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

Musterlösung 3
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 3.0: Mixed Bag

Diese Aufgabe dient als Hilfestellung sich mit wichtigen Begriffen und Konzepten aus dem Skript auseinander zu setzen.

1. Was ist der Unterschied zwischen `$1` und `1` in einer MIPS Instruktion?
2. Welche Kategorien von MIPS Befehlen haben Sie kennengelernt?
3. Warum gibt es die vier Additionsinstruktionen „add“, „addi“, „addu“ und „addiu“?
4. Was ist `.text`? Welche weiteren Direktiven haben Sie kennengelernt oder können Sie noch im Skript finden?
5. Wofür wird z.B. `blub:` in MIPS genutzt? Schreiben Sie das Beispiel „Endadresse“ aus dem Skript um, ohne `loop:`, `cmp:` oder andere Marken zu benutzen. Um Ihre Lösung auf Korrektheit zu testen können Sie sich die Assemblierung des Originalbeispiels in Mars anschauen, dort werden die Sprungmarken zu absoluten Werten übersetzt. Welche Nachteile fallen Ihnen zu der Schreibweise ohne Sprungmarken ein?

Lösung

1. Mit `$1` bezieht man sich auf ein Register, wohingegen 1 einfach eine Konstante ist.
2. Insgesamt wurden in der Vorlesung vier Kategorien angesprochen:
   - Rechenbefehle: Mit diesen Befehlen werden Berechnungen auf Registern gemacht. Zum Beispiel `addu`, `subu`, `and`, `slt` ...
   - Speicherbefehle: Diese transferieren Daten zwischen dem Speicher und den Registern. Zum Beispiel `lw`, `lh`, `sb`, ...
   - Sprungbefehle: Mit diesen Befehlen wird der Programmablauf beeinflusst, es kann zu anderen Instruktionen als der nächsten gesprungen werden. Zum Beispiel `beq`, `bne`, `jr` ...
   - Pseudobefehle: Befehle, die dazu dienen, dem Programmierer Arbeit zu ersparen. Sie können in mehrere echte Hardware Instruktionen umgeschrieben werden. Zum Beispiel `li`, `la`, `not` ...
3. Es gibt zwei Hauptgründe aus denen sich die Notwendigkeit für diese Befehle ergibt. Ein `i` wird benutzt, wenn nicht zwei Register sondern ein Register und eine Konstante (immediate) verwendet werden soll. Ein `u` bedeutet, dass das Ergebnis der Addition als vorzeichenlos (unsigned) interpretiert werden soll, was lediglich zur Folge hat, dass bei einem zeichenbehafteten Überlauf im Gegensatz zu `add` keine Ausnahme geworfen wird. Falls es zu keinem zeichenbehaftetem Überlauf kommt, verhalten sich daher `add` und `addu` komplett gleich. Betrachten wir uns im Folgenden zum Beispiel vier Bit große Zahlen, so können wir vorzeichenlos die Zahlen von 0 bis 15 ($2^4 - 1$) und vorzeichenbehaftet mit der Zweierkomplementdarstellung die Zahlen von -8 (-2^{4-1}) bis 7 ($2^{4-1} - 1$) darstellen.

					vorzeichenlos	vorzeichenbehaftet
	1	0	0	1	9	-7
+	0	0	1	1	3	3
0	0	1	1			
=	1	1	0	0	12	-4

→ Kein zeichenbehafteter Überlauf, wodurch sich `add` und `addu` gleich verhalten.

					vorzeichenlos	vorzeichenbehaftet
	0	1	1	0	6	6
+	0	0	1	1	3	3
0	1	1	0			
=	1	0	0	1	9	-7

→ Zeichenbehafteter Überlauf ($6 + 3 = -7$), wodurch `add` eine Ausnahme wirft. Vorzeichenlos ist das Ergebnis korrekt und `addu` wirft keine Ausnahme. Ein zeichenbehafteter Überlauf lässt sich leicht an den letzten beiden Stellen des Übertrags (von rechts nach links gelesen, oben rot markiert) ablesen. Falls sich die Ziffern unterscheiden, liegt ein zeichenbehafteter Überlauf vor, sonst nicht.

					vorzeichenlos	vorzeichenbehaftet
	1	1	1	1	15	-1
+	1	1	0	0	12	-4
1	1	0	0			
=	1	0	1	1	11	-5

