Rechnerorganisation
Sommersemester 2023 – 5. Vorlesung

Prof. Stefan Roth, Ph.D.

Technische Universität Darmstadt

22. Mai 2023

Inhalt

1. Organisatorisches & Fragen
2. Konzepte der maschinennahen Programmierung
3. Unterprogramme
4. Stack
5. Rekursion
6. Compilieren, Assemblieren und Linken
7. Zusammenfassung und Ausblick
8. Literatur

Organisatorisches & Fragen

Organisatorisches

- 1. Theorietestat ist noch bis heute Abend 23:59 offen. Zwischenergebnisse sehr erfreulich.
- 2. Theorietestat wird heute nach der Vorlesung geöffnet.

- Morgen (23.05.2023) ist ab 13 Uhr TU Meet & Move – Übungen am Nachmittag fallen aus.
- Kommenden Montag (29.05.2023) ist Pfingsten – Vorlesung fällt aus.

- 1. Programmiertestat startet kommende Woche:
  - 2 Wochen Bearbeitungszeit
  - Anmeldung für Testierung vsl. ab kommender Woche
  - Testierung im 2-wöchigen Zeitraum nach der Bearbeitungszeit

Fragen aus der / zur letzten Vorlesung

- F: Wieso verwendet man **`mov pc, lr`** zum Rücksprung aus einem Unterprogramm und nicht einfach **`b lr`**?
- A: Der Branch-Befehl **`b`** <op> akzeptiert nur feste Adressen als Operand <op>.

- F: Wenn ich eine Exception / Ausnahme (im Betriebssystem) behandele, wie komme ich dann wieder in mein ‚normales' Programm zurück?
- A: Der ARM-Prozessor hat verschiedene Prozessormodi, davon einer um mit Ausnahmen umzugehen. Manche der Register, u.a. das Link Register `lr`, sind gespiegelt, d.h. es existieren für die diversen Modi separate Versionen der Register. Man kann also die Rücksprungadresse abspeichern, ohne das ‚normale' Link Register zu überschreiben.

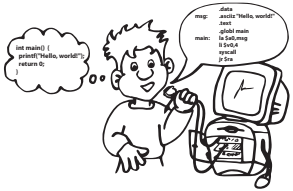
- F: Segmentation fault bei Ausführung von Beispielcode.
- A: Siehe Forum zu Vorlesung 4.

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
  - Feld (Array) Zugriffe
  - Unterprogramme
    - Funktion
    - Prozedur
    - **Rekursion**

Unterprogramme



Aufrufargumente und Rückgabewert – Hochsprache

```
1  int main()
2  {
3    int y;
4    ...
5    y = diffofsums(14, 3, 4, 5); /* 4 Argumente */
6    ...
7  }
8
9  int diffofsums(int f, int g, int h, int i) /* 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

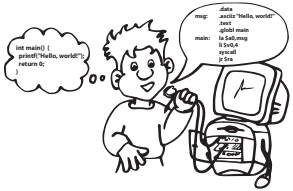
- Nach Abarbeitung der Zeilen 3 – 6 sind die zu übergebenden Argumente in den Registern `r0` – `r3` abgelegt.
- Zeile 7 ruft das Unterprogramm mit dem Befehl **`bl`** (branch & link) auf. Dieser Befehl sorgt dafür, dass die Rückkehradresse 8 in das Register `r14` (`lr`) geschrieben wird.
- Das Unterprogramm 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.

Stack


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

Stack (dt. auch Stapel- oder Kellerspeicher)

- Speicher für temporäres Zwischenspeichern von Werten.
- Agiert wie ein Stapel (Beispiel: Stapel von Tellern):
  - Zuletzt aufgelegter Teller wird zuerst heruntergenommen.
  - "last in, first out" (LIFO)
- Dehnt sich aus: Belegt mehr Speicher, wenn mehr Daten unterzubringen sind.
- Zieht sich zusammen: Belegt weniger Speicher, wenn zwischengespeicherte Daten nicht mehr gebraucht werden.
- Wächst bei ARM nach unten (von hohen zu niedrigen Speicheradressen):
  - Übliche Realisierung (deshalb auch Kellerspeicher genannt).
- stack pointer (Stapelzeiger): `sp (r13)`
  - zeigt auf zuletzt auf dem Stack abgelegtes Datenelement.

Verwendung des Stacks in Unterprogrammen

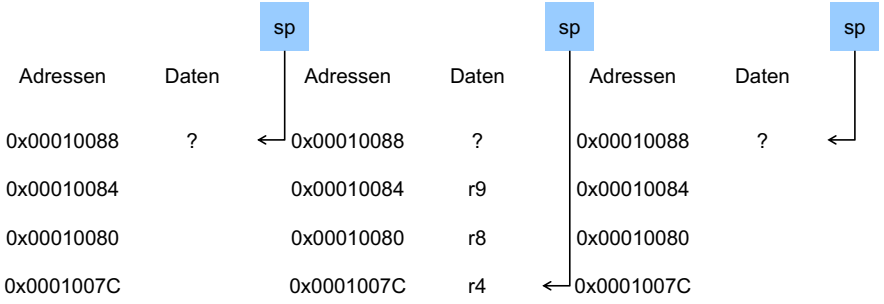
- Aufgerufene Unterprogramme dürfen keine unbeabsichtigten Nebenwirkungen („Seiteneffekte“) haben.
- Problem: diffofsums überschreibt die drei Register `r8`, `r9`, `r4`.

