Grundlagen der Informatik
VO und KV

Programmierung

Überblick

- **Programmiersprachen**
- **“Hello World” in C**
- **Programmierung - Abstraktion**
- **Kompilieren von C-Programmen mit GNU**
- **Von-Neumann-Architektur, CPU und Register**
- **C vs. Assembler**
- **(Schleifen, Bedingungen, Variabel, Bit-Operationen in C)**

Programmiersprachen - Algorithmus - Compiler

- Programmiersprachen sind künstliche, also nicht natürliche, Sprachen, die formal spezifiziert sind. Das Verhalten einer Rechenmaschine kann durch ein syntaktisch korrektes Programm einer solchen Sprache gesteuert werden.

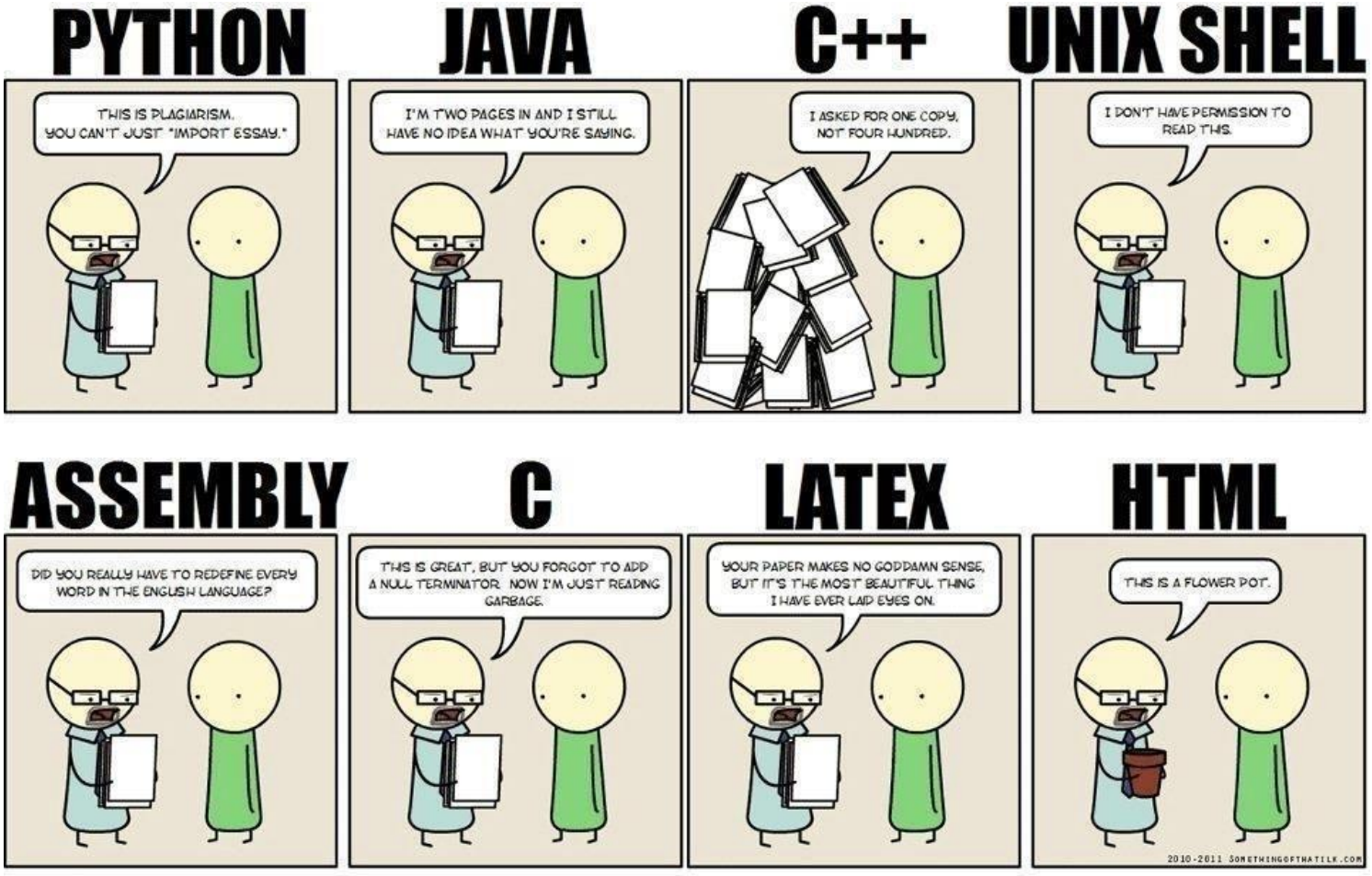
- Ein Algorithmus - die schrittweise Anleitung zur Lösung eines Problems - kann in einer Programmiersprache notiert werden, damit dieser dann automatisiert von unserem Computer ausgeführt werden können.

- Compiler (Computerprogramme) ermöglichen es, dass Programme nicht als direkte Instruktionen in eine CPU (Prozessor) geschrieben werden, sondern ermöglichen es einen Algorithmus in einer höheren Sprache - einer Programmiersprache - zu schreiben (Abstraktion)

Lukas Prokop: Grundlagen der Informatik, Skriptum, S.61-64. http://gdi.ist.tugraz.at/skriptum.pdf .

