

Prof. L. Thiele

Technische Informatik 1 - HS 2011

Lösungsvorschläge für Übung 1

Datum: 13.10.2011

1 Aufgaben

Diese Übung soll Ihnen einen praktischen Einstieg in die MIPS-Programmierung geben. In der ersten Aufgabe erstellen Sie ein einfaches MIPS-Assembler Programm, welches das Skalarprodukt implementiert. Im zweiten Teil der Übung analysieren Sie das Resultat der Compilation einer Matrixmultiplikations-Funktion.

Bevor Sie mit der Übung beginnen, laden Sie von der TI1-Homepage das Archiv mit den benötigten Rahmenprogrammen herunter und entpacken Sie es in Ihrem Homeverzeichnis.

1.1 Skalarprodukt in MIPS Assembler

In dieser Aufgabe soll das Skalarprodukt zweier Vektoren ausgerechnet werden. Die Formel

$$s = \sum_i A_i \cdot B_i$$

soll als Funktion `skalar(int* A, int* B, int n)` in Assembler implementiert werden. Die Vektoren sind Arrays von 32-bit Integer Werten.

Im Verzeichnis "skalar" befinden sich entsprechende Vorlagen. Das C-Programm `skalar-main.c` ist das Rahmenprogramm zum Test der Skalarprodukt-Funktion. Das Skalarprodukt selbst wird im File `skalar.S` implementiert. Für Ihre Skalarprodukt-Implementierung gelten folgenden Pre- und Postconditions:

- In Register `a0` steht die Adresse des Vektors A. In Register `a1` steht die Adresse des Vektors B. Register `a2` enthält die Länge der Vektoren. Auf dem MIPS-Prozessor ist es eine Konvention, die `a`-Register (`a` steht für Argument) für Funktionsparameter zu verwenden.
- Ist das Skalarprodukt berechnet, soll das Resultat in Register `v0` abgespeichert werden. Dies ist wiederum eine Konvention, die auch vom C-Compiler eingehalten wird. D.h. wenn in einer aufrufenden Funktion `x = skalar(A,B, 100);` steht, erzeugt der Compiler Code, um Register `v0` nach dem Sprung zu `skalar` der Variablen `x` zuzuweisen (`x` ist ein Register oder steht im Speicher, je nach Typ der Variablen).
- In der Assemblerfunktion `skalar` im File `skalar.S` steht im vorgegebenen Code die fixe Zuweisung `li v0, 999`, damit die Funktion einen definierten Wert zurückgibt. Ersetzen Sie diese konstante Zuweisung nach der Implementierung mit dem tatsächlichen Skalarprodukt.

1.1.1 Arbeiten mit dem Debugger

Untersuchen Sie zunächst das gegebene Programmskelett mit Hilfe des MIPS-Simulators. In dieser einführenden Übung betrachten wir Compiler, Assembler und MIPS-Simulator als Blackbox. In einer nächsten Übung werden Sie Gelegenheit haben, die Funktion dieser Tools im Detail zu untersuchen.

Folgen Sie den folgenden Anweisungen, um das Programm zu compilieren und zu simulieren:

- Verwenden Sie das vorgegebene Makefile um `skalar.S` und das Rahmenprogramm `skalar-main.c` zu compilieren. Benutzen Sie dazu das Kommando `make`. Dabei wird das auf dem MIPS-Simulator ausführbare Programm `skalar` erstellt.
- Das kompilierte Programm lässt sich nun mit einem MIPS-CPU-Simulator ausführen. Wir verwenden dazu den im GNU Debugger (GDB) integrierten Simulator sowie Insight, ein grafisches Frontend zu GDB. Rufen sie Insight zusammen mit dem auszuführenden Binary auf:
`mips-elf-insight skalar &`
- Führen Sie das Programm mittels `Run` aus. Beim ersten Starten werden Sie aufgefordert, das Target einzurichten. Wählen sie unter Target `Simulator` und deselektieren Sie die Optionen `Set breakpoint at 'main'` und `Set breakpoint at 'exit'`. Kontrollieren Sie ausserdem, dass unter `More Options` die Optionen `Attach to Target`, `Download Program` und `Run Program` selektiert sind.

Erkunden Sie nun die Funktionen des Debuggers:

- Lassen Sie den Sourcecode `skalar-main.c` in verschiedenen Ansichten anzeigen (als C-Sourcecode, Assembler oder im gemischten C/Assembler-Modus)
- Setzen Sie nun einen Breakpoint an den Anfang der `main`-Funktion indem sie auf das Minus am Anfang der entsprechende Source-Code-Zeilen klicken.
- Führen Sie das Programm noch einmal aus. Bewegen Sie sich dann mit Hilfe der Funktionen `Step`, `Next`, `Finish` und `Continue` bis zum Anfang der Funktion `skalar`.
- Öffnen Sie nun die Registeransicht (View → Register), sowie die Speicheransicht (View → Memory). Finden Sie die Eingabevektoren A und B im Speicher. Was enthalten Sie?