```
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 */
```

Register auf Stack zwischenspeichern

```
1  diffofsums:
2    sub sp, sp, #12  /* Speicher auf Stack reservieren */
3    str r9, [sp, #8] /* sichern von r9 auf Stack */
4    str r8, [sp, #4] /* sichern von r8 auf Stack */
5    str r4, [sp]     /* sichern von r4 auf Stack */
6
7    add r8, r0, r1   /* Rechnung durchfuehren */
8    add r9, r2, r3
9    sub r4, r8, r9
10   mov r0, r4       /* Lege Rueckgabewert in r0 ab */
11
12   ldr r4, [sp]     /* herstellen von r4 */
13   ldr r8, [sp, #4] /* herstellen von r8 */
14   ldr r9, [sp, #8] /* herstellen von r9 */
15   add sp, sp, #12  /* Speicher auf Stack freigeben */
16
```

Veränderung des Stacks während diffofsums




```
sub sp,sp,#12
str r9,[sp,#8]
str r8,[sp,#4]
str r4,[sp]
[...]
ldr r4,[sp]
ldr r8,[sp,#4]
ldr r9,[sp,#8]
add sp,sp,#12
[...]
```

Mehrfache Unterprogrammaufrufe – Sicherung von `lr` I

```
1  main:             /* Hauptprogramm */
2    mov r0, #14     /* Argument 0 = 14 */
3    mov r1, #3      /* Argument 1 = 3 */
4    mov r2, #4      /* Argument 2 = 4 */
5    mov r3, #5      /* Argument 3 = 5 */
6    mov r10, lr     /* sichern Ruecksprungadresse in r10 */
7    bl diffofsums   /* Funktionsaufruf */
8    mov r4, r0      /* y = Rueckgabewert */
9    mov lr, r10     /* herstellen der Ruecksprungadresse */
10   mov r0, r4
11   bx lr           /* zurueck zum Aufrufer */
```

Mehrfache Unterprogrammaufrufe – Sicherung von `lr` II – push und pop

