Programmierung 2 (SS 2021)
Universität des Saarlandes
Fakultät MI
Compiler Design Lab

Musterlösung 4
MIPS

Prof. Dr. Sebastian Hack
Julian Rosemann, B. Sc.
Thorsten Klößner, M. Sc.
Julia Wichlacz, M. Sc.
Maximilian Fickert, M. Sc.

Im Vorlesungskalender finden Sie Informationen über die Kapitel des Skripts, die parallel zur Vorlesung bearbeitet werden sollen bzw. dort besprochen werden. Die Übungsaufgaben dienen der Vertiefung des Wissens, das in der Vorlesung vermittelt wird und als Vorbereitung auf Minitests und Klausur.
Weitere Aufgaben zu den Themen finden Sie jeweils am Ende der Skriptkapitel.
Die Schwierigkeitsgrade sind durch Steine des 2048-Spiels gekennzeichnet, von 512 „leicht“ bis 2048 „schwer“. 4096 steht für Knobelaufgaben.

Aufgabe 4.0: Aufrufschachtel

Betrachten Sie das MIPS-Programm aus Abbildung 1.

```
1 .text
2     .globl confusion
3 confusion:
4     and       $t0 $t0 $zero
5     or        $t0 $t0 $a0
6     add       $t1 $a1 $t0
7     or        $s1 $t0 $t1
8     move      $s0 $t0
9     jal       mystery
10    add       $t0 $t1 $v0
11    add       $v0 $a0 $t0
12    jr        $ra
```

Abbildung 1: Programm, das die Aufrufkonvention missachtet.

1. Vervollständigen Sie das Programm so, dass es die Aufrufkonvention implementiert. Seien Sie dazu so speichereffizient wie möglich, das heißt sichern Sie nur die notwendigen Register. Geben Sie jeweils an, welchen Code Sie zwischen welchen Zeilen einfügen müssen.
2. Zeichnen Sie ein Abbild des Stacks (Kellers) vor der Ausführung des Befehls `jal mystery`. Geben Sie zusätzlich die Adresse des Stackpointers zu diesem Zeitpunkt an. Nehmen Sie an, dass der Stackpointer beim Aufruf von `confusion` den Wert 0x7ffffffc hat. Vervollständigen Sie dazu Tabelle1.

Inhalt	Adresse
	0x7ffffffc
...	...

Tabelle 1: Initiales Abbild des Stacks zum Zeitpunkt des Aufrufs von `confusion`.

3. Erklären Sie, warum Sie eine solch „umständliche“ Sicherung auf dem Stack vornehmen müssen. Wäre es nicht einfacher, wenn ein Unterprogramm `foo` seine Register an eine fest vereinbarte Stelle im statischen Bereich des Datensegments speichert?

Lösung

1. Prolog zwischen 3 und 4:

```
1     addiu     $sp $sp -8
2     sw        $s0 ($sp)
3     sw        $s1 4($sp)
```

sichern zwischen 8 und 9

```
1     addiu     $sp $sp -12
2     sw        $t1 ($sp)
3     sw        $a0 4($sp)
4     sw        $ra 8($sp)
```

laden zwischen 9 und 10

```
1     lw        $t1 ($sp)
2     lw        $a0 4($sp)
3     lw        $ra 8($sp)
4     addiu     $sp $sp 12
```

Epilog zwischen 11 und 12

```
1     lw        $s0 ($sp)
2     lw        $s1 4($sp)
3     addiu     $sp $sp 8
```

2.

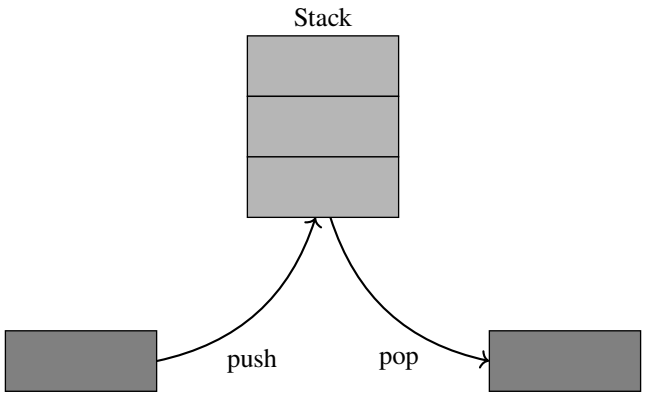
Inhalt	Adresse
	0x7ffffffc
$s1	0x7fffffff8
$s0	0x7fffffff4
$ra	0x7fffffff0
$a0	0x7fffffec
$t1	0x7fffffe8

Die aktuelle Adresse des Stackpointers lautet: 0x7fffffe8.

3. Rekursive Funktionen würden damit ihre Daten aus dem vorherigen Aufruf überschreiben.

Aufgabe 4.1: Der Stack

Der Stack (Keller) dient der dynamischen Speicherverwaltung. In der folgenden Aufgaben werden Sie die wichtigsten Instruktionen eines Stacks, `push` und `pop`, kennen lernen. Beachten Sie dabei, dass der Stack nach unten wächst.



a) Wie können Sie die Operation `push $reg` in MIPS darstellen? Hierbei soll `push $reg` den Inhalt des Registers `$reg` auf den Stack legen. Sie dürfen annehmen, dass dieses Register immer `$a0` ist.

b) Wie können Sie die Operation `pop $reg` in MIPS darstellen? Hierbei soll `pop $reg` das oberste Element des Stacks in Register `$reg` legen. Sie dürfen annehmen, dass dieses Register immer `$a0` ist.

Lösung

Implementierung von `push` und `pop` für `$a0|`:

```
1 push:
2     subiu $sp $sp 4
3     sw $a0 ($sp)
4
5 pop:
6     lw $a0 ($sp)
7     addiu $sp $sp 4
```

Implementierung von `push` und `pop` als Macros für allgemeine Register:

```
1 .MACRO push (%reg)
2     subiu $sp $sp 4
3     sw %reg ($sp)
4 .END_MACRO
5
6 .MACRO pop (%reg)
7     lw %reg ($sp)
8     addiu $sp $sp 4
9 .END_MACRO
```

Aufgabe 4.2: Funktionen

Implementieren Sie folgende Funktionen in MIPS. Die Argumente der Funktion werden in den Registern $\$a0$ und $\$a1$ übergeben. In $\$v0$ soll der berechnete Funktionswert zurückgegeben werden.

