss     dec     hex filename
    888       0     260    1148     47c blink.elf
```

6: size -O0

```
Micha2@micha-PC MINGW64 ~/AppData/Local/Energia15/packages/energia/
c/1.0.3/system
$ arm-none-eabi-size blink.elf
   text    data     bss     dec     hex filename
    844       0     260    1104     450 blink.elf
```

7: size -O2

- text: Code Größe, data: Konstante Daten, bss: globale Variablen oder statisch allozierte Variablen, dec: Summe von text, data und bss als Dezimalzahl, hex: Summe als Hexadezimalzahl
- Wie man erkennen kann ist die Code Größe bei der Optimierungsoption -O2 kleiner
- Blink2.asm enthält im Disassembly der Sektion .text nun nicht mehr <delay>, dafür ist die <main> dann aber etwas länger
- Die anderen Bereiche <ResetISR>, <NmiSR>, <FaultISR>, <IntDefaultHandler> und <zeroLoop> sind auch kleiner geworden
- .init und .fini sind gleich geblieben
- Außerdem hat sich auch noch g_pfnVectors verändert, welcher die ganzen Handler beinhaltet

Aufgabe 3

```
Micha2@micha-PC MINGW64 ~/AppData/Local/Energia15/packages/energia/hardware/tiva
c/1.0.3/system
$ arm-none-eabi-size blink.elf
   text    data     bss     dec     hex filename
    804       0     260    1064     428 blink.elf
```

8: size ohne volatile

- Wie man erkennen kann ist der Code ohne volatile nochmal etwas geschrumpft
- Volatile sagt dem Compiler, dass der Inhalt einer Variablen sich auch außerhalb des normalen Programmflusses ändern kann, deshalb wird der Wert der Variable vor jedem Zugriff neu aus dem Hauptspeicher eingelesen
- Daher nimmt der Compiler mit dem Operator volatile keine Optimierungen vor, was den kleineren Code ohne volatile erklärt
- So kann es vorkommen, dass der Code nicht funktioniert wenn volatile nicht gesetzt ist und wir die Compileroptimierungen vornehmen, da der Compiler nicht erkennt dass die Variable global ist und so wo anders geändert werden kann und der Code dadurch „falsch“ optimiert wird

Quellen für die Abbildungen

Abbildung 1: https://www.rapidtables.com/code/linux/gcc/gcc-o.html

Abbildung 2/3/4/5: Cortex™-M4 Devices Generic User Guide