1 Bits und Bytes, Rechnerarchitektur

Dieses Kapitel soll die nötigen Einsichten und Konzepte liefern, damit die Grundprinzipien eines (elektronischen) Computers verstanden und nachvollzogen werden können.

1.1 Bits und Bytes

Definition 1 Bit und Byte

Ein **Bit** (von **bi**nary digi**t**) ist die kleinste Informationseinheit und kann genau **2** verschiedene Zustände annehmen: **wahr** oder **falsch** (`True, False` in Python).
Ein **Byte** ist eine Zusammenfassung von üblicherweise 8 Bits. Ein Byte kann somit $2^8 = 256$ Zustände annehmen.

Physikalisch werden Bits auf unterschiedliche Art dargestellt, z.B. Strom, bzw. Spannung an oder aus, Magnetisierung Nord-Süd oder Süd-Nord, langer oder kurzer Puls etc.
In verschiedenen modernen Datenspeicher- und -übertragunsmedien werden heute mehr als 1 Zustand dargestellt (z.B. 5 Zustände bei Gigabit Ethernet, oder bis zu 16 Zustände bei SSD-Speicherzellen).

1.2 Logische Operatoren

Logische Operatoren operieren auf **Bool'schen** Werten, d.h. `True` oder `False`
In Python sind folgende logische Operatoren definiert: `not`, `and` und `or`.

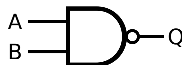
Aufgabe 1 Ein Python-Code zur Erzeugung von Wahrheitstabellen befindet sich auf dem Wiki. Kopieren Sie diesen Code und führen Sie diesen aus.

a) Studieren Sie die Ausgabe. Erklären Sie die Logik der Abfolge der Werte für A und B. Erklären Sie auch die letzte Zeile für die OR-Tabelle.

b) Die XOR-Operation liefert genau dann `True`, wenn genau einer der beiden Werte `True` ist. Implementieren Sie mit den 3 logischen Operationen die XOR-Operation und erweitern Sie den Code so, dass auch die Tabelle für XOR generiert wird.

c) Der **NAND**-Operator ist definiert als `a nand b = not(a and b)`. Erweitern Sie den Python-Code so, dass dieser auch die Tabelle für NAND generiert.

Aufgabe 2 Elektronisch ist die NAND-Operation am einfachsten (mit am wenigsten Bauteilen) zu implementieren. Man spricht vom **NAND-Gate**, bzw. NAND-Logikgatter (Eingänge A, B, Ausgang Q):



Die NAND-Operation hat die Besonderheit, dass alle anderen logischen Operationen damit ausgedrückt werden können.
Schreiben Sie die Operationen `not`, `and`, `or` und `xor` nur mit `nand` und zeichnen Sie die entsprechende Schaltpläne mit dem NAND-Gatter.
Überprüfen Sie Ihre Formeln mit dem Wahrheitstabellen-Code von Aufgabe 1 und einer selbst definierten `nand` Funktion.

Quelle der Grafiken: Wikipedia

1.3 Darstellung von natürlichen Zahlen und Adder

Wie funktioniert das Zehnersystem?

42’062 = ✎

Die Stellen im Dezimalsystem stehen also für Einer, Zehner, Hunderter, ..., 10^ner, ...

Das Zweiersystem funktioniert analog:

0b101010 = ✎

Die Stellen im Zweiersystem stehen also für Einer, Zweier, Vierer, Achter, Sechzehner, ..., 2^ner.

Binärzahlen werden in vielen Programmiersprachen (auch Python) mit dem **Prefix 0b** (Null be) geschrieben. Gewisse Programmiersprachen (Python nicht) erlauben es, Zahlen mit dem Bodenstrich (Underscore) zu gruppieren. Binärzahlen werden normalerweise in **Vierergruppen** aufgeteilt.

Aufgabe 3 Rechnen Sie vom Zehner- ins Zweiersystem um, bzw. umgekehrt.

a) 0b10
b) 10
c) 0b1000’0001
d) 1023
e) 0b1’0000’0000
f) 123

Aufgabe 4

a) Wie sieht man einer Binärzahl an, ob diese durch 2 teilbar ist?

b) Was geschieht mit einer Binärzahl, wenn diese mit 2 multipliziert wird?

Aufgabe 5 Addieren Sie 0b1100’0110’0101 und 0b111’0111’0110 schriftlich ohne Umweg über das Dezimalsystem. Das Resultat ist ebenfalls binär anzugeben.

Aufgabe 6 Diese Fragen beziehen sich auf das Spiel «2048» mit der «Vereinfachung», dass pro Zug genau eine neue Eins erscheint.

a) Wie viele Züge sind mindestens notwendig, um eine 2048er Kachel zu erhalten?

b) Was ist $\log_2(2048)$?

c) Welche Kacheln sind nach 42 Zügen idealerweise noch übrig?

d) Wenn man annimmt, dass alle möglichen Kacheln zusammengeschoben wurden, welche Kacheln sind nach n Zügen auf dem Spielfeld?

e) Was gilt für $n-1$ und n wenn nach n Zügen unmöglich die theoretische minimale Anzahl Kacheln erreicht werden kann?

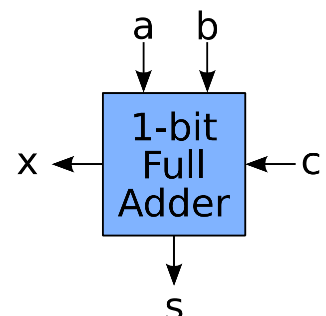
Aufgabe 7 Ziel ist es, eine logische Schaltung zu erstellen, die 3 Bits addieren kann ($a+b+c$). Man nennt diese Schaltung einen «Full Adder». Zwei Bits der Operanden a, b plus das «Behalte» (Carry) c. Das Resultat dieser Berechnung sind 2 Bits s (die Summe) und x (der Übertrag), wobei $2 \cdot x + s = a + b + c$, d.h. s ist das resultierende Bit und x das «Behalte». Vergleichen Sie dazu Aufgabe 5.

a) Erstellen Sie von Hand eine Wahrheitstabelle mit a, b, c als Input und x, s als Output.

b) Finden Sie Formeln für s und x mit den logischen Operationen `not`, `and`, `or` und `xor`.

c) Mit nebenstehendem Full-Adder, zeichnen Sie eine Schaltung, die zwei 4-Bit Zahlen addieren kann.

d) Wer Lust hat, kann einen Full-Adder aus NAND-Gates entwerfen.



1.4 Plexer und Coder

Multiplexer, Demultiplexer, Encoder und Decoder sind weitere Bausteine, um einen Digitalcomputer zu konstruieren.

Definition 2 Multiplexer

Ein Multiplexer (MUX) hat 2^n Dateneingänge, n Steuereingänge und einen Ausgang. Werden die Steuereingänge als Binärzahl i interpretiert, entspricht der Ausgang dem i-ten Dateneingang (nummeriert von 0 bis $2^n - 1$).

Aufgabe 8

a) Bauen Sie mit Logisim einen Multiplexer mit 2 Dateneingängen, 1 Signalleitung und 1 Ausgang. Nennen Sie die Schaltung «2to1mux».

b) Bauen Sie mit der «2to1mux»-Schaltung einen Multiplexer mit 4 Dateneingängen und 2 Signalleitungen. Nennen Sie die Schaltung «4to1mux».

c) Bauen Sie damit einen «8to1mux».

