Technische Universität München

Fakultät für Informatik

Rechnerarchitektur - Praktikum (IN0005)

Gruppe 12

Projekt 2 - Ausarbeitung

Verfasser

Vasil Sarafov (vasil.sarafov@tum.de)
Franz Fuchs (franz.fuchs@tum.de)
Yoav Schneider (yoav.schneider@tum.de)

Verantwortlicher

Yoav Schneider (yoav.schneider@tum.de)

Tutor

Artur Faltenberg (artur.faltenberg@tum.de)

9. Juli, 2016

Sommersemester 16

Inhalt

1 Vorwort **2**

2 Anwenderdokumentation **2**
2.1 Einleitung . . . 2
2.2 Voraussetzungen und Installation . . . 2
2.3 Assemblerprogramm . . . 2
2.4 Mikroprogramm . . . 3
2.5 Tests . . . 3
2.6 Allgemeine Hinweise zur Anwendung . . . 4

3 Entwicklerdokumentation **5**
3.1 Einleitung . . . 5
3.2 Assemblerprogramm . . . 5
3.2.1 Befehle . . . 5
3.2.2 Algorithmus . . . 6
3.3 Mikroprogramm . . . 7
3.4 Vergleich beider Implementierungen . . . 8
3.4.1 Assemblerprogramm . . . 8
3.4.2 Mikroprogramm . . . 9

1 Vorwort

Im Rahmen des ERA-Praktikums an der TU München sollen zwei Projekte in Dreier-Teams ausgearbeitet werden. Unsere Gruppe besteht aus den Mitgliedern Franz Fuchs, Yoav Schneider und Vasil Sarafov und muss die folgenden Aufgaben bearbeiten:

- Assembler Realisierung der Funktion $y = arcsin(x)$
- *strcpy* (Kopieren von Zeichenketten) als AM2910 Mikro- und Assemblerprogramm

Im folgenden soll die Realisierung des zweiten Projektes *strcpy* in Ausführung einer schriftlichen Ausarbeitung beschrieben werden. Dabei werden sowohl die Anwender- als auch die Entwicklerdokumentation des Projektes vorgestellt.

2 Anwenderdokumentation

2.1 Einleitung

Dieser Abschnitt der Ausarbeitung dient dazu:

- dem Benutzer zu helfen das Programm möglichst schnell zu installieren
- die Verwendung des Programmes zu beschreiben

Hierbei wird auf technische Details verzichtet. Für eine fachliche Beschreibung des Projektes, lesen Sie bitte die Entwicklerdokumentation.

2.2 Voraussetzungen und Installation

Damit man das Programm verwenden kann, muss man über den Simulator JMic verfügen und den Sourcecode heruntergeladen haben.

2.3 Assemblerprogramm

Der Ordner assembler beinhaltet die Implementierung vom Befehl strcpy RA RB als Assemblerprogramm (Abbildung 1). Die Hauptdatei ist strcpy-assembler.mpr.

Bei dieser Implementierung werden die Start- und Zieladressen in den Registern R0 bzw. R1 übergeben.

Adresse	Wert	Mnemonic
0	d800	push ra x
1	d810	push ra x
2	d820	push ra x
3	202	mov [ra] rb
4	5002	tst x rb
5	8100	jmpz x x
6	d	
7	621	mov ra [rb]
8	4000	inc x rb
9	4001	inc x rb
a	202	mov [ra] rb
b	b000	jmp x x
c	4	
d	621	mov ra [rb]
e	da02	pop x rb
f	da01	pop x rb
10	da00	pop x rb

Abb. 1: strcpy als Assemblerprogramm im Hauptspeicher.

2.4 Mikroprogramm

Der Ordner mikro beinhaltet die Implementierung vom Befehl strcpy ra rb als Mikrobefehl mit Opcode 1 (Abbilding 2). Die Hauptdatei ist strcpy-mikro.mpr.

Bei dieser Implementierung werden die Start- und Zieladressen in den Befehlsparametern übergeben. Der Inhalt der Register kann manuell gesetzt werden.

Ein Bespiel für Kopieren die Zeichenkette von der Adresse in R2 in R3 wäre:

0x0123 ⇔ strcpy R2 R3.

Adresse	Wert	Mnemonic
0	101	strcpy load RA...

Abb. 2: strcpy als Mikroprogramm im Hauptspeicher, wobei RA := R0 und RB := R1

2.5 Tests

Beide Implementierungen lassen sich leicht testen, indem man manuell Daten in den Hauptspeicher ablegt. Es empfiehlt sich, den Zielbereich durch andere Zeichen als 0

zu befüllen, damit das Kopieren des letzten Null leicht sichtbar ist.

Der Nutzer muss darauf achten, dass die gewählte Zieladresse im Speicher genug Platz für die zu kopierende Zeichenkette lässt. Da sich der Stack am Ende des Speicher befindet und mindestens drei Register gepusht werden müssen, empfiehlt es sich, der genutzte Adressbereich bis 0x3fc zu beschränken. Werden weitere Werte gepusht, soll der Adressbereich entsprechend eingeschränkt werden.

