Rechnerorganisation

Sommersemester 2023

Prof. Stefan Roth, Ph.D.

TECHNISCHE UNIVERSITÄT DARMSTADT

6. Aufgabenblatt mit Lösungsvorschlag

05.06.2023
Compilieren, Assemblieren und Linken

Aufgabe 1: Theoriefragen

(a) Welche (wichtigen) Informationen werden im Header eines Objektprogramms (z. B. ELF-Format) abgelegt.

(b) Erläutern Sie die Aufgaben eines Binders/Linkers und eines Laders.

Lösungsvorschlag:

(a) Data: 2's complement little endian (Zahldarstellung)
Machine: ARM (Prozessortyp)

(b) Der Binder/Linker hat die Aufgabe, aus einer Menge von einzelnen verschiebbaren Objekt-Files ein ausführbares Objektprogramm zu erzeugen, indem die noch offenen externen Referenzen aufgelöst werden. Das Objektprogramm kann durch einen Lader zur Ausführung gebracht werden. Ein Lader ist ein Systemprogramm, das die Aufgabe hat, Objektprogramme in den Speicher zu laden und ggf. deren Ausführung anzustoßen. Dazu wird das Objektprogramm in den Speicher kopiert.

Aufgabe 2: Codeanalyse (optimierender) Compiler

Gegegeben ist folgendes Code-Fragment:

```
#include <stdio.h>

int main (void)
{
  int n = 20;
  int erg;

  if(n == 20) {
    erg = n * 2;
  }

  printf("Ergebnis␣%d\n",erg);

  return 0;
}
```

Im Folgenden sollen ein paar Analysen vorgenommen werden.

a) Wie kann der Ausdruck `erg = n * 2;` des C-Programms effizient berechnet werden?

Lösungsvorschlag:

Es gibt zwei Möglichkeiten. Durch das Schieben um eine Bitposition nach links wird eine Multiplikation mit zwei durchgeführt. Alternativ kann natürlich auch eine Addition des Registerwertes (der n repräsentiert) mit sich selbst vorgenommen werden.

b) Analysieren Sie den vom GCC Compiler generierten Code. Das Programm soll mit gcc test.c (also ohne Optionen) übersetzt werden.[1] Disassemblieren Sie den vom GCC erzeugten Code (objdump -S ./a.out). Geben Sie das Programmfragment in Assembler an, welches den Ausdruck `erg = n * 2;` berechnet.

Lösungsvorschlag:

Der Ausdruck `erg = n * 2;` wird durch ein `lsl r3,r3,#1` realisiert.

```
1044c: e3a03014 mov r3, #20
10450: e50b300c str r3, [fp, #-12]
10454: e51b300c ldr r3, [fp, #-12]
10458: e3530014 cmp r3, #20
1045c: 1a000002 bne 1046c <main+0x2c>
10460: e51b300c ldr r3, [fp, #-12]
10464: e1a03083 lsl r3, r3, #1
```

c) Der Ausdruck im obigen C-Programm hat nun die Form `erg = n * 8;`. Disassemblieren Sie den vom GCC erzeugten Code (objdump -S ./a.out). Geben Sie das Programmfragment in Assembler an, welches den Ausdruck `erg = n * 8;` berechnet.

Lösungsvorschlag:

Der Ausdruck `erg = n * 8;` wird durch ein `lsl r3,r3,#3` realisiert.

```
1044c: e3a03014 mov r3, #20
10450: e50b300c str r3, [fp, #-12]
10454: e51b300c ldr r3, [fp, #-12]
10458: e3530014 cmp r3, #20
1045c: 1a000002 bne 1046c <main+0x2c>
10460: e51b300c ldr r3, [fp, #-12]
10464: e1a03183 lsl r3, r3, #3
```

d) Der Ausdruck im obigen C-Programm hat nun die Form `erg = n / 8;`. Disassemblieren Sie den vom GCC erzeugten Code (objdump -S ./a.out). Geben Sie das Programmfragment in Assembler an, welches den Ausdruck `erg = n / 8;` berechnet. Erklären Sie das Programmfragment.