Lösung:

Die Adresse des Vektors A kann aus Register `a0` ermittelt werden, die des Vektors B aus `a1`. An der Entsprechenden Adresse im Speicher befinden sich die Zahlen 0, 2, 4, ..., 20 jeweils als 32-Bit-Wort gespeichert.

1.1.2 Implementierung

Vervollständigen Sie die Implementierung der Skalarprodukt Funktion in `skalar.S`. Sie können das Programm anschliessend mit dem MIPS-Prozessor-Simulator testen.

Für die Implementierung benötigen Sie die Instruktion `mult`. Diese erwartet zwei Register als Eingabe und legt das Ergebnis in den Spezialregistern `hi` und `lo`¹ ab. Diese können Sie mit Hilfe der Befehle `mfhi` und `mflo` auslesen:

```
mult $Rsrc1, $Rsrc2          {$hi, $lo} = $Rsrc1 * $Rsrc2
mfhi $Rdst                  $Rdst := $hi
mflo $Rdst                  $Rdst := $lo
```

Neben den in der Vorlesung eingeführten MIPS-Instruktionen versteht der Compiler auch noch folgende Pseudo-Instruktionen, die sie verwenden können:

```
move $Rdest, $Rsrc          $Rdest := $Rsrc
li $Rdest, immediate       $Rdest := immediate
bgt $Rsrc1, $Rsrc2, $Label  if ($Rsrc1 > $Rsrc2) then goto $Label
bge $Rsrc1, $Rsrc2, $Label  if ($Rsrc1 >= $Rsrc2) then goto $Label
blt $Rsrc1, $Rsrc2, $Label  if ($Rsrc1 < $Rsrc2) then goto $Label
ble $Rsrc1, $Rsrc2, $Label  if ($Rsrc1 <= $Rsrc2) then goto $Label
blez $Rsrc, $Label         if ($Rsrc <= 0) then goto $Label
```

¹Zwei Ergebnisregister werden benötigt, da das grösstmögliche Ergebnis der Multiplikation zweier 32-Bit-Zahlen eine 64-Bit-Zahl ergibt. Für diese Übung reicht es, wenn sie davon ausgehen, dass das Ergebnis maximal 32 Bit gross ist und entsprechend das `hi`-Register ignorieren.

Lösung:

```
/* Skalarprodukt in MIPS assembler
 *
 * interface: extern skalar(int* a, int* b, int n);
 */

#include "regdef.h"

    .globl skalar
    .ent skalar,0
skalar:
    .frame sp,0,ra
    /* precondition:
     * a0 enthaelt adresse von vektor A
     * a1 enthaelt adresse von vektor B
     * a2 enthaelt die anzahl der elemente in den vektoren
     */

    move v0, zero /* resultat initialisieren */

skalar_loop:
    beq a2, zero, skalar_end_loop /* abbruch wenn keine elemente mehr */
    sub a2, a2, 1
    lw t0, 0(a0) /* ites element aus A laden */
    addi a0, a0, 4 /* adresse auf naechstes element A setzen */
    lw t1, 0(a1) /* ites element aus B laden */
    addi a1, a1, 4 /* adresse auf naechstes element B setzen */
    mult t0, t1 /* multiplizieren */
    mflo t2 /* resultat aus lo register abholen */
    add v0, v0, t2 /* produkte aufsummieren */
    j skalar_loop

skalar_end_loop:
    /* postcondition: resultat ist in register v0 gespeichert */
    j ra
.end skalar
```

1.2 Matrixmultiplikation in C (Zusatzaufgabe)

Am Beispiel der Matrixmultiplikation $C = A \times B$ soll analysiert werden, wie der C-Compiler Code für die MIPS-Architektur erzeugt. Der Einfachheit halber wird angenommen, dass die beiden Matrizen gleich gross sind: $A: (n \times m)$, $B: (m \times n)$, $C: (n \times n)$. Das Produkt ist dann definiert als

$$c_{ij} = \sum_{k=1}^m a_{ik} \cdot b_{kj}$$

und kann in C zum Beispiel folgendermassen implementiert werden:

```
static void matrix(int *A, int *B, int *C, int n, int m) {
    /* multipiziere die Matrizen A(n x m), B(m x n), Resultat in C */
    int i,j,k,sum;
    for (i = 0; i < n; i++) { /* alle Zeilen */
        for (j = 0; j < n; j++) { /* alle Spalten */
            sum = 0;

```

```

    for (k = 0; k < m; k++) { /* Skalarprodukt */
        sum = sum + (A[i * m + k] * B[k * n + j]);
    }
    C[i * n + j] = sum;
}
}
}

```

Genau dieser C-Code (verwenden Sie das vorgegebene File matrix.c) soll nun übersetzt werden mit dem Befehl `mips-elf-gcc -fno-delayed-branch -O -c matrix.c`.

