Rechnerorganisation
Sommersemester 2023 – 4. Vorlesung

Prof. Stefan Roth, Ph.D.

Technische Universität Darmstadt

15. Mai 2023

Inhalt

Konzepte der maschinennahen Programmierung

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

Erinnerung: Komponenten und Struktur eines Rechnersystems

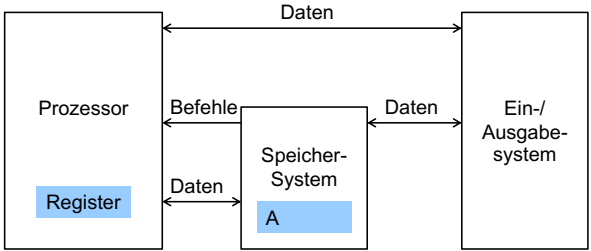


Abbildung: Komponenten eines Rechnersystems (verfeinerte Darstellung)

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

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)

Nutzung des Hauptspeichers I

- Bisher wurden „die Daten“ direkt in Register geschrieben (z. B. durch `mov r1, #42`)
- Laden der Werte der Variablen durch Angabe des Variablennamens:

```
1  /* speicher_0.s */
2
3  .data /* Daten Bereich */
4  var1: .word 5  /* Variable 1 im Speicher, Wert 5 */
5  var2: .word 12 /* Variable 2 im Speicher, Wert 12 */
6
7  .global main   /* Definition Einsprungpunkt Hauptprogramm */
8
9  main:            /* Hauptprogramm */
10     ldr r0, var1 /* laedt Wert von var1 in r0 */
11     ldr r1, var2 /* laedt Wert von var2 in r1 */
12     add r0, r0, r1
13     bx lr        /* Springe zurueck zum aufrufenden Programm */
```

Nutzung des Hauptspeichers II

- Laden der Werte der Variablen durch indirekte Adressierung:

```
1  /* speicher_I_a.s */
2
3  .data /* Daten Bereich */
4  var1: .word 5  /* Variable 1 im Speicher, Wert 5 */
5  var2: .word 12 /* Variable 2 im Speicher, Wert 12 */
6
7  .global main   /* Definition Einsprungpunkt Hauptprogramm */
8
9  main:                /* Hauptprogramm */
10     ldr r0, adr_var1 /* laedt Adresse von var1 in r0 */
11     ldr r1, adr_var2 /* laedt Adresse von var2 in r1 */
12     ldr r2, [r0]     /* Lade Inhalt von Adresse r0 in r2 */
13     ldr r3, [r1]     /* Lade Inhalt von Adresse r1 in r3 */
14     add r0, r2, r3
15     bx lr            /* Springe zurueck zum aufrufenden Programm */
16
17 adr_var1: .word var1 /* Adresse von Variable 1 */
18 adr_var2: .word var2 /* Adresse von Variable 2 */
```

Nutzung des Hauptspeichers III

- Laden der Werte der Variablen durch indirekte Adressierung
- Vereinfachte Syntax: =var

```
1  /* speicher_I_b.s */
2
3  .data /* Daten Bereich */
4  var1: .word 5  /* Variable 1 im Speicher, Wert 5 */
5  var2: .word 12 /* Variable 2 im Speicher, Wert 12 */
6
7  .global main   /* Definition Einsprungpunkt Hauptprogramm */
8
9  main:             /* Hauptprogramm */
10     ldr r0, =var1 /* laedt Adresse von var1 in r0 */
11     ldr r1, =var2 /* laedt Adresse von var2 in r1 */
12     ldr r2, [r0]  /* Lade Inhalt von Adresse r0 in r2 */
13     ldr r3, [r1]  /* Lade Inhalt von Adresse r1 in r3 */
14     add r0, r2, r3
15     bx lr         /* Springe zurueck zum aufrufenden Programm */
```