Lösungsvorschlag:

Man sieht in dem Programmfragment mehrere interessante Konstrukte. Zum einen ist es bei ARM möglich, Befehle konditional auszuführen. Beispiele sind `movlt` und `movge` an den Adressen 1046c und 10470. Das Dividieren passiert durch einen Rechtsshift. Dieser muss ein arithmetischer Rechtsshift sein, da beim Datentyp int auch negative Werte möglich sind und die notwendigen Einsen nachgezogen werden müssen, damit das Vorzeichen erhalten bleibt. Der Additionsbefehl an Adresse 10464 sorgt dafür, dass die Division auf Integerwerten als Modulo-Division implementiert wird. Wird die Addition nicht durchgeführt, würde z. B. die Division der Zahl -7 (mit 4 Bit dargestellt 1001) durch die Zahl 2 als Ergebnis -4 (oder 1100) ergeben. Durch Addition mit 0001 wird die Bitkombination zu 1010 und der Rechtsshift führt zu 1101, was der Darstellung der -3 entspricht.

```
1044c: e3a03014 mov r3, #20
10450: e50b300c str r3, [fp, #-12]
10454: e51b300c ldr r3, [fp, #-12]
10458: e3530014 cmp r3, #20
1045c: 1a000006 bne 1047c <main+0x3c>
```

[1] Nutzer*innen des clientssh-arm bitte daran denken, den `arm-linux-gnueabihf-gcc` aufzurufen.

```
10460: e51b300c ldr r3, [fp, #-12]
10464: e2832007 add r2, r3, #7
10468: e3530000 cmp r3, #0
1046c: b1a03002 movlt r3, r2
10470: a1a03003 movge r3, r3
10474: e1a031c3 asr r3, r3, #3
```

Aufgabe 3: Reverse Engineering

Gegeben ist folgender Object Dump:

```
00010440 <up>:
   10440: e92d4800  push {fp, lr}
   10444: e28db004  add fp, sp, #4
   10448: e24dd010  sub sp, sp, #16
   1044c: e50b0010  str r0, [fp, #-16]
   10450: e51b3010  ldr r3, [fp, #-16]
   10454: e3530000  cmp r3, #0
   10458: 1a000001  bne 10464 <up+0x24>
   1045c: e3a03000  mov r3, #0
   10460: ea000010  b 104a8 <up+0x68>
   10464: e51b3010  ldr r3, [fp, #-16]
   10468: e3530001  cmp r3, #1
   1046c: 1a000001  bne 10478 <up+0x38>
   10470: e3a03001  mov r3, #1
   10474: ea00000b  b 104a8 <up+0x68>
   10478: e51b3010  ldr r3, [fp, #-16]
   1047c: e2433002  sub r3, r3, #2
   10480: e1a00003  mov r0, r3
   10484: ebffffed  bl 10440 <up>
   10488: e50b0008  str r0, [fp, #-8]
   1048c: e51b3010  ldr r3, [fp, #-16]
   10490: e2433001  sub r3, r3, #1
   10494: e1a00003  mov r0, r3
   10498: ebfffe8  bl 10440 <up>
   1049c: e1a02000  mov r2, r0
   104a0: e51b3008  ldr r3, [fp, #-8]
   104a4: e0823003  add r3, r2, r3
   104a8: e1a00003  mov r0, r3
   104ac: e24bd004  sub sp, fp, #4
   104b0: e8bd8800  pop {fp, pc}

000104b4 <main>:
   104b4: e92d4800  push {fp, lr}
   104b8: e28db004  add fp, sp, #4
   104bc: e24dd008  sub sp, sp, #8
   104c0: e3a03014  mov r3, #20
   104c4: e50b3008  str r3, [fp, #-8]
   104c8: e51b0008  ldr r0, [fp, #-8]
   104cc: ebffffdb  bl 10440 <up>
   104d0: e50b000c  str r0, [fp, #-12]
   104d4: e51b100c  ldr r1, [fp, #-12]
   104d8: e59f0010  ldr r0, [pc, #16] ; 104f0 <main+0x3c>
   104dc: ebffff81  bl 102e8 <printf@plt>
   104e0: e3a03000  mov r3, #0
   104e4: e1a00003  mov r0, r3
   104e8: e24bd004  sub sp, fp, #4
```

```
104ec: e8bd8800  pop {fp, pc}
104f0: 00010564  .word 0x00010564
```

a) Analysieren Sie den Object Dump. Was für eine Funktion wird realisiert.

