# Dokumentation

Rechenarchitektur

Hangman

in

RISC-V Assembler

Von

Tobias Lukasewitz

[4659023]

# Inhaltsverzeichnis

# 1. Hangman

Bei dem Spiel „Hangman“ geht es um mindestens zwei Spieler, die das Wort von einem der Spieler erraten müssen. Derjenige, der das Wort ausdenkt, zeigt den restlichen Spielern die Länge des Wortes in Form von Unterstrichen an. Denkt sich einer der Spieler das Wort „Hangman“ aus, so sehen die restlichen Spieler zu Spielbeginn sieben Unterstriche.

Die restlichen Spieler versuchen durch das Raten von Buchstaben das Wort so weit zu vervollständigen, dass einer der Spieler das Wort erraten kann. Ob dies in Form eines Wettkampfes oder als Team stattfindet, ist von der Spielergruppe zu entscheiden.

Wird ein falscher Buchstabe geraten, so vervollständigt sich die Zeichnung. Die Anzahl der Versuche wird ebenfalls meist von der Spielergruppe ausgemacht. Ist die Zeichnung vollständig, ist das Spiel verloren. Konnte das Wort erraten werden, ist das Spiel gewonnen.

# 2. Programmstruktur

Das Programm ist in vier Dateien aufgespalten. Jede Datei hat ihre eigene Aufgabe:

- **Controller** – Die main-Datei des Programms. Sie koordiniert den Programmablauf von Anfang bis Ende. Als Steuerung inkludiert sie dementsprechend alle Dateien des Programms. Bis auf die Koordination trägt sie keinen Eigenbeitrag zur Logik des Spiels bei.
- **Words** – Diese Datei enthält die zur Verfügung stehenden Wörter. Sie kümmert sich über alles was zur Bestimmung des Wortes oder zur Ausgabe von QoL beiträgt.
- **Game** – Diese Datei enthält die Logik des Spiels. Sie enthält Funktionen wie die Ausgabe des Spielstandes, das Ersetzen der Buchstaben und das Validieren des Spielstandes und der Eingabe.
- **Interactive** – Diese Datei ist verantwortlich für den User-Input. Sie nimmt einzelne Buchstaben des Keyboard-Simulator von RARS entgegen und gibt die Information an die Steuerung weiter.

Das Spiel selbst ist ein Einzelspielerspiel. Anstatt eines zweiten Spielers tritt der Spieler gegen den Computer an, der aus einer Liste an Wörtern ein Wort zufällig auswählt. Folglich präsentiert der Computer die Länge des Wortes dem Spieler mit Unterstrichen.

Gespielt wird ausnahmslos mit Kleinbuchstaben. Die Eingabe von anderen Zeichen wird vom Programm so lange ignoriert, bis der Spieler wieder ein valides Zeichen angibt. Das Programm ist so aufgebaut, dass es nur das Keyboard-Interface von RARS benötigt. Die Hangman-Zeichnung wird dem Spieler als numerischen Wert angezeigt.

# 3. Spielablauf

Vor Beginn des eigentlichen Spiels müssen Vorbereitungen getroffen werden. Der Spielverlauf wird in der Controller.asm-Datei geregelt. Die [Abb. 1.1] zeigt den Code für die Vorbereitung. Bevor das Spiel startet, wird folgendes getan:

- **get_random_word** – Computer wählt ein zufälliges Wort auss
- **after_last_address** – Ermittlung der Adresse hinter dem Wort-Array. Das Ergebnis ist die Start-Adresse des Check-Arrays, das den jeweiligen Spielzustand speichert.
- **fill_dst_str** – Initialisierung des Check-Arrays mit Null. Das Check-Array speichert den Spielzustand. Eine Null bedeutet, dass der Buchstabe noch nicht aufgedeckt wurde. Ist ein Buchstabe bereits aufgedeckt, wird ist an der entsprechenden Stelle im Check-Array der ASCII-Wert eingetragen. Der Startzustand ist alle Indizes auf null. Der gezielte Endzustand ist, dass das Check-Array dem Wort-Array entspricht, also alle Buchstaben aufgedeckt sind.
- **print_enter** – Gibt eine Eingabeaufforderung an den Spieler aus
- **print** – Ausgabe des Spielstandes. Gibt am Start das Wort als Unterstriche aus. Wird auch verwendet, um die Ausgabe zu aktualisieren, wenn ein Spielzug gemacht wurde.
- **print_tries** – Ausgabe der noch zur Verfügung stehenden Versuche

```
75  main:
76      #a0 = word-array (should never be changed)
77      #a1 = check-array (should never be changed)
78      #a2 = key
79      #a3 = states
80      #a4 = number of tries (global)
81
82      #------------------------------------------------------------
83      # Start - Preperations and Initializations
84      #------------------------------------------------------------
85      li    a4, TRIES              # initialize a4 with the number of tries the user has
86
87      jal   ra, get_random_word    # get a random word from wordlist
88      mv    a1, a0                 # move the returned word to its register
89
90      jal   ra, after_last_address  # calculate the address for the check-array
91      jal   ra, fill_dst_str        # initialize check-array by 0
92
93      jal   ra, print_enter         # prints a user prompt to enter a lower case letter
94      jal   ra, print               # prints the game state (initially only in undercores)
95      jal   ra, print_tries         # prints the number of tries left (initially max)
96
97      #used for debugging (prints the word)
98      mv    t0, a0
99      mv    a0, a1
100     li    a7, 4
101     ecall
102     mv    a1, a0
103     mv    a0, t0
104
105     #------------------------------------------------------------
106     # Game Loop
107     #------------------------------------------------------------
108     loop:                        # game round ends the program
109     jal   ra, game.round         # call the game round
110     j     loop                   # loop
111
112     #------------------------------------------------------------
113     # End of program
114     #------------------------------------------------------------
115     end.program:
116     li    a7, 10
117     ecall
```

Abbildung 3.1: Ausschnitt aus der main. Von Vorbereitung bis zur Game Loop

Im unteren Abschnitt der [Abb. 3.1] ist die Game-Loop zu sehen. Die Game Loop ist eine Endlosschleife, die von der `game_round` abgebrochen wird, sobald einer von zwei Endzuständen erreicht ist. Das heißt, dass der Spieler entweder gewonnen oder verloren hat.

Der Code für eine Spielrunde ist in [Abb. 3.2] abgebildet und befindet sich ebenfalls in der Controller.asm. Die game_round koordiniert den Spielablauf von der Eingabe des Users bis zur Überprüfung auf die Endzustände.

- **wait_key_pressed** – Ruft die Methode in interactive.asm auf, die auf die User-Eingabe wartet. Erhält sie eine Eingabe des Users speichert sie en Key ins Rückgaberegister `a0` ab. Folglich wird die Information in ihr zugehöriges Register `a2` geschoben.
- **replace** – Wird nach der Eingabe aufgerufen und schaut, ob der Buchstabe eins oder mehrere Unterstriche ersetzen kann. In dem Fall, dass eine Ersetzung erfolgt, wird das state-Register `a3` auf 1 gesetzt.
- **print_[prompt]** – Ausgabe von Aufforderungen oder Informationen
- **print_[state]** – Ausgabe des jeweiligen Endzustandes
- **check** – Überprüft, ob eins der Endzustände, primär der Siegzustand, erreicht wurde.

```
26    #-------------------------------------------------------------
27    # get user-input
28    #-------------------------------------------------------------
29
30    jal   ra, wait_key_pressed # wait for user-input and get the key
31    mv    a2, a0               # move the key in its register
32    lw    a0, 4(sp)            # restore a0
33    lw    a1, 8(sp)            # to make sure restore a1 as well
34
35    #-------------------------------------------------------------
36    # check if user-input is in word
37    #-------------------------------------------------------------
38
39    jal   ra, replace          # look if user-input is in word
40
41    #-------------------------------------------------------------
42    # print game state
43    #-------------------------------------------------------------
44
45    jal   ra, print_enter      # print user prompt
46    jal   ra, print            # print the current game state
47
48    #-------------------------------------------------------------
49    # check game-conditions
50    #-------------------------------------------------------------
51    bnez  a3, cwc              # branch if state says that a replacement happened
52    li    t0, 1                # if no replacement happened a trie should be subtracted
53    sub   a4, a4, t0           # decrement the try counter
54    beqz  a4, print_lose       # check if lose-condition has been reached (a4==0)
55    j     skip                 # skip further checks for win-conditions
56
57    cwc:                       # check for win-condition if a replacement happened
58    jal   ra, check            # call the needed function
59    bnez  a3, print_win        # check if win-condition has been reached (a4!=0)
60
61    skip:
62    jal   ra, print_tries      # print number of tries the user has left
```

