Test und Verlässlichkeit
Grosse Übung 3

Prof. G. Kemnitz

Institut für Informatik, Technische Universität Clausthal

11. Juni 2015

Fehlererkennende Codes

Aufgabe 3.1: Arithmetischer Code

- Bilden Sie für den Bitvektor

  $$1100100010000011101$$

  das fehlererkennende Codewort durch Multiplikation seines Wertes als vorzeichenfrei ganze Binärzahl mit der Primzahl 10313 (Bestimmung des Dezimalwerts, Multiplikation und Konvertierung des Produkts in einen Binärvektor).
- Mit welcher Wahrscheinlichkeit werden mit dem gewählten fehlererkennenden Code Datenverfälschungen erkannt?
- Werden mit dem gewählten Code Verfälschung erkannt, die die Bitstellen 3 bis 14 invertieren?

Hinweis: Der Code ist linear, so dass das Erkennen eines verfälschten Codeworts nur von der Differenz und nicht vom codierten Wert abhängt.

Lösung Aufgabe 3.1: Arithmetischer Code

- Codewort berechnen:
  Eingabewert hexadezimal:

$$11.0010.0010.0001.1101 = 0\text{x}3221\text{D}$$

  Mit Octave (Matlab) Produkt als hexadezimal:

```
octave:3> printf('CW=%x\n',0x3221D*10313)
CW=7e394245
```

  Binärwert: 111.1110.0011.1001.0100.0010.0100.0101

- Erkennungswahrscheinlichkeit:

$$p_{\text{E}} \approx 1 - \frac{1}{10313} = 99{,}990\%$$

- Werden Verfälschung erkannt, die die Bitstellen 3 bis 14 invertieren?

gültiges CW	111.1110.0011.1001.0100.0010.0100.0101
verfälschtes CW	111.1110.0011.1001.0000.0010.0100.1101

Divisionsrest mit Octave (Matlab):

```
printf('Div.-Rest=%x\n',mod(0x7E39024D,10313));
Div.-Rest=109a
```

Verfälschung erkennbar.

Aufgabe 3.2: Prüfsummen

Bilden Sie für die Bytefolge

```
0x13, 0xF2, 0x33, 0xE6, 0x8A, 0x3D, 0x30, 0x51
```

die Prüfsumme:

- durch byteweise Aufsummieren unter Vernachlässigung der Überträge und
- durch bitweise EXOR-Verknüpfung der Bytes.
- Welche der beiden Prüfsummen erkennt, dass die nachfolgenden Datenfolgen verfälscht sind?

```
F1: 0x13, 0x33, 0xF2, 0xE6, 0x8A, 0x3D, 0x30, 0x51
F2: 0x13, 0xF2, 0x35, 0xE2, 0x8A, 0x3D, 0x30, 0x51
```

- Prüfsummenberechnung mit Matlab

```
dat_ok = [0x13, 0xF2, 0x33, 0xE6, 0x8A, 0x3D, 0x30, 0x51];
dat_F1 = [0x13, 0x33, 0xF2, 0xE6, 0x8A, 0x3D, 0x30, 0x51];
dat_F2 = [0x13, 0xF2, 0x35, 0xE2, 0x8A, 0x3D, 0x30, 0x51];
function pkz = PKZ_Add(dat)
 pkz = 0;
 for i = 1:length(dat)
   pkz = pkz + dat(i);
 end
 pkz = bitand(pkz, 0xFF);
end
printf('pkz(dat_ok) = %x\n',PKZ_Add(dat_ok));
printf('pkz(dat_F1) = %x\n',PKZ_Add(dat_F1));
printf('pkz(dat_F2) = %x\n',PKZ_Add(dat_F2));
```

- Ergebnis:

```
pkz(dat_ok) = 66
pkz(dat_F1) = 66 nicht erkennbar (Wert 2 und 3 vertauscht)
pkz(dat_F2) = 64 erkennbar
```

- Dasselbe für »bitweises EXOR«

```
dat_ok = [0x13, 0xF2, 0x33, 0xE6, 0x8A, 0x3D, 0x30, 0x51];
dat_F1 = [0x13, 0x33, 0xF2, 0xE6, 0x8A, 0x3D, 0x30, 0x51];
dat_F2 = [0x13, 0xF2, 0x35, 0xE2, 0x8A, 0x3D, 0x30, 0x51];
function pkz = PKZ_Add(dat)
 pkz = 0;
 for i = 1:length(dat)
   pkz = bitxor(pkz, dat(i));
 end
 pkz = bitand(pkz, 0xFF);
end
printf('pkz(dat_ok) = %x\n',PKZ_Add(dat_ok));
printf('pkz(dat_F1) = %x\n',PKZ_Add(dat_F1));
printf('pkz(dat_F2) = %x\n',PKZ_Add(dat_F2));
```

