Ausgabe der Aufgabe: 2. November 2017 **Eingabefrist: 26. November 2017 24:00**
Hausaufgabe 2 können Sie bis zum Ende der Nachholungswoche (15. Dezember 2017, 24:00) mit der Einzahlung des Extraprozessengebühres eingegeben!
Die Aufgabe soll in elektronischer Form, in PDF Format durch https://hf.mit.bme.hu/ Hausaufgabeneingabeportal eingegeben werden.
Registration ist erforderlich! Erreichbare Punktzahl: max 15.

Digitaltechnik Hausaufgabe 2

Maschinenkodenentwurf

Name:	**NEPTUN:**	**Email:**
Bajczi Levente	XAO5ER	levente.bajczi@edu.bme.hu
Lehrkreis:	**Jahrgang:**	**DIGITKODE:**
Deutschkurs	2017	2745136

Die Aufgaben wurden selbständig, ohne unerlaubtes Hilfsmittel und ohne unmittelbare Mitwirkung der Anderen gelöst

(A feladatokat önállóan, meg nem engedett segédeszközök használata és mások közvetlen közreműködése nélkül oldottam meg:)

.....

Unterschrift (Aláírás)

Zusammenfassung der Ergebnisse:

Die Anzahl von 1-er Bits im 56 Bits langen Datenvektor, der sich auf meinem DIGITKODE basiert war: 24.

Die Parameter für die Effizienz waren wie folgt:

Programmversion	Anzahl der Befehle (P)	Benötigter Datenspeicher (D)	Ausführungszeit (I)	Programmaufwand
Herkömmlich	14	0	171	4788
Tabelle	15	16	99	4554
Arithmetisch	24	0	171	8208

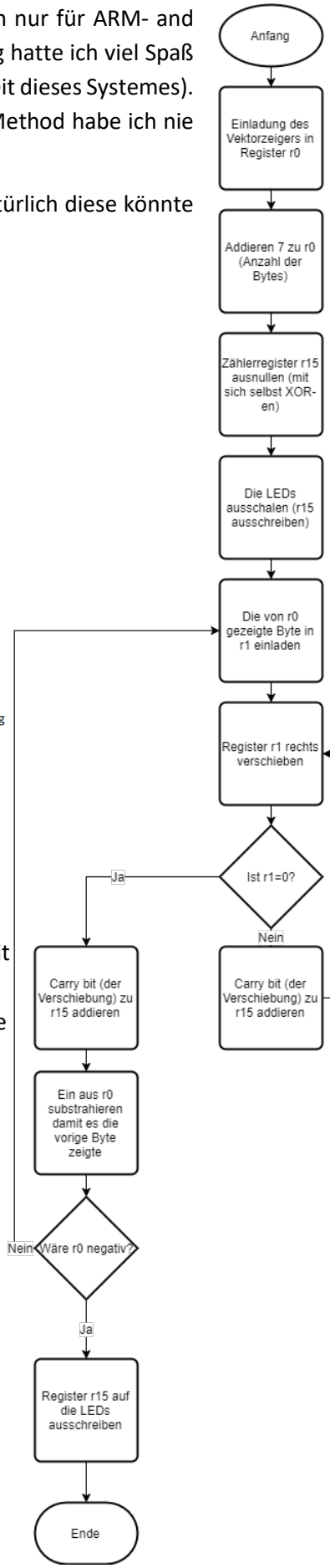
Diese Hausaufgabe war mich sehr interessant zu entwickeln, weil bevor habe ich nur für ARM- and amd64-Systeme in Assembly-Sprache programmiert, und anhand dieser Erfahrung hatte ich viel Spaß beim passende Befehle auszusuchen (Viele Befehle vermisse ich wegen der RISC-heit dieses Systemes). Für mich persönlich war die dritte Lösung der interessanteste, weil über diesen Method habe ich nie gehört.

Als die Tabelle zeigte, ist für meinen Digit-Kode der zweite Lösung am besten. Natürlich diese könnte sich ändern falls ein anderer Kode getestet wäre.

