Rechnerorganisation
Sommersemester 2023 – 3. Vorlesung

Prof. Stefan Roth, Ph.D.

Technische Universität Darmstadt

8. Mai 2023

Inhalt

Organisation

Organisation

Zeitplanung Testate

- 1. Theorietestat: 15. Mai – 22. Mai 2023
- 2. Theorietestat: 22. Mai – 29. Mai 2023
- 1. Praxistestat: 29. Mai – 11. Juni 2022

Übungstestat verfügbar

- `https://moodle.tu-darmstadt.de/mod/quiz/view.php?id=1179448`

Konzepte der maschinennahen Programmierung

Erinnerung: Komponenten und Struktur eines Rechnersystems

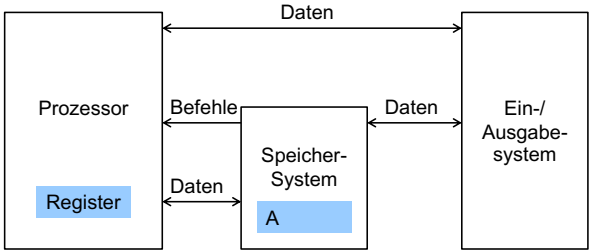


Abbildung: Komponenten eines Rechnersystems (verfeinerte Darstellung)

Programmiermodell des ARM®-Prozessors – Registersatz I

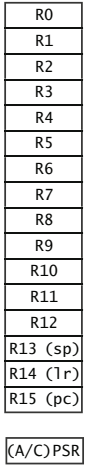


Abbildung: Quelle: DEN0024A_v8_architecture_PG. S. 4-14

Programmiermodell des ARM®-Prozessors – Registersatz II

- R0
- R1 .. R12
- R13 (sp) - stack pointer („Stapelzeiger“)
- R14 (lr) - link register (Rückkehradresse)
- R15 (pc) - program counter („Befehlszähler“)
- CPSR - Current Program Status Register (enthält u. a. Statusflags)

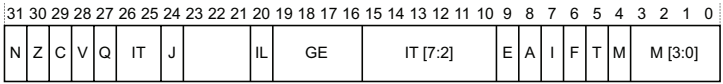


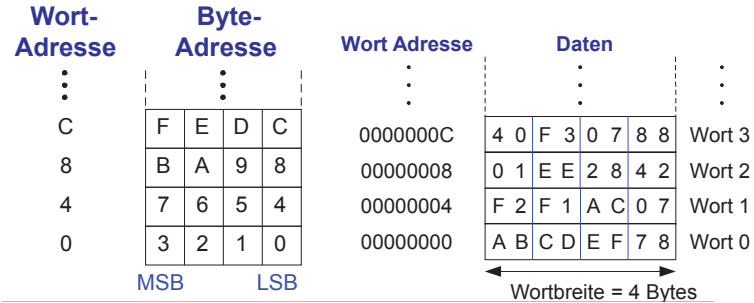
Abbildung: Quelle: DEN0024A_v8_architecture_PG. S. 4-16

Nutzung von Registern und Hauptspeicher

- Daten passen nicht alle in 13 Register (16 Register – R13, R14, R15)
- Lösung: Lege Daten im Hauptspeicher ab
- Hauptspeicher ist groß (GB . . . TB) und kann viele Daten aufnehmen, ist aber langsamer als die Register
- Speichere häufig verwendete Daten in Registern
- Kombiniere Register und Hauptspeicher zum Halten von Daten
  - Ziel: Greife schnell auf gerade benötigte Daten zu

Wort- und Byte-Adressierung von Daten im Speicher

- ARM® ist byte-adressiert, das heißt jedes Byte hat eine eindeutige Adresse
- 32-bit Wort = 4 Bytes
- Adressen von Worten sind also Vielfache von 4



Lesen aus byte-adressiertem Speicher

- Lesen geschieht durch Ladebefehle (load)
- Befehlsname: load register from memory (**ldr**)
- Beispiel

```
1 /* Hochsprache a = mem[2] */
2 /* ARM - Assemblersprache r7 = a */
3 mov r5, #0
4 ldr r7, [r5,#8]
```

