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`.

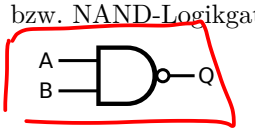
Aufgabe 1 Ein Python-Code zur Erzeugung von Wahrheitstabellen befindet sich auf dem Wiki. Kopieren Sie diesen Code und führen Sie diesen aus.

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):



A, B → NAND → A nand B

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

not A = A nand A

A and B = (A nand B) nand (A nand B)

A or B = (A nand A) nand (B nand B)

A	B	C
0	0	0
1	0	0
0	1	0
1	1	0
0	0	1
0	1	1
1	0	1
1	1	1

0 False
1 True

XOR

A	B	A XOR B
0	0	0
1	0	1
0	1	1
1	1	0

A xor B = ...

(A or B) and (not (A and B))

truthtable.py

```
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");
```

Funktion

$= 2^{\text{numvar}}$ 100

liefert True od. False

Liste True/False

Tabulator → 8 Leerschläge

1.3 Darstellung von natürlichen Zahlen und Adder

Wie funktioniert das Zehnersystem?

$42'062 =$ ✎ $2\cdot 10^0 + 6\cdot 10^1 + 0\cdot 10^2 + 2\cdot 10^3 + 4\cdot 10^4$

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

Das Zweiersystem funktioniert analog:

0b101010 = ✎ $0\cdot 2^0 + 1\cdot 2^1 + 0\cdot 2^2 + 1\cdot 2^3 + 0\cdot 2^4 \cdot 1\cdot 2^5 = 42$

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.

halbes Byte = eine Hexziffer

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 → 1111011 bin(123)

Aufgabe 4

a) Wie sieht man einer Binärzahl an, ob diese durch 2 teilbar ist? → 1er Stelle ist 0 → 2^0

b) Was geschieht mit einer Binärzahl, wenn diese mit 2 multipliziert wird? 1 Stelle nach links schieben

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? 2048 (+11)

b) Was ist $\log_2(2048)$? = 11 → 42 x 1 erschienen.

c) Welche Kacheln sind nach 42 Zügen idealerweise noch übrig? 32 8 2

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

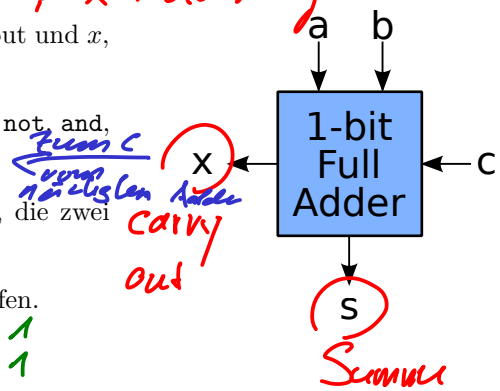
e) Was gilt für $n-1$ und n wenn nach n Zügen unmöglich die theoretische minimale Anzahl Kacheln erreicht werden kann? n-1 = 01... 11... n = 01... 00...

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. s: Summe, x: Übertrag

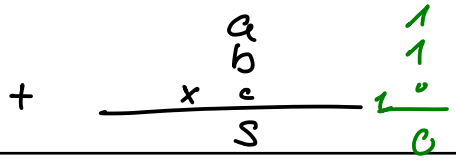
a) Erstellen Sie von Hand eine Wahrheitstabelle mit a, b, c als Input und x, s als Output.

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.

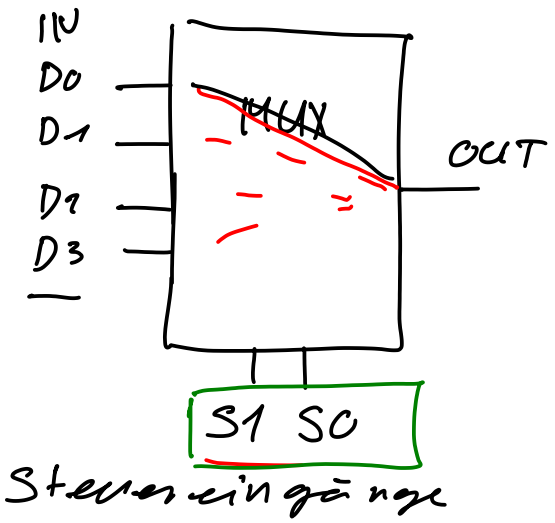


```
  1100  0110  0101
+  111  0111  0110
 11    11    10
10011  1101  1011
```

