Informatik 3 – Übung 07 – Georg Kuschk

7.1) CRC-Verfahren

7.1 a)

Für die Übertragung der Nachricht mittels des Polynoms T(x) werden 40 Bit benötigt.

32 Bit für die eigentliche Nachricht M(x) und 8 Bit für den Divisionsrest, der sich aus der Polynomdivision von $M(x) \cdot x^8$ durch C(x) ergibt.

7.1 b)

M(x)=1000 1101 1001 0010 0000 0000 1011 1111
C(x)= 100000111

T(x) berechnen :

1. Das Polynom, welches der Bitfolge M(x) entspricht mit x^8 multiplizieren
   (da Grad des Generatorpolynoms C(x) = 8) , d.h. 8 Nullen anhängen
2. Das Ergebnis E durch C(x) teilen, und den Rest R bestimmen
3. Addition von E und R ergibt die gesuchte Nachricht T(x)

(um die Rechnung einigermaßen übersichtlich zu gestalten, wurden die 4Bit-Blöcke beibehalten)

```
E =  1000 1101 1001 0010 0000 0000 1011 1111 0000 0000
     1000 0011 1
     0000 1110 0001 0
          1000 0011 1
          0110 0010 10
           100 0001 11
           010 0011 011
            10 0000 111
            00 0011 1000 000
                 10 0000 111
                 01 1000 1110
                  1 0000 0111
                  0 1000 1001 0
                    1000 0011 1
                    0000 1010 1000 1
                         1000 0011 1
                         0010 1011 001
                            10 0000 111
                            00 1011 1101 1
                               1000 0011 1
                               0011 1110 011
                                  10 0000 111
                                  01 1110 1001
                                   1 0000 0111
                                   0 1110 1110 0
                                     1000 0011 1
                                     0110 1101 10
                                      100 0001 11
                                      010 1100 010
                                       10 0000 111
                                       00 1100 1010 0
                                          1000 0011 1
                                          0100 1001 10
                                           100 0001 11
                                           000 1000 0100 = R
```

```
  E =  1000 1101 1001 0010 0000 0000 1011 1111 0000 0000
+ R =  0000 0000 0000 0000 0000 0000 0000 0000 1000 0100
T(x) = 1000 1101 1001 0010 0000 0000 1011 1111 1000 0100
```

7.1 c)
Fehler konstruieren, der nicht erkannt wird:

- Das erzeugte Codewort entspricht einem Vielfachen des Generatorpolynoms.

- Bei der Fehlererkennung wird nun das dem empfangenen Wort entsprechende Polynom durch das Generatorpolynom geteilt. Ist der Rest ungleich Null, so ist ein Fehler aufgetreten.

- Beim Auftreten eines Fehlers wurden in dem gesendeten Wort ein oder mehrere Bits invertiert. Wird dieses Wort als Polynom aufgefasst, dann entspricht das Invertieren eines Bits der Addition eines Fehlerpolynoms, welches eine 1 an dieser Position hat.

- Wenn das Fehlerpolynom ungleich Null ist und durch das Generatorpolynom teilbar ist, wird der Fehler also nicht erkannt.
  (Umgekehrt wird der Fehler erkannt, wenn das Fehlerpolynom nicht durch das Generatorpolynom teilbar ist.)

- Im gegebenen Beispiel ist der einfachste Fall eines solchen, nicht erkennbaren Fehlerpolynoms das Generatorpolynom C(x) selbst.

Die korrekt übertragene Nachricht war
$T(x)$ = 1000 1101 1001 0010 0000 0000 1011 1111 1000 0100

Die fehlerhaft übertragene Nachricht mit dem Fehlerpolynom C(x) ist
$\overline{T}(x)$ = 1000 1101 1001 0010 0000 0000 1011 111*0* 1000 0*011*
(Die invertierten Bits sind kursiv dargestellt.)