Abbildung 3.2: Code der Game-Round

# 4. Funktionen

In diesem Kapitel sollen die einzelnen Hauptfunktionen näher betrachtet werden. Methoden die grundsätzlich dem QoL dienen und damit nicht sonderlich zur Funktion des Spiels beitragen, sollen hier nicht näher analysiert werden.

## 4.1. Auswahl des Wortes

Das Array, das auf die Wörter verweist, ist über Labels aufgebaut. Die Struktur ist in [4.1] zu sehen. Jedes Label verweist auf eine Null-Terminierte Zeichenkette. Der Null-Terminator repräsentiert das Ende des Wortes. Intern handelt es sich bei Labels um Adressen. Diese Adressen werden dem Array als Elemente zugewiesen. Beim hinzufügen von Wörtern muss nur die Konstante für die Anzahl der Wörter aktualisiert werden.

```
175 .data
176 enter:    .asciz   "Enter a lower-case letter: "
177 tries:    .asciz   "Number of tries left: "
178
179 lose.msg: .asciz   "You lose!"
180 win.msg:  .asciz   "You win!"
181
182 #------------------------------------------------------------------
183 # Collection of available words
184 #------------------------------------------------------------------
185 word0:    .asciz   "dampfschiff"
186 word1:    .asciz   "gumba"
187 word2:    .asciz   "hallo"
188 word3:    .asciz   "computer"
189 word4:    .asciz   "motoren"
190 word5:    .asciz   "geburtstag"
191 word6:    .asciz   "langeweile"
192 word7:    .asciz   "probleme"
193 word8:    .asciz   "dokumentation"
194 word9:    .asciz   "assembler"
195
196 #------------------------------------------------------------------
197 # Array that contains the location address of each word
198 #------------------------------------------------------------------
199 wordlist:
200     .word    word0
201     .word    word1
202     .word    word2
203     .word    word3
204     .word    word4
205     .word    word5
206     .word    word6
207     .word    word7
208     .word    word8
209     .word    word9
```

Abbildung 4.1: Aufbau der `wordlist`

Die [Abb. 4.2] zeigt die Auswahl des Wortes. Der Generierung des zufälligen Index wird der RARS ecall verwendet. Dieser erlaubt die Generierung eines (pseudo-)zufälligen Integers innerhalb einer Grenze. Dieser Wert wird über das Rückgaberegister `a0` dem Aufrufer zurückgegeben.

Um auch die Startadresse zu erhalten, muss der Offset berechnet werden. Der Offset entspricht dem generierten Index multipliziert mit den Faktor 4. Wie in der Zeigerarithmetik kann der Offset auf die Startadresse des Arrays addiert werden. Diese Summe der Rechnung ist die Startadresse des

gewählten Wortes. Das Wort wird danach in das Rückgaberegister `a0` geladen und dann in sein zugehöriges Register `a1`.

```
#-----------------------------------------------------------------
# word determonation
#-----------------------------------------------------------------
# function gets no input and returns a random word from the wordlist-array. The
# random index is generated via the get_random_number function.
# output:   a0 - random word chosen from wordlist
get_random_word:
   addi  sp, sp, -4
   sw    ra, (sp)            # save return address

   jal   get_random_number   # call get_random_number to generate random number
   li    t0, 4               # factor for offset calculation
   mul   a0, a0, t0          # calculation of offset
   la    t0, wordlist        # load start-address of wordlist-array
   add   t0, t0, a0          # add the offset to the start-address
   lw    a0, (t0)            # load the word at the random generated index

   lw    ra, 0(sp)           # load return address
   addi  sp, sp, 4
   ret



# function gets no input and return a random Integer from the RARS ecall
# output:   a0 - random number
get_random_number:
   li    a0, 2               #generator (2, seemed to word nicely)
   li    a1, LENGTH_ARRAY    #bounds
   li    a7, 42              #ecall id-number
   ecall                     #ecall: random-int
   ret
```