Lösungsvorschlag:

Rekursive Berechnung der Fibonacci-Zahlen.

b) Implementieren Sie das Programm in C und testen Sie die Funktionsweise.

Lösungsvorschlag:

```c
#include <stdio.h>

int up(int n)
{
  int x;
  if(n == 0)
  {
    return 0;
  }
  else if(n == 1)
  {
    return 1;
  }
  else
  {
    x = up(n - 2);
    return x + up(n - 1);
  }
}

int main(void)
{
  int n = 20;
  int erg;

  erg = up(n);

  printf("Ergebnis_%d\n", erg);

  return 0;
}
```

Aufgabe 4: Vervollständigen von Codestücken

Gegeben ist nachfolgend ein Programm als C-Code und Assembler-Code. In Beiden fehlen jeweils einzelne Codestücke.

```c
#include <stdio.h>

#define EZIS 2

int d[EZIS] = {6, 12};
int e[EZIS] = {1, 0};
```

```
.global main
.data
varZ: ________________
varY: ________________
string: .asciz "%d\n"
.text
label_1:
```

```c
int a() {
    int c = 0;
    for (int f = 0; f < EZIS; f++) {
        ________________
    }
    return c;
}

int b(int f[EZIS]) {
    ________________
    for (int h = 0; h < EZIS; h++) {
        c += f[h];
    }
    return c;
}

int main() {
    ________________
    f = f * b(e);
    ________________
    return 0;
}
```

```
    ________________
    mov r0, #0
    mov r1, #0
    ldr r2, =varZ
    ldr r3, =varY
________________
    cmp r1, #2
    bge label_3
    ________________
    ldr r5, [r2, r4]
    ldr r6, [r3, r4]
    mul r5, r6, r5
    add r0, r0, r5
    add r1, r1, #1
    b label_2
label_3:
    ________________
    bx lr
label_4:
    push {r4}
    mov r2, r0
    mov r0, #0
    mov r1, #0
label_5:
    cmp r1, #2
    bge label_6
    ________________
    ________________
    ________________
    add r1, r1, #1
    b label_5
label_6:
    pop {r4}
    bx lr
main:
    push {lr}
    bl label_1
    mov r1, r0
    push {r1}
    ldr r0, =varY
    ________________
    pop {r1}
    mul r1, r0, r1
    ldr r0, =string
    bl printf
    mov r0, #0
    pop {lr}
    bx lr
```

a) Vollständigen Sie den C-Code und Assembler-Code.

Tipp: Verwenden Sie dazu die Informationen aus dem Code der jeweils anderen Sprache.

Lösungsvorschlag:

```c
#include <stdio.h>

#define EZIS 2
```

```
.global main
.data
varZ: .word 6, 12
varY: .word 1, 0
```

```c
int d[EZIS] = {6, 12};
int e[EZIS] = {1, 0};

int a() {
    int c = 0;
    for (int f = 0; f < EZIS; f++) {
        c += d[f] * e[f];
    }
    return c;
}

int b(int f[EZIS]) {
    int c = 0;
    for (int h = 0; h < EZIS; h++) {
        c += f[h];
    }
    return c;
}

int main() {
    int f = a();
    f = f * b(e);
    printf("%d\n", f);
    return 0;
}
```

```
string: .asciz "%d\n"
.text
label_1:
    push {r4, r5, r6}
    mov r0, #0
    mov r1, #0
    ldr r2, =varZ
    ldr r3, =varY
label_2:
    cmp r1, #2
    bge label_3
    lsl r4, r1, #2
    ldr r5, [r2, r4]
    ldr r6, [r3, r4]
    mul r5, r6, r5
    add r0, r0, r5
    add r1, r1, #1
    b label_2
label_3:
    pop {r4, r5, r6}
    bx lr
label_4:
    push {r4}
    mov r2, r0
    mov r0, #0
    mov r1, #0
label_5:
    cmp r1, #2
    bge label_6
    lsl r3, r1, #2
    ldr r4, [r2, r3]
    add r0, r0, r4
    add r1, r1, #1
    b label_5
label_6:
    pop {r4}
    bx lr
main:
    push {lr}
    bl label_1
    mov r1, r0
    push {r1}
    ldr r0, =varY
    bl label_4
    pop {r1}
    mul r1, r0, r1
    ldr r0, =string
    bl printf
    mov r0, #0
    pop {lr}
    bx lr
```

