Protokoll Versuch 2

Adrian Schollmeyer

Inhaltsverzeichnis

1 Summe zweier Zahlen

Bei diversen Operationen – vor allem arithmetischen – werden je nach Resultat dieser Operationen Flags gesetzt, welche verschiedene Informationen über das Ergebnis und Ereignisse während der Berechnung liefern. Einige Flags davon sind:

Sign Ist das Flag gesetzt, ist das Ergebnis negativ.

Zero Ist das Flag gesetzt, ist das Ergebnis gleich 0.

Overflow Ist das Flag gesetzt, ist bei der Berechnung ein Overflow aufgetreten, d. h. in der 2K-Darstellung hat sich das Vorzeichen geändert.

Carry Ist das Flag gesetzt, ist das tatsächliche Ergebnis der Berechnung größer als das Register, in dem das Ergebnis gespeichert wird, d. h. ein Übertrag wurde erzeugt.

Tabelle 1 zeigt, in welchem Muster die Flags bei der Abarbeitung der Beispieldaten aufgetreten sind.

Tabelle 1: Flags während der Abarbeitung der Beispieldaten zu Aufgabe 1. Eine 1 bedeutet, dass das Flag gesetzt wurde, eine 0 bedeutet, dass es nicht gesetzt wurde

Datensatz	Sign	Zero	Overflow	Carry
1	0	0	0	1
2	0	0	1	1
3	0	1	0	1
4	1	0	1	0

1.1 Quellcode

Listing 1: A1.asm

```
1  ;Es folgen die Beispieldaten, bitte entsprechende Zeilen einkommentieren
2  ; include "data1_1.inc"
3  ; include "data1_2.inc"
4  ; include "data1_3.inc"
5  ; include "data1_4.inc"
6
7  name "Aufgabe 1"
8
9  ;Hier den eigenen Code einfgen
10 add ah, al
11 mov bl, ah
12 hlt  ;
```

2 Kontrollstrukturen

Kontrollstrukturen in Assembly werden über bedingte Sprünge realisiert. Die Entscheidung, ob ein solcher Sprung durchgeführt wird, basiert auf den zuvor durch andere Operationen gesetzten Flags. Für den Vergleich zweier Werte gibt es dazu den Befehl `cmp`, welcher (lt. Befehlsreferenz) die beiden Operanden voneinander subtrahiert, das Ergebnis im Gegensatz zu `sub` nicht in einem Register ablegt, sondern verwirft. Die dabei gesetzten Flags können dann für bedingte Sprünge genutzt werden, wodurch sich verschiedene Kontrollstrukturen umsetzen lassen.

Bei den bedingten Sprüngen gibt es viele redundante Befehle. Diese Redundanz dient lediglich der einfacheren Lesbarkeit.

2.1 Quellcode

Listing 2: A2a.asm

```
1
2 name "Aufgabe 2"
3
4 MOV AX, 5;
5 MOV CX, 1;
6
7 ;Hier den eigenen Code einfuegen
8 mov cx, ax
9 jcxz ax_zero
10 jmp as_not_zero
11
12 ax_zero:
13     mov bx, 0100
14 as_not_zero:
15     mov bx, 0100h
16
17 hlt  ;
```

Listing 3: A2b1.asm

```
1
2 name "Aufgabe 2"
3
4 MOV AX, 5;
5 MOV CX, 1;
6
7 ;Hier den eigenen Code einfuegen
8 loop_in:
9     dec ax
10    cmp ax, 0
11    jnz loop_in
12
13 hlt  ;
```

Listing 4: A2b2.asm

```
1
2 name "Aufgabe 2"
3
4 MOV AX, 5;
5 MOV CX, 1;
6
7 ;Hier den eigenen Code einfuegen
8 loop_in:
9     sub ax, 1
10    jnz loop_in
11
12 hlt  ;
```

Listing 5: A2c1.asm

```
1
2 name "Aufgabe 2"
3
4 MOV AX, 5;
5 MOV CX, 1;
6
7 ;Hier den eigenen Code einfuegen
8 loop_in:
9     inc ax
10    mov bx, ax
11    sub bx, 100
12    jnz loop_in
13
14
15 hlt  ;
```