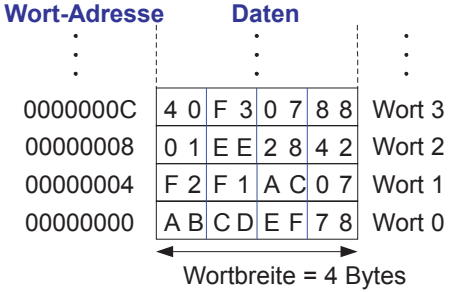
- Funktion: Lese ein Datenwort von der Speicheradresse (r5 + 8) und schreibe dieses in das Register r7
- Adressarithmetik: Adressen werden relativ zu einem Register angegeben
  - Basisadresse (r5) plus Distanz (offset) (8) $\Rightarrow$ Adresse = (r5 + 8)
- Ergebnis: r7 enthält das Datenwort von Speicheradresse (r5 + 8)
- Jedes Register darf als Basisadresse verwendet werden

Lesen aus byte-adressiertem Speicher

- Beispiel: Lese Datenwort 3 (Speicheradresse 0xC) nach r7
- Befehle:

```
1  mov r5,#0
2  ldr r7, [r5,#0xC]
```

- Adressarithmetik:
  - Basisadresse (r5) plus Distanz (offset) (0xC) $\Rightarrow$ Adresse = (r5 + 12) = 12
- Nach Abarbeiten des Befehls hat r7 den Wert 0x40F30788



Schreiben in byte-adressierten Speicher

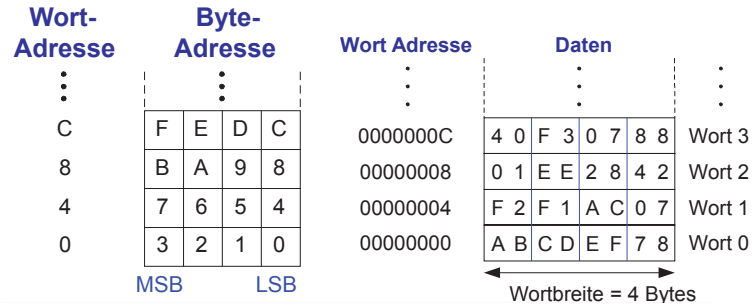
- Schreiben geschieht durch Speicherbefehle (store)
- Befehlsname: store register to memory (**str**)
- Beispiel

```
1 /* Hochsprache mem[5] = 42  */
2 /* ARM - Assemblersprache */
3 mov r1,#0
4 mov r9,#42
5 str r9, [r1,#0x14]
```

- Funktion: Schreibe (speichere) den Wert aus r9 in Speicherwort 5 (bzw. Speicheradresse 5x4 = 20 = 0x14)
- Adressarithmetik: Adressen werden relativ zu einem Register angegeben
  - Basisadresse (r1) plus Distanz (offset) (0x14) $\Rightarrow$ Adresse = (r1 + 20)
- Ergebnis: Nach Abarbeiten des Befehls enthält Speicheradresse (r1 + 20) das Datenwort von r9 (42)

Byte-addressierbarer Speicher

- Transportbefehle können auf Worten oder Bytes arbeiten
- Speichern: **`str`** / **`strb`**
- Laden: **`ldr`** / **`ldrb`**



Befehle eines Rechnersystems

- Mehrere Befehlsformate erlauben Flexibilität ...
- **`add`**, **`sub`**: verwenden z. B drei Register als Operanden
- **`ldr`**, **`str`**: verwendet zwei Register und eine Konstante als Operanden
- ... aber die Anzahl von Befehlsformaten sollte klein sein.
  - macht Hardware weniger aufwändig (und damit billiger)
  - erlaubt ggf. höhere Verarbeitungsgeschwindigkeit

Operanden: Konstante Werte in Befehlen (immediates)

- **ldr** und **str** zeigen die Verwendung von konstanten Werten (immediates)
  - Direkt im Befehl untergebracht, deshalb auch Direktwerte genannt
  - Brauchen kein eigenes Register oder Speicherzugriff
- Befehl **add** addiert Direktwert auf Register
- Direktwert ist Zweierkomplementzahl, die 12 Bit breit ist

```
1 /* Hochsprache */
2 a = a + 4;
3 b = a - 12;
```