Eigenschaften von Algorithmen

Platzkomplexität	…	Speicher ?
Laufzeit / Zeitkomplexität	…	wie lange?
Determinismus	…	kommt bei gleichem Input immer gleicher Output?
Strategie	…	heuristische Verfahren, Divide and Conquer, Dynamic Programming...



https://www.pinterest.at/pin/570831321499698240/

https://de.overleaf.com/learn/latex/Articles/LaTeX_is_More_Powerful_than_you_Think_-_Computing_the_Fibonacci_Numbers_and_Turing_Completeness

Die Programmiersprache “Brainfuck”

Beispielprogramme in Brainfuck [Bearbeiten | Quelltext bearbeiten]

Hello World! [Bearbeiten | Quelltext bearbeiten]

Das folgende Programm gibt „Hello World!“ und einen Zeilenumbruch aus.

```
++++++++++
[
 >+++++++>++++++++++>+++>+<<<<-
]                       Schleife zur Vorbereitung der Textausgabe
>++.                    Ausgabe von 'H'
>+.                     Ausgabe von 'e'
+++++++.                'l'
.                       'l'
+++.                    'o'
>++.                    Leerzeichen
<<+++++++++++++++.      'W'
>.                      'o'
+++.                    'r'
------.                 'l'
--------.               'd'
>+.                     '!'
>.                      Zeilenvorschub
+++.                    Wagenrücklauf
```

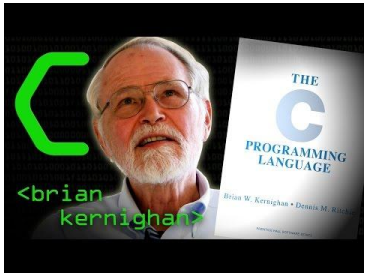
Zur besseren Lesbarkeit ist dieser Brainfuckcode auf mehrere Zeilen verteilt und kommentiert worden. Brainfuck ignoriert alle Zeichen, die keine Brainfuckbefehle sind. Alle Zeichen mit Ausnahme von `+-<>[],.` können deswegen zur Kommentierung des Quellcodes genutzt werden.

https://de.wikipedia.org/wiki/Brainfuck
http://www.bf.doleczek.pl/

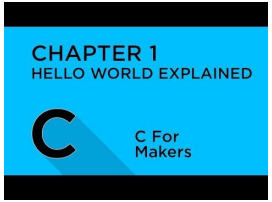
Die Programmiersprache **C**

- **C** ist eine imperative und prozedurale Programmiersprache
- Maschinennah: viele Operationen werden unmittelbar in Maschinenbefehle überführt
- Volles Potenzial des Rechners nutzbar.
  Das Hauptgewicht liegt auf Performance (und ursprünglich Einfachheit des Compilers) und nicht auf “Überwachung”.
- Sprache in der Hardware-Treiber / Betriebssysteme programmiert werden
- Verzeiht nicht so leicht Fehler, man muss alles „selber machen”
  (dafür kann man auch hinter die Kulissen schauen)

https://de.wikipedia.org/wiki/C_(Programmiersprache)



Mein erstes C-Programm




```c
/*    HelloWorld.c    */
#include <stdio.h>

int main(){
  printf("Hello World \n");
  /* "\n" - new line  */
 return 0;
}
```

`include`

sagt dem Compiler, dass weiterer Code importiert werden soll.

`<stdio.h>`

Die Bibliothek stdio.h (Standard Input/Output): Verarbeitung von In- und Output (Keyboard); oder die Ausgabe am Bildschirm.

`#`

Vor dem `include`. Sagt dem Compiler, dass dieser Code vorher verarbeitet werden muss (pre-processing)

```c
/* ich bin ein Kommentar */
```

https://medium.com/backticks-tildes/the-simplest-c-program-explained-in-detail-756ddca208ca

```c
/*    HelloWorld.c    */
#include <stdio.h>

int main(){
  printf("Hello World \n");
  /* "\n" - new line  */
 return 0;
}
```

Online Editor für C-Programme:
https://www.onlinegdb.com/online_c_compiler

`int` Datentyp des Rückgabewerts einer Funktion

`main()` Aufruf der Funktion “main” In runde Klammern kommen die Argumente für eine Funktion.

`{}` Body des Programms

`printf("Hello World \N”);` Aufruf der Funktion aus stdio.h. Ausgabe am Bildschirm. In “ “ steht etwas vom Datentyp String.

`return 0;` Rückgabe der Funktion

https://medium.com/backticks-tildes/the-simplest-c-program-explained-in-detail-756ddca208ca

Ein Programm in Maschinencode

```
00110001 00000000 00000000
00110001 00000001 00000001
00110011 00000001 00000010
01010001 00001011 00000010
00100010 00000010 00001000
01000011 00000001 00000000
01000001 00000001 00000001
00010000 00000010 00000000
01100010 00000000 00000000
```

https://eloquentjavascript.net/00_intro.html

Ein Programm in Maschinencode

```
1. 00110001 00000000 00000000
2. 00110001 00000001 00000001
3. 00110011 00000001 00000010
4. 01010001 00001011 00000010
5. 00100010 00000010 00001000
6. 01000011 00000001 00000000
7. 01000001 00000001 00000001
8. 00010000 00000010 00000000
9. 01100010 00000000 00000000
```

Verbalisierung

1. Store the number 0 in memory location 0.
2. Store the number 1 in memory location 1.
3. Store the value of memory location 1 in memory location 2.
4. Subtract the number 11 from the value in memory location 2.
5. If the value in memory location 2 is the number 0, continue with instruction 9.
6. Add the value of memory location 1 to memory location 0.
7. Add the number 1 to the value of memory location 1.
8. Continue with instruction 3.
9. Output the value of memory location 0.

https://eloquentjavascript.net/00_intro.html

Abstraktion!