Listing 6: A2c2.asm

```
1
2 name "Aufgabe 2"
3
4 MOV AX, 5;
5 MOV CX, 1;
6
7 ;Hier den eigenen Code einfuegen
8 loop_in:
9     inc ax
10    cmp ax, 100
11    jne loop_in
12
13
14 hlt  ;
```

Listing 7: A2d.asm

```
1
2 name "Aufgabe 2"
3
4 MOV AX, 5;
5 MOV CX, 1;
6
7 ;Hier den eigenen Code einfuegen
8 cmp cx, 0
9 je case_0
10 cmp cx, 1
11 je case_1
12 cmp cx, 2
13 je case_2
14 jmp case_default
15
16 case_0:
17     shl cx, 1
18 case_1:
19     shr cx, 1
20 case_2:
21     xor cx, 007eh
22 case_default:
23     and cl, 0073h
24 hlt  ;
```

2.2 Erläuterungen

ax = 0 Wird umgesetzt durch `mov cx, ax` und `jcxz`. Der erste Befehl kopiert den Wert von Register `ax` nach `cx`, der zweite Befehl prüft, ob das Register `cx` 0 ist und springt, sollte dies der Fall sein.

While ax != 0 Wird umgesetzt durch ein Label `loop_in` und `jnz`, welches (hier) prüft, ob der Wert des Registers `ax` 0 ist und genau dann springt, wenn dies nicht der Fall ist. Das nötige Flag für diese Überprüfung (Zero) wird bei `dec` gesetzt.

While ax != 0 Wird umgesetzt durch `cmp ax, 0` bzw. `sub ax, 1` und `jne` bzw. `jnz`. In Variante 1 wird `ax` mit 0 verglichen und gesprungen, sofern die beiden Werte ungleich sind. Variante 2 nutzt das Resultat der `sub`-Operation und springt, sofern das Ergebnis nicht 0 ist, das Zero-Flag also nicht gesetzt wurde.

Do-While AX != 100 Wird umgesetzt durch `sub bx, 100` bzw. `cmp ax, 100` und `jnz` bzw. `jne`. In Variante 1 wird 100 von `bx` (Kopie von `ax`) abgezogen und gesprungen, sofern das Ergebnis nicht 0 ist. In Variante 2 wird `ax` mit 100 verglichen und gesprungen, sofern die Werte ungleich sind.

Fallauswahl cx Wird umgesetzt durch mehrere `cmp`-`je`-Kombinationen.

3 Zähler – Interrupts und Ausgabe

Interrupts unterbrechen die aktuelle Programmausführung und schieben anderen Code zur Ausführung ein. Der Interrupt `15,86` unterbricht das aktuelle Programm und wartet für eine vorgegebene Zeitspanne in Mikrosekunden, die in `cx:dx` übergeben wird.

Der Befehl `out` gibt Speicherinhalte an externe Ports aus, an denen wiederum Peripheriegeräte o. Ä. (z. B. Konsolen oder Displays) angeschlossen sein können. In diesem Beispiel ist an Port 199 eine 7-Segment Anzeige verbunden, welche die übergebenen 2K-Zahlen anzeigt.

Das LED-Display kann, da es einen 2-byte-Wert übergeben bekommt, kann es Zahlen im Bereich von -2^{15} bis $2^{15} - 1$ anzeigen.

3.1 Quellcode

Listing 8: A3.asm

```
1  #start=led_display.exe#
2  name "Aufgabe 3"
3
4
5  ;Hier den eigenen Code einfuegen
6
7  main:             ; begin main loop
8      call step_num
9      call disp_num
10     call wait_sec
11 jmp main          ; end main loop
12
13 step_num:
14     inc ax
15     cmp ax, 21
16     jne return
17     mov ax, 0ffech ; -20 in 2's complement
18
19 return:
20     ret
21
22 disp_num:
23     out 199, ax
24     ret
25
26 wait_sec:
27     push dx        ; buffer register values
28     push cx
29     push ax
30     mov cx, 00fh   ; == 1 sec.
31     mov dx, 04240h ;
32     mov ah, 86     ; prepare interrupt
```

```
33      int 15h
34      pop ax          ; clean up
35      pop cx
36      pop dx
37      ret
38
39  hlt  ;
```

3.2 Erläuterungen

Der `main`-Block ruft nacheinander die jeweiligen Unterroutinen wiederholt auf.

