## Aufgabe 9

a) do ρ(ρ(γ)+100):=α;

```
lw $1, 100($1) # γ':=ρ(γ)+100
               # ich gehe davon aus, dass die Adressierung bereits
               # umgerechnet wurde, sonst müsste hier 25($1) stehen
sw $0, 0($1)   # ρ(γ'):=α
```

b) do α:=ρ(ρ(γ)+50);

```
lw $1, 50($1)  # γ':=ρ(γ)+50, zur Adressierung siehe Aufgabe a)
lw $0, 0($1)   # α:=ρ(γ')
```

c) if α=γ goto j;

```
beq $0, $1, j
```

d) do α:=α+γ;

```
add $0, $0, $1 # ich gehe davon aus, dass der Rechner in der Lage ist,
               # in einem Takt α auszulesen, neu zu berechnen und dann
               # zu beschreiben - falls nicht, müsste man ein drittes,
               # temporäres Register benutzen
```

## Aufgabe 10

a) In Pseudocode formuliert sich die Gleichung um als (Umrechnung 8-Bit-Adressen in 32-Bit-Adressen bereits beachtet):
ρ(168+$9) = ρ(512+$9) - ($18 - ρ(40+$9))
Der dazugehörige Assemblercode sieht dann folgendermaßen aus:

```
lw  $7, 40($9)       # lade A[10] in Register $7
sub $8, $18, $7      # in Register $8 steht nun h-A[10]
lw  $7, 512($9)      # lade A[128] in Register $7
sub $8, $7, $8       # $8 enthält das Resultat obiger Gleichung
sw  $8, 168($9)      # $8 in A[42] abspeichern
```

Etwas kritisch ist für mich der Befehl in Zeile 4, da er Register $8 als Source und als Destination benutzt. Man könnte die Klammer in obiger Gleichung auflösen und dann
ρ(168+$9) = ρ(512+$9) - $18 + ρ(40+$9)
berechnen, wodurch dieses Register-Problem entfällt, da man sehr schön von links nach rechts die Gleichung ausrechnen lassen kann.

b) Im folgenden liste ich die einzelnen Belegungen der Bestandteile der Befehle in Dezimalschreibweise auf, in der zweiten Tabelle steht dann der komplette Befehl als Binärcode:

| Befehl | op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|---|
| `lw $7, 40($9)` | 35 | 9 | 7 | 40 | | |
| `sub $8, $18, $7` | 0 | 18 | 7 | 8 | 0 | 34 |
| `lw $7, 128($9)` | 35 | 9 | 7 | 512 | | |
| `sub $8, $7, $8` | 0 | 7 | 8 | 8 | 0 | 34 |
| `sw $8, 168($9)` | 43 | 9 | 8 | 168 | | |

| Befehl | Binärcode |
|---|---|
| `lw $7, 40($9)` | `100011 01001 00111 0000000000101000` |
| `sub $8, $18, $7` | `000000 10010 00111 01000 00000 100010` |
| `lw $7, 128($9)` | `100011 01001 00111 0000001000000000` |
| `sub $8, $7, $8` | `000000 00111 01000 01000 00000 100010` |
| `sw $8, 168($9)` | `101011 01001 01000 0000000010101000` |

| | |
|---|---|
| op | Operation |
| rs | 1. Registersource |
| rt | 2. Registersource |
| rd | Registerdestination |
| shamt | shift |
| funct | Funktion bzgl. op |

**Aufgabe 13**

a) An der Adresse 0x00400020 befindet sich der erste Assemblerbefehl, der direkt aus unserem Programm `test01.s` stammt und nicht automatisch von SPIM generiert wurde:
`addu $s6,$s4,$s5`.

b) Die Argumentregister $a1 und $a2 werden mit Zeigern auf die dem Programm übergebenen Parameter und den Umgebungsvariablen gefüllt, $a0 wird 0 gesetzt. Weiterhin wird in Register $31 eine Adresse abgelegt, zu der ein Sprung das Programm sofort beendet.

c) Der Befehl `la $a0,mesg3` steht an Adresse 0x0040005c.
Die Registerbelegung sieht folgendermaßen aus:

```
R20             0000007b          123
R21             00000007            7
```

An Adresse 0x10010000 (also im Datensegment) beginnt die nullterminierte Zeichenkette mesg1. Hexadezimal wird sie geschrieben als:

```
6569530a 62616820 52206e65 73696765
20726574 62203032 67656c65 696d2074
00203a74
```

Nicht ganz erwartungsgemäß lautet die entsprechende Umsetzung in ASCII-Zeichen (Unterstriche entsprechen Leerzeichen, Sonderzeichen kursiv):

```
e i S \n b a h _  R _ n e  s i g e
_ r e t  b _ 0 2  g e l e  i m _ t
00_ : t
```

Diese scheinbare Unordnung rührt von der Darstellung der Daten auf der MIPS in der LSB- (least significant bit first) Systematik. In jedem 32-Bit-Wort müsste man zum korrekten Verständnis Byte 0 mit Byte 3 und Byte 1 mit Byte 2 vertauschen. Dann ergibt sich der offensichtlichere Inhalt des Datensegments:

```
\nS i e  _ h a b  e n _ R  e g i s
t e r _  2 0 _ b  e l e g  t _ m i
t : _ 00
```

d) Das Programm erzeugt folgende Bildschirmausgabe:

```
Sie haben Register 20 belegt mit: 123
Sie haben Register 21 belegt mit: 7
Das Ergebnis der Berechnung ist: 130
```

Die vom Benutzer manipulierten Register $20 und $21 wurden summiert und das Ergebnis in Register $22 geschrieben. Zur Bildschirmausgabe musste die Summe noch nach Register $a0 (d.h. 4) kopiert werden.

e) Die von SPIM automatisch erzeugten Initialisierungsroutinen (siehe Aufgabe b) werden nicht eingefügt, es steht nur der von uns gewollte Code im Speicher der virtuellen Maschine.