→ Kein zeichenbehafteter Überlauf, wodurch sich `add` und `addu` gleich verhalten und keine Ausnahme geworfen wird. Betrachtet man jedoch das vorzeichenlose Ergebnis, so fällt auf, dass es dort zu einem vorzeichenlosen Überlauf gekommen ist ($15 + 12 = 11$). Falls man also mit vorzeichenlosen Zahlen arbeitet, muss man solche Überläufe selbst erkennen. Dies lässt sich jedoch einfach abprüfen, da ein solcher Überlauf vorliegt, wenn das Ergebnis kleiner als mindestens einer der Summanden ist (vorzeichenloser Vergleich!). Hier ist $11 < 15$ (und $11 < 12$), weshalb ein Überlauf vorliegt.

					vorzeichenlos	vorzeichenbehaftet
	1	0	0	1	9	-7
+	1	0	1	0	10	-6
1	0	0	0			
=	0	0	1	1	3	3

→ Zeichenbehafteter Überlauf, wodurch `add` eine Ausnahme wirft, `addu` jedoch nicht. Vorzeichenlos betrachtet liegt aber ebenfalls ein Überlauf vor.

4. Die Direktive `.text` markiert den Beginn des Code-Segments. Eine weitere besondere Direktive ist `.data`, diese markiert den Beginn des Datensegments. Es gibt noch die Direktiven `.ascii`, `.asciiz`, `.byte`, `.half` und `.word`, die alle Dinge verschiedener Größe im Speicher ablegen. Um Speicher zu allozieren kann die Direktive `.space` verwendet werden. Mit `.align` wird das nächste Datum ausgerichtet. Als letztes gibt es noch `.globl`, welches ein Symbol für andere Dateien sichtbar macht.

5. Sprungmarken werden genutzt, um Punkte zu markieren, an die Sprunginstruktionen springen können. Sie dienen zum einen dazu das Programm leichter für den Menschen lesbar zu machen. Schreibt man ein Programm ohne Sprungmarken, hat das den Nachteil, dass die Wartbarkeit des Programms schlechter wird. Fügt man beispielsweise eine Instruktion in Code ein, der absolute Sprungziele benutzt, kann es sein, dass die absoluten Sprungwerte hinterher nicht mehr stimmen.

   Das Skriptbeispiel:

```
1     # $a0 enthaelt die Basisadresse einer Reihung von Wörtern
2     # $a1 enthaelt die Endadresse dieser Reihung
3     li    $v0 0          # Initialisiere $v0 mit 0
4     b     cmp            # Springe zum Test der Bedingung
5 loop:
6     lw    $t0 0($a0)     # Lade ein Wort (4 Byte) von Adresse $a0
7     addu  $v0 $v0 $t0    # Addiere den geladenen Wert zu $v0
```

```
 8     addiu $a0 $a0 4      # Erhöhe Basisadresse um 4
 9 cmp:
10     bne $a0 $a1 loop     # springe zu loop wenn $a0 != $a1
```

Die Sprünge in der Disassemblierung:

```
1     b cmp
2     bgez $0, 0x00000003
```

Der unbedingte Sprung wird zu einem Sprung wenn $0 \geq 0$ übersetzt und er überspringt die nächsten 3 Instruktionen.

```
1     bne $a0 $a1 loop
2     bne $4 $5 0xfffffffc
```

Der bedingte Sprung bekommt den Offset -4. Der Wert der Hexadezimalzahl sieht auf den ersten Blick hoch aus, das liegt daran, dass er als vorzeichenbehaftet zu interpretieren ist. Dies dient dazu wie in diesem Fall auch negative Sprünge darstellen zu können.

Aufgabe 3.1: Finde den Fehler

In dieser Aufgabe sollen Sie Fehler in Programmen finden (Programme s. buggy.zip). Finden Sie die Fehler und korrigieren Sie sie. In jeder Aufgabe gibt es zwei Fehler. Versuchen Sie zuerst ohne Hilfe von Mars.

1. In buggyProgram1.asm sollen alle Zahlen einer Reihung aufaddiert werden und das Ergebnis soll auf der Konsole ausgegeben werden.