Mit dem Befehl `mips-elf-objdump --disassemble matrix.o` lässt sich der vom Compiler generierte MIPS-Code anschauen ('disassemblieren'). Die Aufgabe besteht nun darin, die Matrixmultiplikation zu analysieren und zu dokumentieren. Schreiben Sie zu jeder Assembler-Zeile einen Kommentar, was gerade gemacht wird! Verwenden Sie dazu auch Referenzen auf den ursprünglichen C-Code.

Hinweis zur Parameterübergabe: Da die Matrixmultiplikations-Funktion 5 Parameter benötigt, die MIPS-Konvention aber nur 4 Register für Funktionsparameter reserviert, wird das letzte Argument auf dem Stack übergeben werden. In der Funktion matrix wird der Parameter `m` mit dem Assembler-Befehl `lw` von der Adresse `16(sp)` (vom Stack) in ein temporäres Register geladen. `A`, `B`, `C` und `n` sind in `a0` bis `a3` abgelegt.

Lösung:

Bemerkung: die Zuweisung zu temporären Registern kann variieren.

```

a0 = a
a1 = b
a2 = c
a3 = n

t0 = k
t1 = &a[i*m + k]
t2 = sum
t3 = j
t4 = &c[i*n + j]
t5 = m
t6 = i*m
t7 = i

00000000 <matrix>:
0:      8fad0010      lw      t5,16(sp)          // t5 = m
4:      18e00031      blez   a3,cc <matrix+0xcc> // if (n<=0) goto end
8:      00000000      nop

c:      00007821      move   t7,zero           // i = 0
10:     08000028      j      a0 <matrix+0xa0>   // goto init_outer_loop
14:     00000000      nop

inner_loop:
18:     01070018      mult   t0,a3
1c:     00001012      mflo   v0                // v0 = k * n
20:     01621021      addu   v0,t3,v0          // v0 = k * n + j
24:     00021080      sll    v0,v0,0x2         // * 4 (word-addr)
28:     00451021      addu   v0,v0,a1          // v0 = &b[k*n + j]
2c:     8d230000      lw     v1,0(t1)          // v1 = a[i*m + k]
30:     8c420000      lw     v0,0(v0)          // v0 = b[k*n + j]
34:     00000000      nop
38:     00620018      mult   v1,v0
3c:     00001812      mflo   v1                // v1 = a... * b...
40:     01435021      addu   t2,t2,v1          // sum = sum + v1
44:     25080001      addiu  t0,t0,1           // k++
48:     25290004      addiu  t1,t1,4           // t1 = nächste Adresse in a (&a[i*m +k])
4c:     15a8fff2      bne   t5,t0,18 <matrix+0x18> // if (k!=m) goto inner_loop
50:     00000000      nop

```

```

store:
54:      ad8a0000      sw      t2,0(t4)          // c[i*n + j] = sum
58:      256b0001      addiu   t3,t3,1          // j++
5c:      258c0004      addiu   t4,t4,4          // t4 = nächste Adresse in c
60:      10eb000c      beq     a3,t3,94 <matrix+0x94> // if (j==n) goto neue_zeile
64:      00000000      nop

testm:
68:      1da00004      bgtz   t5,7c <matrix+0x7c> // if (m>0) goto init_inner_loop
6c:      00000000      nop

70:      00005021      move   t2,zero          // sum = 0
74:      08000015      j      54 <matrix+0x54> // goto store
78:      00000000      nop

init_inner_loop:
7c:      000e1080      sll    v0,t6,0x2         // v0 = 4 * i * m (word-addr)
80:      00824821      addu   t1,a0,v0         // t1 = &a[i*m]
84:      00004021      move   t0,zero         // k = 0
88:      00005021      move   t2,zero         // sum = 0
8c:      08000006      j      18 <matrix+0x18> // goto inner_loop
90:      00000000      nop

neue_zeile:
94:      25ef0001      addiu   t7,t7,1         // i++
98:      10ef000c      beq     a3,t7,cc <matrix+0xcc> // if (i==n) goto end
9c:      00000000      nop

init_outer_loop:
a0:      01ed0018      mult   t7,t5
a4:      00007012      mflo   t6                // t6 = i * m

...

b0:      01e70018      mult   t7,a3
b4:      00001012      mflo   v0
b8:      00021080      sll    v0,v0,0x2         // v0 = 4 * i * n
bc:      00c26021      addu   t4,a2,v0         // t4 = &c[i*n]
c0:      00005821      move   t3,zero         // j = 0
c4:      0800001a      j      68 <matrix+0x68> // goto testm
c8:      00000000      nop

end:
cc:      03e00008      jr     ra                // return
d0:      00000000      nop

```