MUX

n = 2

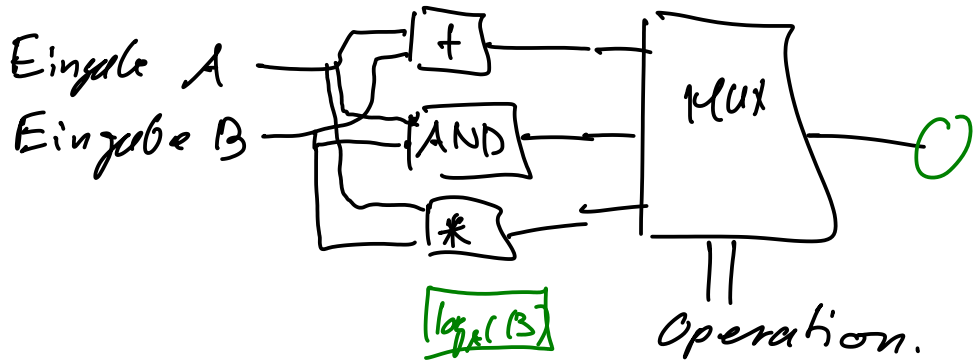
Daten-Eingänge



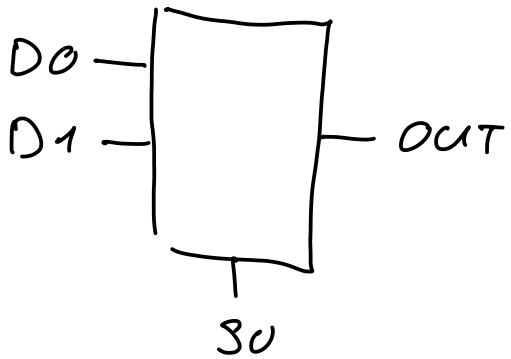
Steuereingänge

S1	S0	OUT
0	0	D0
0	1	D1
1	0	D2
1	1	D3

ALU Arithmetic and Logic Unit



2 to 1 MUX



D0	D1	S0	OUT
0	0	0	0
1	0	0	1
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	0
0	1	1	1
1	1	1	1

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

Auf Donnerstag

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 $c = 3 \cdot 10^8$ m/s

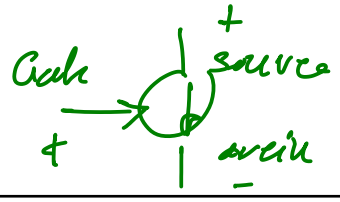
a) Bestimmen Sie die Taktfrequenz Ihres Laptop-Prozessors. ~2 GHz

1 clock tick 0.5 ns

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?

500 Gatter → ~1 THz
1 ps



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).

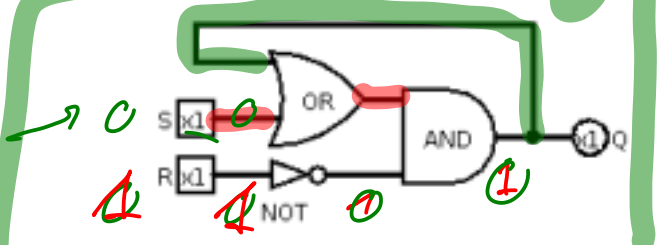
RAM Arbeitsspeicher. Eher langsam (in clock ticks gemessen, so ca. 10 ns). Einige GB.

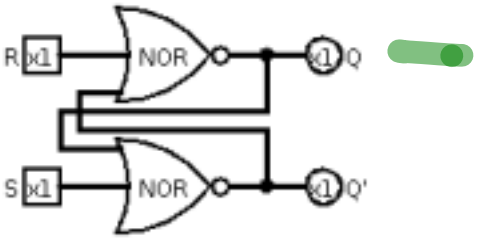
Externer Speicher Harddisks (10-15 ms), SSD (0.1 ms), Netzwerkspeicher (1 ms bis 50 ms?). Sehr langsam (für die CPU eine Ewigkeit). Kapazität einige TB.

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.

RGB - Display 10 x 15

LED - Streifen	15.–	12
Netzteil	15.–	12
Holz	10.–	6
Cutting	5.–	5
Arduino	5.–	5
Platine, Knöpfe, Kabel	5.–	5
Widerstände	–	–
	55.–	45.–

Joystick, ~~MP3~~ MP3 - Chip.

~~Bluetooth~~ Bluetooth

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.

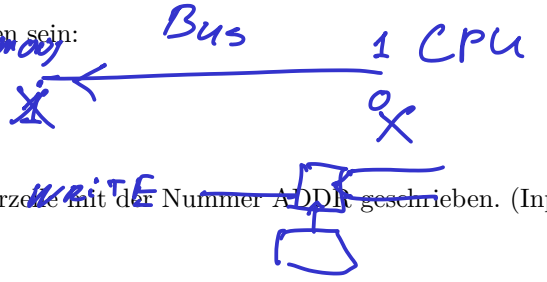
1.6.4 Three-state Logic

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 Segment 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).

