How to TGI

Zahlensysteme **2**
- Umrechnen 2
  - Irgendeins ins Dezimal 2
  - Dezimal in Irgendeins 2
    - VK: 2
    - NK: 3
  - Irgendwas in Irgendwas (Wo es leichter geht) 3
  - Hexa <==> binär 3

Darstellen in Binär **4**
- Vorzeichen und Betrag (und Fixpunkt) 4
- Einerkomplement 4
- Zweierkomplement 4
- Exzess 4
  - Rechnen mit exzess 4
- Runden 5

IEEE 754 Gleitpunktzahlen **5**
- Aussehen 5
- Umwandeln 7
  - Binär -> IEEE 7
- Sonderfälle 8
  - Expo alles Einser 8
  - Expo alles Nuller 8
- Rechnen mit dem Spaß 8
  - Addieren & Subtrahieren 8
  - Multiplizieren und Dividieren 9

Codes **9**
- Barcodes = EAN13 9
- Rechnen 9
- NRZ 9
- Arten von Binärcodes 10
- Hamming-Distanz 10
  - Hamming-Distanz erhöhen 10
- Prüfstellen-Codes 10
  - Hamming-Code codieren 10
    - Rausfinden welches falsch ist 11

Polynomcodes 11
Rausfinden ob ein Fehler passiert ist 12

Zahlensysteme

…. b^8 b^7 b^6 b^5 b^4 b^3 b^2 b^1 b^0 . b^{-1} b^{-2} …

b = Basis = Anzahl der Ziffern, Ziffer * (basis ^ position) = “Wert der Ziffer”
Darstellung: Ziffern in Klammern, dann Basis tiefgestellt $(212511)_{10}$

WIchtig: Einerstelle ist immer b^0 also mal 0 -> die Ziffer selbst.

Umrechnen

Irgendeins ins Dezimal

Es gibt **2** gute **möglichkeiten**, je nach Geschmack anzuwenden:
1. Möglichkeit: Fette **Addition der einzelnen ziffern mal basis ^ position**
Zb Zahl z = $(423)_9$:
-> Dezimalsystem: 3 + 2*9 + 4 * 9^2 = 3 + 18 + 4*81 = 21 + 324 = $\mathbf{(345)_{10}}$

2. Möglichkeit: **Horner-Schema**
Von links beginnend:
((((1. Ziffer * Basis + 2. Ziffer) * Basis) + 3. Ziffer)* Basis … * basis + letzte ZIffer
Also immer ziffer * basis, Ergebnis plus nächster Ziffer bis zur einerstelle:
z = $(423)_9$: 4*9 + 2 -> 38 *9 + 3 = 342 + 3 = $\mathbf{(345)_{10}}$

Beide liefern dasselbe ergebnis, lern eins davon.

Dezimal in Irgendeins

Vorkommerteil und Nachkommerteil trennen.

VK:

dezimalzahl / neue basis = rest
Ergebnis obriger rechnung / neue basis = rest
Ergebnis obriger rechnung …..

...
Macht man bis man 0 als rest hat. Die Zahl im anderen System ist dann der rest **von unten nach oben** gelesen. Wichtig, falls man in ein größeres System umwandelt muss man den Rest (Ziffern), die größer als 10 sind entsprechend ‘umwandeln’ . zb. 10 in dez = A in hex

NK:

0 . NK * basis = VK_0 . NK_0 → VK_0
0 . NK_0 * basis = VK_1 . NK_1 → VK_1
…
Repeat bis genug stellen oder rest = 0.
Vorkommerteil ist hier jeweils die Ziffer nach dem Kommer.
Bei der Rechnung immer vor dem * basis den Vorkommerteil auf 0 setzen.
Von oben nach unten zu lesen

Irgendwas in Irgendwas (Wo es leichter geht)

Geht grundsätzlich auch wie oben (siehe Dezimal in Irgendeins), aber meistens haben wir guade Systeme, die uns die arbeit erleichtern.