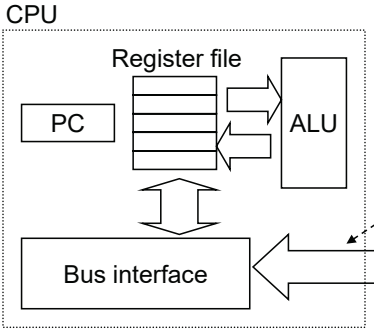
```
1 /* ARM - Assemblersprache */
2 /* r0 = a, r1 = b */
3 add r0, r0, #4
4 add r1, r0, #-12
```

Programmierung in Assembler

- Hochsprachen:
  - z. B. C, C++, Java, Python, Scheme
  - Auf einer abstrakteren Ebene programmieren
- Häufige Konstrukte in Hochsprachen:
  - if/else-Anweisungen
  - while-Schleifen
  - for-Schleifen
  - Array-Zugriffe
  - Unterprogramme
    - ⋆ Funktion
    - ⋆ Prozedur
    - ⋆ Rekursion
- Im Folgenden: Hochsprachenkonstrukte in Assembler
- Learning Nugget 02 - ARM Assembler Befehle

Programmierung in Assembler

- In Hochsprachen gibt es zahlreiche Konstrukte, mit denen der Programmablauf (Kontrollfluss) beeinflusst wird. Beispiel ist eine if/else-Anweisung.
- In einem Rechnersystem werden die Konstrukte zur Veränderung des Programmablaufs im Wesentlichen durch sogenannte **Statusflags** abgebildet.
- Statusflags sind Ergebnisse einer Rechnung (z. B. sub). Die Rechnungen werden in der ALU vorgenommen und Statusflags entsprechend dem Ergebnis gesetzt.



Statusflags

- Statusflags[1] (die wichtigsten) sind:
  - C – Carry flag (Übertragsflag)
  - Z – Zero flag (Nullflag)
  - N – Negative flag (Vorzeichenflag)
  - V – Overflow flag (Überlaufflag)
- Für was werden diese Flags verwendet?
  - Z: Hochsprache $\Rightarrow$ (t == 0)
  - N: Hochsprache $\Rightarrow$ (t $<$ 0)
- Erinnerung: beim ARM®-Prozessor werden die Flags im CPSR[2] abgespeichert.

[1] Statusbits
[2] Current Program Status Register

Berechnung von Statusbits I

- Ein paar Beispiele: Unterschied zwischen Carry und Overflow
- Angenommen: 2K-Darstellung. Die Zahlen sind 4 Bit breit. Der Zahlbereich ist also von -8 bis +7

```
   0101  5
   0001  1
E: 0110  6
```

- Es tritt kein Carry und kein Overflow auf.

  `C = 0, V = 0`

Berechnung von Statusbits II

- Ein paar Beispiele: Unterschied zwischen Carry und Overflow
- Angenommen: 2K-Darstellung. Die Zahlen sind 4 Bit breit. Der Zahlbereich ist also von -8 bis +7

```
   0101  5
   0011  3
E: 1000  8
```

- In diesem Fall tritt kein Carry, aber ein Überlauf auf. Beide Operanden 5 und 3 sind positiv. Das Vorzeichen aber ist negativ.

  `C = 0, V = 1`

Berechnung von Statusbits III

- Ein paar Beispiele: Unterschied zwischen Carry und Overflow
- Angenommen: 2K-Darstellung. Die Zahlen sind 4 Bit breit. Der Zahlbereich ist also von -8 bis +7

```
     0101  5
     1111 -1
E: 1 0100  4
```

- In diesem Fall tritt ein Carry, aber kein Überlauf auf.