1. $avg : \mathbb{N}^2 \to \mathbb{N}, avg(x, y) \mapsto (x + y)/2$
2. $sgn : \mathbb{N} \to \mathbb{N}, sgn(x) \mapsto \begin{cases} 0, & \text{falls } x = 0 \\ 1, & \text{falls } x > 0 \end{cases}$
3. $max : \mathbb{N}^2 \to \mathbb{N}, max(x, y) \mapsto \begin{cases} 0, \text{falls } x \leq y \\ 1, \text{falls } x > y \end{cases}$

Lösung

1. avg:

```
1 avg:
2     addu $t0 $a0 $a1
3     srl $v0 $t0 1 # divide by 2
4     jr $ra
```

2. sgn:

```
1 sgn:
2     # test whether a0 == 0
3     beq       $a0 $zero zero
4     li        $v0 1
5     jr $ra
6
7 zero:
8     li        $v0 0
9     jr $ra
```

3. max:

```
1 max:
2     sltu $v0 $a1 $a0 #$v0 <- 1 if $a1 < $a0 else 0
3     jr $ra
```

Aufgabe 4.3: Schleifen vs. Rekursion am Beispiel der Fibonacci-Zahlen

Schreiben Sie zwei MIPS-Assemblerprogramme, die beide die *n-te Fibonacci-Zahl* berechnen. Die n-te Fibonacci-Zahl $\mathrm{fib}(n)$, $n \in \mathbb{N} \setminus \{0\}$ ist definiert als:

$$\mathrm{fib}(n) = \begin{cases} 1 & \text{falls } n <= 2 \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & \text{falls} n > 2 \end{cases}$$

Ein Beispiel:

$$\mathrm{fib}(4) = \mathrm{fib}(3) + \mathrm{fib}(2) = (\mathrm{fib}(2) + \mathrm{fib}(1)) + 1 = 1 + 1 + 1 = 3$$

Das erste Programm (`fib_rec`) soll die n-te Fibonacci-Zahl rekursiv entsprechend obiger rekursiver Definition berechnen. Hierzu ruft sich das `fib`-Unterprogramm selbst auf. Die zweite Variante (`fib_iter`) soll die n-te Fibonacci-Zahl ohne Unterprogrammaufruf mittels einer Schleife berechnen. Beachten Sie den zusätzlichen Aufwand (Aufsetzen der Aufrufschachtel, Register retten, etc.), die die rekursive Variante erzeugt. Der Parameter n soll in dem Register `$a0` übergeben werden, das Ergebnis in Register `$v0`.
Hinweis: Orientieren Sie sich an dem Beispielprogramm aus der Vorlesung, das die Fakultät berechnet.

Lösung

Rekursive version:

```
 1 .text
 2   .globl fib_rec
 3
 4 fib_rec:
 5   li $v0 1
 6   ble $a0 2 end    #if n <= 2 we end the recursion and return 1
 7
 8 recurse:
 9   addiu $sp $sp -8  # we need some space in the stack
10   sw $ra 4($sp)    # we store the adress where we need to jump back
11   sw $a0 ($sp)  # we store n because for the second call we need to calculate n-2
12   subiu $a0 $a0 1 # we call fib(n-1)
13   jal fib_rec
14   lw $a0 ($sp)     # we load n which we stored before
15   sw $v0 ($sp)  # and store the return value from our function call fib(n-1)
16   subiu $a0 $a0 2    # then we call fib(n-2)
17   jal fib_rec
18   lw $t0 ($sp)     # we load fib(n-1)
19   addu $v0 $v0 $t0  # and store fib(n-1)+fib(n-2) in $v0
20   lw $ra 4($sp)    # we load the adress so we can jump back
21   addiu $sp $sp 8    # we reset $sp to default
22   b end
23
24
25 end:
26   jr $ra
```

Iterative Version:

```
.text
  .globl fib_iter

fib_iter:
  li $t0 1
  li $t1 1

loop:
  ble $a0 2 end    # if n <= 2 we end the loop

  # we add the contents of t0 and t1,
  #where t1 stores the i-2th fib and t0 the i-1th fib number
  #-> after this step in $t0 is the ith fib

  add $t0 $t0 $t1
  move $t2 $t0  # we swap the contents of t0 and t1
  move $t0 $t1  #because in the next step we need the i-1th and ith fib
  move $t1 $t2
  subi $a0 $a0 1  # we decrement n
  b loop

end:
  move $v0 $t1  # if the loop has ended we find the nth fib in $t1
  jr $ra
```

main:

```
.text
  .globl main

main:
  li $a0 10
  jal fib_rec
  #jal fib_iter
  move $a0 $v0
  li $v0 1
  syscall
  li $v0 10
  syscall
```

Aufgabe 4.4: Mips - string parsing

In dieser Aufgabe sollen Sie einen String von der Konsole einlesen und wieder ausgeben.

a) Machen Sie sich mit `syscall 8` vertraut, mit dem Sie einen String einlesen können. Legen Sie zunächst einen Buffer im Datensegment an, mit dem sie 20 Zeichen einlesen können.

b) Schreiben Sie ein Hauptprogramm, dass einen String (maximal 20 Zeichen) aus der Konsole einliest und in den Buffer aus Aufgabenteil a) schreibt.

c) Geben Sie alle Vokale aus dem eingelesenen Wort in ursprünglicher Reihenfolge aus, wobei jeder dieser Vokale in einer einzelnen Zeile stehen soll. Verwenden Sie hierzu einen statischen String aus dem Datensegment, der den newline character `\n` enthält. Beachten Sie ebenfalls die Instruktionen `syscall 4` und `syscall 11`.

d) Geben Sie alle Konsonanten in dem Wort, getrennt durch je ein Leerzeichen, aus. Verweden Sie `syscall 11` und legen Sie keine zusätzlichen Daten im Datensegment an. Beachten Sie, dass `ASCII(_) = 32`.

e) Geben Sie den gesamten Input in einer neuen Zeile aus. Machen Sie sich dafür nochmals klar, wie `syscall 8` verschieden lange Eingaben handhabt. Stellen Sie sicher, dass keine Zeichen ausgegeben werden, die nicht eingegeben wurden.