HA2_1

```
S Addr  Instr   Source code
-------------------------------------------------------------------------------------------
                ; Meine Digit-Kode: 2 7 4 5 1 3 6 -> 29 7B 49 56 14 39 60

                DEF LD     0x80                 ; LED adatregiszter          (írható/olvasható)
                DEF ANZAHL 0x06                 ; Hilfe fuer die laenge des Zahlenvektors
                DEF MASK   0x01

                    DATA

D 00            DIGIT_CODE:
D 00                DB      0x29, 0x7B, 0x49, 0x56, 0x14, 0x39, 0x60

                    CODE
C 00            init:

C 00    C000        mov     r0, #DIGIT_CODE[00]     ;die Anfangsadresse der Zahlenmuster in r0 speichern
C 01    0006        add     r0, #ANZAHL[06]         ;Die letzte byte ist unser anfangswert
C 02    FF6F        xor     r15, r15                ;Nullieren die r15 Register, darin zaehlen wir die eine
C 03    9F80        mov     LD[80], r15
C 04            loop:
C 04    F1D0        mov     r1, (r0)                ;Einladung des Bytes
C 05            inner_loop:
C 05    F171        sr0     r1                      ;Rechst verschieben, Carry in Carry Flag speichern
C 06    B109        jz      endinner[09]            ;Falls nur 0 Werte es gibt, sind wir mit diesem Byte fertig
C 07    1F00        adc     r15, #0x00              ;Den carrywert zu r15 addieren
C 08    B005        jmp     inner_loop[05]          ;Zurücktreten zu der Anfang dieser Schleife
C 09            endinner:
C 09    1F00        adc     r15, #0x00              ;Den carrywert zu r15 addieren
C 0A    2001        sub     r0, #0x01               ;ein byte zurück
C 0B    B604        jnn     loop[04]                ;Falls nicht, dann zurück zur außere Schleife
C 0C    B00D        jmp     finished[0D]            ;Falls diese wert "negativ" ist, sind wir fertig

C 0D            finished:
C 0D    9F80        mov     LD[80], r15             ;Die Ergebnis auf dem LEDs darstellen
C 0E    B00D        jmp     finished[0D]            ;die LEDS halten
```

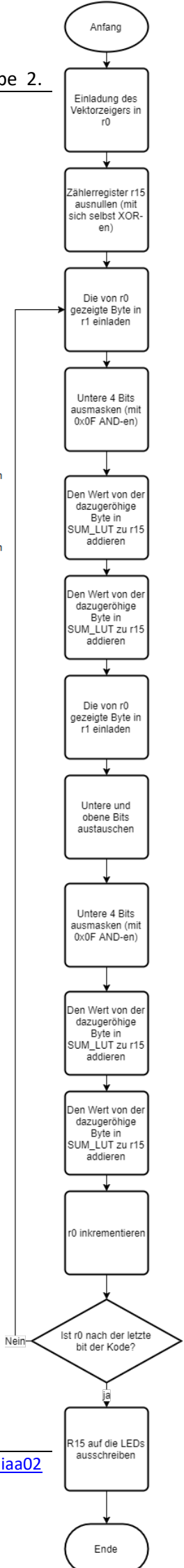
Diese Lösung konzentriert sich auf dem einfachste Methode: wir gehen Bit nach Bit und zählen die 1-Werte. Für diese Method brauchen wir zwei Schleifen: Eine, der Byte nach Byte die Kode einladen lässt, und eine andere, die Jede Byte für 1-Werte testet. Die Implementation ist oben, die ASM ist nach rechts zu sehen.