Außerdem befinden sich die folgenden Tests in den jeweiligen Unterordnern tests und können direkt ausgeführt werden, indem man sie in einem neuen Fenster im Simulator JMic öffnet und den "Play" - Button drückt:

Für die Assembler Implementierung:

- assembler/tests/01-strcpy-assembler-test-copy-2-symbols.mpr
- assembler/tests/02-strcpy-assembler-test-copy-5-symbols.mpr
- assembler/tests/03-strcpy-assembler-test-copy-25-symbols.mpr
- assembler/tests/04-strcpy-assembler-test-copy-50-symbols.mpr
- assembler/tests/05-strcpy-assembler-test-copy-empty-string.mpr

Für die Mikro Implementierung:

- mikro/tests/01-strcpy-mikro-test-copy-2-symbols.mpr
- mikro/tests/02-strcpy-mikro-test-copy-5-symbols.mpr
- mikro/tests/03-strcpy-mikro-test-copy-25-symbols.mpr
- mikro/tests/04-strcpy-mikro-test-copy-50-symbols.mpr
- mikro/tests/05-strcpy-mikro-test-copy-empty-string.mpr

2.6 Allgemeine Hinweise zur Anwendung

Die folgenden Hinweise gelten für die beiden Implementierungen (Assembler und Mikro):

- Die Programmsimulation muss manuell beendet werden, wenn das Null-Zeichen kopiert wird, denn der Simulator wird versuchen den Hauptspeicher bis Ende auszulesen.
- Die Quelladresse wird immer in RA übergeben.
- Die Zieladresse wird immer in RB übergeben.

- Bei allen Tests befinden sich bereits die Quelladresse in R0 und die Zieladresse in R1.
- Die Hauptspeicherzellen, die sich in der Nähe von der Zieladresse befinden, sind absichtlich mit 1111 gefüllt, um der Prozess vom Kopieren besser zu veranschaulichen und leichter zum Korrigieren zu machen.
- **Es ist sehr wichtig, dass man neue Dateien in einem neuen Fenster öffnet**, weil alle Komponenten der MI-Maschine (Statusflags, Speicher usw.) **neugeladen und nicht überschrieben** werden dürfen.

3 Entwicklerdokumentation

3.1 Einleitung

Hierbei sind die technischen Details des Projektes ausführlich dargestellt. Sie sollen Entwicklern dazu dienen, das Programm schneller zu verstehen, um es möglichst einfach weiterentwickeln oder Code übernehmen zu können.

3.2 Assemblerprogramm

3.2.1 Befehle

Die Assemblerimplemntierung basiert auf den folgenden ebenfalls von uns nach der Spezifikation in *Beschreibung der Zielarchitektur* implementierten Mikrobefehle:

Opcode	**Mnemo**
0x02ab	mov [ra] rb
0x06ab	mov ra [rb]
0x40xb	inc rb
0x50xb	tst rb
0x81xx	jmpz imm
0xb0xx	jmp imm
0xd8ax	push ra
0xdaxb	pop rb

3.2.2 Algorithmus

Das Programm wurde wie folgt implementiert:

Algorithm 1 Unicode Zeichenkette Kopieren als MI-Maschinenenprogramm

```
1:  function STRCPY(r0, r1)
2:      push  r0                                   ▷ sichere r0
3:      push  r1                                   ▷ sichere r1
4:      push  r2                                   ▷ sichere r2
5:
6:      mov   [r0],  r2          ▷ Symbol aus der Quelladresse auslesen
7:
8:      while :                                    ▷ Anfang der Schleife
9:      tst   r2          ▷ Schleifenbedingung, Test ob r2 gleich 0 ist
10:     jmpz  end_while   ▷ Wenn das letzte Zeichen 0 war, beende die Schleife
11:                                                ▷ Schleifenkoerper
12:     mov   r2,  [r1]                ▷ Symbol in Zieladresse speichern
13:     inc   r0                         ▷ Quelladresse inkrementieren
14:     inc   r1                         ▷ Zieladresse inkrementieren
15:     mov   [r0],  r2          ▷ Symbol aus der Quelladresse auslesen
16:     jmp   while                        ▷ Zurück zum Schleifenbeginn
17:
18:     end_while :
19:     mov   r2,  [r1]                          ▷ 0 Speichern am Ende
20:
21:     pop   r2                    ▷ schreibe in r2 seinen alten Wert
22:     pop   r1                    ▷ schreibe in r1 seinen alten Wert
23:     pop   r0                    ▷ schreibe in r0 seinen alten Wert
24: end function
```

Es ist zu beachten, dass der Stapel von unten nach oben wächst. Wir haben `0x03ff` als Startadresse des Stapels gewählt und `R15` als Kellerzeiger.