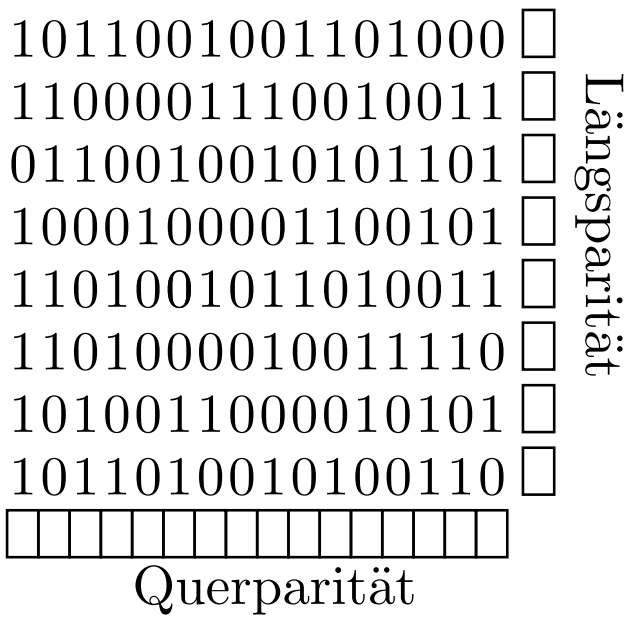
- Ergebnis:

```
pkz(dat_ok) = e2
pkz(dat_F1) = e2 nicht erkennbar (Wert 2 und 3 vertauscht)
pkz(dat_F2) = e0 erkennbar
```

Kreuzparität

Aufgabe 3.3: Berechnung der Kreuzparität

Ergänzen Sie die Bitwerte für die Längs- und Querparität so, dass die Anzahl der Einsen in jeder Zeile und Spalte incl. Paritätsbit gerade ist.



(nur Tafel)

Aufgabe 3.4: Korrektur mit Kreuzparität

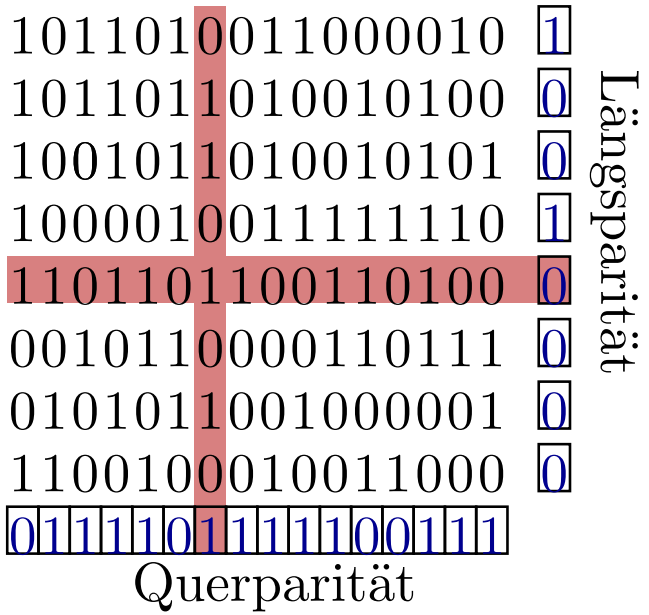
Kontrollieren Sie für die nachfolgenden Bitfelder mit Kreuzparität, ob eine erkennbare oder eine erkenn- und korrigierbare Verfälschung vorliegt und führen Sie, wenn möglich, die Korrektur durch.

1011010011000010	1
1011011010010100	0
1001011010010101	0
1000010011111110	1
1101101100110100	0
0010110000110111	0
0101011001000001	0
1100100010011000	0
0 1 1 1 1 0 1 1 1 1 1 0 0 1 1 1	

Längsparität

Querparität

Lösung »Korrektur mit Kreuzparität«



Korrektur: Bit in Zeile 5, Spalte 6 invertieren (null setzen).