```
1  main:              /* Hauptprogramm */
2    mov r0, #14      /* Argument 0 = 14 */
3    mov r1, #3       /* Argument 1 = 3 */
4    mov r2, #4       /* Argument 2 = 4 */
5    mov r3, #5       /* Argument 3 = 5 */
6    push {lr}        /* sichern Ruecksprungadresse Stack */
7    bl diffofsums    /* Funktionsaufruf */
8    mov r4, r0       /* y = Rueckgabewert */
9    pop {lr}         /* herstellen der Ruecksprungadresse */
10   mov r0, r4
11   bx lr            /* zurueck zum Aufrufer */
```

- **push** und **pop** sind so genannte **Pseudoinstruktionen**.
- vgl. **Reference Guide**
- Learning Nugget 03 – Unterprogramme in Assembler

Rekursion

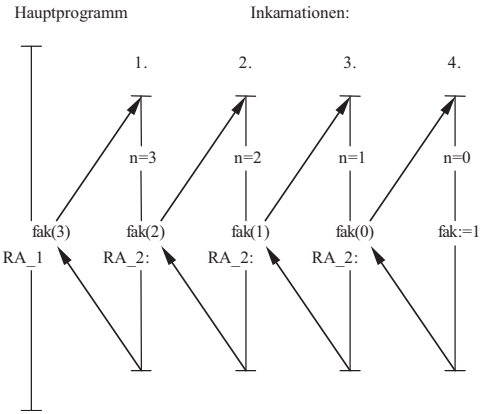

int main() {
printf("Hello, world!");
return 0;
}
msg: .data
.asciiz "Hello, world!"
.text
.globl main
main: la $a0,msg
li $v0,4
syscall
jr $ra

Rekursiver Unterprogrammaufruf – Fakultätsberechnung

```
1  #include <stdio.h>
2  int fak (int n) {
3    if (n >=1 ) {
4      return fak(n-1) * n;
5    }
6    else { return 1; } /* 0! = 1 */
7  } /* Funktionsende */
8
9  int main(void) { /* Hauptprogramm */
10   int n = 3;      /* 3! = 6 */
11   int z;          /* lokale Variable */
12   z = fak(n);     /* Funktionsaufruf */
13   printf("Fakultaet %d\n:", z); /* Ausgabe des Ergebnisses */
14   return 0;
15 } /* Ende Hauptprogramm */
```

Rekursiver Unterprogrammaufruf – Fakultätsberechnung

- Vor der Umsetzung erstmal einige Überlegungen
- Ablauf eines Unterprogramms wird auch als Inkarnation (wiederholter Unterprogrammaufruf) bezeichnet



Rekursiver Unterprogrammaufruf – Fakultätsberechnung

- Die Abbildung der Inkarnationen zeigt, dass es zwei verschiedene Rückkehradressen gibt.
- Die Adresse RA_1 ist die Adresse im Hauptprogramm.
- Die Adresse RA_2 ist die Adresse im Unterprogramm.
- Für alle Inkarnationen des Unterprogramms fak ist die Rückkehradresse RA_2 dieselbe Adresse, da alle Inkarnationen Abläufe desselben Codes sind.
- Learning Nugget 04 – Rekursion in Assembler

Rekursiver Unterprogrammaufruf – Fakultätsberechnung

Hauptprogramm

```
1  .data
2  mystring : .asciz "Fakultaet␣%d␣\n"
3  [...]
4  .global main /* Einsprungpunkt Hauptprogramm */
5
6  main:           /* Hauptprogramm */
7        push {lr}
8        mov r0, #3 /* Fak von 3 */
9        bl fak  /* Funktionsaufruf UP */
10 RA_1: mov r1, r0 /* Ergebnis UP in r1 */
11       ldr r0, =mystring
12       bl printf /* Ausgabe Ergebnis */
13       pop {lr}
14       bx lr     /* zurueck zum Aufrufer */
```

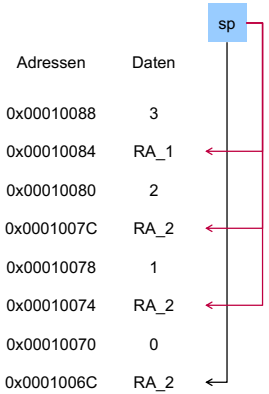
Rekursiver Unterprogrammaufruf – Fakultätsberechnung

Unterprogramm