Lösung

```
 1 .data
 2 buffer:               # a)
 3     .space 21
 4 newline:
 5     .asciiz "\n"
 6 .text
 7
 8 # $t0 - buffer
 9 # $t1 - max. length
10 # $t2 - counter
11 # $t3 - current address
12 # $t4 - current char
13 # $t5 - lower case char
14
15 main:
16     # b) begin
17     li $v0 8
18     la $a0 buffer
19     li $a1 21
20     syscall # read max. 20 chars
21     # b) end c) begin
22     li $v0 4
23     la $a0 newline
24     syscall # print newline
25
26     la $t0 buffer
27     li $t1 21
28
29     li $t2 0 # init counter
30 loop_vocals:
31     bge $t2 $t1 end_vocals
32     addu $t3 $t2 $t0
33
34     lb $t4 ($t3)
35     # consider upper and lower case
36     ori $t5 $t4 32
37     beq $t5 'a' voc
38     beq $t5 'e' voc
39     beq $t5 'i' voc
40     beq $t5 'o' voc
41     beq $t5 'u' voc
42     j counter
43
44 voc:
45     li $v0 11
46     move $a0 $t4
47     syscall # print the current char
48     li $v0 4
49     la $a0 newline
50     syscall # print new line
51     li $v0 11
52 counter:
53     addiu $t2 $t2 1 # inc. counter
54     j loop_vocals
55 end_vocals:
56     # c) end d) begin
57     li $t2 0 #reset counter
58 loop_consonants:
59     bge $t2 $t1 end_consonants
60     addu $t3 $t2 $t0
61
62     lb $t4 ($t3)
63     # consider upper and lower case
64     ori $t5 $t4 32
65     beq $t5 'a' next
66     beq $t5 'e' next
67     beq $t5 'i' next
68     beq $t5 'o' next
69     beq $t5 'u' next
70     # maybe it is not a letter
71     bgt $t5 'z' next
72     blt $t5 'a' next
73
74     li $v0 11
75     move $a0 $t4
76     syscall # print the current char
77     li $v0 11
78     li $a0 32
79     syscall # print space
80 next:
81     addiu $t2 $t2 1
82     j loop_consonants
83 end_consonants:
84     # d) end e) begin
85     la $a0 newline
86     li $v0 4
87     syscall # print a new line
88     la $t3 buffer
89     lb $t5 ($a0) # new line char
90 loop_find_end:
91     lb $t4 ($t3)
92     # check for new line
93     beq $t4 $t5 end_found
94     beqz $t4 end_found
95     addiu $t3 $t3 1
96     j loop_find_end
97 end_found:
98     # write 0x00 at the end
99     sb $zero ($t3)
100    la $a0 buffer
101    li $v0 4
102    syscall # print string
103    li $v0 10
104    syscall
```

a) `syscall 8` erhält in `$a0` die Adresse eines Input-Buffers und in `$a1` die Anzahl der maximal zu lesenden Zeichen. Darin ist aber auch ein Byte für ein abschließendes Nullbyte enthalten. Also werden maximal $\$a0 - 1$ Zeichen gelesen. Für den Buffer verwenden wir entweder `.space 21` oder `.byte 0,0,0,0 ...` um entsprechenden Speicher zu reservieren. Während `.space` den Speicher nur reserviert und keine direkte Aussage

über seinen Inhalt liefert, wird der Speicher bei `.byte` direkt inizialisiert.

b) Wir verweden `syscall 8` und überreichen diesem in `$a0` die Addresse unseres Buffers.

c) Wie wir in einer früheren Aufgabe gesehen haben, können wir mit `ori $t0 $t0 32` einen Buchstaben in `$t0` in einen Kleinbuchstaben umwandeln. In unserem `loop_vocals` wandeln wir jedes Byte innerhalb des Buffers in einen Kleinbuchstaben um und vergleichen es mit den fünf Möglichkeiten für einen kleinen Vokal. Wenn einer der Vergleiche erfolgreich ist, so wird das unveränderte Byte mit ausgegeben. Für einen Zeilenumbruch muss anschließend mit `syscall 4` der im Datensegment angelegte String `\n` ausgegeben werden.

d) In `loop_consonats` wollen wir nun alle Konsonaten ausgeben. Es ist jedoch einfacher, erneut auf alle Vokale zu überprüfen, um diese dann zu überspringen. Desweitern wollen wir alle Zeichen größer 'z' und kleiner 'a' überspringen, also alle Zeichen, die keinen Buchstaben darstellen. 'z' ist hierbei der Kleinbuchstabe mit dem größten und 'a' der mit dem kleinsten ASCII-Wert. Alle Werte außerhalb dieser Spanne werden nicht benötigt, da wir erneut das Bitmuster aus dem vorherigen Aufgabenteil verwenden.

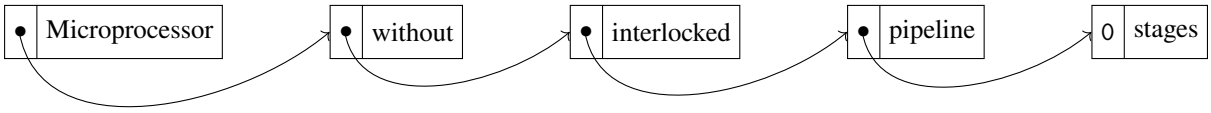
e) Wir bemerken, dass `syscall 8` zu kurze Strings mit einer neuen Zeile abschließt. Da wir diese aber nicht eingegeben haben, wollen wir sie nun auch nicht mit ausgeben. Daher durchsuchen wir den Buffer nach dem Byte $0x04$ (Neue Zeile) oder dem Byte $0x00$ (Nullbyte). Beide können wir nicht eingeben. Wenn wir eines dieser Bytes gefunden haben wissen wir also, dass wir das Ende des Strings gefunden haben. Wir ersetzen $0x04$ durch ein Nullbyte und geben den gesamten Input mit `syscall 4` aus.