D.h. die fehlerhafte Bitfolge $\overline{M}(x)$, die vom Empfänger aus der als fehlerfrei erkannten Nachricht $\overline{T}(x)$ gewonnen wurde, lautet :
$\overline{M}(x)$ = 1000 1101 1001 0010 0000 0000 1011 1110
und unterscheidet sich von der ursprünglichen Nachricht
$M(x)$ = 1000 1101 1001 0010 0000 0000 1011 1111
somit im letzten Bit.

zu 7.1 c)

Zur Sicherheit noch überprüfen, ob die Polynomdivision $\overline{T}(x)$ durch C(x) wirklich den Rest Null hat :

1000 1101 1001 0010 0000 0000 1011 1110 1000 0011
1000 0011 1
0000 1110 0001 0
1000 0011 1
0110 0010 10
100 0001 11
010 0011 011
10 0000 111
00 0011 1000 000
10 0000 111
01 1000 1110
1 0000 0111
0 1000 1001 0
1000 0011 1
0000 1010 1000 1
1000 0011 1
0010 1011 001
10 0000 111
00 1011 1101 1
1000 0011 1
0011 1110 011
10 0000 111
01 1110 1000
1 0000 0111
0 1110 1111 1
1000 0011 1
0110 1100 00
100 0001 11
010 1101 110
10 0000 111
00 1101 0010 0
1000 0011 1
0101 0001 10
100 0001 11
001 0000 0111
1 0000 0111
0 0000 0000 = Rest

7.2) Huffman-Code

7.1 a) Zeigen : Huffman-Code ist Präfixcode

Da im Huffman-Codebaum notwendigerweise kein Pfad von der Wurzel zu einem Blatt Anfangsteil eines anderen Pfades ist, handelt es sich um einen Präfixcode.

7.1 b) Zeigen : Trennsymbole zwischen den Codewörtern werden nicht benötigt

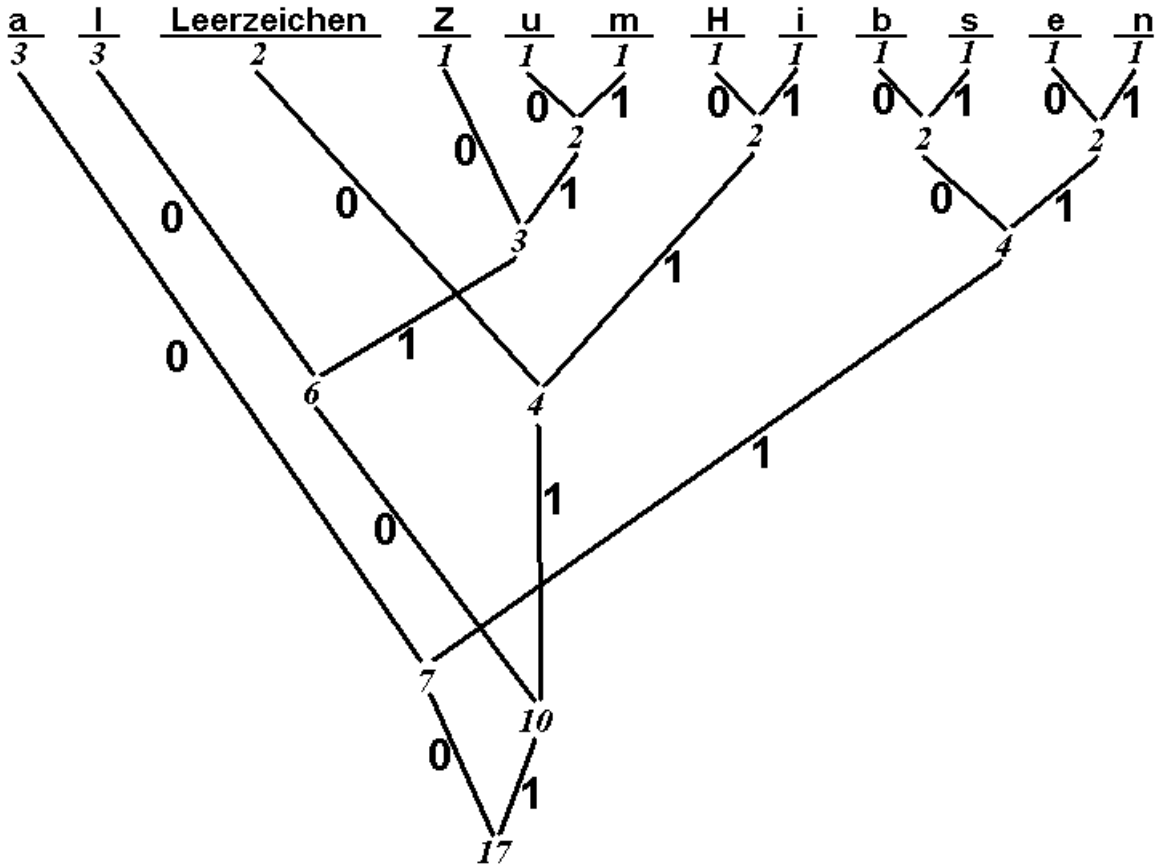
Dies wird anhand der Decode-Funktion gezeigt, welche die Bitfolge anhand des Huffman-Codebaums zurückübersetzt.