```
 1  .text
 2  fak: sub sp, sp, #8    /* Speicher Stack reservieren */
 3       str r0, [sp, #4]  /* sichern von r0 auf Stack */
 4       str lr, [sp]      /* sichern Ruecksprungadresse */
 5       cmp r0, #1        /* Rekursionsende pruefen */
 6       blt else          /* branch less */
 7       sub r0, r0, #1    /* n - 1 */
 8       bl fak            /* Funktionsaufruf */
 9  RA_2:ldr r1, [sp, #4]  /* laden von n */
10       mul r0, r1, r0    /* fak (n-1) * n */
11  fin: ldr lr, [sp]      /* laden Rueckkehradresse */
12       add sp, sp, #8    /* Speicher Stack freigeben */
13       bx lr             /* Sprung zum Aufrufer */
14  else:mov r0, #1        /* Rekursionsanker */
15       b fin
```

Rekursiver Unterprogrammaufruf – Fakultätsberechnung
Aufbau des Stacks

- Maximale Ausdehnung des Stacks (Stackpointer schwarze Linie). Die Rückadresse steht an SP, der Wert von n an SP+4.



Zusammenfassung Unterprogrammaufrufe

- Unter Aufrufkonvention[1] versteht man u. a. die Methoden, mit der einem Unterprogramm Daten übergeben werden.
- Die 32-Bit ARM® Architektur nutzt die 16 Register wie folgt:
  - `r15`: program counter (`pc`)
  - `r14`: link register (`lr`)
  - `r13`: stack pointer (`sp`)
  - `r4` bis `r11`: Lokale Variablen
  - `r0` bis `r3`: Werte, die an das Unterprogramm übergeben werden und Ergebnisse, die das Unterprogramm zurück gibt.
- Umgang mit Registern:
  - Unterprogramme müssen den Inhalt der Register `r4` bis `r11` und `r13` (`sp`) sichern; sollte es innerhalb des Unterprogramms weitere Unterprogrammaufrufe geben, muss auch `r14` (`lr`) gesichert werden.

[1] engl. calling convention

Vorgehensweise Unterprogrammaufrufe

- Typisches Unterprogramm führt folgende Operationen aus:
  - Zum Beginn des Unterprogramms werden die Inhalte der Register `r4` bis `r11` und `r14` auf dem Stack gesichert.
  - Die dem Unterprogramm übergebenen Werte aus `r0` bis `r3` werden in die Register `r4` bis `r11` kopiert.
  - Ggf. werden weitere lokale Variablen in den Registern `r4` bis `r11` definiert.
  - Implementierung der Rechnung.
  - Rückgabewert in `r0` schreiben.
  - Zum Ende des Unterprogramms werden die auf dem Stack gesicherten Inhalte der Register `r4` bis `r11` und `r14` wieder hergestellt.
  - Rücksprung zum Hauptprogramm/Aufrufer.

Compilieren, Assemblieren und Linken

Ein erstes Assemblerprogramm – noch in C

```
1  /* Addition */
2  #include <stdio.h>
3
4  int main()
5  {
6    int p = 5;
7    int q = 12;
8    int result = p + q;
9    printf("result␣ist␣%d␣\n",result);
10   return 0;
11 }
```

- `gcc addition.c` übersetzt das C-Programm[2]
- `gcc -S addition.c` generiert das Assemblerprogramm

[2] Hinweis: Auf `clientssh-arm` verwenden wir einen Cross-Compiler und nicht den systemeigenen `gcc`.

Ein erstes Assemblerprogramm – addition.s – ARM® Architektur

```
1           .file   "addition.c"
2           .section        .rodata
3           .align  2
4   .LC0:
5           .ascii  "result_ist_%d_\012\000"
6           .text
7           .align  2
8           .global main
9           .syntax unified
10          .arm
11          .fpu vfp
12          .type   main, %function
13  main:
14          @ args = 0, pretend = 0, frame = 16
15          @ frame_needed = 1, uses_anonymous_args = 0
16          push    {fp, lr}
17          add     fp, sp, #4
18          sub     sp, sp, #16
19          mov     r3, #5
20          str     r3, [fp, #-8]
21          mov     r3, #12
22          str     r3, [fp, #-12]
23          ldr     r2, [fp, #-8]
24          ldr     r3, [fp, #-12]
25          add     r3, r2, r3
26          str     r3, [fp, #-16]
27          ldr     r1, [fp, #-16]
28          ldr     r0, .L3
29          bl      printf
30          [...]
```

Ein zweites Assemblerprogramm – addition.s – ARM® Architektur

```
1  /* first.s */
2  .global main /* Einsprungpunkt Hauptprogramm */
3
4  main:              /* Hauptprogramm */
5      mov r1, #5     /* Schreibe eine 5 in das Register r1 */
6      mov r2, #12
7      add r0, r1, r2 /* Addiere r1 und r2,Ergebnis in r0 */
8      bx lr          /* Springe zurueck zum aufrufenden Programm */
```

Ein erstes Assemblerprogramm – noch in C