b) Benennen Sie die Variablen und Funktionen bzw. Labels entsprechend ihrer Funktionalität um. Kommentieren Sie den Assembler-Code.

Lösungsvorschlag:

C-Code

```c
#include <stdio.h>

#define SIZE 2

int one[SIZE] = {6, 12};
int two[SIZE] = {1, 0};

int dotProduct() {
    int sum = 0;
    for (int i = 0; i < SIZE; i++) {
        sum += one[i] * two[i];
    }
    return sum;
}

int oneNorm(int vector[SIZE]) {
    int sum = 0;
    for (int i = 0; i < SIZE; i++) {
        sum += vector[i];
    }
    return sum;
}

int main() {
    int number = dotProduct();
    number = number * oneNorm(two);
    printf("%d\n", number);
    return 0;
}
```

Assembler-Code

```
.global main

.data
vectorOne: .word 6, 12
vectorTwo: .word 1, 0
string: .asciz "%d\n"

.text
dotProduct:
    push {r4, r5, r6}   /* store registers r4, r5, r6 (calling conventions) */
    mov r0, #0          /* r0 <- sum = 0 */
    mov r1, #0          /* r1 <- i = 0 */
    ldr r2, =vectorOne  /* r2 <- one */
    ldr r3, =vectorTwo  /* r3 <- two */

dotProduct_for:
    cmp r1, #2          /* i < 2 */
    bge dotProduct_done /* if false, goto dotProduct_done */

    lsl r4, r1, #2      /* r4 <- i * 4 */
    ldr r5, [r2, r4]    /* r5 <- one[i] */
    ldr r6, [r3, r4]    /* r6 <- two[i] */
    mul r5, r6, r5      /* r5 <- one[i] * two[i] */
    add r0, r0, r5      /* r0 <- sum += one[i] * two[i] */

    add r1, r1, #1      /* r1 <- i = i + 1 */
```

```
    b dotProduct_for    /* goto dotProduct_for */

dotProduct_done:
    pop {r4, r5, r6}    /* restore registers r4, r5, r6 (calling conventions) */
    bx lr               /* return sum */

oneNorm:                /* int oneNorm(r0 <- int matrix[2]) */
    push {r4}           /* store register r4 (calling convetions) */
    mov r2, r0          /* r2 <- matrix */
    mov r0, #0          /* r0 <- sum = 0 */
    mov r1, #0          /* r1 <- i = 0 */

oneNorm_for:
    cmp r1, #2          /* i < 2 */
    bge oneNorm_done    /* if false, goto oneNorm_done */

    lsl r3, r1, #2      /* r3 <- i * 4 */
    ldr r4, [r2, r3]    /* r4 <- matrix[i] */
    add r0, r4, r0      /* r0 <- sum = sum + matrix[i] */

    add r1, r1, #1      /* r1 <- i = i + 1 */
    b oneNorm_for       /* goto oneNorm_for */

oneNorm_done:
    pop {r4}            /* restore register r4 (calling convetions) */
    bx lr               /* return sum */

main:
    push {lr}           /* store link register */

    bl dotProduct       /* dotProduct() */
    mov r1, r0          /* r1 <- number = dotProduct() */

    push {r1}           /* store register r1 */
    ldr r0, =vectorTwo  /* r0 <- two */
    bl oneNorm          /* oneNorm(two) */
    pop {r1}            /* restore register r1 */
    mul r1, r0, r1      /* r1 <- number = number * oneNorm() */

    ldr r0, =string     /* r0 <- string */
    bl printf           /* printf("%d\n", number) */

    mov r0, #0          /* r0 <- 0 */
    pop {lr}            /* restore link register */
    bx lr               /* return 0 */
```