Abbildung 4.2: Wort Auswahl

## 4.2. Speicherung des Spielstandes

Gespeichert wird der Spielzustand in einem zusätzlichen Array, dem Check-Array, dass sich genau hinter dem Wort-Array befindet und genau die Länge des ausgewählten Wortes hat. Dazu wird in `after_last_address` in [Abb. 4.4] die Startadresse des Arrays ermittelt. Die Startadresse des Check-Arrays bildet sich aus der Summe der Array-Startadresse und der Länge des Arrays.

```
#-----------------------------------------------------------------
# check-Array initialization
#-----------------------------------------------------------------
# function gets the desination and source address as input. It sets all the following bytes of the dst-address
# to zero, until the src-word has reached the null-terminator. This function is used to initialize the check-array
# with zeros. The zero-sate of the check-array is the starting-state. After a letter is discovered, the indizes
# will be replaced by their ASCII-value. The loop-time is depended on the word-length.
# input:    - a0: destination-address (array)
#           - a1: source-array
fill_dst_str:
   #used to fill the cpy-array with zeros
   #a0 = dst
   #a1 = src
   mv    t0, a0              #mv dst to seperate register
   mv    t1, a1              #mv src to seperate register
   li    t2, 0               #src letter
   li    t3, 0               #dst letter

   copy.s:
   lb    t2, 0(t1)           #get current src letter
   sb    t3, 0(t0)           #store zero at current position
   beq   t2, zero, copy.e    #check for null-terminator

   addi  t0, t0, 1           #advance
   addi  t1, t1, 1           #advance
   jal   zero, copy.s        #get back to start

   copy.e:
   jalr  zero, 0(ra)
```

Abbildung 4.3: Initialisierung des Check-Arrays

```
124 # function gets no input and returns the address after the last word in the wordlist-array.
125 # The address is needed for the second check-array, which keeps track of the letters that
126 # have been dicovered or are still undiscovered.
127 # output:   a0 - address after last word in wordlist-array
128 after_last_address:
129     li    a0, 4                 # factor for offset calculation
130     li    t0, LENGTH_ARRAY      # number of words inside the array
131     mul   t0, t0, a0            # calculate the required offset
132
133     la    a0, wordlist          # load start-address of wordlist-array
134     add   a0, a0, t0            # add offset to start-address
135     jalr zero, 0(ra)
```

Abbildung 4.4: Ermittlung der Startadresse des Check-Arrays

Um den Anfangszustand auch sicherzustellen, wird jedes Element des Arrays mit Null besetzt. Dazu dient die Funktion `fill_dst_str` in [Abb. 4.3]. Im weiteren Spielverlauf ändern sich die Elemente in ihre jeweiligen ASCII-Werte, sodass der Inhalt des Adressraums dem des Wortes entspricht.

## 4.3. Ausgabe

Die [Abb. 4.5] zeigt den Code für die Konsolenausgabe. Die [Abb. 4.6] zeigt, wie die Ausgabe auf der Konsole aussieht. Um die Registerwerte von a0 und a1 konstant zu halten, werden die Inhalte für die Iteration in ein separates Register geschoben. In jedem Schleifendurchlauf wird am Anfang das aktuelle Zeichen von beiden Arrays geholt.