$42 = 2 \cdot 16^1 + 10 \cdot 16^0$

$= 0x2a$

$0x2a$

$0b\ 10'1010$

$0xb = 11 = 0b\ 1011$

1000b

$0x42 = 2 \cdot 16^0 + 4 \cdot 16^1 =$

$2 + 64 = 66$

$176 = 11 \cdot 16^1 + C \cdot 16^0$

$0xbC$

$160 + 16 = 0xa0 + 0x1C$

$2738 = 8 \cdot 10^0 + 3 \cdot 10^1 + 7 \cdot 10^2 + 2 \cdot 10^3$

$0x17 = 7 \cdot 16^0 + 1 \cdot 16^1 =$

$7 + 16 = 23$

$0x16 = 6 \cdot 16 + 16 = 22$

---

c) $0b\ 101'1010 = 0x5a = 90$

d) $hex(3735928559) =$

$0xdeadbeef =$

$0b1101'1110'1010'1101'1011'1110'1110'1111$ (e, f)

$0x42 =$

$0b0100'0010$

Farben: r g b

#ffff00

Binär → Dezimal im Computer

06 1001'1101 von Hand 128 + 16 + 8 + 4 + 1

2917 : 12 = 243 Rest 1

24
51
48
37
1 Rest

2917 : 10 = 291 Rest 7

157 10 15 7

1001'1101 : 1010 = 1111 Rest 111

1 1010 1
0100 11
1010
10010
1010
01001
1010
0111

Einerstelle
7

1111 : 1010 = 1 Rest 101
1010
0101

Zehnerst. 5

1 : 1010 = 0 Rest 1

157

Pseudo-Code

Eingabe: Zahl $z \in \mathbb{N}_0$ (binär)
Ausgabe: Ziffern der dezimalen Darstellung von z
start: wenn z == 0 : Ausgabe '0', fertig
so lange wie z > 0 wiederhole:
    r = z % 10 (Rest durch 10)
    Ausgabe '0' + r (ASCII-Code der Ziffer)
    z = z / 10 (Ganzzahl Division)

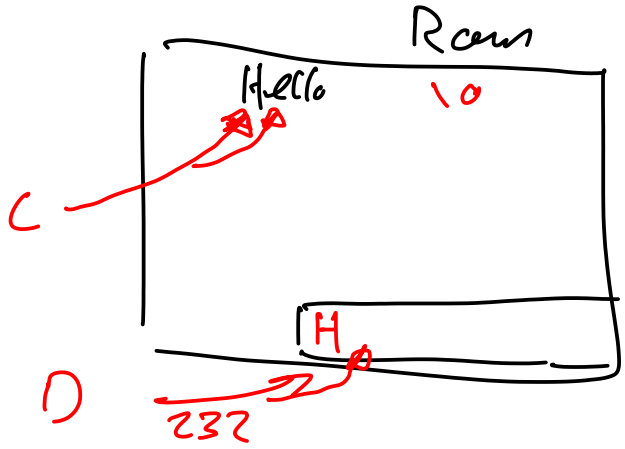
```
; Simple example
; Writes Hello World to the output

        JMP start
hello: DB "Hello World!" ; Variable
       DB 0  ; String terminator

start:
       MOV C, hello     ; Point to var
       MOV D, 232       ; Point to output
       CALL print
       HLT              ; Stop execution

print:                  ; print(C:*from, D:*to)
       PUSH A
       PUSH B
       MOV B, 0
.loop:
       MOV A, [C]       ; Get char from var
       MOV [D], A       ; Write to output
       INC C
       INC D
       CMP B, [C]       ; Check if end
       JNZ .loop  ; jump if not

       POP B
       POP A
       RET
```



B=0

A='H'

A18 Zahl and 011111 → Einerstelle

Contents

Thumbnails

Reviews

Bookmarks

Contents

Bits...
Ass...
H...
A...
U...
Si...
Fl...
8

Lösung zu Aufgabe 18 ex-assembler-hex-ascii

C-15

0-9 → Ziffer

ASCII 48...

10-15 → a-f

ASCII 97-

```
        MOV B,232   ; Addresse vom ersten Buchstaben
        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,'0'   ; Offset '0' = = 48
            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
```

rechts

Michelle

'0' = = 48 = = 0x30

0 f

1010'1111 und 0x'1111

a f

Lösung zu Aufgabe 19

Michelle