Aufgabe 4.5: Listendarstellung

Wir stellen einfach verkettete Listen im Speicher dar. Ein Listenelement besteht aus der Adresse des Nachfolgerelements und den Nutzdaten. Wir verwenden die Adresse 0 um anzuzeigen, dass kein Nachfolgerelement existiert. Die Liste, die als Nutzdaten die Zeichenfolgen

Microprocessor, without, interlocked, pipeline, stages

enthält, sieht abstrakt dargestellt folgendermaßen aus:



Wir betrachten nun eine Darstellung der obigen Liste im Speicher. Dazu sehen wir uns den sogenannten Speicherauszug der entsprechenden Speicherregion an. Eine Zeile eines Speicherauszug besteht aus drei Teilen: Links steht die Speicheradresse in Hexadezimaldarstellung, gefolgt von der Bytefolge der Länge 16, die an dieser Adresse im Speicher beginnt. Rechts steht eingegrenzt von | der Speicherinhalt interpretiert als ASCII-Zeichenfolge. Nicht druckbare Zeichen werden als . dargestellt.

```
10010000 14 00 01 10 4d 69 63 72 6f 70 72 6f 63 65 73 73 |....Microprocess|
10010010 6f 72 00 00 30 00 01 10 77 69 74 68 6f 75 74 00 |or..0...without.|
10010020 40 00 01 10 70 69 70 65 6c 69 6e 65 00 00 00 00 |@...pipeline....|
10010030 20 00 01 10 69 6e 74 65 72 6c 6f 63 6b 65 64 00 | ...interlocked.|
10010040 00 00 00 00 73 74 61 67 65 73 00 00 00 00 00 00 |....stages......|
10010050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

1. Geben Sie die Adresse des Bytes `4d` in der ersten Zeile an. Welchem Zeichen in der ASCII-Spalte entspricht es? Geben Sie zu jedem Element der Liste die Adresse im Speicherauszug an. Markieren Sie jetzt alle Bytes im Speicherauszug, die zur Darstellung einer Folgeelementadresse gehören. In welcher Endianness sind die Folgeelementadressen abgelegt?

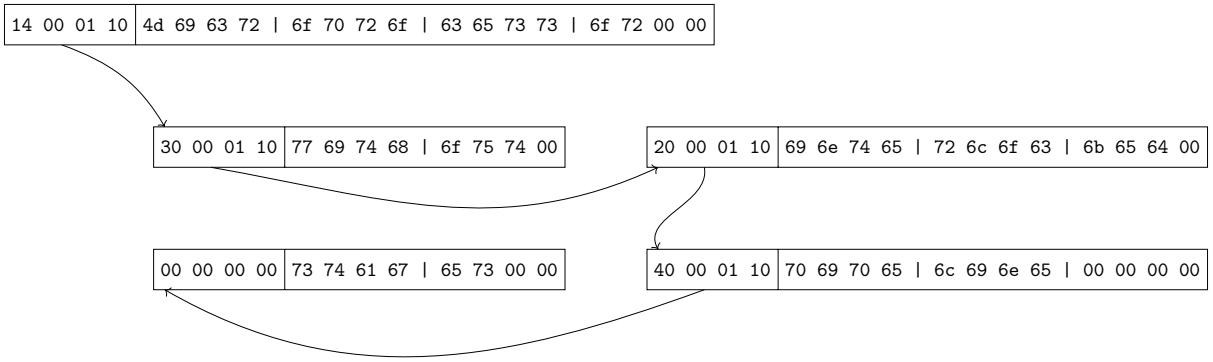
2. Erklären Sie mithilfe von alignment zu welchem Listenelement das Byte mit der Adresse `0x10010013` gehört.

3. Sie sollen die im Speicherauszug gezeigte Liste mithilfe von MIPS-Assemblerdirektiven im Datensegment anlegen. Hinweis: Die folgende Vereinbarung legt ein Wort im Datensegment an, das seine eigene Adresse enthält:

```
1       .data
2       label:
3       .word label
```

4. Schreiben Sie ein Programm, das gegeben der Adresse eines Listenelements die Zeichenketten der Elemente ausgibt. Testen Sie ihr Programm mit der Liste aus dem oben angelegten Datensegment.

Lösung



Die obige Zeichnung entspricht der abstrakten Darstellung der Liste. Hierbei wurden lediglich die Adressen der Nachfolger und die Nutzdaten durch ihre Repräsentation im Speicher ersetzt.
Achtung! Bei der Darstellung des Speichers in MARS kann es zu Verwirrung kommen, da MARS das Datensegment wortweise anzeigt und Wörter in Little Endian gespeichert sind.

1. Das Byte `4d` befindet sich an der Adresse `0x10010004` und stellt das Zeichen `M` dar.
   Die Elemente der Liste starten bei `0x10010000`, `0x10010014`, `0x10010030`, `0x10010020` und `0x10010040`.

   Die vier Bytes auf die die Listenelemente folgen, stellen jeweils die Adresse des Nachfolgers da.
   Die Folgeadressen werden dementsprechend als Little Endian gespeichert.

```
10010000 14 00 01 10 4d 69 63 72 6f 70 72 6f 63 65 73 73 |....Microprocess|
10010010 6f 72 00 00 30 00 01 10 77 69 74 68 6f 75 74 00 |or..0...without.|
10010020 40 00 01 10 70 69 70 65 6c 69 6e 65 00 00 00 00 |@...pipeline....|
10010030 20 00 01 10 69 6e 74 65 72 6c 6f 63 6b 65 64 00 | ...interlocked.|
10010040 00 00 00 00 73 74 61 67 65 73 00 00 00 00 00 00 |....stages......|
10010050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
```

2. Da hier die Daten als Datenwort (`word`) gespeichert werden, müssen immer vier Byte zusammenhängen (alignment). Demzufolge gehört das Byte `0x10010013` zu dem ersten Listenelement.

3.