Nutzung des Hauptspeichers IV

- Veränderung des Programms: Ergänzung eines Offsets in Zeile 13, Ausgabe ist 24

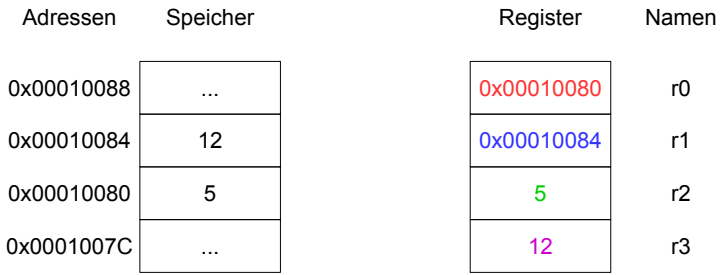
```
1  /* speicher_II.s */
2
3  .data /* Daten Bereich */
4  var1: .word 5  /* Variable 1 im Speicher, Wert 5 */
5  var2: .word 12 /* Variable 2 im Speicher, Wert 12 */
6
7  .global main   /* Definition Einsprungpunkt Hauptprogramm */
8
9  main:                /* Hauptprogramm */
10     ldr r0, adr_var1 /* laedt Adresse von var1 in r0 */
11     ldr r1, adr_var2 /* laedt Adresse von var2 in r1 */
12     ldr r2, [r0,#4]  /* Lade Inhalt von Adresse r0 + Offset 4 in r2 */
13     ldr r3, [r1]     /* Lade Inhalt von Adresse r1 in r3 */
14     add r0, r2, r3
15     bx lr            /* Springe zurueck zum aufrufenden Programm */
16
17 adr_var1: .word var1 /* Adresse von Variable 1 */
18 adr_var2: .word var2 /* Adresse von Variable 2 */
```

Nutzung des Hauptspeichers V

- Veränderung des Programms: Ergänzung um Zeile 13 und Angabe eines Registers in Zeile 14, Ausgabe ist 24

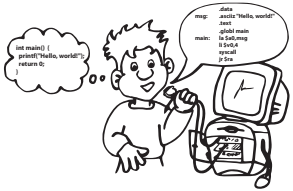
```
1   /* speicher_III.s */
2
3   .data /* Daten Bereich */
4   var1: .word 5  /* Variable 1 im Speicher, Wert 5 */
5   var2: .word 12 /* Variable 2 im Speicher, Wert 12 */
6
7   .global main   /* Definition Einsprungpunkt Hauptprogramm */
8
9   main:               /* Hauptprogramm */
10      ldr r0, adr_var1 /* laedt Adresse von var1 in r0 */
11      ldr r1, adr_var2 /* laedt Adresse von var2 in r1 */
12      mov r5, #4
13      ldr r2, [r0,r5]  /* Lade Inhalt von Adresse r0 + Offset r5 in r2 */
14      ldr r3, [r1]     /* Lade Inhalt von Adresse r1 in r3 */
15      add r0, r2, r3
16      bx lr            /* Springe zurueck zum aufrufenden Programm */
17
18  adr_var1: .word var1 /* Adresse von Variable 1 */
19  adr_var2: .word var2 /* Adresse von Variable 2 */
```

Erklärung der Nutzung der Register und des Hauptspeichers



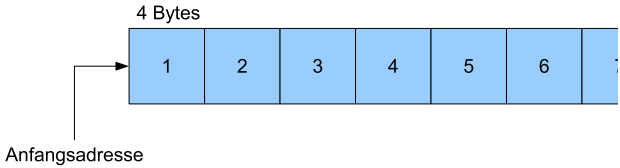

```
[...]
ldr r0,adr_var1
ldr r1,adr_var2
ldr r2,[r0]
ldr r3,[r1]
[...]
```

Arrays


int main() {
printf("Hello, world!");
return 0;
}
msg: .data
.asciiz "Hello, world!"
.text
.globl main
main: la $a0,msg
li $v0,4
syscall
jr $ra

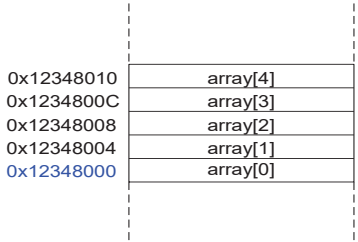
Arrays (Datenfelder)

- Laden (und Speichern) von Worten, s. 3. Vorlesung
- Arrays (Datenfelder) bestehen aus mehreren Worten
- Nützlich um auf eine große Zahl von Daten gleichen Typs zuzugreifen
- Zugriff auf einzelne Elemente über Index
- Größe eine Arrays: Anzahl von Elementen im Array
- Veranschaulichung des Arrays:



Verwendung von Arrays

- Array mit 5 Elementen
- Basisadresse, hier 0x12348000
  - Adresse des ersten Array-Elements
  - Index 0, geschrieben als array[0]
- Erster Schritt für Zugriff auf Element: Lade Basisadresse des Arrays in Register



Bearbeite Array in for-Schleife I

```
1  /* Hochsprache */
2  int i;
3  int scores[200];
4  ...
5  for (i = 0; i < 200; i = i + 1)
6    scores[i] = scores[i] + 10;
```

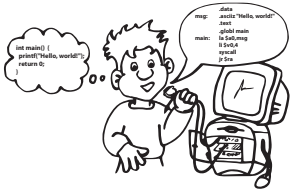
```
1  /* ARM - Assemblersprache */
2  /* r0 = Basisadresse von Array, r1 = i */
3  /* Initialisierung */
4  mov r0, #0x14000000 /* Basisadresse in r0 */
5  mov r1, #0
6  ...
```

Beispiel entnommen aus [HH16] (S. 314). Bemerkung: der mov-Befehl läßt beim zweiten Operanden maximal 16 Bit Werte zu. Die angegebene Adresse führt zu einer Fehlermeldung beim Assemblieren.