Wenn die Basis des 1. system eine potenz des 2. Systems ist, können wir leichter umrechnen.
Zb: 2 in 16, 2 in 8, 4 in 16, 3 in 9…
Dazu kann man die Ziffern des größeren systems immer als mehrere Ziffern des kleineren anschauen und Ziffer für Ziffer umrechnen, was ca. 18 mal schneller ist.
wie viele Ziffern? → immer die hochzahlen anschauen:
16 in 2 → 16 = 2^z -> z = 4 Ziffern
8 in 2 -> 8 = 2^3 -> 3 Ziffern
16 in 4 -> 16 = 4^2 -> 2 Ziffern
9 in 3 -> 9 = 3^2 -> 2 Ziffern

Wichtig: immer volle anzahl an Ziffern verwenden:
$(B\ A\ .\ 3)_{16}$ -> $()_4$

B = 11_{10} = 23_4
A = 10_{10} = 22_4
3 = 3 -> 03 und nicht 30!

-> Zahl im 4er = 2322.03

Hexa <==> binär

Weil mans oft braucht ein paar tipps zum schneller rechnen:
Jede Hex-Ziffer ist eine Summe von 8, 4, 2 und 1 (was den Binärziffern entspricht).

Wenn man sich also fragt, was $C_{16} = 12_{10}$ in binär ist, fragt man sich aus was 12 besteht: 8 + 4 -> 1100.
Diese Denkweise hilft für speed und ein paar zahlen auswendig zu wissen hilft auch:
1010 in binär = 10 in dez = A in hex
1111 in binär = 15 in hex
0110 = 6

Darstellen in Binär

Vorzeichen und Betrag (und Fixpunkt)

Vorzeichen ist das 1. bit (- ist ein 1er, + ist 0), danach kommt einfach der Betrag binär.
Fixpunktdarstellung ist bei kommazahlen relevant. Die anzahl der Nachkommerstellen ist gegeben und man schreibt die Zahl ohne dem komma hin, “weil der computer eh weiß wo das komma ist”

Einerkomplement

Positive Zahlen bleiben gleich
Negative Zahlen werden invertiert (Vorzeichenbit bleibt gleich).
-> negative zahlen kommen im Direktvergleich nach positiven ->

Zweierkomplement

Wie Einerkompliment nur +1 gerechnet wenn negativ (positive bleiben unberührt).

Exzess

Der Exzess ist gegeben.
Zahl + Exzess = Zahl in Exzessdarstellung.
Wenn Zahl negativ → Exzess - | Zahl | (eh wie bei normaler addition, wenn eine der zahlen negativ ist)
sinn: negative zahlen werden positiv gemacht. Der Betrag der Zahl muss kleinergleich (<=) dem Exzess sein, sonst klappt der spaß nicht.

Rechnen mit exzess

Der Exzess wird im IEEE 754 verwendet um negative Zahlen zu killen.
Addition $A_E + B_E$:
$A_E + B_E$ - Exzess

nach normaler rechnung - Exzess rechnen, weil A_E (heißt A in Exzessdarstellung) = A + E und B_E = B + Exzess -> $A_E + B_E$ = A + E + B + E = A+B + 2 * E -> ein E müss ma wegnehmen.

Subtraktion:
$A_E + B_E$ + Exzess (Begründung siehe oben).

Runden

Truncate	->	abschneiden nach angegebenen stellen
Round to nearest	->	0.01_2 = ~0 ; 0.11_2 = ~1, bei $.100_2$: keine entscheidung möglich -> 2. Rundungsvorschrift zieht
Round to even	->	meistens als 2. Vorschrift nach (RTE) angegeben, relevant bei $.10_2$ (nur noch nuller nach dem 1er) -> wenn direkt vor dem Komma eine 0 -> einfach abschneiden (weil 0 ist even), wenn 1 -> + 1 rechnen.