Hamming-Code

Aufgabe 3.1: Hamming-Code

- Bilden Sie für den (8,12)-Hamming-Code

$$
\begin{aligned}
q_0 &= x_0 \oplus x_1 \oplus x_3 \oplus x_4 \oplus x_6 \\
q_1 &= x_0 \oplus x_2 \oplus x_3 \oplus x_5 \oplus x_6 \\
q_2 &= x_1 \oplus x_2 \oplus x_3 \oplus x_7 \\
q_3 &= x_4 \oplus x_5 \oplus x_6 \oplus x_7
\end{aligned}
$$

  die Codeworte für die darzustellenden Werte: 0x73, 0x1D und 0xD6.
- Handelt es sich bei den Codeworten 0xA24, 0x5D6 und 0x141 um zulässige Codeworte, Codeworte mit korrigierbaren oder Codeworte mit erkenn- aber nicht korrigierbaren Verfälschungen? Wenn sie korrigierbar sind, wie lauten die zugeordneten korrekten Werte?

- Bitzuordnung mit Beispiel:

b_{12}	b_{11}	b_{10}	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
x_7	x_6	x_5	x_4	q_3	x_3	x_2	x_1	q_2	x_0	q_1	q_0
1	0	0	1	1	0	1	1	1	0	0	0

- Coder als C-Funktion

```
unsigned short Hamming_8_12(unsigned char x){
  unsigned char x0 = (x>>0) & 1;
  unsigned char x1 = (x>>1) & 1;
  ...
  unsigned char x7 = (x>>7) & 1;
  unsigned char q0 = x0 ^ x1 ^ x3 ^ x4 ^ x6;
  unsigned char q1 = x0 ^ x2 ^ x3 ^ x5 ^ x6;
  unsigned char q2 = x1 ^ x2 ^ x3 ^ x7;
  unsigned char q3 = x4 ^ x5 ^ x6 ^ x7;
  unsigned short y = q0 | (q1<<1) | (x0<<2) | (q2<<3);
  y |=  ((x & 0xE)<<3) | (q3<<7) | ((x & 0xF0) << 4);
  return y;
}
```

Bestimmung des verfälschten Bits

```
unsigned char fBit_H8_12(unsigned short x){
 unsigned char dq0, dq1, dq2, dq3;
 dq0 = 1&((x>>0)^x>>2)^(x>>4)^(x>>6)^(x>>8)^(x>>10));
 dq1 = 1&((x>>1)^(x>>2)^(x>>5)^(x>>6)^(x>>9)^(x>>10));
 dq2 = 1&((x>>3)^(x>>4)^(x>>5)^(x>>6)^(x>>11));
 dq3 = 1&((x>>7)^(x>>8)^(x>>9)^(x>>10)^(x>>11));
 return dq0 | (dq1<<1) | (dq2<<2) | (dq3<<3));
}
```

- Für das Beispiel aus der Tabelle:

```
int main(){
   unsigned char x; unsigned short y;
   x = 0x96; y = Hamming_8_12(x);
   printf("CW: 0x%x => HCW: 0x%x;  ", x, y);
}  printf("dq=%x\n", fBit_H8_12(y ^ (1<<2)));
```

- Ergebnis: CW: 0x96 => HCW: 0x93a; dq=3

Lösung der Aufgabe

- Darzustellende Werte für : 0x73, 0x1D und 0xD6:

```
int main(){
  printf("CW:0x73 => HCW:0x%x\n", Hamming_8_12(0x73));
  printf("CW:0x1D => HCW:0x%x\n", Hamming_8_12(0x1D));
  printf("CW:0xD6 => HCW:0x%x\n", Hamming_8_12(0xD6));
}
```

- Ergebnis:

```
CW:0x73 => HCW:0x79e
CW:0x1D => HCW:0x1e7
CW:0xD6 => HCW:0xdb9
```

- Handelt es sich bei den Codeworten 0xA24, 0x5D6 und 0x141 um zulässige Codeworte, Codeworte mit korrigierbaren oder Codeworte mit erkenn- aber nicht korrigierbaren Verfälschungen?
  Lösung: Bestimmung der Nummer des verfälschen Bits.
  Wenn 0, zulässiges CW, wenn 1 bis 12 korrigierbares CW sonst nicht korrigierbar.

```
int main(){
  printf("dq(0xA24)=%x\n", fBit_H8_12(0xA24));
  printf("dq(0x5D6)=%x\n", fBit_H8_12(0x5D6));
} printf("dq(0x141)=%x\n", fBit_H8_12(0x141));
```