Set “total” to 0.

Set “count” to 1.

[loop]

Set “compare” to “count”.

Subtract 11 from “compare”.

If “compare” is zero, continue at [end].

Add “count” to “total”.

Add 1 to “count”.

Continue at [loop].

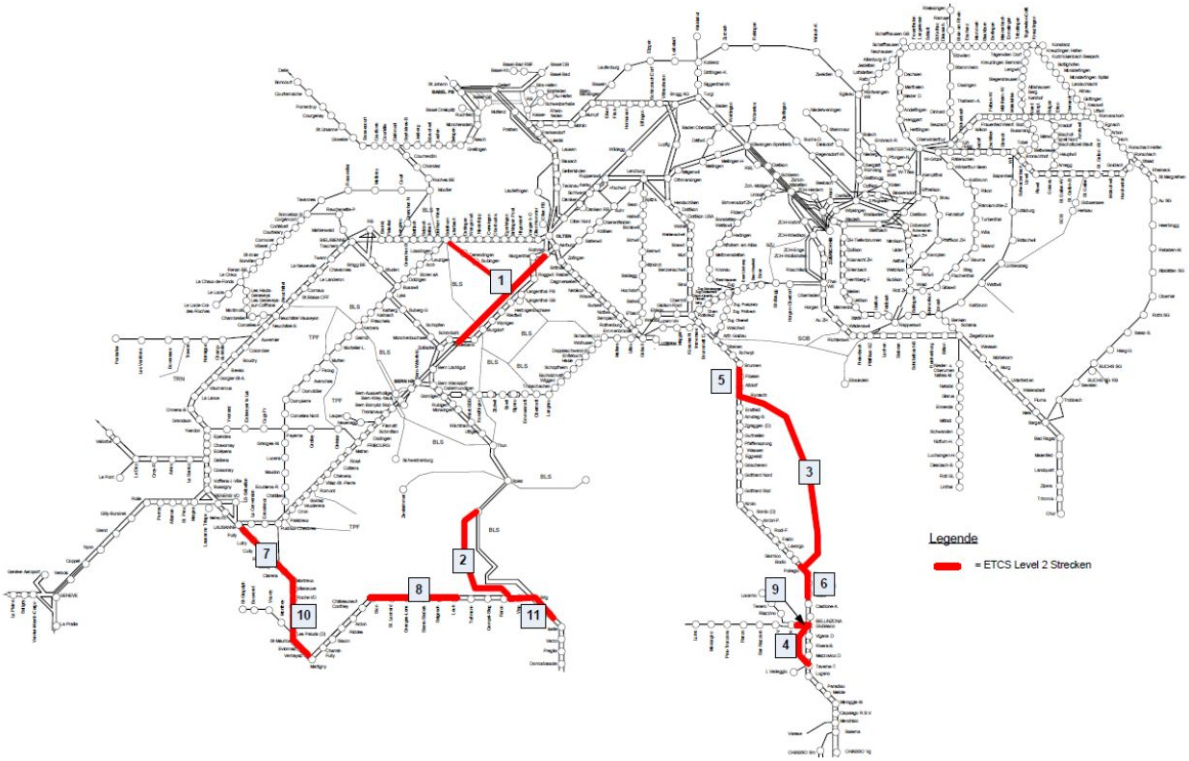
[end]

Output “total”.

C-Programm

```c
#include <stdio.h>

int main(){

  int total = 0;
  int count = 1;

  while (count <= 10) {
    total = total + count;
    count = count + 1;
  }

  printf("%d",  total);
  return 0;

}
```

