ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Technische Informatik 1 – Übung 1
Assembler (Computerübung)

Olga Saukh

13.-14. Oktober 2011

Ziele der Übung

Aufgabe 1

- das erste eigene Assembler-Programm
- Umgang mit dem Debugger

Aufgabe 2 (Zusatzaufgabe)

- Lesen und Analysieren von Maschinencode

Voraussetzungen

- einfache Programmierkenntnisse
- MIPS-Assembler
  - Vorlesung Kapitel 2 (v.a. Folien 7-18)
- Übungsmaterialien von der TI1-Homepage
  http://www.tik.ee.ethz.ch/tik/education/lectures/TI1/
  - Entpacken mit
    `unzip exercise_01.zip`

Aufgabe 1.1: Skalarprodukt

```
int skalar(int* A, int* B, int n) {
```

$$s = \sum_{i=1}^{n} A_i \cdot B_i$$

```
}
```

Aufgabe 1.1: skalar.S

```
...
         .globl   skalar
         .ent     skalar,0
skalar:
         .frame   sp,0,ra

         /* pre-condition:
          * a0 enthaelt Adresse von Vektor A
          * a1 enthaelt Adresse von Vektor B
          * a2 enthaelt die Anzahl der Elemente in den Vektoren
          */

         /* Implementierung */

         /* post-condition:
          * das resultat des skalarproduktes ist in register v0 gespeichert
          */

         j        ra
         .end     skalar
```

Aufgabe 1.1: Register und Speicher

- Register sind typ-lose 32-Bit-Werte
- interpretiert als
  - vorzeichenbehaftete Zahl

```
add $a0, $a0, 4
slt $t0, $a0, $a1
```

  - vorzeichenlose Zahl

```
addu $a0, $a0, 4
sltu $t0, $a0, $a1
```

  - Adresse

```
lw $t0, 0($a0)
sw $t0, 0($a0)
```

Aufgabe 1.1: Assembler 1x1

- Planen Sie voraus.
  – Notieren Sie die Lösung in Pseudocode.
  – Überlegen Sie, welche Register Sie verwenden.
- Dokumentieren Sie den Code.
  – Verwenden Sie aussagekräftige Labels.
  – Kommentare, Kommentare, Kommentare. /* … */
- Machen Sie vom Debugger Gebrauch.

Aufgabe 1.1: Technische Hinweise

- Register mit Namen verwenden

  `t0, t1, a0, v0, …` anstatt `$1, $2, $3`

- Grösse der Felder beim Zugriff auf Arrays beachten

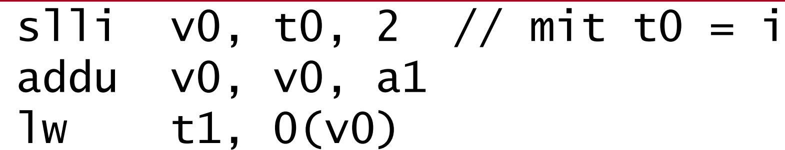
```
lw v0, 0(a0)  // lädt a0[0] für Int-Arrays
lw v0, 4(a0)  // lädt a0[1] für Int-Arrays
```

- bei Multiplikation 32-Bit-Ergebnis verwenden (`mflo`)

```
mult t0, t1   // {hi,lo} = t0 * t1
mflo t2       // t2 := lo
```

Lösung 1.1: Naiver Ansatz

```
int skalar(int* A, int* B, int n) {
   int result = 0;
   for (int i = 0; i < n; i++) {
      result += A[i] * B[i];
   }
   return result;
}
```



```
slli  v0, t0, 2  // mit t0 = i
addu  v0, v0, a1
lw    t1, 0(v0)
```

- **wenn immer möglich Zeiger direkt inkrementieren**

Lösung 1.1: Zeigerarithmetik

```
int skalar(int* A, int* B, int n) {
   int result = 0;
   for (int i = 0; i < n; i++) {
        result += *A * *B;
        A++;
        B++;
   }
   return result;
}
```

- **for-Schleifen in while-Schleifen umwandeln**
- **wenn möglich, Argumente direkt als Schleifenvariable verwenden**

Lösung 1.1: Finaler Pseudocode

```
int skalar(int* A, int* B, int n) {
   int result = 0;
   while (n>0) {
       result += *A * *B;
       A++;
       B++;
       n--;
   }
   return result;
}
```