Aufgabe 9 Ziel der Aufgabe ist es, eine kleine Recheneinheit (ALU: arithmetic logic unit) zu bauen. Diese Einheit hat zwei 8 Bit grosse Eingänge A und B und kann folgende 4 Operationen ausführen: $A + B$ bitweise A and B, bitweise A or B und not A. Es werden immer **alle** Operationen parallel ausgeführt.

Welches Resultat am Ende ausgegeben wird, bestimmt ein 2-Bit Wert.

a) Bauen Sie die oben beschriebene ALU.

b) Zusätzlich soll die ALU ein Status-Bit ausgeben, wenn ein überlauf passiert ist (carry).

c) Zusätzlich soll die ALU ein Status-Bit ausgeben, wenn das Resultat Null ist (zero).

1.5 Clock

Je nach Operation, die eine ALU ausführt dauert es mehr oder weniger lang, bis sich das Ausgangssignal stabilisiert hat. Eine Addition ist viel aufwendiger als ein «and», weil jedes Carry «weitergereicht» werden muss.

Es muss also eine bestimmte Zeit gewartet werden, bis das Resultat effektiv benutzt werden kann.

Fast alle Prozessoren haben darum einen internen Taktgeber (clock). Für jeden Befehl ist bekannt, wie viele Taktzyklen (clock ticks) zu warten ist, bis das Resultat der ALU verlässlich ist.

Eine Clock produziert ein regelmässiges Rechtecksignal, das zwischen «False» (normalerweise 0V) und «True» (irgendwo zwischen 1.2 V - 5 V) hin und her wechselt. Das Signal wird normalerweise mit einen Quarzkristall erzeugt und ist typischerweise im Bereich von einigen MHz bis einigen GHz.

Aufgabe 10

a) Bestimmen Sie die Taktfrequenz Ihres Laptop-Prozessors.

b) Wie weit (Strecke in cm) kommt ein Signal höchstens zwischen zwei clock ticks?

c) Schätzen Sie die Anzahl Gatter ab, die ein Signal maximal durchlaufen muss, um zwei 64 bit Ganzzahlen zu addieren. Was hiesse das für die Schaltfrequenz dieser Gatter, wenn eine solche Addition in einem Taktzyklus erledigt werden soll?

1.6 Memory

Ein Computer braucht natürlich Datenspeicher. Die sind ganz unterschiedlicher Natur, meistens ein Kompromiss aus Zugriffsgeschwindigkeit und Grösse:

Register Interne Datenspeicher der CPU. Extrem schnell, dafür nur einige Dutzend.

Cache Abbild von Teilen des Arbeitsspeichers (RAM), sehr schnell, einige kB bis MB. (Oft noch in verschiedene Levels unterteilt).

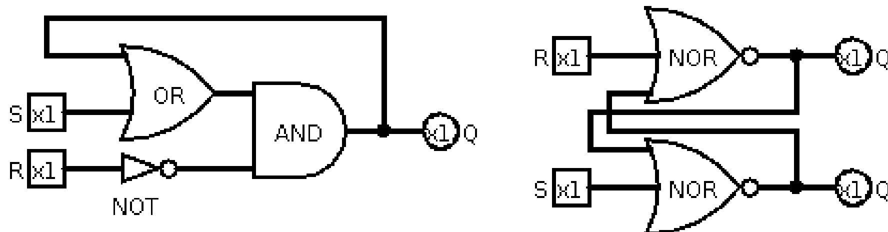
RAM Arbeitsspeicher. Eher langsam (in clock ticks gemessen, so ca. 10 ns). Einige GB.

Externer Speicher Harddisks (10-15 ms), SSD (0.1 ms), Netzwerkspeicher (1 ms bis 50 ms?). Sehr langsam (für die CPU eine Ewigkeit). Kapazität einige TB.

1.6.1 Flip Flop

Ein Flip Flop hat zwei stabile Zustände, die durch äussere Pulse beeinflusst werden können. Einfachste Flip Flops können aus zwei NOR Gattern gebaut werden. Wir werden ein RS Flip Flop aus je einem OR, NOT und AND Gatter bauen. Das ist etwas einfacher zu verstehen und hat den Vorteil, dass auch wenn beide Eingänge R und S True sind, dass der Ausgang weiterhin definiert ist (in diesem Fall 0).

Die Eingänge S und R stehen für **S**et und **R**eset. Werden diese einzeln auf 1 gesetzt, ändert der Zustand vom Ausgang Q entsprechend. Sind beide Eingänge 0, ändert sich der Ausgang Q nicht. Je nach Ausführung des Flip Flops dürfen nicht beide Eingänge 1 sein.



⚒ **Aufgabe 11** Erstellen Sie ein RS Flip Flop mit Logisim. Ein Link zum entsprechenden Screencast finden Sie auf dem Wiki.

1.6.2 Counter

Ein Counter zählt, wie viel mal ein Eingangssignal (typischweise eine clock) von 0 auf 1 gewechselt hat. Die Zahl wird binär dargestellt. Je nach Anwendung haben diese Counter mehr oder weniger Bits.

Läuft ein Counter über, kann Verschiedenes passieren. Im einfachsten Fall läuft der Counter bei Null weiter, bleibt stehen, oder löst weitere Aktionen aus, wie z.B. ein Interrupt, wo der normale Programmfluss in einem Prozessor unterbrochen wird und eine spezielle Routine aufgerufen wird.

⚒ **Aufgabe 12** Mit Hilfe der RS Flip Flops, die schon in Logisim vorgegeben sind, bauen Sie einen 1-Bit Counter.

Neben den Eingängen S und R, ist ein Eingang für die Clock vorhanden und ein «enable pin», der auf 1 sein muss, damit die Clock überhaupt einen Einfluss hat.

Die Idee ist, die Ausgänge Q und $\overline{Q}$ wieder so als Eingänge zu verwenden, dass der Flip Flop seinen Zustand wechselt beim nächsten Übergang der clock von 0 auf 1.

Ihr Schaltkreis soll einen Eingang (CLK) und einen Ausgang (OUT) haben.

Machen Sie dann einen neuen Schaltkreis und bauen Sie mit ihrem 1-Bit Counter einen 4-Bit Counter, indem Sie den Ausgang eines 1-Bit Counters als Eingang des nächsten Counters benutzen. Fassen Sie die vier Ausgänge auf eine 4-Bit Leitung zusammen und stellen Sie die Binärzahl mit einer Hex-Ziffer dar.

1.6.3 Daten- und Adressbus

Ein Bus ist eine Verbindung zwischen verschiedenen Teilen eines Computers. Es können mehr als zwei Teilnehmer «angeschlossen» sein. Ein typisches Beispiel für einen Bus ist die Anbindung vom Arbeitsspeicher (RAM) and den Prozessor (CPU). Die Anzahl paralleler Leitungen nennen wir «Busbreite». Die Busbreite sagt uns, wieviele Bits nebeneinander durch ein Bus passen.

