Aufgabe 1: MIPS-Macrobefehle

Im Anhang finden Sie einen Ausschnitt aus dem MIPS Befehlssatz. Dieser Befehlssatz ist turingvollständig, aber etwas klein.

a) Erweitern Sie den verfügbaren Befehlssatz um folgende Macro-Befehle (`%reg`, `%rs`, `%rt` sind Register, `%imm` und `%label` Konstanten bzw. Labels):

```
mov (%rd, %rs)           # Kopiert Wert von %rs nach %rd
push (%reg)          # Schiebt Wert %reg auf den Stack
pop (%reg)         # Holt den obersten Wert vom Stack
mult (%rd, %rs, %rt)  # %rd = %rs * %rt; %rd: least 32bit
div (%rd, %rs, %rt)   # %rd = %rs / %rt
mod (%rd, %rs, %rt)   # %rd = %rt % %rt
not (%reg)         # %reg = ~%reg
clear (%reg)       # %reg = 0
ror (%reg, %imm)     # Rotiert %imm Bits nach rechts
rol (%reg, %imm)     # Rotiert %imm Bits nach links
bgt (%rs, %rt, %label)  # if (%rs > %rt) goto %label
ble (%rs, %rt, %label)  # if (%rs <= %rt) goto %label
```

Beispiel:

```
.macro mov (%rd, %rs)
  add %rd, %rs, $zero
.end_macro
```

Hinweis: Der MIPS Stack ist empty descending. Alle MIPS-Register sind 32 bit *groß.*

b) Wann bzw. von wem werden in einer RISC Architektur Macrobefehle in Microbefehle übersetzt? Wann bzw. von wem in einer CISC Architektur (wenn überhaupt)?

Name/Syntax	Semantik	Bemerkung
`add $d, $s, $t`	`$d = $s + $t`	Addiert Register
`sub $d, $s, $t`	`$d = $s - $t`	
`addi $d, $s, C`	`$d = $s + C`	Addiert Konstante („Immediate“)
`subi $d, $s, C`	`$d = $s - C`	
`mult $s, $t`	`HI:LO = $s * $t`	`HI` und `LO` sind spez. Reg.
`multu $s, $t`	`HI:LO = $s * $t`	Wie `mult` nur vorzeichenlos.
`div $s, $t`	`LO = $s / $t, HI = $s % $t`	`LO` enthält Erg., `HI` Rest
`divu $s, $t`	`LO = $s / $t, HI = $s % $t`	Wie `div` nur vorzeichenlos.
`lw $t, C($s)`	`$t = Memory[$s + C]`	Lädt Speicher-Wort von Adresse
`sw $t, C($s)`	`Memory[$s + C] = $t`	Speichert Speicher-Wort an Adresse
`mfhi $d`	`$d = HI`	Move From `HI` Reg. in norm. Reg.
`mflo $d`	`$d = LO`	
`and $d, $s, $t`	`$d = $s & $t`	Bitweises UND
`andi $d, $s, C`	`$d = $s & C`	
`or $d, $s, $t`	`$d = $s | $t`	Bitweises OR
`ori $d, $s, C`	`$d = $s | C`	
`xor $d, $s, $t`	`$d = $s ^ $t`	Bitweises XOR
`xori $d, $s, C`	`$d = $s ^ C`	
`nor $d, $s, $t`	`$d = ~($s | $t)`	`~` ist der Bitwise Not-Operator in C
`slt $d, $s, $t`	`$d = ($s < $t)`	d wird 1, wenn $s < t$ ist, sonst 0
`sltu $d, $s, $t`	`$d = ($s < $t)`	wie `slt` nur Vorzeichenlos
`slti $d, $s, C`	`$d = ($s < C)`	
`sllv $d, $t, $s`	`$d = $t << $s`	t wird um s nach links geschoben
`srlv $d, $t, $s`	`$d = $t >> $s`	
`srav $d, $t, $s`	`$d = $t >> $s`	Wie `srlv` nur mit Vorzeichenbit
`sll $d, $t, C`	`$d = $t << C`	
`srl $d, $t, C`	`$d = $t >> C`	
`sra $d, $t, C`	`$d = $t >> C`	Wie `srl` nur mit Vorzeichenbit
`beq $s, $t, C`	`if ($s == $t) goto C`	Branch if Equal
`bne $s, $t, C`	`if ($s != $t) goto C`	
`j C`	`$gp = C`	Jump
`j $s`	`$gp = $s`	
`jal C`	`$ra = $gp + 4; $gp = C`	Jump And Link („call“)

Register	Nutzung
`$zero`	Konstant 0
`$at`	Assembler temporary
`$v0-$v1`	Funktionsrückgabewerte
`$a0-$a3`	Funktionsargumente
`$t0-$t9`	Zwischenspeicher
`$s0-$s7`	Nicht flüchtiger Zwischenspeicher
`$k0-$k1`	Kernel-Register
`$gp`	Instruction Pointer
`$sp`	Stack Pointer
`$fp`	Frame Pointer
`$ra`	Return Address

Tabelle 1: MIPS-Instruktionen & Registersatz

Aufgabe 2: Funktionspointer

Implementieren Sie eine Sortierungsfunktion in Assembler.

```
void sort(int64_t *base, size_t nel,
          int (*compar)(const int64_t *, const int64_t *));
```

Diese Funktion nimmt drei Argumente:

base Adresse eines Arrays mit 64 bit breiten Einträgen, bspw. `int64_t arr[10]`.

nel Die Anzahl der Elemente in diesem Array.

compar Eine Funktion die jw. zwei *Pointer* zu solchen Einträgen aus dem Array vergleichen kann.

Der Rückgabewert von `compar` ist kleiner, gleich, oder größer als 0, wenn der Vergleich zwischen den beiden übergebenen Werten kleiner, gleich öder größer ergeben hat.

Eine solche Funktion die selber eine Funktion (bzw. Adresse einer Funktion) als Argument bekommt nennt man eine Higher-Order Function.

Beispielaufruf der `compar`-Funktion mit den ersten beiden Elementen von `base`:

```
; rdi contains base == &base[0]
mov rsi, rdi
add rsi, 8
; rsi now contains base+8 == &base[1]
; call the hof
call rdx
; rax is < 0 if rdi < rsi
;        = 0 if rdi = rsi
;        > 0 if rdi > rsi
```

Sie können das Programm dann bspw. wiefolgt testen:

```
$ ./hofs 3 0 -1 7 2
-1
0
2
3
7
```