Round towards +infinity -> wenn zahl positiv aufrunden, wenn negativ abrunden
Round away from zero -> betragsmäßig aufrunden (eh kloar)

absoluter Rundungsfehler: Gerundetes Ergebnis - Genaues Ergebnis
relativen Rundungsfehler: Absoluter Rundungsfehler / Genaues Ergebnis (bitte nie ganz ausrechnen, Bruch aufschreiben reicht ihnen)

IEEE 754 Gleitpunktzahlen

Aussehen

Zahlen mit denen computer echt rechnen -> klassische 32 oder 64 bit:
Zahlen werden immer normalisiert (auf 1.nachkommastellen * 2 ^ exponent gebracht)
Single precision: 32bit: F(**2**, **24**, **-127**, **128**, **true**)

2: **basis** -> binär (ist immer ein 2er)
24: **Mantissen-precision**: Mantisse heißt das 1.nachkommastellen ohne exponent -> in binär haben wir immer einen 1er vorne, also lassen wir den Vorkommaeinser weg -> **23 nachkommastellen** schreiben wir hin
-127: **kleinst**möglicher **Exponent**. **= wenn** im **Exponent nur eine 1 ganz rechts** steht.
128: **größt**möglicher **Exponent;** Gibt **gleich**zeitig den **Exzess** an. Ist erreicht, wenn **alles 1er außer ganz rechts ein 0er**

true: wenn true (es ist eig. Immer true): denormalisierte Zahlen sind darstellbar = **0.nachkommastellen * 2 ^ eMin** -> **gekennzeichnet** durch **kleinsten Exponenten minus 1** = im **Exponent** stehen **nur nullen**

Hat den sinn, dass man zahlen näher bei null darstellen kann

0		0	0	0	0	0	0	0	0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

(abstand nur für lesbarkeit)
1 bit: Vorzeichen: - = 1, + = 0
8 bit Exponent in Exzessdarstellung
23 bit Mantisse (mantisse sind immer die Nachkommerstellen nach dem impizierten 1er)
!!! die 24 in der Angabe stehen für precision, den 1er vorm komma kann man sich sparen -> 23 stellen.

Wie merk ich mir diese Zahlen??
Mantisse = 3/4 von 32 bit -> **24 ->** auf ein bit pfeifen -> 23. -> 32 - **1** (VZ-bit) - 23 = **8** bit für den exponeten.

Double precision: 64bit: F(2, 52, -1023, 1024, true)
1 bit: Vorzeichen: - = 1, + = 0
11 bit Exponent in Exzessdarstellung
52 bit Mantisse (mantisse sind immer die Nachkommerstellen nach dem impizierten 1er)
!!! die 53 in der Angabe stehen für precision, den 1er vorm komma kann man sich sparen -> 23 stellen.

-> kommt zum rechnen eh nicht, aber zum merken: expo: 11 ist die nächste Fibonaccizahl nach 8, rest errechnet sich aus den 64.

Wir rechnen meistens mit einem kleineren 16bit-Format: F(2, 11, -14, 15, true) ->
10 bit für **Mantisse** und **5** für **Exponent**

Umwandeln

Binär -> IEEE

1. Vorzeichen kann man gleich ins Ergebnis schreiben (- wird zu 1, + zu 0),
2. Zahl normalisieren (auf 1.irgendwas * 2^irgendwas bringen. Das hoch irgendwas gibt verschiebung an)
3. Exponent (die hochzahl überm 2er) + dem Exzess rechnen (Exzess = höchster Exponent = 01111...1). -> wenn das voller einser ist, haben wir einen sonderfall, bei lauter nuller auch, siehe Sonderfälle
4. Exponent + exzess hinschreiben (nullen vorne dazufetzen, nicht hinten)

5. Nachkommastellen als Mantisse hinschreiben

Sonderfälle

Expo alles Einser

1. In **Mantisse** irgendwo ein oder mehr **einser** stehen (**!= 0**) -> **NaN**.
2. in **Mantisse** lauter **Nuller** -> **unendlich** (+ oder - **unendlich** je nach Vorzeichen)

Expo alles Nuller

1. **Mantisse** nur **Nuller** -> + oder - Null (je nach vorzeichenbit)
2. **Mantisse ungleich null** -> **denormalisierte** Zahl -> 0.mantisse (. = komma) und nicht 1.mantisse!