HA2_2

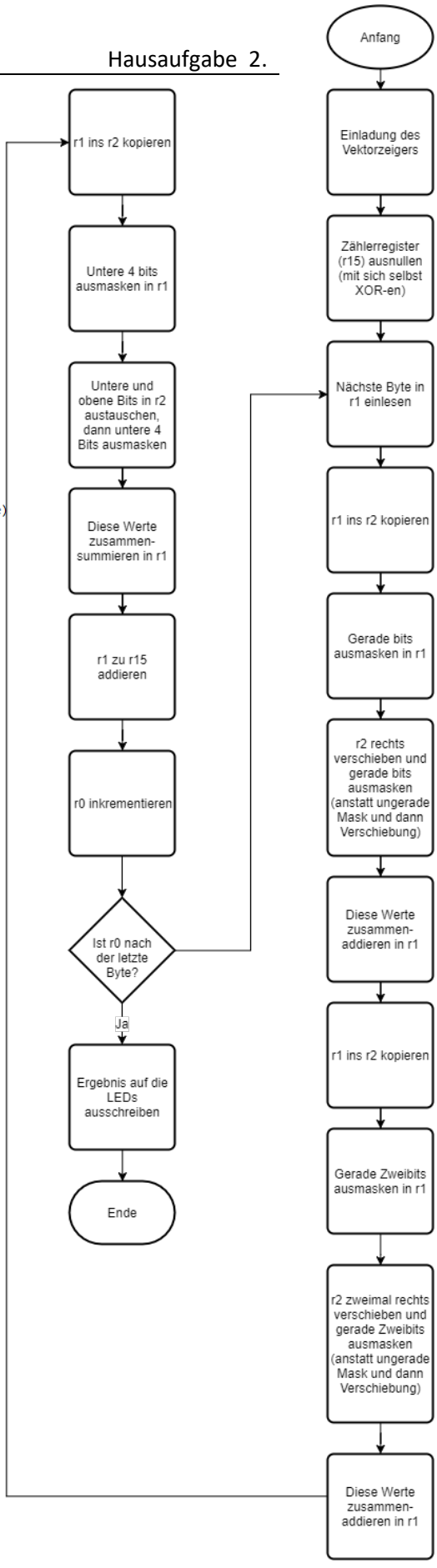
```
S Addr  Instr   Source code
------------------------------------------------------------------------------------------
                DEF NACHLETZTE 0x28
                DEF BYTEMASK   0x0F
                DEF LD         0x80

                    DATA

D 00            SUM_LUT:
D 00                DB      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03, 0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04
                    ORG     0x20
D 20            DIGIT_CODE:
D 20                DB      0x29, 0x7B, 0x49, 0x56, 0x14, 0x39, 0x60

                    CODE
C 00            start:
C 00    C020        mov     r0, #DIGIT_CODE[20]     ;Einladung von der Anfangsadresse
C 01    FF6F        xor     r15, r15                ;Den Register, den wir als Zaehler benutzen werden, ausnullen
C 02            loop:
C 02    F1D0        mov     r1, (r0)                ;Dateneinladung
C 03    410F        and     r1, #BYTEMASK[0F]       ;untere 4 bits ausmasken
C 04    FED1        mov     r14, (r1)               ;basisadresse ist 0, also die Offset allein koennte als Adresse benutzt werden
C 05    FF0E        add     r15, r14
C 06    F1D0        mov     r1, (r0)                ;Dateneinladung noch einmal
C 07    7100        swp     r1                      ;Untere und obene Bits austauschen
C 08    410F        and     r1, #BYTEMASK[0F]       ;untere 4 bits ausmasken
C 09    FED1        mov     r14, (r1)               ;basisadresse ist 0, also die Offset allein koennte als Adresse benutzt werden
C 0A    FF0E        add     r15, r14
C 0B    0001        add     r0, #0x01               ;Adresse inkrementieren
C 0C    A028        cmp     r0, #NACHLETZTE[28]     ;Falls sie die gleiche Wert betragen, sind wir fertig
C 0D    B202        jnz     loop[02]
C 0E            endloop:
C 0E    9F80        mov     LD[80], r15             ;Ergebnis auf der LEDs darstellen
C 0F    B00E        jmp     endloop[0E]
```