2. Ein Kommilitone bittet Sie um Hilfe. Er hat versucht, das Fakultätsprogramm der Vorlesung selbst noch einmal zu schreiben, aus irgendeinem Grund funktioniert es aber nicht. In buggyProgram2a.asm und buggyProgram2b.asm ist sein Versuch zu finden.

   Hinweis: Um diese Programme in Mars zu testen, kopieren Sie sie in den selben Ordner und setzen Sie in Mars unter `Settings` die Häkchen bei `Assemble all files in directory` und `Initialize Program Counter to global 'main' if defined`.

Lösung

1. Die Endadresse der Reihung ist falsch. Ein word hat vier byte, um an das Element hinter dem letzten zu kommen müssen wir hinter das vierte Wort zeigen, also Adresse von Wort1 + 4x4 Byte.

   ```
   1 addiu $a2 $a1 4      # falsch
   2 addiu $a2 $a1 16     # richtig
   ```

   Die Daten sind als word abgelegt, werden aber byteweise geladen. Um Wörter zu laden sollte man lw nutzen statt lb.

   ```
   1 lb $t0 0($a1)        # falsch
   2 lw $t0 0($a1)        # richtig
   ```

2. Das Unterprogram fac hat seine Definition nicht nach außen sichtbar gemacht, daher kann es nicht von main genutzt werden.

   ```
   1 .globl fac           # unter .text hinzufügen
   ```

   Die Schleife läuft von n bis 0, was dazu führt das das Ergebnis immer 0 ist.

   ```
   1  bgez $a0 loop       # falsch
   2  bgtz $a0 loop       # richtig
   ```

Aufgabe 3.2: Pseudoinstruktionen

Der MIPS Assembler definiert eine Reihe von Pseudo-Instruktionen. Dies sind hilfreiche Instruktionen, die nicht in Hardware implementiert sind, aber leicht durch ein oder zwei existierende Befehle ausgedrückt werden können. Geben Sie Implementierungen der Pseudo-Instruktionen `blt`, `bgt`, `ble`, `neg`, `not`, `bge`, `li`, `la`, `lw`, `move`, `sge`, `sgt` an.

Lösung

- `blt $t8 $t9 label`

```
slt $at $t8 $t9
bne $at $zero label
```

- `bgt $t8 $t9 label`

```
slt $at $t9 $t8
bne $at $zero label
```

- `ble $t8 $t9 label`

```
slt $at $t9 $t8
beq $at $zero label
```

- `neg $t8 $t9`

```
sub $t8 $zero $t9
```

- `not $t8 $t9`

```
nor $t8 $t9 $zero
```

- `bge $t8 $t9 label`

```
slt $at $t8 $t9
beq $at $zero label
```

- `li $t8 i`
  Ein Befehl kann maximal eine 16 Bit große Konstante enthalten. Deshalb muss die Konstante in zwei Teilen in ein Register kopiert werden, wenn sie größer ist.
  $i = \langle b_{31} \cdots b_0 \rangle$ $i_u = \langle b_{31} \cdots b_{16} \rangle$ $i_l = \langle b_{15} \cdots b_0 \rangle$
  wenn $i < 2^{16} - 1$

```
ori $t8 $zero i
```

  wenn $i \geq 2^{16} - 1$

```
lui $at iu
ori $t8 $at il
```

- `la $t8 i($t9)`

```
ori $at $zero i
add $t8 $t9 $at
```

- `lw $t8 add` Abhängig von der Größe der Adresse muss diese, ähnlich wie bei `li` in zwei Schritten geladen werden.

- `move $t8 $t9`

```
addu $t8 $zero $t9
```

- `sge $t8 $t9 $t1`

```
slt $t8 $t1 $t9
```

- `sgt $t8 $t9 $t1`

```
sle $t8 $t1 $t9
```

Aufgabe 3.3: Codeverständnis

Betrachten Sie das folgende MIPS Programm. Versuchen Sie die folgenden Aufgaben ohne Hilfe eines MIPS Emulators zu lösen.

```
1  main:
2    li $a1 5
3    li $a2 10
4    li $a3 7
5
6    sltu $t1 $a1 $a2
7    beqz $t1 case2
8
9  case1:
10   sltu $t1 $a2 $a3
11   beqz $t1 reta2
12   b reta3
13
14 case2:
15   sltu $t1 $a3 $a1
16   beqz $t1 reta3
17   b reta1
```

```
18 reta1:
19   addu $a0 $0 $a1
20   b end
21
22 reta2:
23   addu $a0 $0 $a2
24   b end
25
26 reta3:
27   addu $a0 $zero $a3
28   b end
29
30 end:
31   li $v0 1
32   syscall
33   li $v0 10
34   syscall
```