Rechnen mit dem Spaß

Addieren & Subtrahieren

1. Exponenten angleichen (kleineren größer machen) -> damit Zahl gleich bleibt:
2. Mantisse um abstand der beiden Exponenten nach rechts schieben (dabei die 1 vor dem komma nicht vergessen) zB. A = .101 * 2^2 soll auf 2^4 gebracht werden
   -> A = .01101.
   Dabei schiebt man die Manitsse aus dem Format raus -> nur die letzten 3 bits außerhalb sind relevant Guard, Round und Sticky. G und R sind einfach das was da steht und Sticky gibt an, ob irgendwann eine 1 drüber geschoben wurde (ist fürs runden wichtig).
3. Mantissen mit den letzten überschuss Bits rechnen und nicht vergessen, dass eine der beiden Vorkommazahlen jetzt ein Nuller ist -> schreib sie dir in Klammer zw. Expo und mantisse.
4. Jetzt runden wir:
   wenn G 0 -> abrunden,
   wenn G und R 1 -> aufrunden,
   wenn G1R0S1 -> aufrunden,
   wenn G1 R0 S0 -> grenzfall siehe 2, Rundungsbedingung

5. Ist Sonderfall?
   a. während dem rechnen wurde durch übertrag das impizierte erste bit zu einem 10 (2er wertigkeit) -> einfach expo + 1 rechnen.
6. Vorzeichen hinschreiben nach Logik

Multiplizieren und Dividieren

1. Exponenten addieren/subtrahieren (auf das +/- Exzess nicht vergessen)
2. Mantissen multiplizieren/dividieren (auf das 1. Nicht vergessen)
3. Vorzeichen hinschreiben nach Logik.
4. Ist Sonderfall?
    a. Expo kleiner min expo -> versuchen denormalisert darzustellen indem man die Mantisse nach rechts verschiebt bis im Exponenten nur nullen stehen, wenn dann nach dem GRS runden noch was in der Mantisse steht -> gut, sonst ist es Null (0).

Codes

Barcodes = EAN13

Immer **7bit je Ziffer (2 dunkle und 2 helle Streifen)**. Mitte 5 bit Anfang und Ende je 3 bit.
Insgesamt **13 Ziffern**, **aber 1. ist impliziert aus Codemuster des ersten Teils.**
Letzte Zahl ist Prüfziffer, wird auch hincodiert.

Rechnen

1. Implizierte Ziffer, kann man aus der Tabelle je nach Muster ablesen (kommt nicht zum test)
2. Prüfsumme (letzte Ziffer) kann man sich mit folgernder Formel ausrechnen:

$10 \mid (Z_0 + Z_1*3 + Z_2 + Z_3*3 \ldots. + Z_{11}*3 + \text{Prüfziffer})$ // 10 | heißt “zehn teilt restlos”

-> alle Ziffern addieren (mit erster implizierten ohne Prüfziffer [schreib einfach p hin]), jede 2. * 3 rechenen.
Dabei kann man, weil ja nur relevant ist, obs durch 10 restlos teilbar ist, alles wegstreichen, was zusammen 10 ergibt. Zb 2 und 8 oder 8,8,4 (=20) oder 8,6,6 oder …
Dann kommt sowas raus wie 10 | 7 + p -> p = 3 = Prüfziffer.
Als Aufgabe wird gerne eine andere Ziffer als die Prüfziffer freigelassen -> dann rechnet man mit x und setzt die prüfziffer als echte ziffer ein.

NRZ

Wirklich nur ablesen

Arten von Binärcodes

Blockcode -> **jedes codewort** hat **dieselbe länge**. Alle untenstehende sind Blockcodes
Lineare Codes -> **2 Codewörter addiert** (Modulo-Arithmetik: ohne Überschlag -> 1 + 1 = 0 ohne 1 weiter) ist **wieder** ein **Codewort** des Codes.
Zyklischer Code -> Linearer Code. für jedes codewort gilt: erstes bit nach hinten verschieben liefert wieder valides codewort
Systematische Codes -> es gibt **eigene Datenbits** und **abhängig von der anzahl** dieser **Prüfbits**.

Hamming-Distanz

Gibt **minimalen** Abstand zweier Codewörter eines Codes an (**wie viele bits unterschiedlich** sind).
Wenn Hamming-Distanz 2 ist, kann man einen Fehler erkennen, wenn nur 1 bit falsch ist -> fällt zwischen 2 Codewörtern.