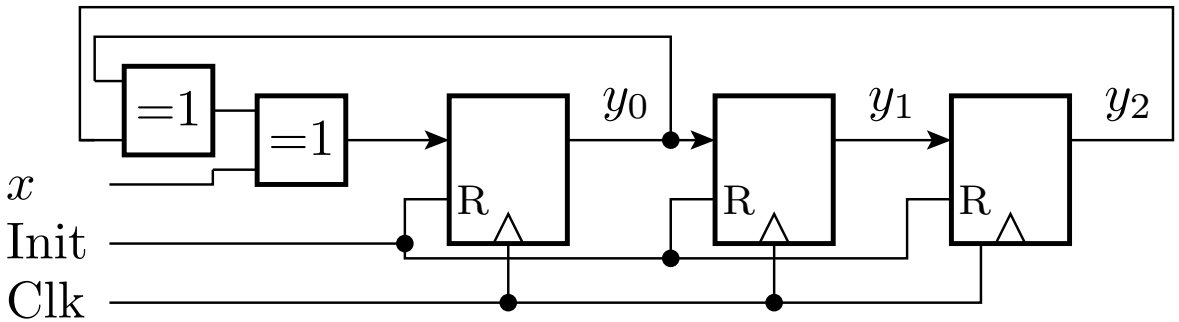
- Ergebnis

```
dq(0xA24)=3 korrigierbar, inv. x0
dq(0x5D6)=9 korrigierbar, inv. x4
dq(0x141)=f nicht korrigierbar
```

Prüfkennzeichen mit LFSR

Aufgabe 3.7: Prüfkennzeichen mit LFSR

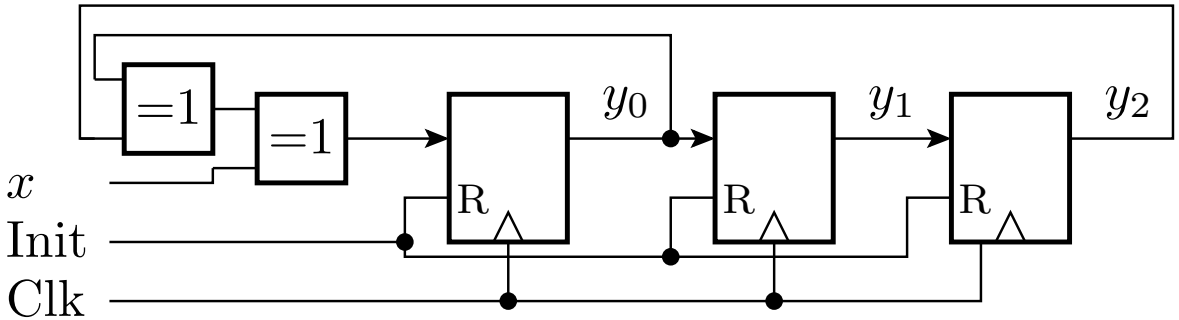
Gegeben ist folgendes linear rückgekoppelte Schieberegister:



	x	y_2	y_1	y_0
0	1	0	0	0
1	0			
2	0			
3	1			
4	1			
5	0			
6	0			
7	1			
8	0			
9	1			
10	1			
11	1			
12	1			
13	0			
14	1			
15	0			
PKZ:				

1. Wie hoch ist Fehlererkennungswahrscheinlichkeit?
2. Welches Prüfkennzeichen $\mathbf{y} = y_2 y_1 y_0$ hat die Datenfolge »1001100101111010« bei Abbildung beginnend mit dem höchstwertigen Bit. Startwert 000. Füllen Sie dazu die Tabelle in der Abbildung aus.