`step_num` inkrementiert das Register `ax` und setzt dieses auf den Wert $ffec_{16}$, was -20 im Zweierkomplement[1] entspricht.

`disp_num` übergibt den Wert des Registers `ax` an Port 199, also unser LED-Display.

`wait_sec` puffert zunächst die Werte der Register `ax`, `cx` und `dx` auf dem Stack, da diese später für den Interrupt benötigt werden. Danach werden in `cx` und `dx` die Parameter für den Interrupt gesetzt (hier: eine Sekunde in Mikrosekunden umgerechnet). Danach wird Interrupt `15,86` aufgerufen, welcher eine Sekunde wartet.

4 BCD Korrektur

Bei der BCD Korrektur muss das Auxiliary-Flag beachtet werden. Dieses wird gesetzt, wenn bei der Berechnung ein Übertrag von der ersten Tetrade zur zweiten Tetrade des bytes stattgefunden hat.

4.1 Quellcode

Listing 9: A4.asm

```
1  name "Aufgabe 4"
2
3
4  MOV AL, 07H
5  MOV BL, 08H
6  ADD AL,BL
7  CALL bcdcorrect
8  push ax
9  ;Testfall 1: Erwartetes Ergebnis in AX: 0015H
10
11 MOV AL, 79H
12 MOV BL, 79H
13 ADD AL,BL
14 CALL bcdcorrect
15 push ax
```

[1] Erläuterung zum Zweierkomplement: Das MSB gibt an, ob die Zahl positiv oder negativ ist. Ist die Zahl positiv, so enthalten die restlichen Bits den Betrag der Zahl. Ist die Zahl negativ, enthalten die restlichen Bits den bitweise invertierten und um 1 erhöhten Betrag der Zahl.

```
16 ;Testfall 2: Erwartetes Ergebnis in AX: 0158H
17
18 MOV AL, 32H
19 MOV BL, 45H
20 ADD AL,BL
21 CALL bcdcorrect
22 push ax
23 ;Testfall 3: Erwartetes Ergebnis in AX: 0077H
24
25 MOV AL, 72H
26 MOV BL, 93H
27 ADD AL,BL
28 CALL bcdcorrect
29 push ax
30 ;Testfall 4: Erwartetes Ergebnis in AX: 0165H
31
32 ; Save results to register for better readability when checking results
33 pop dx ; Ergebnis Testfall 4
34 pop cx ; Ergebnis Testfall 3
35 pop bx ; Ergebnis Testfall 2
36 pop ax ; Ergebnis Testfall 1
37 hlt  ;
38
39
40 bcdcorrect PROC
41     ; This code has been moved to another destination
42     call carry_backup
43
44     ; Hier eigenen Code einfuegen
45     cmp cl, 1
46     jne start_correction
47     mov ah, 1
48
49 start_correction:
50     call check_needs_correction
51     call correct
52 RET                 ;Ende der Funktion
53 bcdcorrect ENDP
54
55 carry_backup:
56     MOV CL, 1H ;Carry Backup nach CL
57     JC saveNC
58     MOV CL, 0H
59     saveNC:
60
61     LAHF      ; Auxillary Flag (AF) nach CH
62     MOV CH, AH
63     AND CH, 10H
64     SHR CH, 4
```

```
65      MOV AH, CL
66      ret
67
68  ; Check for needed corrections
69  check_needs_correction:
70      mov dx, 0
71      call check_upper_pseudo_tetrade
72      jae set_upper_correction
73      cmp cl, 0 ; required carry set?
74      jne set_upper_correction
75  check_lower_needs_correction:
76      call check_lower_pseudo_tetrade
77      jae set_lower_correction
78      cmp ch, 0
79      jne set_lower_correction
80      ret
81
82  check_upper_pseudo_tetrade:
83      mov bh, al
84      shr bh, 4
85      cmp bh, 10
86      ret
87
88  check_lower_pseudo_tetrade:
89      mov bh, al
90      and bh, 00fh
91      cmp bh, 10
92      ret
93
94  set_upper_correction:
95      mov dh, 1
96      jmp check_lower_needs_correction
97
98  set_lower_correction:
99      mov dl, 1
100     ret
101
102 ; Execute correction
103 correct:
104     cmp dx, 0
105     je return
106     cmp dx, 00001h
107     je correct_lower
108     cmp dx, 00100h
109     je correct_upper
110     jmp correct_both
111 set_carry_register_and_recheck:
112     jnc recheck
113     mov ah, 1
```

```
114 recheck:
115     mov cx, 0
116     call check_needs_correction
117     call correct
118     ret
119
120 correct_lower:
121     add al, 006h
122     jmp recheck
123
124 correct_upper:
125     add al, 060h
126     jmp set_carry_register_and_recheck
127
128 correct_both:
129     add al, 066h
130     jmp set_carry_register_and_recheck
131
132 return:
133     ret
```