Beim RAM wird unterschieden zwischen dem Adressbus und dem Datenbus.

Über den Adressbus teilt die CPU die Adresse, d.h. die Nummer der Speicherzelle, mit, an die geschrieben, bzw. von wo gelesen werden soll. Der Adressbus vermittelt keine Daten, sondern nur den Ort im Speicher, für den sich der Prozessor gerade interessiert. Der Bus ist unidirektional, denn Adressen gehen an den Arbeitsspeicher, wo sie eine bestimmte Speicherstelle ansteuern. Die Busbreite des Adressbus sagt uns, wieviele verschiedene Speicherstellen angesprochen werden können.

Der Datenbus übermittelt Daten zwischen CPU und RAM und ist somit bidirektional. Die CPU kann sowohl Daten zum Arbeitsspeicher senden, um sie zu verwahren, wie auch Daten vom RAM anfordern, um sie zu verarbeiten. Es muss also darauf geachtet werden, dass nicht beide Chips gleichzeitig auf den Bus schreiben (d.h. 0 V oder eine Spannung anlegen), weil sonst Kurzschlüsse entstehen könnten. Es darf also nicht vorkommen, dass der Arbeitsspeicher Daten via den Datenbus an die CPU schickt, während die CPU gleichzeitig den Datenbus für sich einnimmt, um ein Berechnungsresultat an den Arbeitsspeicher zu übermitteln.

Die Lösung für dieses Problem ist Three-state Logic.

1.6.4 Three-state Logic

Es gibt in der Elektronik neben low (d.h. False, 0 V, kann Strom schlucken) und high (d.h. True, z.B. 5 V, kann Strom liefern) noch einen dritten Zustand, nämlich «high impedance», was in etwa einem durchgetrennten Draht entspricht. Dies bedeutet, dass kein Strom fliessen kann und die Spannung nicht definiert ist. Das ist der Zustand, in welchem z.B. die Eingänge eines Logik-Gatters sind (die natürlich in nennenswerten Mengen weder Strom liefern noch schlucken sollen).

In Logisim kann three-state logic mit einem «controlled buffer» implementiert werden. Ist der Kontroll-Eingang auf 0, ist der Ausgang auf «high impedance», d.h. weder 0 noch 1 und kein eindeutiger Zustand wird weitergeleitet. Ist der Kontroll-Eingang des Buffers auf 1, wird der Eingang auf den Ausgang kopiert, so dass ein eindeutiger Zustand 0 oder 1 ausgegeben wird.

⚒ **Aufgabe 13** Bauen Sie ein 2-Bit RAM. Die beiden Speicherzellen sollen aus je einem RS Flip Flop bestehen.

Folgende (1 Bit weite) Inputs und Outputs sollen vorhanden sein:

- ADDR: Adresse der Speicherzelle. (Input)
- IN: Bit, das geschrieben werden soll. (Input)
- WRITE: Wenn 1, wird das Bit von IN in die Speicherzelle mit der Nummer ADDR geschrieben. (Input)
- OUT: Bit, das gelesen wurde. (Output)
- READ: Wenn 1, wird das Bit der Speicherzelle mit der Nummer ADDR auf OUT kopiert. (Input)

Es soll genau einen (1 Bit weiten) Datenbus geben. Die einzelnen In- und Outputs sind über «Controlled Buffers» von einander isoliert. Je nach READ oder WRITE werden die richtigen Buffer geöffnet. Dazu benötigen Sie je einen **Demultiplexer**, der aus der Adresse das READ bzw. WRITE Signal auf die richtige Speicherzelle leitet.

Wenn die Schaltung steht und Sie diese verstanden haben, erweitern Sie diese auf einen 4 Bit breiten Datenbus und einen 2 Bit breiten Adressbus. Es können so 4 Zeilen à 4 Bits adressiert werden.

1.7 Von Neumann Architektur, Program Counter

Der Programmcode kann entweder im gleichen Speicher liegen wie die Daten (von Neumann Architektur) oder in getrennten Speicherbereichen (Harward Architektur). Letztere Architektur findet man öfters bei Mikroprozessoren. Erstere ist gängig bei «stärkeren» Prozessoren.

Unabhängig davon hat jede CPU einen «Program Counter», oder kurz PC. Dieser gibt die Nummer der Speicherzelle an, wo die nächste auszuführende Instruktion liegt.

Diese Instruktion wird geladen (fetch), die ALU entsprechend vorbereitet (decode) und dann ausgeführt (execute). Das Resultat wird an die entsprechende Stelle gespeichert und der Programm Counter erhöht, bzw. angepasst, wenn ein Sprung (jump) nötig ist.

Die Instruktionen selbst sind binär codiert als sogenannter **Maschinencode**.

Ein Beispiel einer vollständigen CPUs in Logisim finden Sie z.B. auf http://minnie.tuhs.org/CompArch/Tutes/week03.html.

2 Assembler

Assembler bezeichnet eine Programmiersprache, die praktisch eins zu eins den Maschinencode abbildet. Assembler ist aber lesbarer Text anstatt binär. So stünde z.B. «ADD A,B» für «addiere Register B zu Register A», oder «MOV A,[123]» für «speichere den Inhalt der Speicherzelle mit Nummer 123 im Register A».

2.1 Hexadezimalsystem

Wird systemnah programmiert (wie z.B. auf Microprozessoren) ist das Hexadezimalsystem gebräuchlich. Das entspricht dem Sechzehnersystem, welches mit den Ziffern 0-9 und a-f (für 10 bis 15) arbeitet. Das Präfix ist fast universell 0x.

Aufgabe 14 Rechnen Sie folgende Zahlen ins 16er-, 10er- und 2er-System um

a) 42 b) 0x42 c) 0b101'1010 d) 3'735'928'559

Finden Sie auch die entsprechenden Python-Fuktionen für die Umrechnungen.

Aufgabe 15 Erklären Sie, wie man zwischen 2er- und 16er-System umrechnet und warum das so einfach ist. Wie viele Hex-Ziffern werden für ein Byte benötigt?

2.2 ASCII Code

Der ASCII-Code ist eine standardisierte Zuordnung von Zahlen (0-126) zu Zeichen (Buchstaben, Ziffern, Satzzeichen, einige Sonderzeichen). Diese Codierung ist praktisch auf jedem textfähigen System gleich.

Einige «Besonderheiten»:

- Die Differenz zwischen Gross- und Kleinbuchstaben ist genau 32, d.h. deren ASCII-Code unterscheidet sich nur im Bit Nummer 5 (d.h. das 6. Bit). Z.B. ist 'A'=65=0b100'0001 und 'a'=97=0b110'0001.
- ASCII-Codes kleiner 31 sind Steuerzeichen (z.B. 10 steht für Zeilenumbruch).
- Die Ziffer Null hat den ASCII-Code 48=0b11'0000, die weiteren Ziffern folgen darauf. Um eine Ziffer in das entsprechende Zeichen umzurechnen, kann also «logisch oder» gerechnet werden. Eine Addition nicht nicht nötig, da keine Überläufe auftreten können.

Aufgabe 16 Studieren Sie den «Hello World» Code im «Simple 8-bit Assembler Simulator» zu finden auf https://fginfo.ksbg.ch/~ivo/assembler-simulator/ (Link auf dem Wiki). Beachten Sie dazu auch die Hilfsseite mit dem «Instruction Set».