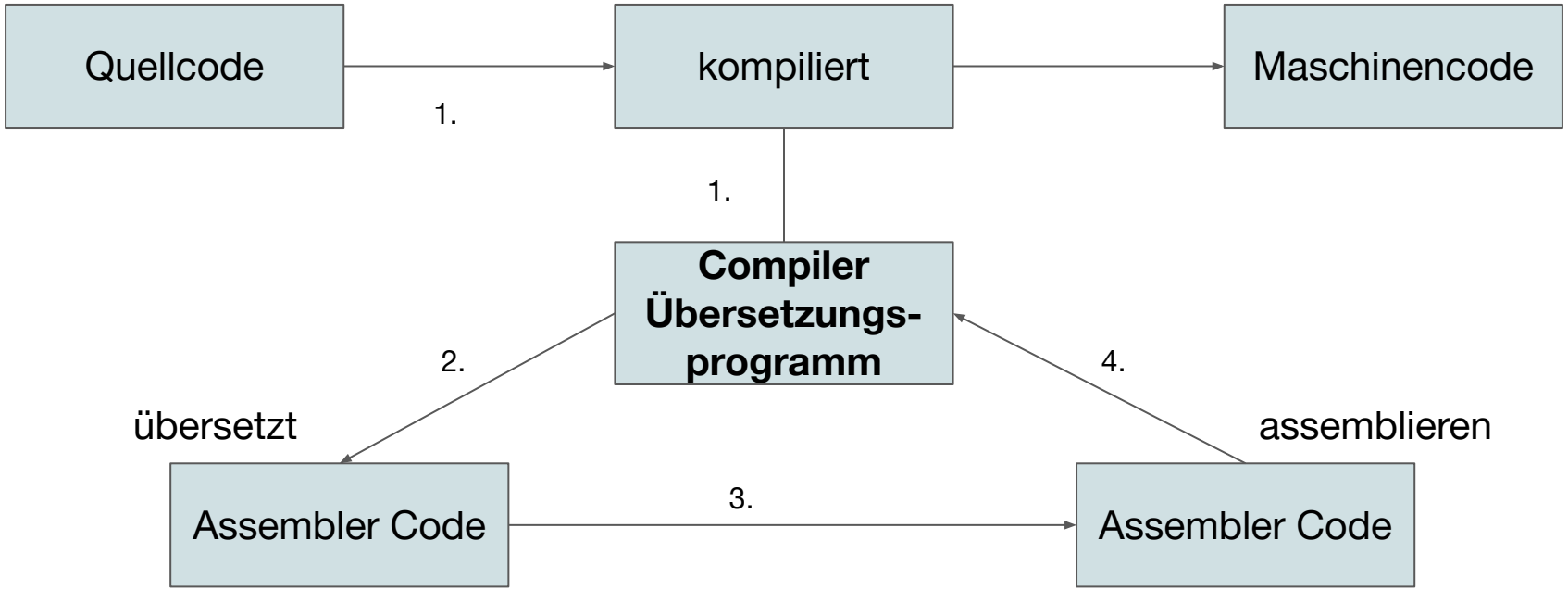
Abstraktion



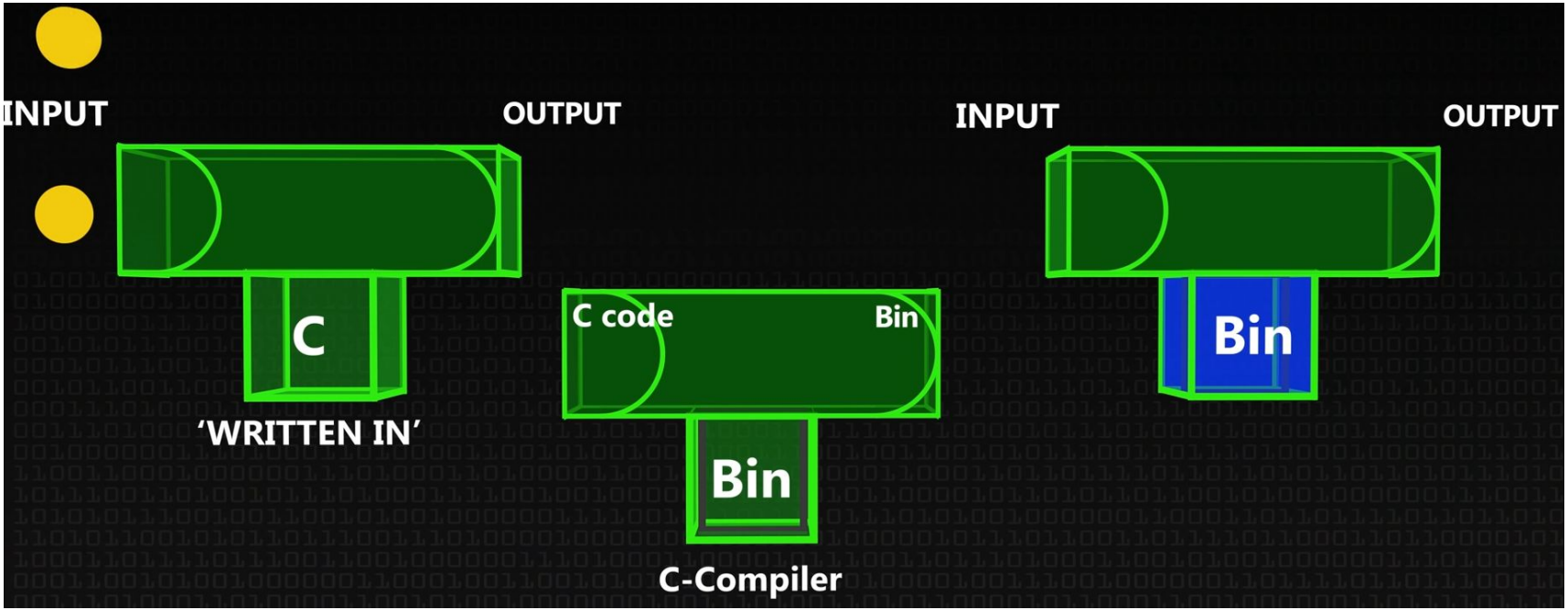
http://gdi.ist.tugraz.at/skriptum.pdf

Compiler

Programm, das Quellcodes einer Programmiersprache in eine Form übersetzt, die von einem Computer (direkt) ausgeführt werden kann.