Bearbeite Array in for-Schleife II

```
1  LOOP:
2      cmp r1, #200     /* i < 200 */
3      bge L3           /* if i >= 200, exit loop */
4      lsl r2, r1, #2   /* r2 = i * 4 */
5      ldr r3, [r0,r2]  /* r3 = scores[i] */
6      add r3, r3, #10  /* r3 = scores[i] + 10 */
7      str r3, [r0,r2]  /* scores[i] = scores[i] + 10 */
8      add r1, r1, #1   /* i = i + 1 */
9      b LOOP           /* repeat */
10 L3:
```

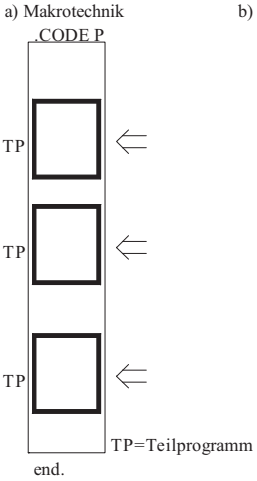
Unterprogramme



Unterprogramme – Einführung

- Unterprogramme helfen bei der strukturierten Programmierung
- Im Prinzip gibt es zwei grundlegende Konzepte zur Realisierung von Unterprogrammen (eine Funktion stellt ein spezielles Unterprogramm da)
- Betrachtet wird ein Programm (Hauptprogramm), in dem ein Teilprogramm (TP) an verschiedenen Stellen ausgeführt werden soll.
- Die zwei Konzepte sind **Makrotechnik** und **Unterprogrammtechnik**
- Veranschaulichung (s. folgende Folie)

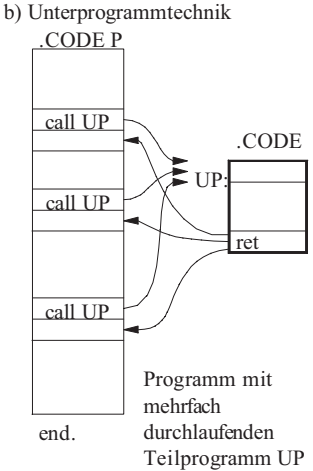
Unterprogramme – Makrotechnik



Unterprogramme – Makrotechnik

- Makrotechnik: Dabei wird das Teilprogramm an den Stellen, wo es gebraucht wird, einkopiert (vgl. Präprozessor in C)
- Dazu wird dem Teilprogramm, dem sogenannten Makro, ein Name zugeordnet (Makroname).
- Das passiert durch die sogenannte Makrodefinition.
- An den Stellen, an denen das Makro einkopiert werden soll, wird dann der Makroname genannt.
- $\Rightarrow$ Makroaufruf

Unterprogramme – Unterprogrammtechnik



Unterprogramme – Unterprogrammtechnik

- Unterprogrammtechnik: Hier ist das Teilprogramm (Unterprogramm, UP) nur einmal im Code vorhanden. Es wird durch eine Sprungmarke (Unterprogrammname) gekennzeichnet.
- An den Stellen, an denen das Unterprogramm ausgeführt werden soll erfolgt der Aufruf durch einen Sprungbefehl mit dem Unterprogrammbezeichner als Operand.
- Am Ende des Unterprogramms erfolgt die Rückkehr in das aufrufende Programm wiederum durch einen speziellen Sprungbefehl auf die Rückkehradresse, d. h. die Adresse des Befehls, der im aufrufenden Programm nach dem Befehl zum Aufruf folgt.
- Rückkehradresse wird gespeichert (Register, Stack).

Unterprogramme – Unterscheidungen und Probleme

- Das Konzept der Funktionen und das Konzept der Prozeduren ist aus höheren Programmiersprachen abgeleitet (z. B. C/C++, Pascal)
- Zwei grundsätzliche Fragen sind dabei zu betrachten:
  - Wie erfolgt die Parameterübergabe und wie werden die Ergebnisse zwischen dem aufrufenden Programm und dem Unterprogramm ausgetauscht?
  - die Implementierung von lokalen Variablen der Funktion
- Wichtig dabei: Sichtbarkeit von Variablen
  - globale Variablen, stehen allen Funktionen eines Programms zur Verfügung
  - lokale Variablen, stehen nur in der Funktion zur Verfügung
- Lebensdauer: Während ihrer Lebensdauer (vgl. [DBG08, S. 108]) besitzt eine Variable einen Speicherplatz