Aufgabe 17 Die Speicherzellen mit Adressen 253-255 beeinflussen auch die 7-Segment-Anzeigen. Die einzelnen Segemente sind von oben im Uhrzeigersinn nummeriert, das mittlere Segement ist das siebte. So beinflusst z.B. das 5. Bit (Bit Nummer 4) das Segment unten links.

Schreiben Sie ein Assembler-Programm, das LOL auf die 7-Segment-Anzeigen ausgibt. Beachten Sie, dass Binärzahlen mit dem Postfix 'b' notiert werden, also z.B. 1010b (für dezimal 10).

2.2.1 Unterprogramme, Stack und Stackpointer (SP)

Der Assembler Befehl «`CALL adresse`» bewirkt einen Sprung in ein Unterprogramm, das an der entsprechenden Adresse (Nummer der Speicherzelle) beginnt. Das Unterprogramm wird mit dem Befehl «`RET`» beendet.

Der Stack (auch Stapelspeicher oder Kellerspeicher) ist ein Speicherbereich, in dem Dinge mit «`PUSH`» gespeichert und mit «`POP`» in der umgekehrten Reihenfolge wieder ausgelesen werden können. Konkret zeigt der Stackpointer `SP` auf die nächste freie Speicherzelle. «`PUSH`» schreibt den gewünschten Wert in diese Speicherzelle und vermindert `SP` um eins. «`POP`» erhöht `SP` um eins und liest dann den Inhalt aus.

Der Stack wird insbesondere dafür benutzt, um die Rücksprungadresse bei einem `CALL` Befehl zu speichern. Konkret wird `PC+`x gepusht, wobei x die Grösse vom `CALL` Befehl ist.

Der Stack wird oft auch dazu benutzt, Variablen an ein Unterpogramm zu übergeben oder um Register zu speichern. Typischerweise sieht ein Unterprogramm wie folgt aus:

```
subprg: PUSH A    ;A retten
        PUSH B    ;B retten
        ... Berechnungen, die A, B verändern
        POP B     ;B wiederherstellen (umgekehrte Reihenfolge!)
        POP A     ;A wiederherstellen
        RET       ;Stack muss ausgeglichen sein, sonst falsche Adresse!
```

Wird aus Versehen (oder mit manipulierten Daten absichtlich) der Stack überschrieben, kann eine falsche Adresse als Rücksprungadresse eingetragen werden. Das nutzten Geheimdienste und Cyberkriminelle aus, um eigenen Code auszuführen, der an der neuen Adresse platziert wurde. Heute geht das immer noch, ist aber ein bisschen komplizierter geworden.

Aufgabe 18 Schreiben Sie ein Assembler Programm, das eine beliebige Zahl hexadezimal als Text ausgibt.

Die Zahl soll mit «`zahl: DB 175`» gespeichert und dann mit z.B. «`MOV A,[zahl]`» in ein Register zur weiteren Bearbeitung geladen werden.

Das Textfeld beginnt bei Speicheradresse 232.

Schreiben Sie ein Unterprogramm, das die Hexziffer vom Inhalt von Register C ausgibt und an die Speicheradresse schreibt, die in B steht (indirekte Adressierung). Hier ein Skelett:

```
                ...             ; Richtige Werte in Register B und C schreiben
                CALL ausgabe
                ...             ; Andere Werte in B und C schreiben
                CALL ausgabe
                HLT

ausgabe:        ...             ; ASCII-Code zur Ziffer in C (Wert 0-15) berechnen
                MOV [B],...     ; In die richtige Speicherzelle schreiben.
                RET             ; Rücksprung
```

Aufgabe 19 Schreiben Sie ein Programm, das dezimal hochzählt. Die Einerstelle soll an der Adresse 250 zu stehen kommen.

2.2.2 Arrays

Ein Array von einzelnen Bytes wird als zusammenhängender Speicherbereich aufgefasst. Gespeichert wird die Startadresse vom Array. Wird auf ein Element zugegriffen, wird zur Startadresse der gewünschte Index addiert und auf diese Adresse zugegriffen. In Assembler sieht das wie folgt aus:

```
        ...
        MOV A,feld      ; Arrayadresse in Register A
        ADD A,2         ; Adresse drittes Element
        MOV B,[A]       ; Wert in Register B
        ...
feld:   DB 23
        DB 42
        DB 0xff
        DB 101010b
```

Aufgabe 20 Schreiben Sie ein Programm, das eine beliebige Zahl hexadezimal auf der 7-Segment Anzeige ausgibt. Speichern Sie dazu die einzelnen Ziffern in einem Array.

Aufgabe 21 Schreiben Sie ein Assembler Programm, das eine beiliebige 1-Byte Zahl dezimal ausgibt.

Aufgabe 22 Schreiben Sie ein Unterprogramm, das das Register A mit dem Register B multipliziert und das Resultat in Register C und D ablegt, wobei C das höherwertige Byte des 16-Bit Resultats sein soll.

Aufgabe 23 Es wird ein Assembler Programm geschrieben, das irgendwann garantiert die HLT-Instruktion ausführt und sich damit beendet. Nach wie vielen Schritten geschieht dies spätestens?

Schreiben Sie ein solches Programm und schätzen Sie die Anzahl ausgeführter Schritte für Ihr Programm ab.

2.3 Unicode, UTF-8, Latin1

ASCII reicht nicht für alle Sprachen. Eine (bzw. die) Lösung ist Unicode, wo «jedem» Zeichen eine Nummer zugewiesen wird (neuerdings auch für Emoticons). Damit lassen sich sehr viele Sprachen und Zeichen (z.B. auch mathematische Symbole) darstellen.

Bevor sich Unicode durchgesetzt hat (bzw. durchsetzt, liebe Microsoft) wurde je nach Sprache der ASCII-Code um ein 8. Bit (und damit um maximal 128 Zeichen) erweitert. Für die Schweiz am wichtigsten ist die *Latin1* Codierung. Z.B. müssen Namen in offiziellen Dokumenten mit diesen Zeichen dargestellt werden. In Windows verwenden viele Programme und oft noch das System die Codierung *Windows-1252*, was eine Übermenge von Latin1 ist.

Auf dem Web und in Linux hat sich die UTF8-Codierung durchgesetzt. Das ist eine ASCII-kompatible Codierung für Unicode.

2.3.1 Bitweise Operatoren

Die bitweisen Operatoren operieren je nach Typ auf einem oder mehreren Bytes gleichzeitig.

Name	Python	Beispiel
AND	`&`	`0b1100 & 0b1010 == 0b1000`
OR	`|`	`0b1100 | 0b1010 == 0b1110`
NOT	`~`	`~0b1100'1010 == 0b0011'0101`
Shift left	`<<`	`0b1 << 4 == 0b10000`
Shift right	`>>`	`0b1010 >> 2 == 0b10`

Die **Shift**-Operatoren verschieben die Bits um entsprechend viele Stellen nach links oder rechts. «Herausfallende» Bits gehen verloren. Für binär codierte natürliche Zahlen entspricht dies einer Multiplikation mit oder Division durch eine Zweierpotenz.

