Aufgabe 2.1 Zweierkomplement

Negiere die folgenden 8-Bit-Zahlen schriftlich (siehe Folie 6): 1, 37, -99, 0, -128.

Aufgabe 2.2 Binäre Ausgabe

`printf` bietet leider keinen Formatstring für die binäre Ausgabe von Zahlen. Schreibe eine Funktion `void print_binary(unsigned x)`, die diese Aufgabe übernimmt. Dabei sollen grundsätzlich alle 32 Bit auf die Konsole geschrieben werden.

(Wie man in C an die einzelnen Bits herankommt, kannst Du der letzten Folie entnehmen.)

Schreibe eine weitere Variante, die führende Nullen bei der Ausgabe weglässt. Falls man die Zahl `0` ausgeben will, soll natürlich trotzdem eine Stelle ausgegeben werden.

Aufgabe 2.3 Bytes innerhalb eines Wortes vertauschen

Schreibe eine Funktion `unsigned endianswap(unsigned x)`, welche die beiden Byte-Paare am Rand und in der Mitte jeweils miteinander vertauscht, so dass zum Beispiel `0x12345678` auf `0x78563412` abgebildet wird.

Hierzu bietet es sich an, immer 8 Bits auf einmal zu isolieren, an die passende Stelle zu schieben und am Ende die Zwischenergebnisse zu verschmelzen:

```
x = 12345678

a = 12000000 (Isolieren)
b = 00340000
c = 00005600
d = 00000078

A = 00000012 (Verschieben)
B = 00003400
C = 00560000
D = 78000000

y = 78563412 (Verschmelzen)
```

Teste die Funktion manuell mit `printf` und dem Formatstring `%08x`, d.h.

```
printf("%08x", endianswap(0xba757a9e));
```

sollte `9e7a75ba` auf die Konsole schreiben.

Aufgabe 2.4 Farbraumkonvertierung

Farben werden oft als Kombination von Rot-, Grün- und Blau-Anteil (RGB) gespeichert. Je nachdem, wie viele Bits man pro Anteil reserviert, ergeben sich verschiedene Formate. Heutzutage wird meist RGB24 verwendet (8 Bit pro Farbkanal). Um Arbeitsspeicher zu sparen, waren früher auch RGB15 (5 Bit pro Farbkanal) und RGB16 (1 weiteres Bit für Grün, weil das menschliche Auge Grüntöne am besten unterscheiden kann) beliebt:

```
 0 r5 r4 r3 r2 r1 g5 g4 g3 g2 g1 b5 b4 b3 b2 b1 (RGB15)
r5 r4 r3 r2 r1 g5 g4 g3 g2 g1 g0 b5 b4 b3 b2 b1 (RGB16)
```

Schreibe eine Funktion `void print_rgb15(unsigned short rgb15)`, welche den Rot-, Grün- und Blau-Anteil als Dezimalzahl zwischen 0 und 31 auf die Konsole schreibt. `print_rgb15(0x62cf)` soll zum Beispiel folgendes auf die Konsole schreiben:

```
R: 24/31
G: 22/31
B: 15/31
```

Schreibe eine entsprechende Funktion `void print_rgb16(unsigned short rgb16)`. `print_rgb16(0xc58f)` soll zum Beispiel folgendes auf die Konsole schreiben:

```
R: 24/31
G: 44/63
B: 15/31
```

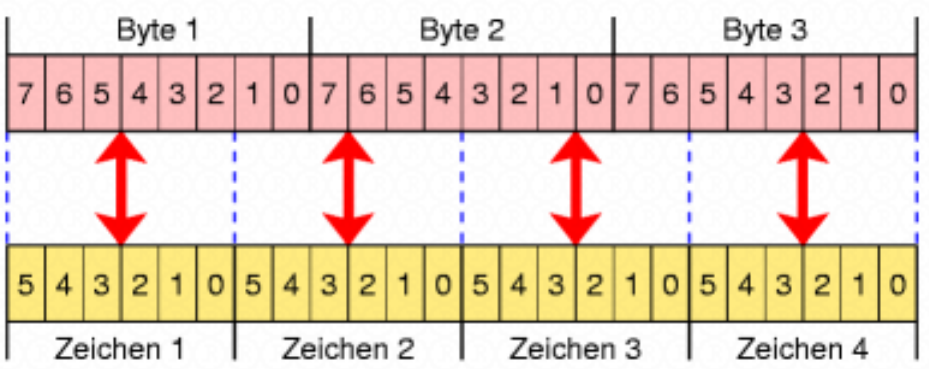
22/31 und 44/63 ist fast derselbe Grün-Anteil (prüfe mittels Taschenrechner). Da in RGB16 ein zusätzliches Grün-Bit zur Verfügung steht, hat sich der Wert von 22 auf 44 verdoppelt.

Schreibe eine Funktion `unsigned short rgb15to16(unsigned short rgb15)`, die einen RGB15-Pixel in einen RGB16-Pixel konvertiert. Wie Du in der obigen Illustration erkennen kannst, bleiben die 5 blauen Bits an derselben Position stehen. Die 5 grünen Bits wandern um eine Position nach links, wobei das neue Bit g0 zunächst den Wert 0 annehmen soll. Die 5 roten Bits müssen ebenfalls um eine Position nach links wandern. Verwende zum Testen das obige Beispiel: `rgb15to16(0x62cf)` sollte `0xc58f` liefern.

Im letzten Schritt soll die Verdoppelung des Grün-Anteils noch eine Rundung erfahren: Für die Werte 0..15 soll der Wert einfach nur verdoppelt werden wie bisher, also 0..30. Für die Werte 16..31 soll zusätzlich noch 1 addiert werden, also 33..63 (statt 32..62). `rgb15to16(0x62cf)` sollte jetzt `0xc5af` liefern statt `0xc58f`.

Aufgabe 2.5 Base64-Kodierung

Beim Verschicken von Mails oder beim Posten in Foren wird gerne das Base64-Format verwendet, um Binärdaten (z.B. Bilder, Musik oder Videos) als Text darzustellen. Dabei kodiert man jeweils 3 Bytes als 4 Zeichen, wobei man sich auf 64 verschiedene sichtbare Zeichen beschränkt (26 Großbuchstaben, 26 Kleinbuchstaben, 10 Ziffern, `+` und `/`).



Schreibe eine Funktion `void encode64(byte src[], int len, char dst[])`, die diese Aufgabe übernimmt. Gehe zunächst davon aus, dass `len` ein Vielfaches von 3 ist.

C hat eigentlich gar keinen Datentyp namens `byte`, aber wir können ihn gut simulieren:

```
typedef unsigned char byte;
```

Schreibe anschließend das entsprechende Gegenstück `decode64` und teste ausführlich.

Verallgemeinere `encode64` und `decode64` zuletzt für beliebige Längen.