Funktions- und Prozeduraufruf – ein Beispiel

- Definitionen
  - Aufrufer – caller: Ursprung des Funktionsaufrufs (hier main)
  - Aufgerufener – callee: aufgerufene Funktion (hier sum)

```
1  int main()
2  {
3     int y;
4     y = sum(42, 7);
5     ...
6  }
7  int sum(int a, int b)
8  {
9     return (a + b);
10 }
```

Funktions- und Prozeduraufruf – Regeln

- Aufrufer – caller:
  - Übergibt Argumente (aktuelle Parameter) an Aufgerufenen
  - Springt Aufgerufenen an
- Aufgerufener – callee:
  - Führt Funktion/Prozedur aus
  - Gibt Ergebnis (Rückgabewert) an Aufrufer zurück (für Funktion)
  - Springt hinter Aufrufstelle zurück
  - **Darf keine Register oder Speicherstellen überschreiben, die im Aufrufer genutzt werden**

Aufrufargumente und Rückgabewert – Hochsprache

```
1  int main()
2  {
3    int y;
4    ...
5    y = diffofsums(14, 3, 4, 5); /* 4 Argumente */
6    ...
7  }
8  int diffofsums(int f, int g, int h, int i)
9  /* 4 formale Parameter */
10 {
11   int result;
12   result = (f + g) - (h + i);
13   return result; /* Rueckgabewert */
14 }
```

Aufrufargumente und Rückgabewert – Assembler I

```
1  /* r4 = y */
2  main:
3    mov r0, #14    /* Argument 0 = 14 */
4    mov r1, #3     /* Argument 1 = 3 */
5    mov r2, #4     /* Argument 2 = 4 */
6    mov r3, #5     /* Argument 3 = 5 */
7    bl diffofsums  /* Funktionsaufruf */
8    mov r4, r0     /* y = Rueckgabewert */
9  /* -------------------------- */
10 diffofsums:
11   add r8, r0, r1
12   add r9, r2, r3
13   sub r4, r8, r9
14   mov r0, r4     /* Lege Rueckgabewert in r0 ab */
15   mov pc, lr     /* Ruecksprung zum Aufrufer */
```

Aufrufargumente und Rückgabewert – Assembler II

- Nach Abarbeitung der Zeilen 3 – 6 sind die zu übergebenden Argumente in den Registern r0 – r3 abgelegt.
- Zeile 7 ruft das Unterprogramm mit dem Befehl bl (branch & link) auf. Dieser Befehl sorgt dafür, dass die Rückkehradresse 8 in das Register r14 (lr) geschrieben wird.
- Das Unterprogramm beginnt an der Adresse 10. Nach Durchführung der arithmetischen Operationen (Zeilen 11 – 13) wird das Ergebnis in das Register r0 geschrieben (Zeile 14). r0 wird auch *return value* Register genannt.
- In Zeile 15 wird die zuvor gespeicherte Rückkehradresse (8) aus dem Register r14 (lr) in das Register r15 (pc) kopiert. Der nächste auszuführende Befehl ist an der Adresse 8 zu finden und lautet mov r4, r0

Aufrufargumente und Rückgabewert – Assembler III

```
1  /* r4 = y */
2  diffofsums:
3    add r8, r0, r1
4    add r9, r2, r3
5    sub r4, r8, r9
6    mov r0, r4  /* Lege Rueckgabewert in r0 ab */
7    mov pc, lr  /* Ruecksprung zum Aufrufer */
```

- diffofsums überschreibt drei Register: r8, r9 und r4
- Wenn in r8, r9 und r4 wichtige Daten sind, die der **Aufrufer** noch benötigt, gibt es bei dieser Implementierung ein Problem.
- Lösung des Problems: diffofsums kann den Inhalt benötigter Register temporär auf dem **Stack** sichern

Zusammenfassung und Ausblick

Zusammenfassung und Ausblick

Zusammenfassung

- Konzepte der maschinennahen Programmierung
- Arrays
- Unterprogramme

Ausblick

- Stack
- Rekursion
- Compilieren, Assemblieren und Linken

Lernkontrolle

- Ich habe das Prinzip der indirekten Adressierung nachvollziehen können und habe den Unterschied zwischen Adresse und Datenwert verstanden ☑
- Die Bedeutung von Unterprogrammen für die strukturierte Programmierung konnte ich nachvollziehen ☑
- Das Zusammenspiel der Register pc und lr habe ich nachvollziehen können ☑
- ...

Literatur


int main() {
printf("Hello, world!");
return 0;
}
msg: .data
.asciiz "Hello, world!"
.text
.globl main
main: la $a0,msg
li $v0,4
syscall
jr $ra

Literatur

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

[DBG08] Dausmann, Manfred, Ulrich Bröckl und Joachim Goll: *C als erste Programmiersprache*.
Teubner, 2008.

[HH16] Harris, David Money und Sarah L. Harris: *Digital Design and Computer Architecture, ARM® Edition*.
Morgan Kaufmann, 2016.