1. Was befindet sich nach der Ausführung des Programms in Register `$a0`?
2. Was berechnet das Programm für allgemeine `$a1`, `$a2` und `$a3` (angenommen, in den Zeilen 2-4 werden andere Zahlen in diese drei Register geladen)?
3. Zwei der Instruktionen können aus dem Programm gestrichen werden, ohne die Semantik (für beliebige `$a1`, `$a2` und `$a3`) zu verändern. Welche Zeilen können gestrichen werden?

Lösung

1. 10
2. Das Maximum der 3 Zahlen.
3. Die Branches in Zeile 17 und 28, denn sie springen nur zur nächsten Instruktion.

Aufgabe 3.4: Alignment

Es sei folgendes Programm gegeben:

```
1 .data
2 msg: .byte 'P','r','o','g',0
3 num: .byte 2,0,0,0
4
5 .text
6 la $a0 msg
7 li $v0 4 #Gebe Zeichenkette aus (syscall 4)
8 syscall
9 lw $a0 num
10 li $v0 1 #Gebe Integer aus (syscall1)
11 syscall
```

1. Warum schlägt die Ausführung des Programmes fehl? Finden Sie den Fehler und ändern Sie das Programm so ab, dass es `Prog2` ausgibt.
2. Ist es auch möglich die Ausrichtung von `num` mit `.space` zu berichtigen? Erklären Sie!
3. Finden Sie eine weitere Möglichkeit, die Zahl 2 aus dem Datensegment zu laden, ohne die Ausrichtung von `num` zu verändern.
4. *Challenge:*
   Der Student Konrad Klug hat eine revolutionäre Idee: Ein Programm, welches ein `word` in ein Register laden kann, ohne dass die Speicheradresse des Wortes eine korrekte Ausrichtung besitzt. Leider hat er seinen Notizzettel verloren, auf dem er den Code notiert hatte. Können Sie ihm helfen?
   Sie können dafür folgendes Codegerüst benutzen:

   

   ```
   1 .data
   2 .space 1
   3 wort:
   4 .byte 4,3,2,1
   5
   6 .text
   7 la $a0 wort
   8
   9 #TODO Lade Wort an Marke wort (0x01020304)
   10
   11 li $v0 1
   12 syscall
   ```

   Das korrekte Programm gibt die Zahl `16909060` aus.

Lösung

1. Das Programm schlägt fehl, versucht wird, ein Wort aus einer Adresse zu laden, welche nicht durch 4 teilbar ist. Deshalb muss man im Code ein `.align 2` hinzufügen, sodass das Label `num` wieder an einer durch 4 teilbaren Adresse liegt. (Zur Erinnerung: `.align n` sorgt dafür, dass die nächste Instruktion an einer durch 2^n teilbaren Adresse liegt!)

   ```
   1 .data
   2 msg: .byte 'P','r','o','g',0
   3 .align 2
   4 num: .byte 2,0,0,0
   5
   6 .text
   ```

```
7 la $a0 msg
8 ...
```

2. Ja, dies ist durch ein `.space 3` an derselben Stelle wie in Aufgabe 1 möglich, da dies das 5te Bit aus `msg` aufnimmt und so die nächste Zeile mit 4 alignt. Dennoch sollte generell die Ausrichtung mit Hilfe von `.align` bevorzugt werden, da sie auch nach Änderung des Programmcodes auf jeden Fall korrekt bleibt.