Über dem des Wort-Arrays wird geprüft, ob das Ende des Worts erreicht wurde. Das Zeichen des Check-Array wird verwendet, um zu überprüfen, ob das Zeichen bereits aufgedeckt wurde. Ist es das, dann wird das Zeichen auch ausgegeben, ansonsten wird ein Unterstrich ausgegeben. Vor der nächsten Iteration folgt immer ein Leerzeichen. Mit dem Erreichen vom Wortende wird ein Zeilenumbruch gemacht.

```
 98 # function prints the game-state which translates to how many letters have been discovered. Letter which are discovered
 99 # have their ASCII-Value placed in the check-array. If the letter is undiscovered their value will be 0. In that case a
100 # underscore will be printed instead of the character.
101 # input:       - a0: check-array
102 #              - a1: word-array
103 print:
104     addi  sp, sp, -16
105     sw    ra, 0(sp)
106     sw    a0, 4(sp)
107     sw    a1, 8(sp)
108     sw    s0, 12(sp)
109
110     li    a7, 11          # set ecall for printChar
111
112     pg.loop.s:
113     lb    t1, 0(a1)       # get word-letter
114     lb    t0, 0(a0)       # get check-letter
115
116     addi  a1, a1, 1       # advance iteration
117     addi  a0, a0, 1       # advance iteration
118
119     mv    s0, a0          # save sate of a0 for ecalls
120
121     beqz  t1, pg.loop.e   # branch when null-terminator has been reached
122     beqz  t0, underscore  # branch when check-letter is zero
123
124     mv    a0, t1          # symbol: key
125     ecall                 # ecall: printchar
126     j     space
127
128     underscore:
129     li    a0, 0x5F        # symbol: '_'
130     ecall                 # ecall: printchar
131     j     space
132
133        space:
134     li    a0, 0x20        # symbol: ' '
135     ecall                 # ecall: printchar
136
137     mv    a0, s0          # restore a0
138     j     pg.loop.s       # loop
139
140     pg.loop.e:
141     li    a0, 13      # carriage return
142     ecall
143     li    a0, 10      # line feed
144     ecall
```

Abbildung 4.6: Code für die Spielausgabe. Das Zurückladen der Register aus dem Stack und der Rücksprung sind nicht in der Abbildung enthalten

```
Enter a lower-case letter: _ _ _ _ _ _ _ _
Number of tries left: 10
computerEnter a lower-case letter: c _ _ _ _ _ _ _
Number of tries left: 10
Enter a lower-case letter: c o _ _ _ _ _ _
Number of tries left: 10
Enter a lower-case letter: c o m _ _ _ _ _
Number of tries left: 10
Enter a lower-case letter: c o m _ _ _ _ _
Number of tries left: 9
Enter a lower-case letter: c o m _ _ _ _ _
Number of tries left: 8
Enter a lower-case letter: c o m _ u _ _ _
Number of tries left: 8
```

Abbildung 4.7: Konsolenausgabe für das Wort „computer“ bei User-Eingabe: „c o m a n u“

# 5. Game-loop

Die Game Loop ist eine Endlosschleife. Jeder Durchlauf repräsentiert eine Spielrunde. Die Spielrunde beendet das Spiel, sobald einer der Endkonditionen erreicht ist. Beendet wird mit einer Ausgabe, die den erreichten Endzustand wiedergibt. Der Code ist in [Abb. 3.2] abgebildet.

Die folgenden Unterkapitel gehen mehr auf die einzelnen Funktionen der Game Loop ein.

## 5.1. User-Input

Für die Entgegennahme von User-Input wird das Keyboard-Interface von RARS verwendet. RARS setzt einen Wert ungleich Null in die Adresse 0xffff0000, wenn eine Eingabe erfolgt ist. Der ASCII-Wert für den gedrückten Key kann dann aus der Adresse 0xffff004 entnommen werden.

Die Funktion loopt so lange, bis der User einen Buchstaben eingibt. Sobald eine Eingabe erfolgt ist, wird die Eingabe validiert. Handelt es sich bei dem ASCII-Wert nicht um einen lateinischen Kleinbuchstaben, wird weiter auf eine Eingabe gewartet. In der Spielrunde ist der Key im Register a2 zu finden. Der gerade beschriebene Code ist in [Abb. 4.6] abgebildet.

```
7  .eqv KEY_PRESSED  0xffff0000     # set when user-input happened
8  .eqv KEY_VALUE    0xffff0004     # the key that was pressed
9
10 # function is used to retrieve input from user. It will loop until the user enters
11 # a valid input which are letters from a to z.
12 # output:       - a0: key pressed
13 wait_key_pressed:
14    li    t0, KEY_PRESSED         # load address
15    li    t1, KEY_VALUE           # load address
16
17    wkp.loop.s:
18    lw    t2, (t0)                # load KEY_PRESSED to check user-input
19    beqz  t2, wkp.loop.s          # check if key was pressed, otherwise loop
20
21    lw    a0, (t1)                #get key pressed
22
23
24    li t3, 97                     # letter: 'a'
25    blt a0, t3, wkp.loop.s        # branch if entered key is not between a and z
26
27    li t3, 122                    # letter: 'z'
28    bgt a0, t3, wkp.loop.s        # branch if entered key is not between a and z
29
30    jalr  zero, 0(ra)
```