Lösung



	x	y_2	y_1	y_0
0	1	0	0	0
1	0	0	0	1
2	0	0	1	1
3	1	1	1	1
4	1	1	1	1
5	0	1	1	1
6	0	1	1	0
7	1	1	0	1
8	0	0	1	1
9	1	1	1	1
10	1	1	1	1
11	1	1	1	1
12	1	1	1	1
13	0	1	1	1
14	1	1	1	0
15	0	1	0	0
PKZ:		1	0	0

Erkennungswahrscheinlichkeit:

$$p_{\mathrm{E}} \approx 1 - 2^{-3} = 87{,}5\%$$

Syntaxtest

Aufgabe 3.8: Mikrorechner Ein- und Ausgabe

Ein einfaches Protokoll für die Kommunikation zwischen Mikrorechnern ist ein ASCII-Zeichen für den Nachrichtentyp

```
t = 'U'|'V'|'W'
```

gefolgt von drei Dezimalziffern:

```
z = '0'|'1'|...|'9'
```

- Beschreiben Sie dieses Eingabeformat als formale Sprache mit den Ersetzungsregeln »...|...«, »[...]« und »{...}«.
- Spezifizieren Sie einen Automaten zum Erkennen von Worten der Sprache zur Rückgewinnung des Kommandos und des Wertes als Ablaufgraph.
- Entwerfen Sie in C eine Parser-Funktion

  ```
  void parsMsg(unsigned char c);
  ```

  zur Nachbildung des Automaten.

```
Private globale Zustandsvariablen:
    unsigned char state; // Automatenzustand
    unsigned char cmd;   // Kommando
    unsigned short val;  // Paramterewert
```

- Testrahmen mit Testausgaben:

```
int main(){
 unsigned char i, dat[]="fgU126W248H77V1F";
 state = 0;
 printf("c: state cmd val\n");
 for (i=0;i<16;i++){
  parsMsg(dat[i]);
  printf("%c:   %i    %c  %3i\n", dat[i],state,cmd,val);
 }
}
```

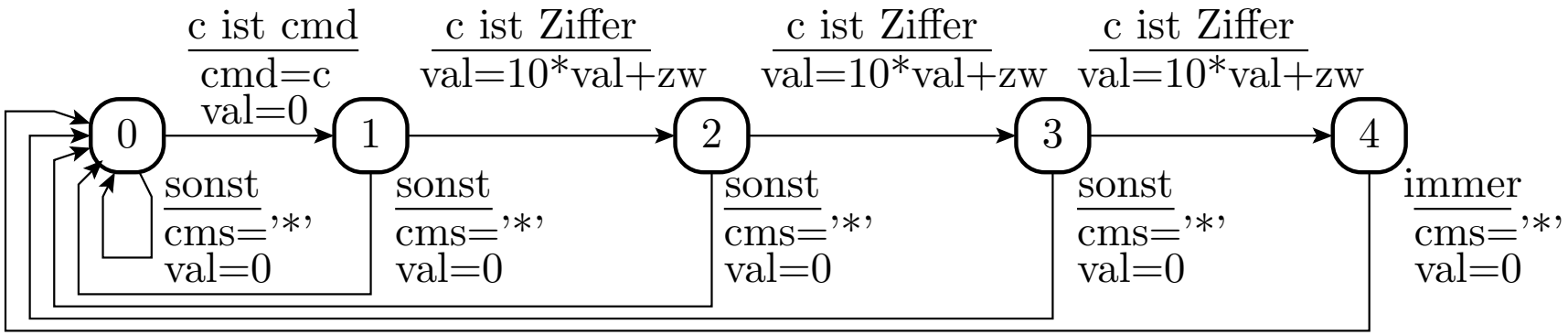
```
    c: state cmd val
    f:   0    *    0
    g:   0    *    0
    U:   1    U    0
    1:   2    U    1
    2:   3    U   12
```

Lösung

- Beschreibung als formale Sprache:

  `gültiges_Kommando => tzzz`

- möglicher Ablaufgraph:




```
c: state cmd val        6:  4   U   126
f:   0    *    0       W:  1   W     0
g:   0    *    0       2:  2   W     2
U:   1    U    0       4:  3   W    24
1:   2    U    1       8:  4   W   248
2:   3    U   12       H:  4   *     0
```

```
unsigned char state, cmd; unsigned short val;
void parsMsg(unsigned char c){
 switch(state){
  case 0:    // unzul. Zeichen empfangen
  case 4:    // gült. Kommando empfangen
    cmd ='*'; val = 0;
    if ((c<'U')||(c>'W')) return;
    else {
      cmd = c;
      state = 1; return;}
  case 1:    // t empfangen
  case 2:    // tz empfangen
  case 3:    // tzz empfangen
    if ((c < '0') || (c >'9')){
      cmd ='*'; val = 0; return;}
    else {
      val = val*10 + c -'0';
      state += 1; return;}
 }
}
```