3. Eine Möglichkeit besteht darin, `lb` statt `lw` zum Laden der Zahl 2 aus der Adresse zu benutzen:

```
1 .data
2 msg: .byte 'P','r','o','g',0
3 num: .byte 2,0,0,0
4
5 .text
6 la $a0 msg
7 li $v0 4 #Gebe Zeichenkette aus (syscall 4)
8 syscall
9 lb $a0 num
10 li $v0 1 #Gebe Integer aus (syscall 1)
11 syscall
```

4. Eine Lösung wäre folgendes Programm:

```
1 .data
2 .space 1
3 wort: .byte 4,3,2,1
4
5 .text
6 la $t0 wort
7
8 lb $a0 0($t0)
9 lb $t1 1($t0)
10 sll $t1 $t1 8
11 or $a0 $a0 $t1
12 lb $t1 2($t0)
13 sll $t1 $t1 16
14 or $a0 $a0 $t1
15 lb $t1 3($t0)
16 sll $t1 $t1 24
17 or $a0 $a0 $t1
18
19 li $v0 1
20 syscall
```

Aufgabe 3.5: Little und Big Endian

In folgender Tabelle ist jeweils zu einer 32 Bit breiten Zahl in Hexadezimaldarstellung die Darstellung im Speicher in Little und Big Endian angegeben. Vervollständigen Sie die Tabelle.

Hexadezimaldarstellung	Little Endian	Big Endian
0x11223344		
	DE AD C0 DE	
		F0 05 BA 11
0xDECAFF01		
	C0 FF EE 00	
		BA 5E BA 11

Lösung

Hexadezimaldarstellung	Little Endian	Big Endian
0x11223344	44 33 22 11	11 22 33 44
0xDEC0ADDE	DE AD C0 DE	DE C0 AD DE
0xF005BA11	11 BA 05 F0	F0 05 BA 11
0xDECAFF01	01 FF CA DE	DE CA FF 01
0x00EEFFC0	C0 FF EE 00	00 EE FF C0
0xBA5EBA11	11 BA 5E BA	BA 5E BA 11

Aufgabe 3.6: Speicherung und Interpretation binärcodierter Daten

Der folgende Speicherauszug eines MIPS Programms zeigt den Speicherinhalt ab Adresse 0x10000000. Dieser enthält die persönlichen Daten eines Studenten. In der folgenden Darstellung sind pro Zeile 4 MIPS-Wörter in Hexadezimal-Schreibweise dargestellt, insgesamt also 16 Bytes.

Adresse	Speicherinhalt			
0x10000000	74733973	64757473	07d00c19	fffffde8
0x10000010	42464e49	00000003	00000000	00000000

1. Welche Werte sind im Speicher abgelegt, wenn man folgende Interpretationen unterstellt. Beachten Sie die Endianess des MIPS-Prozessors.

Offset	Interpretation	Werte
0 – 7	Kennung (8 ASCII-Zeichen)	
8	Geburtsdatum: Tag (unsigned, 1 Bytes)	
9	Geburtsdatum: Monat (unsigned, 1 Bytes)	
10 – 11	Geburtsdatum: Jahr (unsigned, 2 Bytes)	
12 – 15	Semesterbeitrag Saldo (signed, 4 Bytes)	
16 – 18	Studiengang (3 ASCII-Zeichen)	
19	Abschluss (1 ASCII-Zeichen)	
20	Fachsemester (unsigned 1 Byte)	

2. Notieren Sie die Datendefinitionen (`.data`), die nach dem Laden diese Speicherbelegung liefern! Überprüfen Sie die Wirkung Ihrer Datendefinitionen im MIPS Simulator.

Lösung

1. Die Werte:

Offset:	Werte:
0 – 7	s9ststud
8	0x19 bzw. 25
9	0x0c bzw. 12
10 – 11	0x07d0 bzw. 2000
12 – 15	0xfffffde8 bzw. −536
16 – 18	INF
19	B
20	0x03 bzw. 3

2. Die Datendefinitionen:

```
1 .data
2 .ascii "s9ststud"       # Folge von 8 chars, 8 Bytes
3 .byte  0x19             # unsigned integer, 1 Byte
4 .byte  0x0c             # unsigned integer, 1 Byte
```

```
5 .half   0x07d0          # unsigned integer, 2 Bytes
6 .word   0xfde8          # signed integer, 4 Bytes
7 .ascii "INF"            # Folge von 3 chars, 3 Bytes
8 .ascii "B"              # char, 1 Byte
9 .byte   0x3             # unsigned integer, 1 Byte
```

Aufgabe 3.7: Bedingter Sprung (oder nicht)

Schreiben Sie ein Assemblerprogramm, das den Absolutwert eines 32-Bit-Wertes aus dem Datenbereich berechnet und ausgibt. Nehmen Sie dabei an, dass der Wert im Speicher an der Adresse mit dem Label `zahl` steht.