Diese Lösung konzentriert sich über die leichste (für die CPU) Methode. Wir speichern eine Tabelle, mit Hilfe von wem wir jede 4-Bit Werte testen, wie viele 1-Werte sie enthalten.
Diese Look-up-table enthielt 16 bytes, und fängt beim 0x00 Addresse an, damit jede 4-Bit Wert als Offset benutzt werden kann, um die 1-Wert-Anzahl zu bekommen. Das heißt, dass zum Beispiel Wert 0b0011 (=3_{10}) hat 2 1-Werte, also beim Addresse 0x03 hat SUM_LUT die Wert 2.
Für meinen Digit-kode war dieser Method am besten.
Der Implementation ist oben, die ASM ist rechts zu sehen.



HA2_3

```
5 Addr  Instr   Source code
--------------------------------------------------------------------------------
                DEF LD          0x80                ; LED adatregiszter (írható/olvasható)
                DEF GERADE1     0x55                ;MASKE
                DEF GERADE2     0x33                ;
                DEF GERADE4     0x0F                ;
                DEF NACHLETZTE 0x08                 ;Erste adresse, die wir nicht benutzen

                    DATA
0 00            DIGIT_CODE:
0 00                DB      0x29, 0x7B, 0x49, 0x56, 0x14, 0x39, 0x60


                    CODE
0 00            start:
0 00    C000        mov     r0, #DIGIT_CODE[00]     ;Addresse einlesen
0 01    FF6F        xor     r15, r15                ;Ergebnisregister ausnullen

0 02            loop:
0 02    F1D0        mov     r1, (r0)                ;Erste Byte einlesen
0 03    F2C1        mov     r2, r1                  ;Kopieren
0 04    4155        and     r1, #GERADE1[55]        ;Gerade Bits ausmasken (0-te, 2-te, 4-te, 6-te)
0 05    F271        sr0     r2                      ;Rechts verschieben, um gleiche Maske benutzen zu koennen
0 06    4255        and     r2, #GERADE1[55]        ;jetzt gerade Bits ausmasken (vorherig 1-te, 3-te, 5-te, 7-te)
0 07    F102        add     r1, r2                  ;Zusammensummieren
0 08    F2C1        mov     r2, r1                  ;Kopieren
0 09    4133        and     r1, #GERADE2[33]        ;Gerade Zweibits ausmasken (0-1, 4-5)
0 0A    F271        sr0     r2
0 0B    F271        sr0     r2                      ;Zwei Stellen nach rechts verschieben
0 0C    4233        and     r2, #GERADE2[33]        ;jetzt gerade Zweibits ausmasken (vorherig 2-3, 6-7)
0 0D    F102        add     r1, r2                  ;Summieren
0 0E    F2C1        mov     r2, r1                  ;Kopieren
0 0F    410F        and     r1, #GERADE4[0F]        ;Untere 4 bits ausmasken
0 10    7200        swp     r2                      ;Obene und untere Bits austauschen
0 11    420F        and     r2, #GERADE4[0F]        ;jetzt untere 4 bits ausmasken (vorherig obene 4)
0 12    F102        add     r1, r2                  ;Zusammensummieren
0 13    FF01        add     r15, r1                 ;Ergebnis speichern
0 14    0001        add     r0, #0x01               ;Addressenzeiger erhöhen
0 15    A008        cmp     r0, #NACHLETZTE[08]     ;Falls wir fertig sind, sind sie gleich
0 16    B202        jnz     loop[02]

0 17            endloop:
0 17    9F80        mov     LD[80], r15             ;Ergebnis auf die LEDs darstellen
0 18    B017        jmp     endloop[17]
```



Diese Lösung benutzt eine sehr interessante arithmetische methode: Die Anzahl von 1-Werte könnte man mit Maskierungen und Verschiebungen bekommen. In dieser Fall war es am schlechsten, weil zu viele Berechnungen müssen ausgeführt werden.

Die Implementation ist oben, die ASM ist rechts zu sehen.