Sie startet in der Wurzel und wandert anhand der Bitfolge in dem Baum bis zu einem Blatt, liest dort das entsprechende Zeichen und fährt wieder bei der Wurzel beginnend fort, bis die Bitfolge komplett ausgelesen wurde.
Wenn sie ein komplettes Codewort erkannt hat, befindet sie sich also in einem Blatt und „weiss" somit, dass mit dem nächsten Bit aus der Bitfolge ein neues Codewort beginnt.
Trennsymbole sind also überflüssig.

7.2 c)

Zeichenfolge : Zum Halali blasen
Anzahl Zeichen : 17

Codebaum :



Zeichen	a	l	Leerzeichen	Z	u	m	H	i	b	s	e	n
Häufigkeit	3	3	2	1	1	1	1	1	1	1	1	1
Code	00	100	110	1010	10110	10111	1110	1111	0100	0101	0110	0111

Die Zeichenfolge "Zum Halali blasen" wird also wie folgt codiert :
1010 10110 10111 110 1110 00 100 00 100 1111 110 0100 100 00 0101 0110 0111

(Die Leerstellen zwischen den Bits sind nur der Übersichtlichkeit eingefügt -
der reine Code ist 10101011010111110111000100001001111110010010000010101100111)

Gesamtcodewortlänge = 59 Bit

7.3) MIPS

```
#Aufgabe 7.3)
#Georg Kuschk

#In der Datei befinden sich wie gefordert NUR die Unterprogramme, sonst NICHTS.

.data

str_txt1 :   .asciiz "-"

.text


#################################################################################################################
###### Interpretiere $a0 als vorzeichenlose Zahl - Ausgabe als Dezimalzahl (ohne fuehrende Nullen)
out_dez:
        move    $t0,$sp                 #Stackpointer in $t0 retten

        li      $t1,0                   #Indikator um fuehrende Nullen zu entfernen
        li      $t2,1000000000          #hoechste 10er Potenz die in einer 32Bit Zahl auftreten kann (1 Mrd)

loop_out_dez:
        div     $t3,$a0,$t2             #Zahl durch die aktuelle 10er Potenz dividieren


        sne     $t4,$t3,$zero           #ist die Ziffer ungleich Null, dann setze $t4 auf 1
        or      $t1,$t1,$t4             #OR Verknuepfung mit dem Indikator fuer fuehrende Nullen
                                        #zeigt an, ob schon einmal eine Ziffern ungleich Null ausgegeben wurde

        seq     $t4,$t2,1               #wird allerdings gerade die niedrigste 10er Potenz abgearbeitet,
        or      $t1,$t1,$t4             #dann wird Ausgabe der Ziffer erzwungen

        ###### Ausgabe der Ziffer, falls $t1 ungleich Null
        beq     $t1,$zero,leading_zero  #fuehrende Nullen nicht ausgeben


        li      $v0,01                  #print_int
        add     $t4,$a0,0               #$a0 = auszugebende Zahl in $t4 zwischenspeichern
        add     $a0,$t3,0               #10er Stelle laden
        syscall                         #Ziffer ausgeben
        add     $a0,$t4,0               #zwischengespeicherte Zahl wieder nach $a0 schreiben

        beq     $t2,1,exit_out_dez      #wenn die niedrigste 10er Potenz abgearbeitet wurde, Funktion beenden

leading_zero:
        rem     $a0,$a0,$t2             #die restliche auszugebende Zahl ist "Zahl modulo aktueller 10er-Potenz"
        div     $t2,$t2,10              #eine 10er Potenz tiefer gehen
        j       loop_out_dez
exit_out_dez:
        move    $sp,$t0                 #Stackpointer rest aurieren
        jr      $ra                     #return to caller
#################################################################################################################
```