Einzelne Bits einer binär codierten Zahl `x` können mit diesen Operatoren extrahiert oder gesetzt werden:

Bit testen	`x & 0b100` oder `(x >> 2) & 1` testet das 3. Bit (Bit Nummer 2).
Bit setzen	`x | 0b100` bzw. `x | (1 << 2)` setzt das 3. Bit (Bit Nummer 2).
Bit löschen	`x & (~(1 << 2))` löscht das 3. Bit (Bit Nummer 2).

Auf diese Art und Weise können natürlich mehr als ein Bit auf einmal extrahiert, gesetzt oder gelöscht werden. Z.B. mit `x & 0b1111` werden die unteren 4 Bits extrahiert.

Aufgabe 24 Studieren Sie die UTF-8-Codierung online (z.B. Wikipedia). Schreiben Sie dann in Python eine Funktion, die zu einer gegebenen Unicodenummer den entsprechenden UTF-8 String generiert.

Benutzen Sie dazu bitweise logische Operatoren wie `&` für AND und `|` für OR, die Shift-Operatoren `<<` und `>>` und die `chr` Funktion, die eine Zahl in ein entsprechendes Byte als String verwandelt. Z.B. ergibt `chr(0xc3)+chr(0xa4)` den String mit dem UTF-8-codierten Zeichen 'ä'.

Testen Sie Ihre Funktion für verschiedene Zeichen, deren Unicode Sie online finden (z.B. Smilies, Pfeile etc.) Da die Konsole von Tiger Jython dafür nichts taugt (sogar auf Linux ist da wohl CP-1252 Codierung), schreiben Sie den String in eine Datei und öffnen Sie die Datei mit einem Browser. Ein Beispielcode dazu ist auf dem Wiki.

2.4 Signed ints

2.5 Floats

2.6 Lösungen

Hinweise zu den Symbolen:
⚒ Diese Aufgaben könnten (mit kleinen Anpassungen) an einer Prüfung vorkommen. Für die Prüfungsvorbereitung gilt: "If you want to nail it, you'll need it".
⚙ Diese Aufgaben sind wichtig, um das Verständnis des Prüfungsstoffs zu vertiefen. Die Aufgaben sind in der Form aber eher nicht geeignet für eine Prüfung (zu grosser Umfang, nötige «Tricks», zu offene Aufgabenstellung, etc.). **Teile solcher Aufgaben können aber durchaus in einer Prüfung vorkommen!**.
🔭 Diese Aufgaben sind dazu da, über den Tellerrand hinaus zu schauen und oder die Theorie in einen grösseren Kontext zu stellen.

Lösung zu Aufgabe 1 ex-wahrheitstabellen

a) OR heisst: «Das eine, das andere oder beides».

b) Z.B. ist `a xor b = (a or b) and (not (a and b))`

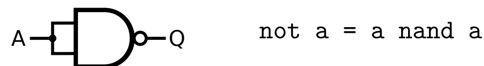
oder `a xor b = (a and (not b)) or (b and (not a))`

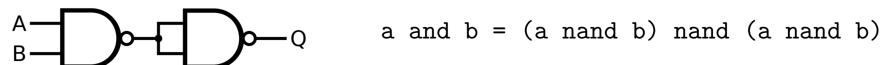
oder `a xor b = (a or b) and ((not a) or (not b))`.

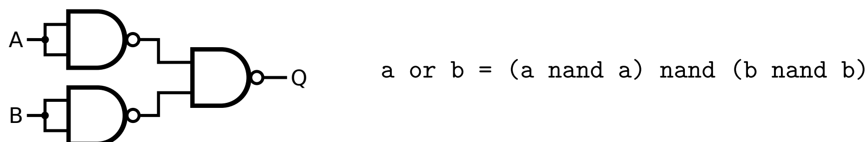
c)

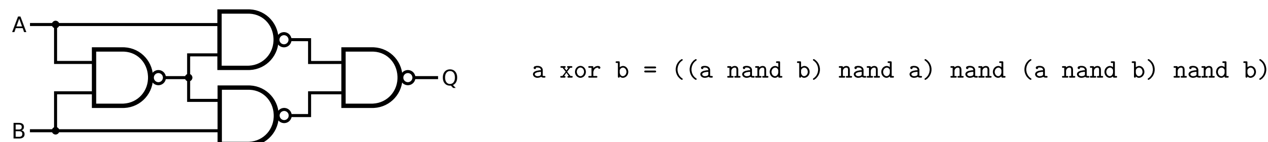
```
def truthTable(op,numvar,title):
    print title
    print "-"*8*(numvar+2)
    for v in range(1 << numvar):
        l=[]
        for i in range(numvar):
            l.append((v>>i & 1)==1)
            print l[-1],
            print "\t",
        print "|\t",op(l)
    print

truthTable(lambda x:not x[0], 1, "A\t|\tnot A")
truthTable(lambda x:x[0] and x[1] ,2," A\tB\t|\tA and B")
truthTable(lambda x:x[0] or x[1], 2, "A\tB\t|\tA or B")
truthTable(lambda x:not (x[0] and x[1]), 2, "A\tB\t|\tA nand B")
```

Lösung zu Aufgabe 2 ex-alles-nand
Es sind auch andere Formeln denkbar und richtig. Folgende finden sich auf den Englischen Wikipedia Seiten zu den entsprechenden Logik-Gattern.









```
def nand(a,b):
    return not(a and b)

def xorFromNand(l):
    return nand(nand(nand(l[0],l[1]),l[0]),nand(nand(l[0],l[1]),l[1]))

truthTable(xorFromNand, 2, "A\tB\t|\tA xor B")
```

Quelle der Grafiken: Wikipedia

Lösung zu Aufgabe 3 ex-umrechnen-binaer-dezimal

a) 0b10=2

b) 10 = 0b1010

c) 0b1000’0001 = 129

d) 1023 = 0b111’1111’1111

e) 0b1’0000’0000 = 256

f) 123 = 0b111’1011

Lösung zu Aufgabe 4 ex-teilbarkeit-verdoppelung-binaer

a) Ist die Einerziffer eine Null, ist die Zahl gerade (alle anderen 2er-Potenzen sind gerade).

b) Die gesamte Zahl wird um eine Stelle nach links verschoben (alle 2er-Potzenzen werden mit 2 multipliziert).

Lösung zu Aufgabe 5 ex-schriftliche-addition

	1	1	0	0		0	1	1	0		0	1	0	1
+		1	1	1		0	1	1	1		0	1	1	0
	1					1	1				1			
1	0	0	1	1		1	1	0	1		1	0	1	1

Lösung zu Aufgabe 6 ex-2048

a) Die Summe aller Kacheln bleibt konstant, abgesehen davon, dass eine Einerkachel hinzukommt. Es sind so also mindestens 2048 Züge nötig (bzw. der entsprechende Bruchteil, je nach Erwartungswert der Summe der erscheinenden Kacheln (z.B. $\frac{1}{6} \cdot 2048$ wenn 2 zufällige Kacheln 2 oder 4 erscheinen).