```
1       .data
2       # Ein Listenenelement besteht aus
3       #der Adresse des Nachfolgers und Nutzdaten
4       element_1:
5       .word element_2
6       .asciiz "Microprocessor"
7
8       element_2:
9       .word element_4
10      .asciiz "without"
11
12      element_3:
13      .word element_5
14      .asciiz "pipeline"
15
16      element_4:
17      .word element_3
18      .asciiz "interlocked"
19
20      # Listenende
21      element_5:
22      .word 0
23      .asciiz "stages"
```

4.

```
1       .text
2       .globl main
3
4       main:
5       # lade die Adresse des Listenkopfs
6       la $t0 element_1
7
8       loop:
9       # breche ab, wenn t0 Null enthält
10      beqz $t0 end
11      # nach den ersten vier Bytes folgen
12      # die Nutzdaten
13      addiu $a0 $t0 4
14      # gebe den string aus
15      li $v0 4
16      syscall
17      # setze das aktuelle Element auf den
18      # Nachfolger
19      lw $t0 0($t0)
20      j loop
21
22      end:
23      li $v0 10
24      syscall
```

Aufgabe 4.6: Reihungen in Mips

In dieser Aufgabe lernen Sie den Umgang mit Reihungen (auf Englisch 'Array') in Mips.

1. Schreiben Sie ein Unterprogramm, das die natürlichen Zahlen, angefangen von 0, aufsteigend in einem `word`-Array (welches im Datensegment liegt) speichert, bis es voll ist. Die Anfangsadresse des Arrays wird Ihnen dabei in Register `$a0` übergeben, die Länge steht in `$a1`.

2. Nun betrachten wir zwei verschiedene Arrays. Ihre Aufgabe ist es ein Unterprogramm zu schreiben, welches jedes zweite Element des ersten Arrays in das zweite Array kopiert. In den Registern `$a0, $a1, $a2, $a3` finden Sie die benötigten Angaben:

   - `$a0`: die Anfangsadresse des Arrays von dem kopiert werden soll
   - `$a1`: die Länge des Arrays von dem kopiert werden soll (in Byte)
   - `$a2`: die Anfangsadresse des Zielarrays
   - `$a3`: die Größe eines einzelnen Elements (in Byte)

Lösung

1.

```
1  .text
2      .globl fill
3      # a0 Adresse des Arrays
4      # a1 Länge des Arrays
5
6  fill:
7      # Zähler erstellen
8      li $t0 0
9
10 loop:
11     # breche ab, falls Zähler gleich Länge der Reihung
12     beq $t0 $a1 end
13     # Zähler in Reihung speichern
14     sw $t0 0($a0)
15     # Zähler erhöhen
16     addiu $t0 $t0 1
17     # a0 auf nächste Adresse der Reihung setzen
18     addiu $a0 $a0 4
19     j loop
20
21 end:
22     jr $ra
```

2.

```
.text
    .globl copy_even
    # a0 Adresse des Arrays aus dem kopiert werden soll
    # a1 Größe des Arrays aus dem kopiert werden soll
    # a2 Adresse des Arrays in das kopiert werden soll
    # a3 Größe eines einzelnen Elements des Array

copy_even:
    # Endadresse berechnen
    addu $a1 $a1 $a0

outer_loop:
    # äußere Schleifenbedingung
    bge $a0 $a1 end
    # inneren Zähler erstellen
    move $t0 $a3

inner_loop:
    lb $t2 0($a0)
    sb $t2 0($a2)
    # ein Byte weiter springen
    addiu $a0 $a0 1
    addiu $a2 $a2 1
    addiu $t0 $t0 -1
    # innere Schleifenbedingung
    bgtz $t0 inner_loop

    # ein Element überspringen
    addu $a0 $a0 $a3
    j outer_loop

end:
    jr $ra
```

Aufgabe 4.7: Fibonacci mit Reihungen in Mips

Konrad Klug hat den Auftrag erhalten eine komplexe mathematische Berechnung durchzuführen. Da er hierbei häufig die ersten n Fibonacci-Zahlen benötigt, möchte er möglichst viele von diesen in einem Array speichern, um sie nicht immer neu berechnen zu müssen. Leider weiß er (noch) nicht, wie er das implementieren kann. Die Berechnungsvorschrift für die Fibonacci-Zahlen lautet:

$$
\begin{aligned}
F_0 &:= 0 \\
F_1 &:= 1 \\
F_n &:= F_{n-1} + F_{n-2} \text{ für } n \in \mathbb{N} \wedge n \geq 2
\end{aligned}
$$

Helfen Sie Konrad Klug, indem Sie die Speicherung der Fibonacci-Folge in einem `word`-Array vornehmen. Sie finden die Anfangsadresse des Arrays in Register `$a0`. Die Endadresse steht in Register `$a1`.

Lösung

```
.text
    .globl fib
    # a0 Anfangsadresse des Arrays
    # a1 Endadresse des Arrays
```