Zu 7.3)

```
#################################################################################################################################
###### Interpretiere $a0 als Einerkomplement
###### Ausgabe als vorzeichenbehaftete Dezimalzahl (ohne fuehrende Nullen)
out_k1:
        move    $t0,$sp                 #Stackpointer in $t0 retten

        ##### Prüfen, ob die Zahl negativ ist
        srl     $t1,$a0,31              #dazu feststellen, ob das 31. Bit gesetzt ist
        beq     $t1,$zero,positiv_k1    #wenn nicht, als positive Zahl behandeln

        ##### Falls die Zahl negativ ist, zuerst ein "-" ausgeben :
        li      $v0,04                  #print_string "-"
        move    $t2,$a0                 #$a0 in $t4 retten
        la      $a0,str_txt1
        syscall
        move    $a0,$t2                 #$a0 restaurieren

        ##### Dann die Zahl invertieren -> dies ergibt die entsprechende positive Zahl (den Betrag also)
        xor     $a0,$a0,4294967295      #die Zahl invertieren (  $a0 XOR ((2^32)-1)  )
        ##### und mit der Ausgabe des Betrages fortfahren :
positiv_k1:
        li      $t1,0                   #Indikator um fuehrende Nullen zu entfernen
        li      $t2,1000000000          #hoechste 10er Potenz die in einer 32Bit Zahl auftreten kann (1 Mrd)

loop_out_k1:
        div     $t3,$a0,$t2             #Zahl durch die aktuelle 10er Potenz dividieren


        sne     $t4,$t3,$zero           #ist die Ziffer ungleich Null, dann setze $t4 auf 1
        or      $t1,$t1,$t4             #OR Verknuepfung mit dem Indikator fuer fuehrende Nullen
                                        #zeigt an, ob schon einmal eine Ziffern ungleich Null ausgegeben wurde

        seq     $t4,$t2,1               #wird allerdings gerade die niedrigste 10er Potenz abgearbeitet,
        or      $t1,$t1,$t4             #dann wird Ausgabe der Ziffer erzwungen

        ###### Ausgabe der Ziffer, falls $t1 un gleich Null
        beq     $t1,$zero,leading_zero_k1       #fuehrende Nullen nicht ausgeben


        li      $v0,01                  #print_int
        add     $t4,$a0,0               #$a0 = auszugebende Zahl in $t4 zwischenspeichern
        add     $a0,$t3,0               #10er Stelle laden
        syscall                         #Ziffer ausgeben
        add     $a0,$t4,0               #zwischengespeicherte Zahl wieder nach $a0 schreiben

        beq     $t2,1,exit_out_k1       #wenn die niedrigste 10er Potenz abgearbeitet wurde, Funktion beenden

leading_zero_k1:
        rem     $a0,$a0,$t2             #die restliche auszugebende Zahl ist "Zahl modulo aktueller 10er-Potenz"
        div     $t2,$t2,10              #eine 10er Potenz tiefer gehen
        j       loop_out_k1
exit_out_k1:
        move    $sp,$t0                 #Stackpointer restaurieren
        jr      $ra                     #return to caller
#################################################################################################################################
```