**Aufgabe 14**

Das Register $sp hat beim Laden eines beliebigen Programmes den Wert 0x7fffeffc. Der Stack ist leer, an der Adresse der ersten Speicherzelle, die belegt werden könnte, steht eine 32-Bit breite Null.

```
.text 0x400000
.globl main
```

```
main:
       li   $s0,1
       li   $s1,3
       li   $s2,11
       li   $s3,7
       li   $s4,17

       li   $s5,4

       sw  $s0,0($sp)
       sub $sp,$sp,$s5
       sw  $s1,0($sp)
       sub $sp,$sp,$s5
       sw  $s2,0($sp)
       sub $sp,$sp,$s5
       sw  $s3,0($sp)
       sub $sp,$sp,$s5
       sw  $s4,0($sp)
       sub $sp,$sp,$s5

       li    $v0,10
       syscall
```

Der Stackpointer $sp zeigt nach Ablauf des Programmes auf 0x7fffefec und der darauffolgende Speicher hat die Struktur:

```
0x7fffefec  0x00000011
0x7ffeff0  0x00000007
0x7ffeff4  0x0000000b
0x7ffeff8  0x00000003
0x7fffeffc  0x00000001
```

**Aufgabe 15**

Im folgenden der dokumentierte Quellcode:

```
# Codesegment startet an Standardadresse
       .text 0x00400000
# Einsprungpunkt global bekannt machen
       .globl main

main:

# laedt den Wert 0x55AA*2^16 (=0x55AA0000) in R07
       lui  $7,0x55AA

# vorzeichenbehaftetes OR von 0xAA55 (=-21930) mit dem Nullregister, R2=0xAA55
       ori  $2,$0,0xAA55

# R2 wird um 16 Bits nach rechts geschoben, das Ergebnis in R3 gespeichert
# da die obersten 16 Bits von R2 nicht gesetzt waren, ist R3=0
       srl  $3,$2,16

# R2 wird um 16 Bits nach links geschoben, das Ergebnis in R4 gespeichert
# da die untersten 16 Bits von R2=0xAA55 waren, ist R4=0xAA550000
       sll  $4,$2,16

# logische OR-Verknuepfung von R2 (=0xAA55) und R7 (=0x55AA0000)
# Ergebnis in R7 ist dann 0x55AAAA55
       or   $7,$7,$2

# addiert zu R7 das Nullregister, Ergebnis in R2=R7=0x55AAAA55
       add  $2,$0,$7

# dieser Befehl wird vom Assembler in 2 Befehle unterteilt:
# zuerst wird die Adresse der Daten im Datensegment in R1 geladen (=0x10010000)
# dann wird in R5 der Offset von Mem1 addiert (in diesem Falle 0), R5=0x10010000
       la   $5,Mem1

# das Datum an der Adresse von Mem1 wird in R5 geladen, R5=0x12345678
```

```
        lw   $6,0($5)

# Register R2 (=0x55AAAA55) wird 4 Bytes hinter Mem1, also bei Mem2 gespeichert
        sw   $2,4($5)

# das Programm tritt in eine Endlosschleife ein
ende:   j    ende


        .data
Mem1:   .word 0x12345678
Mem2:   .word 0
```

Ich habe die Veränderungen im program counter (PC-Register) nicht protokolliert, sie ergeben sich im Simulator durch die jeweiligen Befehlslängen. Einzig erwähnenswert ist die Endlosschleife beim Label `ende`, welche durch das stete Setzen des PC auf 0x00400028 entsteht. Damit sorgt der Befehl `j ende` dafür, dass der auf ihn folgende Befehl er selbst wieder ist.
Die Veränderungen von *Mem1* in *Mem2* lassen sich in zwei Gruppen einteilen:
Der Assembler setzt Hauptspeicherzugriffe in andere Adressen um: der Befehl `la $5, Mem1` verwies auf Adresse 0x10010000, der Befehl `la $5, Mem2` ordnet $5 die Adresse 0x10010004 zu.
Das Programm schreibt im Befehl `sw $2,4($5)` nun auch an einer anderen Speicheradresse (nämlich 0 statt 0x12345678) den Wert 0x55AAAA55.

## Aufgabe 16

Nachdem $16 mit dem Wert 9 und $17 mit dem Wert 0x0fffffff belegt wurden, werden diese Register mit verschiedenen Zahlen per sltiu-Befehl verglichen. Sein Aufgabe leitet sich aus dem Akronym direkt ab: set if lower than immediate unsigned, übersetzt etwa: setze, falls kleiner als vorzeichenlose Parameterzahl. Das boolesche Ergebnis dieses Vergleiches ist 1 für true bzw. 0 für false und wird im ersten Parameter-Register gespeichert. Im Programmverlauf werden die Register $18 bis $23 wie folgt geändert:

| Register | $18 | $19 | $20 | $21 | $22 | $23 |
|---|---|---|---|---|---|---|
| Wert | 1 | 1 | 1 | 0 | 1 | 0 |

Am auffälligsten ist, das negative Zahlen in ihrer Zweierkomplementdarstellung stets größer als positive Zahlen sind. Da aber explizit SPIM mitgeteilt wird, das ein vorzeichenloser Vergleich durchgeführt werden soll, bewirkt der Befehl `sltiu $19,$16,-20` eigentlich die Sequenz `sltiu $19,$16,0xffffffec`, was nun verständlicherweise true, d.h. eine 1 in $19 schreibt.