  `C = 1, V = 0`

Berechnung von Statusbits IV

- Ein paar Beispiele: Unterschied zwischen Carry und Overflow
- Angenommen: 2K-Darstellung. Die Zahlen sind 4 Bit breit. Der Zahlbereich ist also von -8 bis +7
- Beispiel (3+(-5))

```
   0011  3
   1011 -5
E: 1110 -2
```

- In diesem Fall tritt kein Carry und kein Überlauf auf. Das Vorzeichen ist negativ und das Vorzeichenbit wird eins.

```
C = 0, V = 0, N = 1
```

Sprünge/Verzweigungen

- Ändern der Ausführungsreihenfolge von Befehlen
- Arten von Sprüngen/Verzweigungen
- Unbedingte Sprünge
  - **b** target /* Springe (branch) zu target */
- Bedingte Sprünge
  - **beq** target /* branch if equal */
  - **bmi** target /* branch if negative (minus) */
  - ...

Unbedingte Sprünge (**b**)

```
1  ...
2  /* ARM - Assemblersprache */
3  add r1, r2, #17 /* r1 = r2 + 17 */
4  b target /* branch zu target */
5  orr r1, r1, r3 /* nicht ausgefuehrt */
6  and r3, r1, #0xFF /* nicht ausgefuehrt */
7
8  target: /* Sprungmarke (label) */
9  sub r1, r1, #78 /* r1 = r1 - 78 */
10 ...
```

Label sind Namen für Stellen (Adressen) im Programm. Sie müssen anders als Maschinenbefehle (Mnemonics) heißen.
Achtung: Labels müssen mit einem Doppelpunkt abgeschlossen werden.

Der Vergleichsbefehl (**cmp**)

- Der Vergleichsbefehl (**cmp**)[3] führt eine Subtraktion aus und setzt die Statusflags
- Beispiel:

```
1  ...
2  /* ARM - Assemblersprache */
3  cmp r0, r1 /* setze Flags auf Basis der */
4             /* Rechnung r0 - r1 = -4 */
5             /* N=1, da Ergebnis negativ */
6             /* Statusflags: N=1, Z=0, C=0, V=0 */
7  ...
```

- Der Vergleichsbefehl (**tst**)[4] testet den Inhalt des ersten Operanden gegen den zweiten Operanden durch bitweise AND-Verknüpfung. Es können die Statusflags N, Z und C verändert werden.

[3] cmp steht für compare
[4] Test bits

Bedingte Sprünge (**beq**)

```
1  ...
2  /* ARM - Assemblersprache */
3  mov r0, #4 /* r0 = 4 */
4  add r1, r0, r0 /* r1 = r0 + r0 = 8 */
5  cmp r0, r1 /* setze Flags auf Basis der */
6             /* Rechnung r0 - r1 = -4 */
7             /* N=1, da Ergebnis negativ */
8             /* Statusflags: N=1, Z=0, C=0, V=0 */
9  beq there /* kein Sprung (Z != 1) */
10 orr r1, r1, #1 /* r1 = r1 or 1 = 9 */
11
12 there:
13 add r1, r1, #78 /* r1 = r1 + 78 = 87 */
14 ...
```

Bedingte Ausführung von Befehlen

- Die Ausführung z. B. eines Sprungbefehls **beq** erfolgt unter einer Bedingung. Im Fall **beq** ist die Bedingung, dass das Zero flag (Z) gesetzt ist. Ist das Zero flag nicht gesetzt, wird nicht gesprungen.
- Die vorgestellte ARM®-Architektur erlaubt es, dass eine Vielzahl von Befehlen mit Bedingungen[5] verknüpft werden können.
- Die Bedingung wird als Suffix[6] formuliert. Bei einem Additionsbefehl würde man z. B. **addeq** schreiben. Der Additionsbefehl wird nur ausgeführt, wenn das Ergebnis eines Vergleiches (z. B. durch **cmp**) das Zero flag gesetzt hätte.
- Bei der Programmierung einer ARM® 32-Bit Cortex Architektur kann durch Nutzung von Condition Codes vielfach auf Sprungbefehle verzichtet werden. Die ARM® 64-Bit Architekturen unterstützen, bis auf die Sprungbefehle, keine bedingte Ausführung von Befehlen.