3.3 Mikroprogramm

Die Mikroprogrammimplementierung erledigt das Kopieren in einem einzigen Mikrobefehl. Dabei werden wiederholte Aufrufe von IFETCH vermieden.

Algorithm 2 Unicode Zeichenkette Kopieren als MI-Mikroprogramm

```
1:  function STRCPY(r0, r1)
2:      ra → r8                          ▷ r8 enthält die Quelladresse
3:      ra → Adressbus                   ▷ Lesezyklus für das erste Zeichen
4:      [ra] → Datenbus                  ▷ r10 enthält das Zeichen
5:      Datenbus → r10
6:      rb → rq                          ▷ rq enthält die Zieladresse
7:      rb → Adressbus                   ▷ starte Schreibezyklus für das erste Zeichen
8:      r10 → Datenbus
9:      Datenbus → [rq]                  ▷ Zeichen wird in die Zieladresse geschrieben
10:     SetFlags                         ▷ Mikro-Flags werden gesetzt
11:
12:     while not ZF                     ▷ Wenn Zero-Flag ist gesetzt springe zu Ende
13:     r8 + 1 → r8
14:     r8 → Adressbus                   ▷ Lesezyklus für das nächste Zeichen
15:     [r8] → Datenbus
16:     Datenbus → r10
17:     rq + 1 → rq
18:     rq → Adressbus                   ▷ Schreibezyklus für das nächste Zeichen
19:     r10 → Datenbus
20:     Datenbus → [rq]                  ▷ Zeichen wird in die Zieladresse geschrieben
21:     SetFlags                         ▷ Mikro-Flags werden gesetzt
22:     while → Mikrobefehlszaehler      ▷ Springen zu While
23:     IFETCH → Mikrobefehlszaehler     ▷ Springen zu IFETCH
24: end function
```

Wobei die folgenden Schritte parallel erfolgen:

2, 3
4, 5
6, 7
8, 9, 10
13, 14
15, 16
17, 18
19, 20, 21

3.4 Vergleich beider Implementierungen

Hierbei werden beide Implementierungen anhand der Taktanzahl für k Zeichen verglichen.

3.4.1 Assemblerprogramm

Taktanzahl für die jeweiligen Befehle inklusive drei Takte für IFETCH:

Befehl	**Taktanzahl**
push ra x	5
pop x rb	6
mov [ra] rb	5
mov ra [rb]	5
inc x rb	4
tst x rb	4
jmp imm	5
jmpz x x	5 (ZF gesetzt)
jmpz x x	6 (ZF nicht gesetzt)

Die gesamte Taktanzahl für k Zeichen beträgt:

Anfang:

Befehl	**Taktanzahl**	**Insgesamt**
push ra x	5 * 3	15
mov [ra] rb	5 * 1	5
		20

Schleifenkörper * k:

Befehl	**Taktanzahl**	**Insgesamt**
tst x rb	1 * 4	4
mov [ra] rb	5 * 1	5
mov ra [rb]	5 * 1	5
inc x rb	4 * 2	8
jmp imm	5 * 1	5
jmpz x x	6 * 1	6
		33k - 1

Da jmpz im letzten Durchlauf nur 5 Takte benötigt, muss ein Takt subtrahiert werden.

Ende:

Befehl	Taktanzahl	Insgesamt
mov ra [rb]	5 * 1	5
pop x rb	5 * 3	15
		20

Insgesamt: $20 + (33k - 1) + 20 = 39 + 33k$ Takte

3.4.2 Mikroprogramm

IFETCH wird einmal aufgerufen. Die Zeilennummern beziehen sich auf den oben genannten Algorithmus.

Anfang:

Befehl	Taktanzahl	Insgesamt
ra ⇒ r8 ra ⇒ Adressbus	1	1
[ra] ⇒ Datenbus Datenbus ⇒ r10	1	1
rb ⇒ rq rb ⇒ Adressbus	1	1
r10 ⇒ Datenbus Datenbus ⇒ [rq] SetFlags	1	1
		4

Schleifenkörper * k:

Befehl	**Taktanzahl**	**Insgesamt**
while not ZF	1	1
r8 + 1 ⇒ r8 r8 + 1 ⇒ Adressbus	1	1
[r8] ⇒ Datenbus Datenbus ⇒ r10	1	1
rq + 1 ⇒ rq rq ⇒ Adressbus	1	1
r10 ⇒ Datenbus Datenbus ⇒ [rq] SetFlags	1	1
while ⇒ Mikrobefehlszähler	1	1
		6

Ende:

Befehl	**Taktanzahl**	**Insgesamt**
IFETCH ⇒ Mikrobefehlszähler	1	1
		1

Insgesamt: $4 + 6k + 1 = 5 + 6k$ Takte

Wie man deutlich sehen kann, ist das Mikroprogramm mehr als fünf mal schneller als das Assemblerprogramm, was ein wesentlicher Unterschied ist, wenn man lange Zeichenkette kopieren muss.