4.2 Erläuterungen

Die eingeschobenen `push`- und `pop`-Aufrufe am Anfang der Datei dienen der leichteren Überprüfbarkeit der Ergebnisse am Ende der Berechnung. Der Code wurde der einfacheren Lesbarkeit halber in mehrere Unterprozeduren aufgeteilt, welche jedoch größtenteils auch inline geschrieben werden könnten.

`carry_backup` speichert die Werte des Carry-Flags und des Auxiliary-Flags in den Registern `cl` bzw. `ch`.

`check_needs_correction` überprüft, ob die zu korrigierende Zahl überhaupt eine Korrektur benötigt, d. h. ob Pesudotetraden oder Tetradenübergänge aufgetreten sind. Dies wurde jeweils über Bitshifts und Vergleiche realisiert. Zudem sind hier die Werte aus `cl` und `ch` (d. h. die Carry- und Auxiliary-Flags) eingeflossen. Je nachdem, welche Tetraden eine Korrektur benötigen, wird in `dx` das high- und/oder das low-Register gesetzt.

`correct` führt je nachdem, welchen Wert `dx` hat, die eigentliche Korrektur der Tetraden aus. Im Anschluss dazu wird erneut geprüft, ob eine Korrektur nötig ist und gegebenenfalls erneut korrigiert.

5 Vorzeichenbetragszahlen

In einer 8086-CPU werden vorzeichenbehaftete Zahlen im 2K-Format dargestellt, weshalb vor der Rechnung mit VBZ eine Umrechnung nötig ist.

5.1 Quellcode

Listing 10: A5.asm

```
1  include "data5_1.inc" ; Gegeben AX:8024H BX:0137H Erwartetes Ergebnis in AX
       :0113H
2
3  name "Aufgabe 5"
4
5  #start=led_display.exe#
6
7  ;Hier den eigenen Code einfuegen
8  call vbz_to_twok
9  push ax
10 mov ax, bx
11 call vbz_to_twok
12 mov bx, ax
13 pop ax
14 add ax, bx
15 call twok_to_vbz
16
17 ;Resulatat auf das Display ausgeben
18 out 199, ax
19
20 hlt  ;
21
22 vbz_to_twok:
23     mov cx, ax
24     and cx, 08000h ; is the msb set?
25     cmp cx, 0
26     jnz vbz_to_twok_is_negative
27     ; number positive, just return
28     ret
29
30 vbz_to_twok_is_negative:
31     mov dx, ax
32     and dx, 07fffh ; extract the absolute value
33     not dx
34     add dx, 1
35     mov ax, dx
36     ret
37
38 twok_to_vbz:
39     mov cx, ax
40     and cx, 08000h ; is the first bit set?
41     cmp cx, 0
42     jnz twok_to_vbz_is_negative
43     ; number positive, rust return
44     ret
45
46 twok_to_vbz_is_negative:
47     mov dx, ax
48     and dx, 07fffh ; extract the absolute value
```

```
49      not dx
50      add dx, 1
51      or dx, 08000h ; set the msb
52      mov ax, dx
53      ret
```

5.2 Erläuterungen

Die Routine `vbz_to_twok` konvertiert eine VBZ in eine 2K-Zahl, indem sie zunächst prüft, ob die Zahl negativ ist (das MSB gesetzt ist) und, falls dies der Fall ist, die Zahl invertiert und um eins erhöht, was einer Umrechnung in 2K entspricht. Positive Zahlen müssen nicht angepasst werden.