```
 5
 6 fib:
 7     # erstes Element setzen
 8     li $t0 0
 9     sw $t0 0($a0)
10     # zweites Element setzen
11     li $t0 1
12     sw $t0 4($a0)
13     # Adresse erhöhen
14     addiu $a0 $a0 8
15
16 loop:
17     # Schleifenbedingung
18     bge $a0 $a1 end
19     # vorherige Elemente laden
20     lw $t0 -4($a0)
21     lw $t1 -8($a0)
22     # nächstes Element berechnen
23     addu $t0 $t0 $t1
24     # neues Element speichern
25     sw $t0 0($a0)
26     # Adresse erhöhen
27     addiu $a0 $a0 4
28     j loop
29
30 end:
31     jr $ra
```

Aufgabe 4.8: Structs

Konrad Klug hat nun schon - mit No Hau's Hilfe - einige Programme in `MIPS` geschrieben. Jetzt möchte er die persönlichen Daten eines Freundes in `MIPS` darstellen. Er hat gehört, dass er dazu Verbunde nutzen kann (siehe Skript Abschnitt 2.6). Allerdings weiß er nicht, wie er das tun soll. Die Person, die er darstellen möchte, heißt „Peter Meier“ und ist am 23.04.1986 geboren. Nach langer Recherche hat No Hau die folgenden Richtlinien für das Verfahren erstellt:

- Erinnern Sie sich, was Verbunde sind und welche Arten von Verbunden es gibt.
- Überlegen Sie, wie groß die Daten sind und auf welche Weise Sie diese darstellen wollen.
- Überlegen Sie, welche Anweisungen nützlich sind, um Daten bestimmter Größe und Art im Speicher anzulegen.
- Erinnern Sie sich, welche Einschränkung die Befehle `lw`, `sw`, `lh` und `sh` haben.

Mit No Hau's Richtlinien in der Hand fühlt sich Konrad für sein Verfahren gut ausgerüstet. Helfen Sie ihm, indem Sie:

1. Komposition nutzen, um die Person im Speicher darzustellen. Um ihre Arbeit zu testen, schreiben Sie ein Programm, das den Namen, den Vornamen, den Geburtstag, den Geburtsmonat und das Geburtsjahr in dieser Reihenfolge aus dem Speicher liest und ausgibt.
2. Aggregation nutzen und danach ihr Programm in der gleichen Weise testen. Worauf müssen Sie jetzt achten bzw. was müssen Sie jetzt ändern, um die Daten aus dem Speicher zu lesen?
3. Vergleichen Sie nun die beiden Methoden und stellen Sie die Unterschiede dar.

Lösung

1. Komposition

```
.data
person:                 # Verbund Person
    .asciiz "Meier"
    .asciiz "Peter"
    .byte 23
    .byte 4
    .align 1            # Halb-Wort auf durch 2 teilbare Adresse ausrichten
    .half 1986

.text
komposition:
    la $t0 person       # Nachname laden und ausgeben
    move $a0 $t0
    li $v0 4            # Stringausgabe
    syscall

    li $a0 '\n'         # Newline
    li $v0 11
    syscall

    add $t0 $t0 6       # Vorname laden und ausgeben
    move $a0 $t0
    li $v0 4            # Stringausgabe
    syscall

    li $a0 '\n'         # Newline
    li $v0 11
    syscall

    lb $a0 6($t0)       # Tag laden und ausgeben
    li $v0 1            # Integerausgabe
    syscall

    li $a0 '\n'         # Newline
    li $v0 11
    syscall

    lb $a0 7($t0)       # Monat laden und ausgeben
    li $v0 1            # Integerausgabe
    syscall

    li $a0 '\n'         # Newline
    li $v0 11
    syscall

    lh $a0 8($t0)       # Jahr laden und ausgeben
    li $v0 1            # Integerausgabe
    syscall

    li $v0 10           # Programm beenden
    syscall
```

2. Aggregation

```
.data
person:                 # Verbund Person
    .word 0             # Zeiger auf Namen
    .word 0             # Zeiger auf Vornamen
```

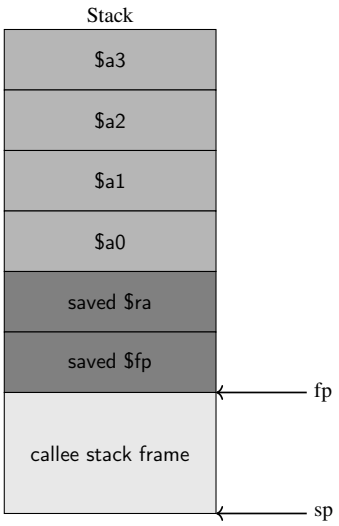
```
 5     .byte 23
 6     .byte 4
 7     .align 1
 8     .half 1986
 9
10 vorname:
11     .asciiz "Peter"
12
13 nachname:
14     .asciiz "Meier"
15
16 .text
17 aggregation:
18     la $t0 person        # Namen in Verbund legen
19     la $t1 nachname
20     sw $t1 ($t0)
21     la $t1 vorname
22     sw $t1 4($t0)
23
24     lw $a0 ($t0)         # Nachname laden und ausgeben
25     li $v0 4             # Stringausgabe
26     syscall
27
28     li $a0 '\n'          # Newline
29     li $v0 11
30     syscall
31
32     lw $a0 4($t0)        # Vorname laden und ausgeben
33     li $v0 4             # Stringausgabe
34     syscall
35
36     li $a0 '\n'          # Newline
37     li $v0 11
38     syscall
39
40     lb $a0 8($t0)        # Tag laden und ausgeben
41     li $v0 1             # Integerausgabe
42     syscall
43
44     li $a0 '\n'          # Newline
45     li $v0 11
46     syscall
47
48     lb $a0 9($t0)        # Monat laden und ausgeben
49     li $v0 1             # Integerausgabe
50     syscall
51
52     li $a0 '\n'          # Newline
53     li $v0 11
54     syscall
55
56     lh $a0 10($t0)       # Jahr laden und ausgeben
57     li $v0 1             # Integerausgabe
58     syscall
59
60     li $v0 10            # Programm beenden
61     syscall
```

3. In der folgenden Tabelle sind die Unterschiede von Komposition und Aggregation gegenübergestellt:

Komposition	**Aggregation**
Verbraucht weniger Speicherplatz bei Daten mit unterschiedlichen Werten.	Verbraucht weniger Speicherplatz, wenn bei viele Daten dieselben Werte eingetragen sind.
Um Daten anzulegen, muss nur Speicherplatz für den einzutragenden Wert angelegt werden.	Um Daten anzulegen, muss Speicher für den Zeiger und die Daten selbst angelegt werden.
Ist derselbe Eintrag bei mehreren Daten falsch, muss er überall abgeändert werden.	Ist derselbe Eintrag bei mehreren Daten falsch, muss er an nur einer Stelle abgeändert werden.
Einträge wie z.B. Namen müssen auf eine gewisse Größe festgelegt werden, damit Daten wie z.B. Personen in eine Reihung geschrieben werden können.	Die Werte können beliebig groß werden und unterschiedliche Größen haben.
Lese- und Schreibzugriff sind direkt.	Lese- und Schreibzugriff werden durch einen Pointer verzögert.

Aufgabe 4.9: Der stack pointer

Die Aufrufschachtel des Stacks bzw. der stack frame wird durch zwei Zeiger begrenzt. Hierbei begrenzt der frame pointer (Schachtelzeiger) den stack frame nach oben, der stack pointer (Kellerpegelzeiger) nach unten.
Bei einem Aufruf eines Unterprogramms muss eine neue Aufrufschachtel angelegt werden, die vor dem Verlassen dieses Unterprogramms wieder aufgeräumt werden muss. Müssen einem Unterprogramm mehr als vier Argumente übergeben werden, kann dies über den Stack geschehen, da die im üblichen verwendeten Argument-Register hierfür nicht ausreichen.
Im folgenden werden Sie die Instruktionen `call $label` und `ret` kennen lernen, die entsprechend eine Aufrufschachtel aufsetzen bzw. aufräumen und zu einem Unterprogramm springen bzw. zu einem Oberprogramm zurückkehren. Beachten Sie hierbei die Argumentreihenfolge inder Abbildung.



a) Wie können Sie die Operation `call $label` in MIPS darstellen? Sie dürfen hierbei annehmen, dass `$label` immer `Callee` heißt. Außerdem folgt auf eine Call-Instruktion immer direkt das Label `Return_Here`, zu dem zurückgekehrt werden soll.

   Beispiel:

```
1     call Function
2 Return_Here:
3       ...
```

Nach dem Verlassen des Unterprogramms `Function` soll also zu dem Label `Return_Here` zurückgesprungen werden.

`call $label` soll dabei die Argument-Register, sowie den jetzigen frame pointer und die return-Adresse auf den Stack legen. Bedenken Sie, wie sich frame pointer und stack pointer hierbei ändern und beachten Sie die Reihenfolge entsprechend der Abbildung.

b) Wie können Sie die Operation `ret` in MIPS darstellen? `ret` soll den ursprünglichen Zustand des Stacks wiederherstellen und zu der gespeicherten Rücksprungadresse zurückkehren.

c) CHALLENGE: Wie können Sie die Operation `call $label` implementieren, ohne dass Ihnen eine Rücksprungadresse gegeben wird? Denken Sie darüber nach, wie Sie den Program Counter in MIPS auslesen können und wie dieser verändert werden muss.

Lösung

Für die Implementierung von `push` und `pop` siehe vorherige Aufgabe.

a) Implementierung von `call $label` als Macro für allgemeine Register:

```
1 .MACRO call ($label)
2     push $a3
3     push $a2
4     push $a1
5     push $a0
6     la $v0 Return_Here
7     push $v0
8     push $fp
9     move $fp $sp
10    j $label
11 .END_MACRO
```

b) Implementierung von `ret`:

```
1 .MACRO ret
2     move $sp $fp
3     pop $fp
4     pop $ra
5     addiu $sp $sp 16
6     jr $ra
7 .END_MACRO
```

c) Implementierung von `call $label` ohne gegebene Rücksprungadresse:

Hilfsfunktion, um den program counter auszulesen:

```
1 getpc:
2     move $v0, $ra
3     jr $ra
```

Angepasstest Macro:

```
1 .MACRO call ($label)
2     push $a3
3     push $a2
4     push $a1
5     push $a0
6     jal getpc
7     addiu $v0, $v0 44         # Erklärung: siehe unten
```

```
 8     push $v0
 9     push $fp
10     move $fp, $sp
11     j $label
12 .END_MACRO
```

Die 44 ergibt sich durch die Anzahl an Zeilen durch das Push-Macro und der Pseudoinstruktionen.

Aufgabe 4.10: Kompilieren und ausführen

1. Schreiben Sie ein C-Programm, welches zwei Variablen vom Typ `int` anlegt, ihnen Werte zuweist und die Werte der Variablen auf der Konsole ausgibt. Erstellen Sie dazu eine Datei `main.c` und schreiben Sie die entsprechende `main`-Funktion.
   Erstellen Sie eine ausführbare Datei `prog` und führen Sie diese aus.

2. Erweitern Sie die Datei `main.c` aus Aufgabenteil (a) um eine Funktion `max`, welche zwei Argumente vom Typ `int` bekommt und das Maximum der beiden Werte zurückgibt.
   Erstellen Sie eine ausführbare Datei `prog` und führen Sie diese aus.

3. Lagern Sie die in Aufgabenteil (b) erstellte Funktion `max` in eine separate Übersetzungseinheit `max.c` aus. Verwenden Sie eine Headerdatei `max.h`, um Ihrem Hauptprogramm den Prototypen des Unterprogramms `max` bekannt zu machen.
   Erstellen Sie eine ausführbare Datei `prog` und führen Sie diese aus.

Lösung

1. Programm

```c
#include <stdio.h>

int main() {
  int a = 10;
  int b = 20;

  printf("a: %d\n", a);
  printf("b: %d\n", b);

  return 0;
}
```

Ausführbares Programm `prog` erstellen und ausführen.