1. Verwenden Sie zur Implementierung einen bedingten Sprungbefehl.
2. Implementieren Sie nun das Programm ohne Sprungbefehle.

```
1 .data
2 zahl:
3   .word -2147
4
5 .text
6 #Hier Ihre Lösung
```

Lösung

```
1 .data
2 zahl:
3   .word -2147
4
5 .text
6   lw $a0 zahl
7   bgez $a0 exit
8   subu $a0 $0 $a0
9 exit:
10  li $v0 1 #Ausgabe
11  syscall
12  li $v0 10
13  syscall
```

```
1 .data
2 zahl:
3   .word -2147
4
5 .text
6   lw $a0 zahl       # $a0 >= 0          $a0 < 0
7   sra   $t0 $a0 31  # $t0   = 0...0     $a0 < 0     $t0   = 1...1
8   xor   $a0 $a0 $t0 # $a0 xor 0 = $a0 $a0 xor 1...1 = Einserkomplement $a0
9   subu $a0 $a0 $t0 # $a0 -= 0 = $a0  $a0 -= -1 -> $a0 enthält Zweierkomplement
10  li $v0 1 #Ausgabe
11  syscall
12  li $v0 10
13  syscall
```

Aufgabe 3.8: Überlauf

Schreiben Sie ein Programm, das die Werte der Register $a0$ und $a1$ addiert, und die Marke `overflow` anspringt, wenn ein Überlauf bezüglich der vorzeichenbehafteten Interpretation vorliegt.

Hinweis: Ein Überlauf kann nur vorliegen, wenn die Vorzeichen der beiden Summanden gleich sind, aber das Ergebnis der Addition ein anderes Vorzeichen hat.

Lösung

```
1   .data
2 input:  0x80000000, 0xffffffff
3
4   .text
5
6 # Lade Testwerte
7   la $a1 input
8   lw $a0 0($a1)
9   lw $a1 4($a1)
10
11 # Zunächst die Zahl tatsächlich addieren
12  addu $t1 $a0 $a1
13
14 # Die folgenden Operationen verwenden zwar alle Bits,
15 # wir beachten am Ende aber nur das erste (das Vorzeichenbit)
16
17 # Ein Überlauf kann nur auftreten, wenn beide Summanden das gleiche Vorzeichen haben:
18  xor $t0 $a0 $a1   # unterschiedliche Bits finden
19  nor $t0 $zero $t0 # invertieren (gleiche Bits finden)
20
21 # Ein Überlauf ist aufgetreten, falls das Vorzeichenbit der Summe abweicht
22 # von dem der Summanden
23 # Da wir schon geprüft haben, ob beide Vorzeichen gleich sind,
24 # müssen wir nur mit einem davon vergleichen
25
26  xor $t1 $t1 $a0
27
28 # Beide Bedingungen vernüpfen
29  and $t0 $t0 $1
30
31 # Wir wollen überprüfen, ob das erste Bit eine 1 ist, alle anderen sind uninteressant
32 # Falls das erste Bit eine 1 ist, gilt die Zahl als negativ
33  bltz $t0 overflow
34
35 # Zum Testen: gebe 'n' aus wenn kein overflow
36  li $a0 'n'
37  b end
38
39 overflow:
40 # Zum Testen: gebe 'o' aus wenn overflow
41  li $a0 'o'
42
43 end:
44  li $v0 11
45  syscall
46  li $v0 10
```

```
47   syscall
```

Aufgabe 3.9: RGB-Basics

Sie bekommen eine vorzeichenlose Zahl (32 Bit), die eine Farbe im RGB-Format enkodiert, als Eingabe in $a0. Filtern Sie die einzelnen Farbwerte heraus und geben Sie sie negiert aus. Eine Farbe wird negiert, indem jeder Farbwert x durch $255 - x$ ersetzt wird.