Allerdings sind nach dem 2047. Zug idealerweise genau 1 Kachel jeder Art bis 1024 übrig. Diese müssen dann in 10 Zügen noch zusammengeschoben werden. Das Minimum ist also 2058.

b) 11.

c) 42 = 0b101010, also eine 32er, 8er und 2er (plus die 1er, die gerade erschien).

d) Die Binärdarstellung von n gibt an, welche Kacheln auf dem Spielfeld sein müssen, um diese Anzahl von Zügen darzustellen.

e) Die Binärdarstellung von $(n-1)$ enthält mindestens 2 konsekutive Einsen, die bei n als Nullen erscheinen. Die den Einsen von $(n-1)$ entsprechenden Kacheln müssen danach in Einzelschritten zusammengeschoben werden.

Lösung zu Aufgabe 7 ex-adder-bauen

a)

a	b	c	s	x
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

Und der Code, der die Tabelle erzeugt hat:

```
def truthTable(ops,numvar,title):
    print title
    for v in range(1 << numvar):
        l=[]
        for i in range(numvar):
            l.append((v>>i & 1)==1)
            print 1 if l[-1] else 0,
            if (i<numvar-1):
                print " & ",
        for op in ops:
            print " & ",1 if op(l) else 0,
        print " \\\\"


# != (not equal) is logic xor for booleans
ops = [lambda x:(x[0] != x[1]) != x[2],
       lambda x: (x[0] and x[1]) or
                 (x[0] and x[2]) or
                 (x[1] and x[2])]
truthTable(ops, 3, "$a$ & $b$ & $c$ & $s$ & $x$ \\\\  \\hline")
```

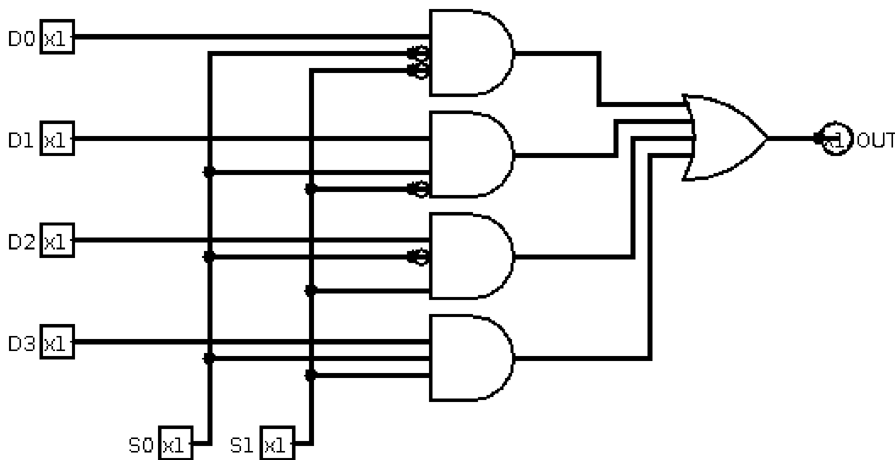
b) $s = a$ xor b xor c

$x = (a$ and $b)$ or $(a$ and $c)$ or $(b$ and $c)$

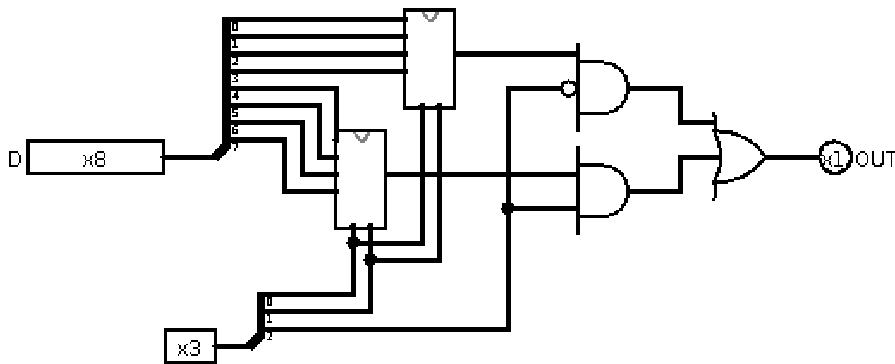
c)

Lösung zu Aufgabe 8 ex-mux-bauen

Hier als Beispiel ein 4 to 1 MUX. Beachten Sie, dass gewisse Eingänge an den AND-Gattern invertiert sind (kleine Kreise vor den Eingängen):

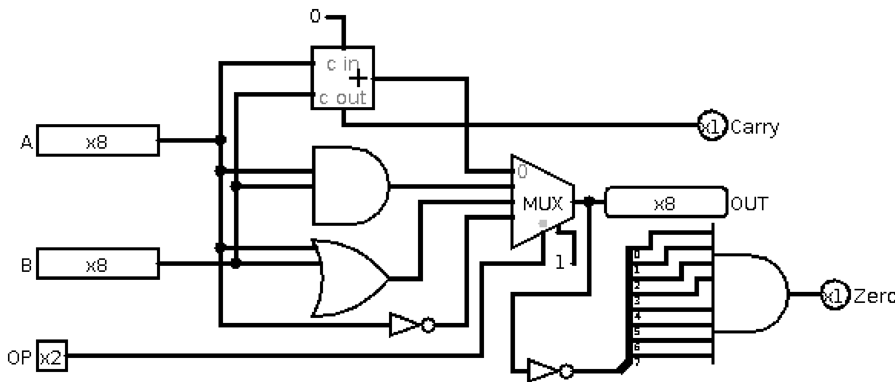


Daraus kann ein 8 to 1 MUX gebaut werden:



Lösung zu Aufgabe 9 ex-pet-alu

Mit OP kann die Operation ausgewählt werden. Die Operationsgatter haben alle 8 Bit breite Ein- und Ausgänge.



Lösung zu Aufgabe 10 ex-signalweg

a) Annahme 2 GHz, d.h. ein Taktzyklus von 0.5 ns $= 5 \cdot 10^{-10}$ s.

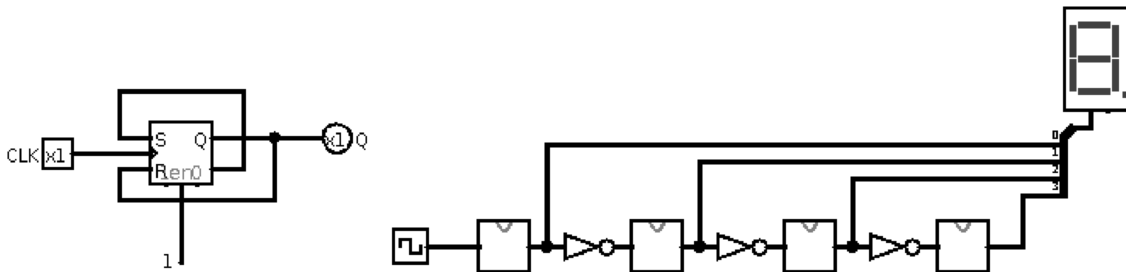
b) Die Lichtgeschwindigkeit im Vakuum ist $c = 3 \cdot 10^8$ m/s. Geht man von einer Signalgeschwindigkeit von $\frac{1}{2}c$ aus, erhält man eine Distanz von
$s = v \cdot t = 1.5 \cdot 10^8$ m /s $\cdot 5 \cdot 10^{-10}$ s $= 7.5 \cdot 10^{-2}$ m $= 7.5$ cm.

c) Ein Full-Adder kann mit 4 Gates gebaut werden, also $4 \cdot 64 = 256$. Soll eine Addition in einem Taktzyklus erledigt werden, müssten die Gatter eine Schaltfrequenz von etwa 500 GHz haben.