Abbildung 5.1: Eingabe und dessen Validierung

## 5.2. Einsetzen von Buchstaben

Zum Einsetzen von Buchstaben werden wieder beide Arrays iteriert. Das Zeichen des Wort-Arrays wird wieder verwendet zur Erkennung des Wortendes. Das Zeichen des Check-Arrays wird verwendet, um zu überprüfen, ob der Buchstabe an der Stelle schon aufgedeckt wurde.

Ist der Platz bereits belegt oder entspricht der Key nicht dem Buchstaben, der an die Stelle gehört, wird weiter iteriert. Am Ende befindet sich das state-Register a3 auf 0, wenn kein Austausch stattfand oder auf 1, wenn ein Zeichen ausgetauscht wurde.

```
53 # function is used to replace a byte in check-array if the entered character is contained in the
54 # word. The current word-byte is used to check the input and if the null-terminator was reached. The
55 # function will set the state to 1 if a replacement happened for further evaluation.
56 # input:       - a0: check-array
57 #              - a1: word-array
58 #              - a2: key (user-input)
59 # outputs:     - a3: game state [a3==1 (replacement happened), otherwise 0]
60 replace:
61     addi  sp, sp, -12
62     sw    ra, 0(sp)
63     sw    a0, 4(sp)
64     sw    a1, 8(sp)
65
66     li    t0, 0             # word-letter
67     li    t1, 1             # check-letter
68     li    a3, 0             # state
69
70     rp.loop.s:
71     lb    t0, 0(a1)         # get word-letter
72     lb    t1, 0(a0)         # get check-letter
73
74     beqz  t0, rp.loop.e     # end loop if null-terminator has been reached
75     bne   t0, a2, rp.neq    # branch if word-letter does not match input-character
76     beq   t0, t1, rp.neq    # branch if check-letter already has a value
77
78     sb    t0, 0(a0)         # place letter in copy-array
79     addi  a3, zero, 1       # set state to 1 since a replacement happened
80
81     rp.neq:
82     addi  a0, a0, 1         # advance iteration
83     addi  a1, a1, 1         # advance iteration
84     j     rp.loop.s         # loop
85
86     rp.loop.e:
87     lw    ra, 0(sp)
88     lw    a0, 4(sp)
89     lw    a1, 8(sp)
90     addi  sp, sp, 12
91     ret
```

Abbildung 5.2: Austauschfunktion

## 5.3. Überprüfung der Konditionen

Unabhängig davon, ob der Buchstabe richtig war oder nicht wird der Spielstand erneut ausgegeben. Bevor die Anzahl der übrigen Versuche ausgegeben werden kann, muss das Ergebnis der Austauschfunktion ausgewertet werden.

Fand keine Ersetzung statt, wird die Anzahl der Versuche dekrementiert. Zudem muss überprüfen werden, ob der Spieler verloren hat. Hat der Spieler verloren wird das Programm beendet, ansonsten werden die weiteren Überprüfungen übersprungen.