Die Routine `twok_to_vbz` konvertiert eine 2K-Zahl in eine VBZ, indem sie zunächst prüft, ob die Zahl negativ ist (das MSB gesetzt ist) und, falls dies der Fall ist, die Zahl invertiert und um eins erhöht, was einer Rückumrechnung in VBZ entspricht. Positive Zahlen müssen nicht angepasst werden.

6 Potenzfunktion – Speicherzugriff

Für den `mov`-Befehl gibt es verschiedene Adressierungsarten:

Unmittelbare Adressierung Der übergebene Operand ist der Wert, mit dem gearbeitet werden soll.

Direkte Registeradressierung Der übergebene Operand ist ein Register, welches den Wert enthält, mit dem gearbeitet werden soll.

Direkte Speicheradressierung Der übergebene Operand enthält direkt die Adresse der Speicherstelle, die den Wert enthält, mit dem gearbeitet werden soll.

Indirekte Speicheradressierung Der übergebene Operand enthält ein Register, welches die Adresse der Speicherstelle enthält, die den Wert enthält, mit dem gearbeitet werden soll.

Indizierte Adressierung Der übergebene Operand ist ein Indexregister und eine Zahl. Der Wert des Registers wird zur Zahl addiert, um die Adresse der Speicherstelle zu erhalten, die den Wert enthält, mit dem gearbeitet werden soll.

Basisadressierung Der übergebene Operand ist ein Register und eine Zahl. Der Wert des Registers wird zur Zahl addiert, um die Adresse der Speicherstelle zu erhalten, die den Wert enthält, mit dem gearbeitet werden soll.

Basisindizierte Adressierung Der übergebene Operand besteht aus einem Register und einem Indexregister, deren Werte addiert werden, um die Adresse der Speicherstelle zu erhalten, die den Wert enthält, mit dem gearbeitet werden soll.

Die höchste in einer Speicherstelle darstellbare nichtnegative Zahl ist 32767.

6.1 Quellcode

Listing 11: A6.asm

```
1  include "data6_1.inc"
2  name "Aufgabe 6"
3
4  ;Hier den eigenen Code einfuegen
5  mov al, 1
6  mov bx, bp
7  mov cl, [bp]
8
9  next_val:
10     cmp al, 0
11     jbe hlt
12     inc bx
13     mov [bx], al
14     mul cl
15     jmp next_val
16
17 hlt:
18 hlt  ;
```

6.2 Erläuterungen

Die Speicheradresse des nächsten Speichersegments wird im Register `bx` gespeichert, `cl` enthält die Basis, die potenziert werden soll, `al` enthält das aktuelle Zwischenergebnis.

Die Schleife prüft, ob die letzte Berechnung eine Zahl ≤ 0 ergeben hat und bricht ab, falls dies der Fall ist. Sonst wird `bx` inkrementiert, um die nächste zu schreibende Speicheradresse zu erhalten und `ax` nach `[bx]` geschrieben und danach mit `cl` (also der Basis) multipliziert, um die nächste Potenz zu berechnen.

7 Temperatursteuerung – Ein-/Ausgabe

7.1 Quellcode

Listing 12: A7.asm

```
1  #start=thermometer.exe#
2  name "Aufgabe 7"
3
4  ;Hier den eigenen Code einfuegen
5  mov cl, [bp] ; T_min
6  mov ch, [bp + 1] ; T_max
7  ; mov cl, 0
8  ; mov ch, 100
9
10 loop:
11 in al, 125
```

```
12 cmp al, cl
13 jna start
14 cmp al, ch
15 jnb stop
16 jmp loop
17 start:
18 mov al, 1
19 out 127, al
20 jmp loop
21 stop:
22 mov al, 0
23 out 127, al
24 jmp loop
25
26 hlt ;
```

7.2 Erläuterungen

In Dauerschleife liest dieses Programm den Wert von Port 125 aus und prüft, ob dieser unter T_{min} (gespeichert in `cl`) liegt und sendet in diesem Fall eine 1 an Port 127, um den Brenner zu aktivieren. Analog dazu wird sonst geprüft, ob T_{max} (gespeichert in `ch`) überschritten worden ist und deaktiviert in diesem Fall den Brenner.