```
1  /* Addition */
2  #include <stdio.h>
3
4  int main()
5  {
6    int p = 5;
7    int q = 12;
8    int result = p + q;
9    printf("result ist %d \n",result);
10   return 0;
11 }
```

- gcc addition.c übersetzt das C-Programm
- gcc -S **-O1** addition.c generiert das Assemblerprogramm

Analyse des Assemblerprogramms – addition.s – ARM® Architektur

```
1  main:
2          @ args = 0, pretend = 0, frame = 0
3          @ frame_needed = 0, uses_anonymous_args = 0
4          push    {r4, lr}
5          mov     r1, #17
6          ldr     r0, .L3
7          bl      printf
8          mov     r0, #0
9          pop     {r4, pc}
10 .L4:
11         .align  2
12 .L3:
13         .word   .LC0
14         .size   main, .-main
15         .section        .rodata.str1.4,"aMS",%progbits,1
16         .align  2
17 .LC0:
18         .ascii  "result␣ist␣%d␣\012\000"
19         .ident  "GCC:␣(Raspbian␣6.3.0-18+rpi1+deb9u1)␣6.3.0␣20170516"
```

Diskussion der Assemblerprogramme

- Die Übersetzung eines Programms aus einer Hochsprache nach Assembler kann offenbar sehr verschiedene Ergebnisse haben.
- Nicht alle in Hochsprachen sichtbare Sprachelemente werden in Assembler abgebildet.
- Beispiele:
  - Nicht jede lokale Variable wird auf dem Stack sichtbar. Ggf. Realisierung von lokalen Variablen bei Schleifen ausschließlich in Registern.[3]
  - Bei konstanten Variablen (=Konstanten) wird ggf. eine sogenannte **Common subexpression elimination** durchgeführt. Das bedeutet, dass Teilausdrücke zur Übersetzungszeit berechnet werden und als Konstanten gespeichert werden.
  - ...

[3] Dies hängt auch davon ab, mit welcher Optimierungseinstellung ein Compiler aufgerufen wird. Beim gcc sind das z. B. (-O1, -O2).

π Berechnung mit Trapezintegration

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main (void)
5  {
6    long i;
7    long num_steps = 2000000000;
8    double x, pi, step, sum = 0.0;
9    step = 1.0 / (double) num_steps;
10   for (i = 1; i <= num_steps; i++)
11   {
12     x = (i - 0.5) * step;
13     sum = sum + 4.0 / (1.0 + x * x);
14   }
15   pi = step * sum;
16   printf("PI_%lf\n", pi);
17  return 0;
18 }
```

Laufzeiten des Beispielprogramms I

- Raspberry Pi, mit ARM® Cortex-A53, 1 GB RAM, 4 Kerne

Variante	Optimierungsstufe	Laufzeit
1	–	1m34,416s
2	-O1	1m5,943s
3	-O2	1m4,409s

- **Achtung**: Raspberry Pi nicht zu lange laufen lassen. Überhitzungsgefahr!

Laufzeiten des Beispielprogramms II

- clientshh-arm, ThunderX, 64 GB RAM, 48 Kerne

Variante	Optimierungsstufe	Laufzeit
1	–	1m15,076s
2	-O1	0m50,051s
3	-O2	0m46,048s

- Parallelisierung kann dies weiter beschleunigen, s. Vorlesung Parallele Programmierung

Zusammenfassung und Ausblick

Zusammenfassung und Ausblick

Zusammenfassung

- Unterprogramme
- Rekursion
- Compilieren, Assemblieren und Linken

Ausblick

- Compilieren, Assemblieren und Linken
- Laufzeitanalyse von Programmen

Lernkontrolle

- Ich habe das Prinzip des Stacks verstanden und kann den Stack durch Maschinenbefehle nutzen ☑
- Die Umsetzung von Variablen aus Hochsprachen in Assemblerprogrammen kann ich nachvollziehen ☑
- Die Implementierung einer Rekursion in Assembler kann ich nachvollziehen ☑
- ...

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.

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