T-Diagrams



https://www.youtube.com/watch?v=PjeE8Bc96HY&list=PLi2jUrtr82FL91OhCnAebNKgOKxR270kq

Compiler

Lexikalische Analyse

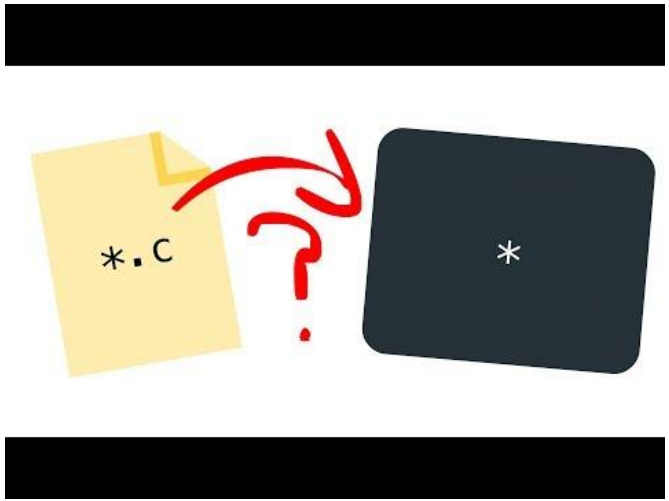
Syntaktische Analyse

Semantische Analyse

Zwischencodeerzeugung

Programmoptimierung

Codegenerierung



Ab 3:40

https://de.wikipedia.org/wiki/Compiler

Wir kompilieren unser C - Programm

`helloworld.c`

```c
#include <stdio.h>

int main(){
printf("Hello World \n");
  /* "\n" - new line  */
 return 0;
}
```

`helloworld.i`

- Kommentare weg
- Macros und Bibliotheken stdio.h expandiert
- Conditional compilation
- etc.

`helloworld.s`

- Assembler Code

`helloworld.o` / `helloworld`

- `helloworld.o` Maschinensprache (kann man mit einem Hex-Editor öffnen)
- `helloworld` Executable file

Quellcode → **Preprocessing** → **Compiling** → **Assembly / Linking**

https://www.geeksforgeeks.org/compiling-a-c-program-behind-the-scenes/

- **`gcc -Wall filename.c -o filename`**
  Kompiliert eine File mit der GNU Compiler Collection (gcc)
  Zeigt uns alle Warnings an (-Wall)
  Nimmt den Quellcode filename.c und macht daraus das ausführbare File mit dem Namen filename.

- **`gcc -Wall -save-temps filename.c -o filename`**
  Gibt uns zusätzlich die .i , .s und .o Files

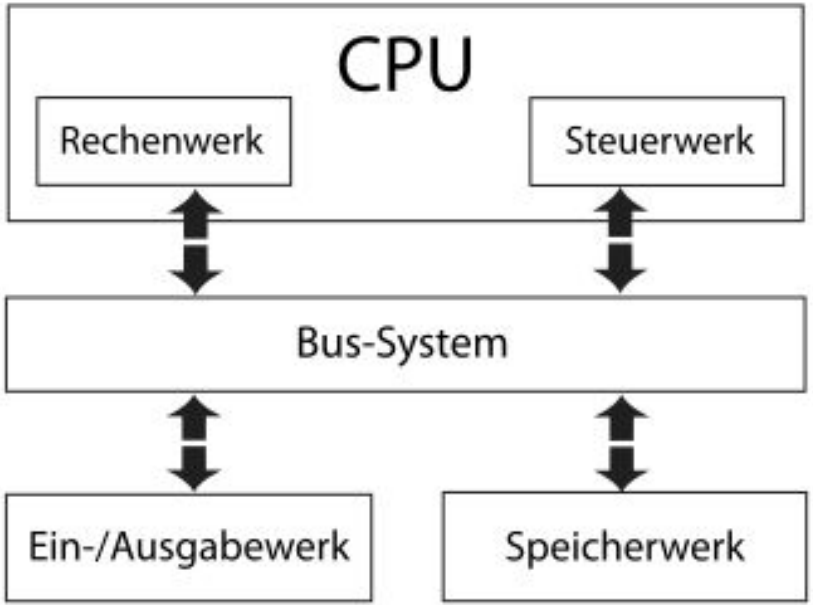
- **`./filename`**
  Führt das Programm aus: der Text im printf wird ausgegeben

```
pollin@pollinEliteBook-820-G3:~/Documents/Lehre/GDI_KV_WS18_19/Scripts/helloWolrd$ gcc -Wall -save-temps helloworld.c -o helloworld
pollin@pollinEliteBook-820-G3:~/Documents/Lehre/GDI_KV_WS18_19/Scripts/helloWolrd$ gcc -Wall helloworld.c -o helloworld
pollin@pollinEliteBook-820-G3:~/Documents/Lehre/GDI_KV_WS18_19/Scripts/helloWolrd$ ls
helloworld  helloworld.c  helloworld.i  helloworld.o  helloworld.s
pollin@pollinEliteBook-820-G3:~/Documents/Lehre/GDI_KV_WS18_19/Scripts/helloWolrd$ ./helloworld
pollin@pollinEliteBook-820-G3:~/Documents/Lehre/GDI_KV_WS18_19/Scripts/helloWolrd$ ./helloworld
Hello World
pollin@pollinEliteBook-820-G3:~/Documents/Lehre/GDI_KV_WS18_19/Scripts/helloWolrd$
```

Assembler

- Assembler Sprachen habe ein 1:1 Mapping zum Maschinencode
- Maschinencode ist eine Folge von 0en und 1en, die im Prozessor abgearbeitet werden.
- **Grace Hopper** hat **A-0**, die erste Assembler-Sprache entwickelt. [https://de.wikipedia.org/wiki/Grace_Hopper]
- Eine Zeile in einer höheren Programmiersprache kann zu vielen Instruktionen in Assembler führen, die auf der CPU dann ausgeführt werden.