Heutige Prozessoren scheinen eine Addition tatsächlich in 1 clock tick zu schaffen, siehe http://www.agner.org/optimize/instruction_tables.pdf

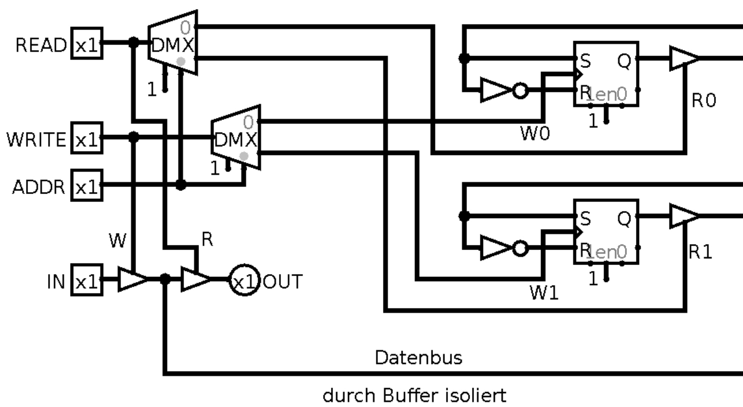
Auch sollen die Signalwege ein grösseres Problem sein als die Schaltgeschwindigkeit der Gatter, die sich im Bereich von 1 THz befinden soll. Eine belastbare Quelle dafür fehlt mir noch, ist durch obige Abschätzung aber plausibel.

Lösung zu Aufgabe 12 ex-counter-bauen

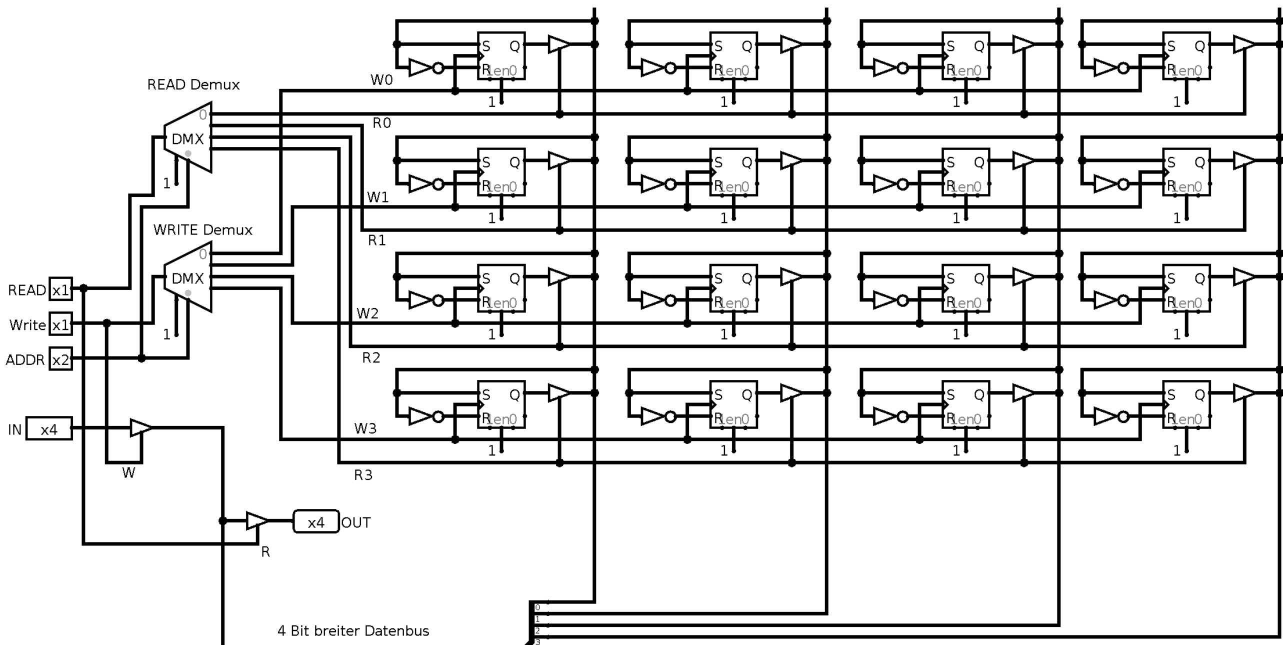


Lösung zu Aufgabe 13 ex-ram-bauen

2-Bit RAM:



16-Bit RAM:



Lösung zu Aufgabe 14 ex-hex-umrechnen

a) 0x2a b) 66 c) 0x5a d) 0xdeadbeef

`hex(zahl)` liefert einen String mit der Darstellung. Eingabe von z.B. `0x42` liefert die dezimale Darstellung.

Lösung zu Aufgabe 15 ex-hex-binaer-umrechnen

4 Bits können eine Zahl zwischen 0 und 15 darstellen, was genau einer Hex-Ziffer entspricht. D.h. für die Umrechnung entspricht jede Hex-Ziffer genau einem 4er-Block in der binären Darstellung und umgekehrt. Ein Umweg über das Dezimalsystem ist unnötig und überkompliziert.

Ein Byte besteht aus 8 Bits (fast überall). Es braucht also *genau* zwei Hex-Ziffern für ein Byte und $0xff=255=2^8-1$.

Lösung zu Aufgabe 17 ex-assembler-lol

Beachten Sie, dass Binärzahlen in diesem Assembler mit einem Postfix 'b' anstatt Präfix '0b' geschrieben werden.

```
MOV [253], 111000b
MOV [254], 111111b
MOV [255], 111000b
```

Lösung zu Aufgabe 18 ex-assembler-hex-ascii

```
        MOV B,232  ; Addresse vom ersten Buchstaben
        MOV C,[zahl] ; Zahl in Register C
        SHR C, 4   ; 4Bits nach rechts verschieben -> 16er-Stelle
        CALL ausgabe
        INC B
        MOV C,[zahl]
        AND C, 0xf   ; untere 4 Bits maskieren -> Einerstelle
        CALL ausgabe
        HLT

        ; gibt Register C hexadezimal an Adresse B aus.
ausgabe:        MOV D,'0'   ; Offset '0'
                CMP C,10   ; Mit 10 vergleichen
                JB ok      ; Wenn kleiner, jump to ok:
                MOV D,87   ; Offset 'a'-10
ok:             ADD D, C   ; Wert und offset addieren
                MOV [B],D  ; in Textausgabe schreiben
                RET        ; Rücksprung

zahl:   DB 0xaf
```