Hamming-Distanz erhöhen

Von 1 auf 2 -> 1 Parity bit anhängen. Das heißt nur, ein bit anzuhängen, das angibt ob die anzahl der 1en gerade oder ungerade ist. (sollt angegeben sein, oder aus richtigem codewort ersichtlich)

Prüfstellen-Codes

Sind systematische Codes

Mehrere Paritätsbits: Die prüfbits (paritätsbits) geben immer unterschiedliche informationen an -> man kann fehler ausbessern (wenn nicht zu viel falsch übertragen wurde).
Zb. Hamming-Code,

Hamming-Code codieren

Zu codierende Daten -> kommen in datenbits.
Um fehler finden zu können, werden prüfbits dazugegeben.
Aber wo wie welche? -> alle 2er Potenzen sind Prüfbits → man beginnt mit 2 prüfstellen, weil 1 und 2 beides 2er-Potenzen sind -> (erstmal leer lassen), die 3. Stelle ist das erste Datenbit -> 1. Ziffer der Daten rein (d1) -> 4 ist wieder eine Prüfstelle. (leer lassen)-> 5, 6, 7 sind weider Datenbits (d2, d3, d4). -> weitermachen bis all die Daten an ihren Stellen stehen.

Prüfbits berechnen: Idee ist, dass der index jedes der Datenbits als summe von 2er Potenzen aufgefasst werden kann, all jene prüfbits beeinflusst dessen Inhalt also.
Bei einer Länge von 5 Datenbits ->

c1	c2	c3	c4	c5	c6	c7	c8	c9
p1	p2		p3				p4	
		d1		d2	d3	d4		d5

-> p1 = c3 (+) c5 (+) c7 (+) c9 [c6 fehlt, weil es 4 + 2 ist ohne +1, nie geraden indizes hier]
-> p2 = c3 (+) c6
-> p3 = c5 (+) c6 (+) c7
-> p4 = c9
(+) addieren und nur auf die letzte stelle schauen = XOR -> 1+1 = 0, 0+0 = 0, 1+0, 0+1 = 1

-> jedes datenbit kommt genau 2 mal vor -> rausfinden welches falsch ist, easy

Rausfinden welches falsch ist

1. prüfbits nachrechnen
2. Wenn 1 falsch ist -> dieses Prüfbit ist falsch
3. Wenn 2 falsch sind, schauen wovon sie gemeinsam abhängen:
   Zb wenn p3 und p2 falsch sind -> c6 muss falsch sein, weil es die einzige stelle ist, von der p3 und p4 abhängen. (ein bisschen wie kgv finden)

Polynomcodes

Sind auch wieder Codes mit mehreren Prüfbits.
Bitfolgen werden als Polynome gesehen: **01010110** = $x^6 + x^4 + x^2 + x^1$
//das nennen wir M(x)

Es gibt ein Generatorpolynom (gegeben). Die Prüfsumme ist so gewählt, dass die Division des Datenpolynoms / Generatorpolynom 0 Rest gibt.

Sagen wir **G(x) =** x^4+x^3

Wie berechnet man die Prüfsumme?

1. Grad von G(x) bestimmen = größte hochzahl -> **4**
2. So viele nullen an binärdaten anhängen -> **1010100 0000** = $x^4 * (x^6 + x^4 + x^2)$
3. Jetzt Polynomdivision x^4***M(x) / G(x)** aber wieder mit Modulo-Arithmetik -> wenn man eine neue stelle runternimmt, wird alles was negativ war wieder positiv.
4. Der Rest der Polynomdivision wird vom Ursprünglichen M(x) abgezogen.
5. done

-> (x^10 + x^8 + x^6) / (x^4 + x^3) = x^6 + x^5
x^10 + x^9 (="geht x^6 mal, x ^6 * x hoch 3 = x ^10, x ^4 * x hoch 3 = x hoch 9)
---
0 + x^9 + x ^8 (="x hoch zehn und x hoch zehn gleich 0, x ^9 nehma mit, x ^8 herab")
+ x^9 + x^8 (="geht x hoch 5 mal, ….)
0 + 0 + x^6 Rest

-> (x^6 **+** x^4 **+** x^2**) - x^6 =** (x^4 **+** x^2**) = Codiert.**

Rausfinden ob ein Fehler passiert ist

Wie beim Codieren vorgehen, wenn bei der Polynomdivision 0 Rest rauskommt ist vermutlich kein Fehler aufgetreten.