Lösung 1.1: Assemblercode

```
skalar:
   move v0, zero                          /* result = 0 */
skalar_loop:
   beq  a2, zero, skalar_end_loop         /* while n > 0 */
   sub  a2, a2, 1                         /* n-- */
   lw   t0, 0(a0)                         /* lade *A */
   addi a0, a0, 4                         /* A++ */
   lw   t1, 0(a1)                         /* lade *B */
   addi a1, a1, 4                         /* B++ */
   mult t0, t1                            /* *A * *B */
   mflo t2
   add  v0, v0, t2                        /* result = result + *A * *B  */
   j    skalar_loop
skalar_end_loop:
   j ra                                   /* return result */
```

Zusatzaufgabe

Aufgabe 1.2: Matrixmultiplikation

```
static void matrix(int *a, int *b, int *c, int n, int m) {
   int i,j,k,sum;
   for(i=0; i < n; i++) {
      for(j = 0; j < n; j++) {
         sum= 0;
         for(k = 0; k < m; k++) {
            sum = sum + (a[i * m + k] * b[k * n + j]);
         }
         c[i * n + j] = sum;
      }
   }
}
```

Lösunghilfen:

- Initial bekannte Registerzuordnung notieren
- Aufteilung in sequentielle Blöcke, d.h Blöcke ohne Jumps
- Analyse von innen nach aussen

Aufgabe 1.2: Registerzuordnung

```
static void matrix(int *a, int *b, int *c, int n, int m)
    {
   int i,j,k,sum;
   for(i=0; i < n; i++) {
      for(j = 0; j < n; j++) {
         sum= 0;
         for(k = 0; k < m; k++) {
            sum = sum + (a[i * m + k] * b[k * n + j]);
         }
         c[i * n + j] = sum;
      }
   }
}
```

a0	→	*a
a1	→	*b
a2	→	*c
a3	→	n
16(sp)	→	m
t0	→	
t1	→	
t2	→	
t3	→	
t4	→	
t5	→	
t6	→	
t7	→	

- **Tabelle mit Register-Variablen-Zuordnung anlegen und mitführen**
- **v0 und v1 werden für Zwischenergebnisse verwendet**
- **Wohin wird m geladen?**

Aufgabe 1.2: Aufteilung in Blöcke

- Neuer Block beginnt nach der *einem Jump-Branch-Befehl folgenden* Anweisung:

```
00000000 <matrix>:
   0:   8fad0010        lw      t5,16(sp)
   4:   18e00031        blez    a3,cc <matrix+0xcc>
   8:   00000000        nop
-----------------------------------------------------
   c:   00007821        move    t7,zero
  10:   08000028        j       a0 <matrix+0xa0>
  14:   00000000        nop
-----------------------------------------------------
  18:   01070018        mult    t0,a3
```

Aufgabe 1.2: Codeanalyse

```
static void matrix(int *a, int *b, int *c, int n, int m)
    {
   int i,j,k,sum;
   for(i=0; i < n; i++) {
      for(j = 0; j < n; j++) {
         sum= 0;
         for(k = 0; k < m; k++) {
            sum = sum + (a[i * m + k] * b[k * n + j]);
         }
         c[i * n + j] = sum;
      }
   }
}
```

```
a0      → *a
a1      → *b
a2      → *c
a3      → n
16(sp)  → m

t0      → k
t1      →
t2      →
t3      →
t4      →
t5      →
t6      →
t7      →
```

= längster Codeblock:

```
18:    mult    t0,a3
1c:    mflo    v0
…
```

Aufgabe 1.2: Compilierung

- Compilierung:

```
mips-elf-gcc -fno-delayed-branch -O -c matrix.c
```

grosses "O" wie in Optimierung!

- Dissassemblierung:

```
mips-elf-objdump --disassemble matrix.o
```

Lösung 1.2: Schleifen-Optimierung

- arithmetische Operationen werden aus der inneren Schleife herausgezogen:

in i-Schleife initialisiert

in j-Schleife initialisiert,
in k-Schleife inkrementiert

Lösung 1.2: Allgemeiner Schleifenaufbau

```
    /* Schleifenzähler initialisieren */
    b [Schleifenkondition falsch], Ende
    j Init
Schleife:
    /* Schleifenkörper */
    /* Schleifenzähler weitersetzen */
    b [Schleifenkondition falsch], Ende
Init:
    /* Schleifenvariablen initialisieren */
    j Schleife
Ende:
```