Von-Neumann-Architektur



John von Neumann

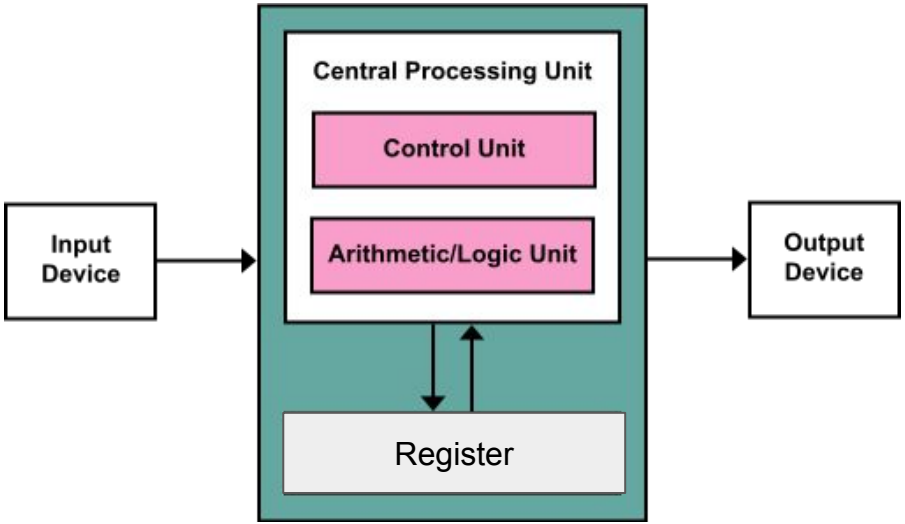
- https://de.wikipedia.org/wiki/Von-Neumann-Architektur
- https://de.wikipedia.org/wiki/John_von_Neumann
- Assembler Tutorial #3 - Der Von Neumann Rechner. https://www.youtube.com/watch?v=P013kCzh4TA

CPU - Central Processing Unit

- CPU ≡ Central Processing Unit ≡ Prozessor ≡ Zentrale Verarbeitungseinheit
- Ist zentraler Bestandteil des Computers: “***Kopf des Computers***”.
- Dient der **Verarbeitung von Daten**, die sich als Bytes im Speicher des Rechners befinden.
- CPU besteht aus **Registern, Rechenwerk** (Arithmetic Logical Unit, ALU), einem **Steuerwerk** (Control Unit, CU) und den **Datenleitungen** (Bus).
    - **ALU** führt **logische und mathematische Operationen** durch.
    - **Register** sind extrem schnelle **Hilfsspeicherzellen**, die mit ALU verbunden sind.
    - **CU** steuert den **Ablauf der Befehlsverarbeitung**.
    - Bus sind Systeme zur Datenübertragung

Gumm, H. P., Sommer, M. (2012). Einführung in die Informatik, S.44-49.

Ablauf eines Befehls



1. **Befehl laden** und mit Befehlsregister abgleichen.
2. Benötige **Daten** werden in Register **geladen**.
3. In der ALU werden **Daten verarbeitet**. Einfache arithmetische und logische Operationen.
4. **Daten werden zurückgeschrieben** entweder in das Register oder in ein Output Device

Joch J., Alnawa T., Guhl H. Computeranatomie - Ein Blick ins Innere deines Rechners, https://open.hpi.de/courses/computeranatomie2021

Register

- Bestehen aus Logik-Gattern. In sogenannten Latches kann 1 Bit gespeichert werden.
- Bringt man mehrere Latches zusammen, hat man ein Register.
- Registers sind Speicherorte für Daten für die CPU.
- Die Größe der Register definiert die Systemarchitektur:

  64bit / 32bit / 16bit / 8bit

  → ein 8-Bit-Register besteht aus 8 Speicherzellen, die jeweils genau 1 Bit speichern können.
- Auf die Register kann man sehr schnell zugreifen und darin passieren arithmetische und logische Operationen.