Zu 7.3)

```
#########################################################################################################################
###### Interpretiere $a0 als Zweierkomplement
###### Ausgabe als vorzeichenbehaftete Dezimalzahl (ohne fuehrende Nullen)
out_k2:
        move    $t0,$sp                 #Stackpointer in $t0 retten

        ##### Prüfen, ob die Zahl negativ ist
        srl     $t1,$a0,31              #dazu feststellen, ob das 31. Bit gesetzt ist
        beq     $t1,$zero,positiv_k2    #wenn nicht, als positive Zahl behandeln

        ##### Falls die Zahl negativ ist, zuerst ein "-" ausgeben :
        li      $v0,04                  #print_string "-"
        move    $t2,$a0                 #$a0 in $t4 retten
        la      $a0,str_txt1
        syscall
        move    $a0,$t2                 #$a0 restaurieren

        ##### Dann eine 1 subtrahieren und das Ergebnis invertieren
        ##### -> dies ergibt die entsprechende positive Zahl (den Betrag also)
        sub     $a0,$a0,1
        xor     $a0,$a0,4294967295      #die Zahl invertieren (  $a0 XOR ((2^32)-1)  )
        ##### und mit der Ausgabe des Betrages fortfahren :

positiv_k2:
        li      $t1,0                   #Indikator um fuehrende Nullen zu entfernen
        li      $t2,1000000000          #hoechste 10er Potenz die in einer 32Bit Zahl auftreten kann (1 Mrd)

loop_out_k2:
        div     $t3,$a0,$t2             #Zahl durch die aktuelle 10er Potenz dividieren


        sne     $t4,$t3,$zero           #ist die Ziffer ungleich Null, dann setze $t4 auf 1
        or      $t1,$t1,$t4             #OR Verknuepfung mit dem Indikator fuer fuehrende Nullen
                                        #zeigt an, ob schon einmal eine Ziffern ungleich Null ausgegeben wurde

        seq     $t4,$t2,1               #wird allerdings gerade die niedrigste 10er Potenz abgearbeitet,
        or      $t1,$t1,$t4             #dann wird Ausgabe der Ziffer erzwungen

        ###### Ausgabe der Ziffer, falls $t1 ungleich Null
        beq     $t1,$zero,leading_zero_k2       #fuehrende Nullen nicht ausgeben


        li      $v0,01                  #print_int
        add     $t4,$a0,0               #$a0 = auszugebende Zahl in $t4 zwischenspeichern
        add     $a0,$t3,0               #10er Stelle laden
        syscall                         #Ziffer ausgeben
        add     $a0,$t4,0               #zwischengespeicherte Zahl wieder nach $a0 schreiben

        beq     $t2,1,exit_out_k2       #wenn die niedrigste 10er Potenz abgearbeitet wurde, Funktion beenden

leading_zero_k2:
        rem     $a0,$a0,$t2             #die restliche auszugebende Zahl ist "Zahl modulo aktueller 10er-Potenz"
        div     $t2,$t2,10              #eine 10er Potenz tiefer gehen
        j       loop_out_k2
exit_out_k2:
        move    $sp,$t0                 #Stackpointer restaurieren
        jr      $ra                     #return to caller
#########################################################################################################################
```

Zu 7.3)

```
#########################################################################################################################
###### $a0 als Binärzahl-Zeichenkette ausgeben (mit fuehrenden Nullen)
###### Anmerkung : die Code-Zeilen, welche die fuehrenden Nullen streichen, wurden auskommentiert
out_bin:
        move    $t0,$sp                 #Stackpointer in $t0 retten

#       li      $t1,0                   #Indikator um fuehrende Nullen zu entfernen
        li      $t2,31                  #zu bearbeitende 2er Potenz initialisieren (=maximale 2er Potenz)

loop_out_bin:

        srl     $t3,$a0,$t2             #Zahl durch die aktuelle 2er Potenz dividieren

#       sne     $t4,$t3,$zero           #ist das Bit ungleich Null, dann setze $t4 auf 1
#       or      $t1,$t1,$t4             #OR Verknuepfung mit dem Indikator fuer fuehrende Nullen
#                                       #zeigt an, ob schon einmal ein Bit ungleich Null ausgegeben wurde
#
#       seq     $t4,$t2,0               #wird allerdings gerade 2^0 abgearbeitet,
#       or      $t1,$t1,$t4             #dann wird Ausgabe des Bits erzwungen
#
#       ###### Ausgabe der Ziffer, falls $t1 ungleich Null
#       beq     $t1,$zero,leading_zero_bin      #fuehrende Nullen nicht ausgeben


        li      $v0,01                  #print_int
        add     $t4,$a0,0               #$a0 = auszugebende Zahl in $t4 zwischenspeichern
        add     $a0,$t3,0               #aktuell auszugebendes Bit laden
        syscall                         #Bit ausgeben
        add     $a0,$t4,0               #zwischengespeicherte Zahl wieder nach $a0 schreiben

        beq     $t2,0,exit_out_bin      #wenn 2^0 abgearbeitet wurde, Funktion beenden

#leading_zero_bin:
        li      $t4,1
        sll     $t4,$t4,$t2
        rem     $a0,$a0,$t4             #die restliche auszugebende Zahl ist "Zahl modulo aktueller 2er-Potenz"


        sub     $t2,$t2,1               #eine 2er Potenz tiefer gehen
        j       loop_out_bin
exit_out_bin:
        move    $sp,$t0                 #Stackpointer restaurieren
        jr      $ra                     #return to caller
#########################################################################################################################
```