Anmerkung: Eine Zahl im RGB-Format enthält jeweils für rot, grün und blau einen Wert zwischen 0 und 255. Dabei werden die Werte in der angegebenen Reihenfolge hintereinander geschrieben.
Beispiel:
Die folgende Bitfolge stellt einen RGB-Wert mit einem Rotanteil von 50, einem Grünanteil von 100 und einem Blauanteil von 150 dar: 00000000 $\underbrace{00110010}_{rot}$ $\underbrace{01100100}_{\text{grün}}$ $\underbrace{10010110}_{blau}$
Wenn wir diese Farbe negieren erhalten wir einen Rotanteil von 205, einen Grünanteil von 155 und einem Blauanteil von 105, also den Farbwert: 00000000 $\underbrace{11001101}_{rot}$ $\underbrace{10011011}_{\text{grün}}$ $\underbrace{01101001}_{blau}$
Was Fällt Ihnen anhand dieses Beispieles auf? Gilt Ihre Beobachtung für alle Farben? Warum?

Lösung

Die einzelnen Bits, die einen Farbanteil darstellen, werden einfach nur geflippt. Dies gilt für alle Farben, weil 255 der größte Wert ist, den man mit 8 Bit unsigned Zahlen erreichen kann.

```
1 # negiere Eingabe
2 li $t0 0x00FFFFFF
3 sub $a0 $t0 $a0
4 # kopiere negiertes Eingabeargument in a1
5 move $a1 $a0
6
7 # shifte rotwert
8 srl $a0 $a0 16
9 # Ausgabe
10 li $v0 1
11 syscall
12
13 # kopiere negiertes Argument zurück in a0
14 move $a0 $a1
15 # schiebe grünwert nach hinten
16 sll $a0 $a0 16
17 srl $a0 $a0 24
18 #Ausgabe
19 syscall
20
21 # kopiere negiertes Argument zurück in a0
22 move $a0 $a1
23 # schiebe blauwert nach hinten
24 sll $a0 $a0 24
25 srl $a0 $a0 24
26 # Ausgabe
27 syscall
```

Aufgabe 3.10: Stringrev

Schreiben Sie ein Programm, welches eine ASCIIZ-Zeichenkette reversiert. Sie erhalten in `$a0` die Anfangsadresse der Zeichenkette und in `$a1` die Anfangsadresse des Zielpuffers. Sie können davon ausgehen, dass der Zielpuffer immer groß genug ist.
Sollten Sie Hilfe brauchen, um die Aufgabe zu lösen, orientieren Sie sich an folgenden Punkten:

a) Machen Sie sich Gedanken, was die Besonderheit eines Strings ist.

b) Beginnen Sie dann damit einen Programmteil zu schreiben, welcher das Ende einer solchen Zeichenkette sucht.

c) Schreiben Sie nun einen weiteren Programmteil, welcher die Zeichen von einer Adresse an eine andere Adresse kopiert.

d) Verbinden Sie die Programmteile sinnvoll, denken Sie dabei daran das Programm korrekt zu beenden.

Lösung

```
1 .text
2
3     move      $t0 $a0          # merke Anfangsadresse der ASCIIZ-Zeichenkette
4
5 # ---b---
6 loop_scan:
7   lbu       $t1 ($t0)
8     addiu     $t0 $t0 1        # nächsten Buchstaben betrachten
9     bnez      $t1 loop_scan    # noch nicht am Ende des Strings angekommen
10
11
12 found_end:
13    addiu     $t0 $t0 -2       # letztes Zeichen der Eingabe
14
15 # ---c/d---
16 loop_write:
17    bltu      $t0 $a0 finish   # wieder am Anfang der Eingabe angekommen
18
19    lbu       $t1 ($t0)        # kopiere Zeichen in Zielpuffer
20    sb        $t1 ($a1)
21    addiu     $t0 $t0 -1       # gehe jeweils ein Zeichen weiter
22    addiu     $a1 $a1 1
23
24    j         loop_write
25
26 # ---d---
27 finish:
28    sb        $zero ($a1)      # String mit 0 abschliessen
29
30    li        $v0 10           # beende Programm
31    syscall
```

Aufgabe 3.11: Eingabe/Ausgabe, ASCII und Schleifen in MIPS

Schreiben Sie ein Assemblerprogramm, das die Caesar-Verschlüsselung einer Zeichenkette mit einer Verschiebung von n berechnet und ausgibt. Bei der Caesar-Verschlüsselung wird jeder Buchstabe durch den Buchstaben ersetzt, der n Stellen weiter im Alphabet steht. Am Ende des Alphabets wird von vorne weitergezählt. Lesen Sie n und eine, bis zu 20 Zeichen lange, Zeichenkette vom Benutzer ein. Sie dürfen dabei annehmen, dass dieser nur die ASCII-Zeichen a-z, oder einen Zeilenumbruch (bei kürzerer Eingabe) eingibt.