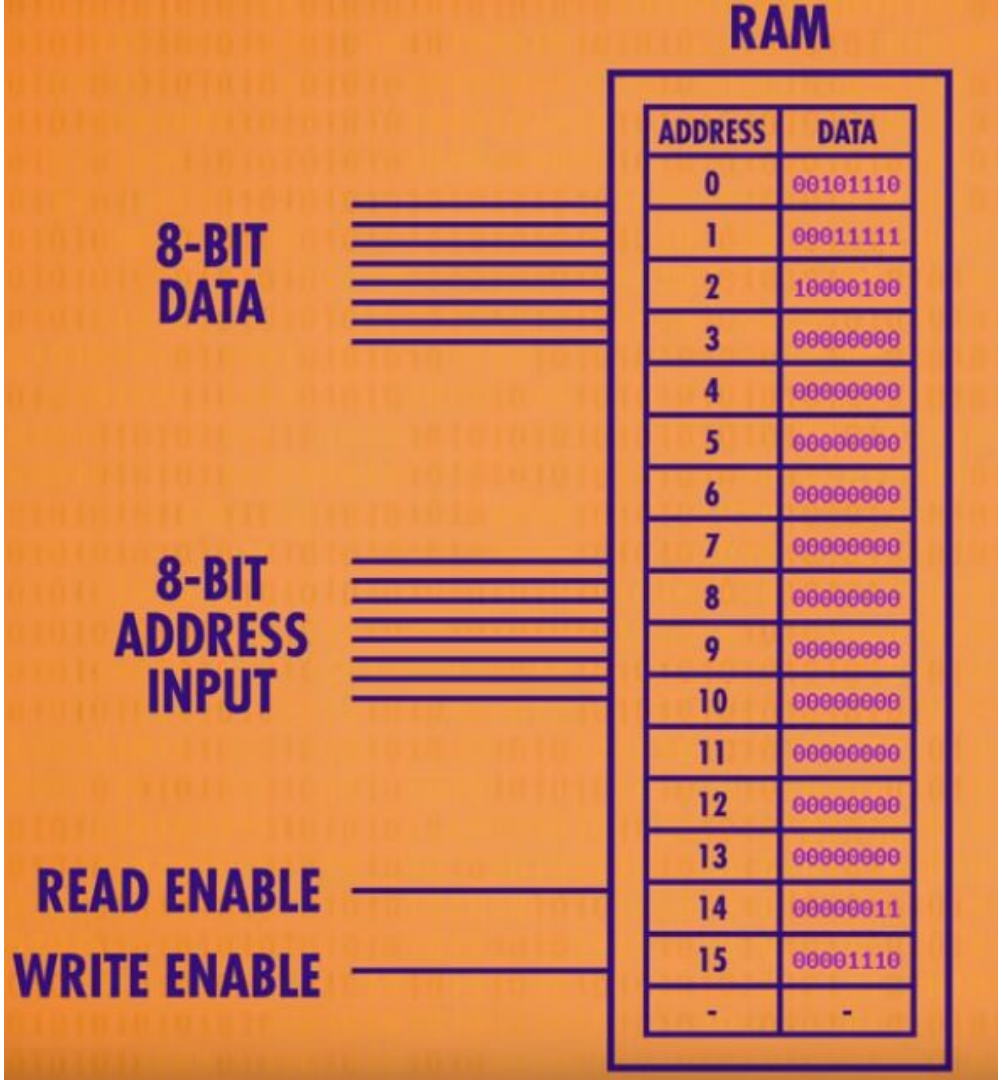
Random-Access Memory (RAM)

Ein Speicher ist eine Folge von Registern, die alle über eine Adresse verfügen.

RAM = Arbeitsspeicher

8 Eingangsleitungen, 8 Bit speichern, Adresse zur direkten Adressierung (BUS)

Lesen und schreiben



C vs. Assembler

```c
#include <stdio.h>
int main() {
  int i;
  for (i = 1; i < 11; ++i)
  {
    printf("%d ", i);
  }
  return 0;
}
```

C to Assembler:
https://godbolt.org

```
.LC0:
     .string "%d "
main:
     push rbp         ; Save address of previous stack frame
     mov  rbp, rsp    ; Address of current stack frame
     sub  rsp, 16     ; Reserve 16 bytes for local variables
     mov  DWORD PTR [rbp-4], 1
.L3:
     cmp  DWORD PTR [rbp-4], 10
     jg   .L2
     mov  eax, DWORD PTR [rbp-4]
     mov  esi, eax
     mov  edi, OFFSET FLAT:.LC0
     mov  eax, 0
     call printf
     add  DWORD PTR [rbp-4], 1
     jmp  .L3
.L2:
     mov  eax, 0
     leave
     ret
```

Assembler Sprache - X86-64 gcc 9.2

- `.LC0: .string "%d "`
  Lokale Variable, .string steht %d von printf
- **`main:`**
  Funktion mit dem Namen “main”
- `push rbp`
  Gib rbp auf den Stack
- **`mov rbp, rsp`**
  Speichere den Wert von rsp in rbp
- rsp und rbp sind Speicheradressen und definieren den Stack auf dem gearbeitet wird.
  rbp = base pointer (esb bei 32 bit, sb bei 16bit)
  rsp = stack pointer
- **`sub rsp, 16`**
  Subtrahiert rsp-16
  Es wird Platz am Stack reserviert
- **`mov DWORD PTR [rbp-4], 1`**
  i = 1
- **`cmp DWORD PTR [rbp-4], 10`**
  i < 11
  Vergleich den Wert in `[rbp-4] mit 10`

```
.LC0:
    .string "%d "
main:
    push rbp
    mov  rbp, rsp
    sub  rsp, 16
    mov  DWORD PTR [rbp-4], 1   i = 1
.L3:
    cmp  DWORD PTR [rbp-4], 10  i < 11
    jg   .L2
    mov  eax, DWORD PTR [rbp-4]
    mov  esi, eax
    mov  edi, OFFSET FLAT:.LC0
    mov  eax, 0
    call printf
    add  DWORD PTR [rbp-4], 1
    jmp  .L3
.L2:
    mov  eax, 0
    leave
    ret
```

- **`jg  .L2`**
  "jump greater" Springe zu L2, wenn in [rbp-4] der Wert 10
- **`mov  eax, DWORD PTR [rbp-4]`**
  Gib den Wert in DWORD PTR [rbp-4] ins Register eax (eax = Allzweckregister)
- **`mov  esi, eax`**
  Gib den Wert von Register eax in Register esi
- **`mov  edi, OFFSET FLAT:.LC0`**
- **`mov  eax, 0`**
- **`call printf`**
  printf("%d ", i);
- **`add  DWORD PTR [rbp-4], 1`**
  Addiere 1 zur 32-bit Integer Repräsentation in 2 Bytes beginnend bei **`[rbp-4]`** Wert in der Speicheradresse;
  ++i
- **`mov  eax, 0`**
- **`call printf`**
- **`jmp  .L3`**
  Springe wieder zu .L3
- `cmp  DWORD PTR [rbp-4], 10`
  SCHLEIFE, solange bis Wert in [rbp-4] 10 ist
- **`mov  eax, 0 / leave / ret`**

```
.LC0:
    .string "%d "
main:
    push rbp
    mov  rbp, rsp
    sub  rsp, 16
    mov  DWORD PTR [rbp-4], 1
.L3:
    cmp  DWORD PTR [rbp-4], 10
    jg   .L2
    mov  eax, DWORD PTR [rbp-4]
    mov  esi, eax
    mov  edi, OFFSET FLAT:.LC0
    mov  eax, 0
    call printf
    add  DWORD PTR [rbp-4], 1
    jmp  .L3
.L2:
    mov  eax, 0
    leave
    ret
```