[5] Condition Codes (CC)
[6] an ein Befehlswort angehängte Zeichenkette

Condition Codes der Befehle

Condition Codes (CC) sind Suffixes von Befehlen, die Bedingungen anhand der Statusflags definieren. Diese Bedingungen müssen erfüllt sein, damit der Befehl ausgeführt wird.

Suffix	**Flags**	**Bedeutung**
EQ	Z gesetzt	Gleich
NE	Z ungesetzt	Nicht gleich
CS oder HS	C gesetzt	$>=$ für unsigned
CC oder LO	C ungesetzt	$<$ für unsigned
MI	N gesetzt	Negative
PL	N ungesetzt	Positiv oder Null
VS	V gesetzt	Overflow
VC	V ungesetzt	Kein Overflow
HI	C gesetzt und Z ungesetzt	$>$ für unsigned
LS	C ungesetzt oder Z gesetzt	$<=$ für unsigned
GE	N und V sind gleich	$>=$ für signed
LT	N und V sind ungleich	$<$ für signed
GT	Z ungesetzt, N und V sind gleich	$>$ für signed
LE	Z gesetzt, N und V sind ungleich	$<=$ für signed

Sprachkonstrukte von Hochsprachen und deren Abbildung in Assembler

- if-Anweisungen
- if/else-Anweisungen
- while-Schleifen
- for-Schleifen

if-Anweisungen

```
1  /* Hochsprache */
2  if (apples == oranges)
3    f = i + 1;
4
5  f = f - i;
```

```
1  /* ARM - Assemblersprache */
2  /* r0=apples; r1=oranges, r2=f, r3=i */
3  cmp r0, r1
4  bne L1
5  add r2, r3, #1
6  L1:
7  sub r2, r2, r3
```

if/else-Anweisungen

```
1 /* Hochsprache */
2 if (apples == oranges)
3   f = i + 1;
4 else
5   f = f - i;
```

```
1 /* ARM - Assemblersprache */
2 /* r0=apples; r1=oranges, r2=f, r3=i */
3 cmp r0, r1
4 bne L1
5 add r2, r3, #1
6 b L2
7 L1:
8 sub r2, r2, r3
9 L2:
```

while-Schleifen

```
1  /* Hochsprache // berechnet x = log2(128) */
2  int pow = 1; int x = 0;
3  while (pow != 128) {
4    pow = pow * 2;
5    x = x + 1; }
```

```
1  /* ARM - Assemblersprache r0 = pow, r1 = x */
2  mov r0, #1
3  mov r1, #0
4  WHILE:
5  cmp r0, #128
6  beq DONE
7  lsl r0, r0, #1 /* pow = pow * 2 */
8  add r1, r1, #1 /* x = x + 1 */
9  b WHILE
10 DONE:
```

for-Schleifen

```
1  /* Hochsprache // addiere Zahlen von 0 bis 9 */
2  int sum = 0; int i;
3  for (i=0; i < 10; i=i+1) {
4    sum = sum + i; }
```

```
1  /* ARM - Assemblersprache r0 = i, r1 = sum */
2  mov r1, #0
3  mov r0, #0
4  FOR:
5  cmp r0, #10
6  bge DONE
7  add r1, r1, r0 /* sum = sum + i */
8  add r0, r0, #1 /* i=i+1 */
9  b FOR
10 DONE:
```

Zusammenfassung und Ausblick

Zusammenfassung und Ausblick

Zusammenfassung

- Konzepte der maschinennahen Programmierung

Ausblick

- Konzepte der maschinennahen Programmierung
- Arrays
- Unterprogramme

Lernkontrolle

- Ich habe ein Verständnis für die Organisaton des Hauptspeichers entwickelt ☑
- Ich habe das Verständnis für die Programmierung mit Maschinenbefehlen vertieft und kann Hochsprachenkontrukte in Assembler formulieren ☑
- Die Bedeutung und Berechnung der Statusflags habe ich verstanden ☑
- ...

Literatur

Literatur

[BO10] Bryant, Randal E. und David R. O'Hallaron: *Computer Systems – A Programmer's Perspective*.
Prentice Hall, 2010.