Lösung

```
.data
input:
  .space 21
eingabestr:
  .asciiz "Geben␣Sie␣den␣Text␣ein␣(max.␣20␣Zeichen):\n"
eingabeint:
  .asciiz "Geben␣Sie␣die␣Verschiebung␣ein:\n"
newline:
  .asciiz "\n"
.text
.globl main
main:
  li $v0,4 #print_string
  la $a0,eingabestr
  syscall
  li $v0,8 #read_string
  la $a0,input
  li $a1,21
  syscall
  li $v0,4 #print_string
  la $a0,newline
  syscall
  li $v0,4 #print_string
  la $a0,eingabeint
  syscall
  li $v0,5 #read_int
  syscall
  li $v0,4 #print_string
  la $a0,newline
  syscall
  move $t0,$v0 #offset in t0 speichern
  li $t1,0 #Iterationsvariable t1 initialisieren
  li $t5,26 #Anzahl Zeichen in [a-z]
loop:
  lbu $t2,input($t1) #input[i] laden
  slti $t3,$t2,97 #input[i] < 'a'; behandelt auch das Stringende
  bgtz $t3,exit #dann Abbruch
  slti $t3,$t2,122 #input[i] < 'z'; nicht zwingend gefordert in der Aufgabe
  blez $t2,exit #wenn nicht dann Abbruch
cipher:
  subi $t2,$t2,97 #t2 = input[i] - 'a'; um 'a' auf 0 abzubilden
  add $t2,$t2,$t0 #t2 = t2 + offset; die Verschiebung anwenden
  divu $t2,$t5
  mfhi $t4 #t4 = t2 mod 26; overflow
  addi $t4,$t4,97 #t4 = t4 + 'a'
  sb $t4,input($t1) #rotiertes Zeichen zurückspeichern
```

```
47   addi $t1,$t1,1 #Iterationsvariable erhöhen
48   b loop
49 exit:
50   sb $zero,input($t1)
51   li $v0,4
52   la $a0,input
53   syscall
54   li $v0,10
55   syscall
```

Aufgabe 3.12: Palindrome

Palindrome sind Zeichenketten, die von vorn und von hinten gelesen dasselbe ergeben. Schreiben Sie ein Programm, welches prüft, ob eine ASCIIZ-Zeichenkette, deren Anfangsadresse in $a0 steht, ein Palindrom ist. Ihr Programm soll im positiven Fall 42 zurückgeben und ansonsten 0. Sie dürfen dabei das Unterprogramm aus der vorherigen Aufgabe verwenden, insbesondere mit der Annahme, dass in $a1 schon ein ausreichend großer Zielpuffer zur Verfügung steht. Dies ist allerdings nicht zwingend notwendig, da Sie den String hierfür nicht reversieren müssen

Lösung

```
 1     move $t0 $a0          # move address
 2     li $t4 0
 3 counting:
 4     lb $t3 ($t0)          # lade das n-te Byte (Ascii-Zeichen)
 5     beqz $t3 counted      # wenn es 0 ist, ist der String zu Ende
 6     addiu $t4 $t4 1       # erhöhe meinen Counter
 7     addiu $t0 $t0 1       # setze die Addresse aufs nächste Byte im Puffer
 8     b counting
 9
10 counted:
11
12     move $t5 $a0
13     addiu $t4 $t4 -1      # setze Offset und Counter um 1 zurück
14     addiu $t0 $t0 -1
15
16
17 loop:
18     lb $t1 ($t5)          # lade die beiden Bytes an der jeweiligen Position
19     lb $t2 ($t0)
20
21     bne $t1 $t2 nopal     # falls sie nicht gleich sind, dann Abbruch
22     addiu $t5 $t5 1       # erhöhe Addresse "von links"
23     addiu $t0 $t0 -1      # verringere Addresse "von rechts"
24
25     bge $t5 $t0 end       # vergleiche Addressen (String komplett geprüft?)
26
27     b loop
28
29
30 end:
31     li $v0 42             # Palindrom gefunden
32     jr $ra
33
34 nopal:
35     li $v0 0              # kein Palindrom
36     jr $ra
```