Abbruch-bedingung

Schleife

Ass 6

Im Folgenden gibt es mehrere kleine C-Programme. Ziel dieses Assignments ist erklären zu können warum sich die Programme so verhalten, wie sie sich verhalten. Nur das Ergebnis alleine ist nicht ausreichend. Aber du kannst (sollst) natürlich die Programme in der Konsole kompilieren und ausführen.
(in der Konsole kompilieren; nur im Notfall mit dem Editor arbeiten : https://code.visualstudio.com)
Schaue dir für jedes Beispiel den Assembler Code (X86-64 gcc 9.2) an und erkläre in grob. Nutze dazu https://godbolt.org/.
1 Punkt jeweils für C-Programm, 1 Punkt für Assembler. Schaue im AMD64-Assembler-Handbuch oder recherchiere im Web.
Du musst nicht alles zu 100% verstehen.

1.1) Bitoperationen
1.2.) Assembler

```
#include <stdio.h>
int main() {
  int x = 1 << 3;
  printf("%d", x);
  return 0;
}
```

2.1) Schleifen
2.2.) Assembler

```
#include <stdio.h>
int main() {
  char letter;
  for(letter = 'A'; letter <= 'H'; letter++){
    printf("%c", letter);
  }
  return 0;}
```

3.1) Bedingungen
3.2.) Assembler

```
#include <stdio.h>
int main() {
  int a, b, c;
  a = 11;
  b = 12;
  c = 20;

if((a + b > c) && (a + c >
b) && (b + c > a)) {

    printf("Dreieck\n");
  }
  else{
  printf("kein Dreieck\n");
}
return 0;
}
```

4.1) Strings
4.2.) Assembler

```
#include <stdio.h>
int main() {
  char str[] = "Banane";
  str[3] = '\0';
  printf("%s", str);
  return 0;
}
```

5.1) Zahlen
5.2.) Assembler

```
#include <stdio.h>
int main () {
  printf("Signed 32-bit:\n");
  printf("%d\n", 46341 * 46341);
  return 0;
}
```

AMD64-Assembler-Handbuch

http://www.complang.tuwien.ac.at/ublu/amd64/amd64h.html

x86 Assembly Guide

http://www.cs.virginia.edu/~evans/cs216/guides/x86.html

C-HowTo, http://www.c-howto.de/

Eines der Tutorials durcharbeiten: https://www.youtube.com/watch?v=1uR4tL-OSNI, http://www.c-howto.de/tutorial/

- **Bitmanipulation:** http://www.c-howto.de/tutorial/variablen/bitmanipulation/
- **Operatoren:** http://www.c-howto.de/tutorial/variablen/operatoren/
- **Datentypen:** http://www.c-howto.de/tutorial/variablen/datentypen/
- **Typumwandlung:** http://www.c-howto.de/tutorial/variablen/typumwandlung/
- **IF ELSE:** http://www.c-howto.de/tutorial/verzweigungen/if-und-else/
- **Logische Operatoren:**

  http://www.c-howto.de/tutorial/verzweigungen/logische-operatoren/
- **FOR-Schleife:** http://www.c-howto.de/tutorial/schleifen/for-schleife/
- **Nullterminierung:** http://www.c-howto.de/tutorial/strings-zeichenketten/nullterminiert/

Ressourcen

Assembler Tutorial #6 - Der Stack,
https://www.youtube.com/watch?v=9i_1T-I8Guo

- https://godbolt.org/
- Kompilieren: `gcc -Wall evenNumber.c - o evenNumber`
- Ausführen: `./evenNumber`
- Alle Zwischenschritte:
  `gcc -Wall -save-temps evenNumber.c - o evenNumber`
- Preprocessing
  - include
  - Kommentar

```
#include <stdio.h>
// even numbers
int main() {
  int count = 0;
  int b = 10;

  for (count; count <= b; count++) {
    if (count % 2 == 0 && count != 0)
    {
    printf("Gerade Zahl: %d \n", count);
    }
  }
}
```

```
.LC0:
        .string "Gerade Zahl: %d \n"
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     DWORD PTR [rbp-4], 0
        mov     DWORD PTR [rbp-8], 10
        jmp     .L2
.L4:
        mov     eax, DWORD PTR [rbp-4]
        and     eax, 1
        test    eax, eax
        jne     .L3
        cmp     DWORD PTR [rbp-4], 0
        je      .L3
        mov     eax, DWORD PTR [rbp-4]
        mov     esi, eax
        mov     edi, OFFSET FLAT:.LC0
        mov     eax, 0
        call    printf
.L3:
        add     DWORD PTR [rbp-4], 1
.L2:
        mov     eax, DWORD PTR [rbp-4]
        cmp     eax, DWORD PTR [rbp-8]
        jle     .L4
        mov     eax, 0
        leave
        ret
```

Assembler Sprache - X86-64 gcc 9.2