Lösung 1.2: k-Schleife

```
for (k = 0; k < m; k++)
    sum = sum + (a[i * m + k] * b[k * n + j]);
}
```

```
...
inner_loop:
  18: mult  t0,a3
  1c: mflo  v0            // v0 = k * n
  20: addu  v0,t3,v0      // v0 = k * n + j
  24: sll   v0,v0,0x2     // * 4 (word-addr)
  28: addu  v0,v0,a1      // v0 = &b[k*n + j]
  2c: lw    v1,0(t1)      // v1 = a[i*m + k]
  30: lw    v0,0(v0)      // v0 = b[k*n + j]
  34: nop
  38: mult  v1,v0
  3c: mflo  v1            // v1 = a... * b...
  40: addu  t2,t2,v1      // sum = sum + v1
  44: addiu t0,t0,1       // k++
  48: addiu t1,t1,4       // t1 = nächstes a
  4c: bne   t5,t0,18      // if (k!=m) goto
    inner_loop
...
```

```
a0      → *a
a1      → *b
a2      → *c
a3      → n
16(sp)  → m

t0      → k
t1      → akt. *a
t2      → sum
t3      → j
t4      →
t5      → m
t6      →
t7      →
```

Lösung 1.2: j-Schleife

```
...
inner_loop:
  ...
store:
  54: sw     t2,0(t4)                      // c[i*n + j] = sum
  58: addiu  t3,t3,1                       // j++
  5c: addiu  t4,t4,4                       // t4 = nächste Adresse in c
  60: beq    a3,t3,94 <matrix+0x94>        // if (j==n) goto neue_zeile
  64: nop
testm:
  68: bgtz   t5,7c <matrix+0x7c>           // if (m>0) goto init_inner_loop
  6c: nop
  70: move   t2,zero                       // sum = 0
  74: j      54 <matrix+0x54>              // goto store
  78: nop
init_inner_loop:
  7c: sll    v0,t6,0x2                     // v0 = 4 * i * m (word-addr)
  80: addu   t1,a0,v0                      // t1 = &a[i*m]
  84: move   t0,zero                       // k = 0
  88: move   t2,zero                       // sum = 0
  8c: j      18 <matrix+0x18>              // goto inner_loop
...
```

```
for (j = 0; j < n; j++)
   {
   sum = 0;
   for (k = …)
   c[i * n + j] = sum;
}
```

```
a0      → *a
a1      → *b
a2      → *c
a3      → n
16(sp)  → m

t0      → k
t1      → akt. *a
t2      → sum
t3      → j
t4      → akt. *c
t5      → m
t6      → i*m
t7      →
```

Lösung 1.2: i-Schleife

```
...
inner_loop:
...
testm:
...
neue_zeile:
  94:  addiu   t7,t7,1                 // i++
  98:  beq     a3,t7,cc <matrix+0xcc>  // if (i==n) goto end
  9c:  nop
init_outer_loop:
  a0:  mult    t7,t5
  a4:  mflo    t6                      // t6 = i * m
         ...
  b0:  mult    t7,a3
  b4:  mflo    v0
  b8:  sll     v0,v0,0x2               // v0 = 4 * i * n
  bc:  addu    t4,a2,v0                // t4 = &c[i*n]
  c0:  move    t3,zero                 // j = 0
  c4:  j       68 <matrix+0x68>        // goto testm
...
```

```
for (i = 0; i < n; i++) {
   for (j = …)
}
```

```
a0      → *a
a1      → *b
a2      → *c
a3      → n
16(sp)  → m

t0      → k
t1      → akt. *a
t2      → sum
t3      → j
t4      → akt. *c
t5      → m
t6      → i*m
t7      → i
```

Lösung 1.2: Funktionsrahmen

```
00000000 <matrix>:
   0: lw     t5,16(sp)              // t5 = m
   4: blez   a3,cc <matrix+0xcc>    // if (n<=0) goto end
   8: nop

   c: move   t7,zero                // i = 0
  10: j      a0 <matrix+0xa0>       // goto init_outer_loop
inner_loop:
...
end:
  cc: jr     ra                     // return
  d0: nop
```

```
a0      → *a
a1      → *b
a2      → *c
a3      → n
16(sp)  → m

t0      → k
t1      → akt. *a
t2      → sum
t3      → j
t4      → akt. *c
t5      → m
t6      → i*m
t7      → i
```