Hat ein Austausch stattgefunden, wird über die Funktion check überprüft, ob der Spieler gewonnen hat. Ist das der Fall, wird das Programm beendet. Ansonsten geht es mit dem Spiel weiter.

```
48    #-------------------------------------------------------------
49    # check game-conditions
50    #-------------------------------------------------------------
51    bnez  a3, cwc              # branch if state says that a replacement happened
52    li    t0, 1                # if no replacement happened a trie should be subtracted
53    sub   a4, a4, t0           # decrement the try counter
54    beqz  a4, print_lose       # check if lose-condition has been reached (a4==0)
55    j     skip                 # skip further checks for win-conditions
56
57    cwc:                       # check for win-condition if a replacement happened
58    jal   ra, check            # call the needed function
59    bnez  a3, print_win        # check if win-condition has been reached (a4!=0)
60
61    skip:
62    jal   ra, print_tries      # print number of tries the user has left
```

Abbildung 5.3: Auswertung der Austauschfunktion und Überprüfung auf Endzustände

Die check-Funktion iteriert wie die anderen Funktionen auch durch beide Arrays. Das Wort-Array wird wieder verwendet, um auf das Wortende zu überprüfen. Ist auch nur eins der check-Zeichen Null, dann wird der weitere Durchlauf abgebrochen und als State wird der Wert 0 zurückgegeben. Sind alle Elemente des check-Arrays mit irgendwelchen ASCII-Werten gefüllt, wird eine 1 dem Aufrufer zurückzugeben.

```
7   #-------------------------------------------------------------
8   # evaluation of game sates
9   #-------------------------------------------------------------
10  # function evaluates if the win-condition has been reached. The condition has been reached
11  # when every letter is filled out which results in the check-array not containing a single
12  # zero. The end will be tracked by the null-terminator in word-array
13  # input:       - a0: check-array
14  #              - a1: word-array
15  # outputs:     - a3: game state [a3==1 (win), otherwise 0]
16  check:
17     addi  sp, sp, -12
18     sw    ra, 0(sp)
19     sw    a0, 4(sp)
20     sw    a1, 8(sp)
21
22     li    t0, 0            # word-letter
23     li    t1, 1            # check-letter
24     li    a3, 1            # state
25
26     ec.loop.s:
27     lb    t0, 0(a1)        # get word-letter
28     lb    t1, 0(a0)        # get check-letter
29
30     beqz  t0, ec.loop.e    # null-terminator reached
31     beqz  t1, ec.no.win    # break when a zero was still found
32
33     addi  a0, a0, 1        # advance iteration
34     addi  a1, a1, 1        # advance iteration
35     addi  a3, zero, 1      # set to win-state
36     j ec.loop.s            # loop
37
38     ec.no.win:
39     addi  a3, zero, 0      # set to zero if a zero was encountered
40
41     ec.loop.e:
42     lw    ra, 0(sp)
43     lw    a0, 4(sp)
44     lw    a1, 8(sp)
45     addi  sp, sp, 12
46     ret
```

Abbildung 5.4: Evaluation, ob der Sieg-Zustand erreicht wurde

# 6. Spielende

Sobald der Spieler gewonnen oder verloren hat, soll eine der Nachrichten ausgegeben werden.

- „You lose!“, wenn der Spieler verloren hat oder
- „You win!“, wenn der Spieler gewonnen hat

Dazu wird über das jeweilige Label die richtige Nachricht in Register `a0` geladen und rufen folgend die Funktion zur Nachrichtenausgabe auf. Die Nachrichtenausgabe verwendet den RARS ecall zur String-Ausgabe. Die [Abb. 6.1] zeigt den Codeausschnitt.

```
64 #------------------------------------------------------------------
65 # Win or Lose Messages
66 #------------------------------------------------------------------
67 # function gets no input and gives the message to display as output. The print function is automatically called
68 # by this funtion. This particular function is used to print the lose-message when the user has lost the game.
69 # output:    - a0: message to be displayed in the console
70 print_lose:
71    la    a0, lose.msg   # load the lose message
72    jal   ra, print_msg  # print the message
73    j     end.program
74 
75 # function gets no input and gives the message to display as output. The print function is automatically called
76 # by this funtion. This particular function is used to print the win-message when the user has won the game.
77 # output:    - a0: message to be displayed in the console
78 print_win:
79    la    a0, win.msg    # load the win message
80    jal   ra, print_msg  # print the message
81    j     end.program
82 
83 # function prints the message given via the register a0 to the console. The message is printed with the RARS ecall
84 # input:     - a0: message to be displayed
85 print_msg:
86    li    a7, 4          # ecall number
87    ecall                # ecall: print string
88    jalr zero, 0(ra)
```

Abbildung 6.1: Ausgabe des Endzustandes