Lösung zu Aufgabe 19 ex-assembler-dezimal-counter

```
; Register A: Adresse, wo hochgezählt werden soll
        MOV A,[base]   ; Addresse auf Einerstelle
        MOV [A],'0'    ; Mit '0' initialisieren

plus:   MOV B,[A]      ; Wert in B
        CMP B,'9'      ; Schon bei '9'?
        JE ueber       ; Falls ja: Übertrag auf nächste Stelle
        INC B          ; Sonst B erhöhen
        MOV [A],B      ; eintragen
        MOV A,[base]   ; auf Einerstelle setzen
        JMP plus       ; und von vorne erhöhen
```

```
ueber:  MOV [A], '0'  ; aktuelle Stelle wird Null
        DEC A         ; Eine Stelle nach links
        MOV B,[A]     ; Inhalt der Stelle in B
        CMP B,'0'     ; Testen, ob schon eine Ziffer dort
        JAE plus      ; Wenn ja, erhöhen (an aktueller Adresse)
        MOV [A],'1'   ; Sonst eine 1 schreiben.
        MOV A,[base]  ; auf Einerstelle setzen
        JMP plus      ; und von vorne erhöhen

base:   DB 250        ; Adresse der Einerstelle
```

Lösung zu Aufgabe 20 ex-assembler-hex-7segment

```
        JMP start
zahl:   DB 175
digits: DB 111111b  ;0
        DB 110b     ;1
        DB 1011011b ;2
        DB 1001111b ;3
        DB 1100110b ;4
        DB 1101101b ;5
        DB 1111101b ;6
        DB 111b     ;7
        DB 1111111b ;8
        DB 1101111b ;9
        DB 1110111b ;A
        DB 1111100b ;b
        DB 0111001b ;C
        DB 1011110b ;d
        DB 1111001b ;E
        DB 1110001b ;F

start:  MOV B,254
        MOV C,[zahl]   ; Zahl in Register C
        SHR C, 4       ; 4Bits nach links verschieben -> 16er-Stelle
        CALL ausgabe
        INC B
        MOV C,[zahl]
        AND C, 0xf      ; untere 4 Bits maskieren -> Einerstelle
        CALL ausgabe
        HLT

; gibt Register C hexadezimal an Adresse B aus.
ausgabe:        MOV D,digits
                ADD D,C
                MOV D,[D]
                MOV [B],D
                RET
```

Lösung zu Aufgabe 21 ex-assembler-dezimal-ausgabe

Die Idee ist, die Zahl durch 10 zu dividieren, der Rest ergibt die Einerstelle. Mit dem Resultat wird weitergefahren, bis das Resultat Null ist.

```
        JMP start
zahl:   DB 242
out:    DB 250 ; Adresse fuer Ausgabe
```

```
start:  MOV D,[zahl]
loop:   CMP D,0         ; Zahl nicht Null?
        JNE  weiter      ; Weiter machen
        HLT             ; Sonst ENDE
weiter: MOV B,D         ; Zahl in A
        CALL divr       ; Dividieren
        ADD B,'0'       ; Zum Rest (Einerstelle) ASCII 48 addieren
        MOV A,[out]     ; Ausgabe Adresse holen
        MOV [A],B       ; Ziffer schreiben
        DEC A           ; Ein Position nach links
        MOV [out],A     ; In out speichern
        MOV D,C         ; Ergebnis der Division in D
        JMP loop        ; Und weiter geht's

        ;Division B/10 gibt C Rest B, verändert A
divr:   MOV A,B
        DIV 10          ; A/10
        MOV C,A         ; A/10 in C
        MUL 10          ; (A/10)*10
        SUB B,A         ; Differenz ist REST in B
        RET
```

Lösung zu Aufgabe 22 ex-assembler-multiplikation

```
        MOV A,123
        MOV B,145
        CALL mymul
        HLT



;INPUT A,B OUTPUT C,D (bigendian)

; [A1,A0] * [B1,B0] =
;              A0*B0
;           A1*B0
;           A0*B1
;        A1*B1
;      --------------
;        C1,C0|D1,D0


mymul:
        MOV C,0
        MOV D,0
        PUSH A
        PUSH B
        AND A, 0xf    ; COMPUTE A0*B0 -> [D1,D0]
        AND B, 0xf
        MUL B
        MOV D,A
        MOV A,[SP+2]   ; Restore A
        SHR A,4       ; COMPUTE A1*B0 -> [C0,D1]
        CALL middleMul
        MOV A,[SP+2]  ; Restore A,B
        MOV B,[SP+1]
```

```
        AND A, 0xf   ; Compute A0*B1
        SHR B, 4
        CALL middleMul
        MOV A,[SP+2] ; Restore A,B
        MOV B,[SP+1]
        SHR A,4
        SHR B,4
        MUL B
        ADD C,A
        POP B
        POP A
        RET


middleMul:
        MUL B
        PUSH A   ; Save result
        SHL A,4  ; Get D1 component
        ADD D,A  ; Add it
        JNB noC0 ; no carry
        ADD C,1 ; add carry to C0
noC0:   POP A   ; restore A
        SHR A,4
        ADD C,A
        RET
```

Lösung zu Aufgabe 23 ex-assembler-longest-running-program

Die Anzahl theoretisch möglicher Zustände des Assemblersimulators lässt sich genau angeben. Diese ergibt sich aus der Grösse des Speichers (256 Bytes), der Register (6 Bytes) und der Status Flags (3 Bits). Das ergibt total $256 \cdot 8 + 6 \cdot 8 + 3 = 2099$ Bits. D.h. die maximale Anzahl Zustände beträgt $2^{2099} \approx 10^{632}$. Da das Programm einmal halten muss, darf sich der Zustand nie wiederholen (das käme einer Endlosschleife gleich). Das Programm hält also nach allerspätestens 2^{2099} Schritten.

Praktisch liegt diese Grenze noch einiges tiefer, da z.B. immer gültige Befehle im Speicher stehen müssen.

Hier mal ein Beispiel, das 22 Bytes für den Programmcode benötigt. Der restliche Speicher wird im 256er-System hochgezählt (Einerstelle an Adresse 255).

```
start:  MOV A,255       ;Adresse
plus:   MOV B,[A]       ;Speicherzelle hochzählen
        ADD B,1         ;Zum Testen, hier 0x80 addieren.
        MOV [A],B
        JNB start       ;Wenn kein Überlauf, weiterzählen
        DEC A           ;Nächste Stelle
        CMP A,ende      ;Alles voll? Zum Test, hier mit 0xfb vergleichen
        JNE plus        ;Wenn nicht, nochzählen
ende:   HLT             ;Danach beginnt der freie Speicher
```

Das Hochzählen braucht mindestens 5 Schritte. Hochgezählt wird bis $2^{8\cdot(256-22)} = 2^{1872} \approx 10^{563}$. Zum Vergleich: Das Alter des Universums wird auf ca. $5 \cdot 10^{27}$ ns geschätzt (ja Nanosekunden, so lange dauert in etwa eine Operation auf heutigen Prozessoren). Selbst wenn jedes Elementarteilchen im Universum (Anzahl auf etwa 10^{86} geschätzt) mit der gesamten irdischen aktuellen Rechenkapazität (geschätzt auf 10^{21} Flops, «floating point operation per second») rechnen würde, kämen «nur» 10^{107} Operationen pro Sekunde zusammen. Das obige Programm würde immer noch 10^{456} Sekunden dauern, d.h. in etwa das 10^{438}-fache des Alters des Universums.