```
$ cc -o main.o -c main.c
$ cc -o prog main.o
$ ./prog
```

Kurz:

```
$ cc main.c -o prog
$ ./prog
```

2. Funktion `max`

```c
#include <stdio.h>

int max(int x, int y) {
  if(x > y) {
    return x;
  } else {
```

```
 7     return y;
 8   }
 9 }
10
11 int main() {
12   int a = 10;
13   int b = 20;
14
15   int res = max(a, b);
16
17   printf("Maximum␣von␣a␣und␣b:␣%d\n", res);
18
19   return 0;
20 }
```

Ausführbares Programm `prog` erstellen: Wie oben.

3. Mit header-Datei
   max.h:

```
1 #ifndef MAX_H
2 #define MAX_H
3
4 int max(int x, int y);
5
6 #endif
```

max.c:

```
1 #include "max.h"
2
3 int max(int x, int y) {
4   if(x > y) {
5     return x;
6   } else {
7     return y;
8   }
9 }
```

main.c:

```
1 #include <stdio.h>
2 #include "max.h"
3
4 int main() {
5   int a = 10;
6   int b = 20;
7
8   int res = max(a, b);
9
10  printf("Maximum␣von␣a␣und␣b:␣%d\n", res);
11
12  return 0;
13 }
```

Ausführbares Programm `prog` erstellen und ausführen:

```
$ cc -o main.o -c main.c
$ cc -o max.o -c max.c
$ cc -o prog main.o max.o
$ ./prog
```

Kurz:

```
$ cc main.c max.c -o prog
$ ./prog
```

Aufgabe 4.11: C Programme verstehen

Untersuchen und verstehen Sie das folgende Programm:

```c
#include <stdio.h>
int main() {
    for (int i = 1; i < 10; i++) {
        if ((i % 2) == 0) {
            printf("Die Zahl %i ist ...\n", i);
        } else {
            printf("Die Zahl %i ist ...\n", i);
        }
    }
    return 0;
}
```

1. Welche Ausgabe liefert es? Wie können Sie die Pünktchen in der Ausgabe sinnvoll ersetzen?
2. Erweitern Sie das Programm so, dass die Ausgabe für den Bereich von $0 - 20$ (inklusive Grenzen) erfolgt.
3. Können Sie die Schleife so abändern, dass die Zahlen aus Teil (b) rückwärts ausgegeben werden?

Lösung

1. Die ersten Pünktchen sind sinnvoller durch gerade und die zweiten Pünktchen durch ungerade zu ersetzen. Dann liefert das Programm folgende Ausgabe:

```
Die Zahl 1 ist ungerade
Die Zahl 2 ist gerade
Die Zahl 3 ist ungerade
Die Zahl 4 ist gerade
Die Zahl 5 ist ungerade
Die Zahl 6 ist gerade
Die Zahl 7 ist ungerade
Die Zahl 8 ist gerade
Die Zahl 9 ist ungerade
```

2. Erweiterung

```c
#include <stdio.h>
int main() {
  for (int i = 0; i <= 20; i++) {
    if ((i % 2) == 0) {
      printf("Die Zahl %i ist gerade\n", i);
    } else {
      printf("Die Zahl %i ist ungerade\n", i);
    }
  }
  return 0;
}
```

3. Rückwärts

```c
#include <stdio.h>
int main() {
  for (int i = 20; i >= 0; i--) {
    if ((i % 2) == 0) {
      printf("Die Zahl %i ist gerade\n", i);
    } else {
      printf("Die Zahl %i ist ungerade\n", i);
    }
  }
  return 0;
}
```

Aufgabe 4.12: C Programme verstehen und erweitern

Konrad Klug hat das Kapitel zu C im Skript gelesen und fühlt sich nun bereit, sein erstes C-Programm zu schreiben. No Hau hat ihm erzählt, dass ihr erstes Programm die Zahlen von 0 bis 100 ausgegeben hat, was Konrad jedoch nicht reicht. Er will nun die Zahlen von 0 bis 100 in umgekehrter Reihenfolge und nur in 5er Schritten ausgeben. Also: 100, 95, 90, ..., 5, 0.

1. Konrad ist mit dem Programm fast fertig. Ihm fehlt nur noch ein wesentlicher Baustein. Ersetzen Sie im folgenden Programm den Kommentar `/* hier fehlt was */` so, dass es die oben genannte Zahlenfolge ausgibt:

```c
int main() {
  int j = 100;
  int z = 5;
  do {
    /* hier fehlt was */
  } while (j >= 0);
  return 0;
}
```

2. Was passiert, wenn die Variable j einen Wert kleiner Null zugewiesen bekommt? Würde sich das ändern, wenn wir anstelle einer offenen Schleife eine geschlossene Schleife verwenden?

3. Nachdem Konrad seine Zahlenfolge nun sehen konnte würde er dasselbe auch gerne mit anderen Zahlen ausprobieren. Er hat aber von No Hau gehört, dass es schlecht sei Code zu duplizieren. Um das zu vermeiden, hat er sich vorgenommen ein kleines Unterprogramm zu schreiben. Dieses soll abhängig von beliebig gewählten Zahlen j und z die Zahlenfolge $j, j-z, j-2z, ..., j-\lfloor\frac{j}{z}\rfloor z$ ausgibt. Leider erinnert er sich nicht mehr genau wie man das macht. Vervollständigen Sie seinen Code analog zu Aufgabenteil (a), jedoch unter Verwendung einer geschlossenen Schleife.

```c
int main() {
  magic(100, 5);
  magic(100, 10);
  magic(5000, 200);
  return 0;
}

void magic(...) {
  ...
}
```

Lösung

1. vervollständigtes Programm:

```c
int main() {
  int j = 100;
  int z = 5;
  do {
    printf("Die␣Zahl␣ist:␣%i␣\n", j ) ;
    j -= z;
  } while (j >= 0);
  return 0;
}
```

2. Wenn die Variable j einen Wert kleiner Null zugewiesen bekommt, wird die Schleife einmal ausgeführt, obwohl die Abbruchbedingung gilt. Im Fall einer geschlossenen Schleife würde der Schleifenrumpf überhaupt nicht ausgeführt werden.

3. modulare Version:

```c
void magic(int j, int z){
  while (j >= 0) {
    printf("Die␣Zahl␣ist:␣%i␣\n", j ) ;
    j-=z;
  }
}

int main(){
  magic(100, 5);
  magic(100, 10);
  magic(5000, 200);
  return 0;
}
```