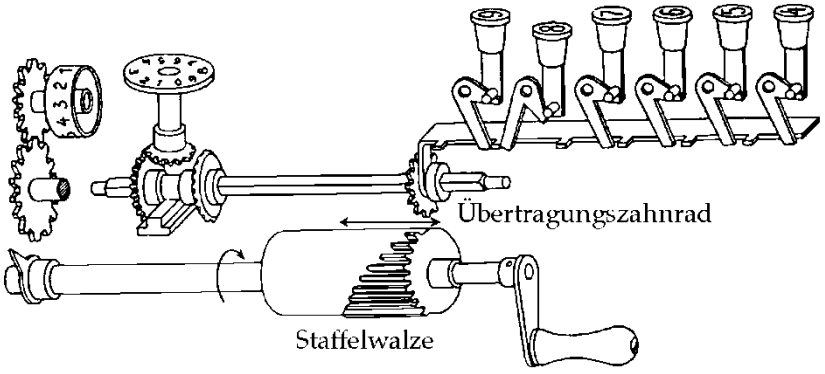
Inhaltsübersicht

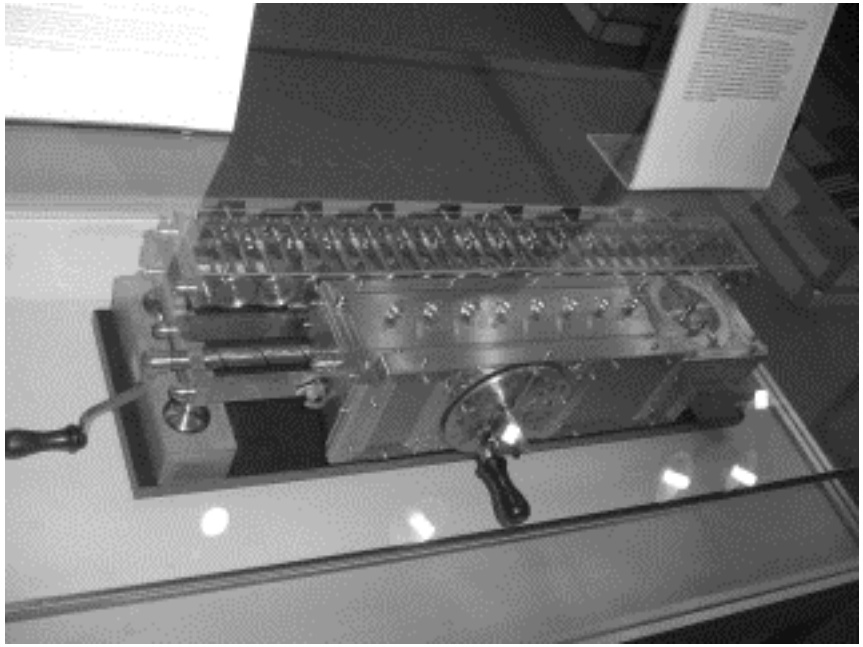
Kapitelübersicht

I. Kurze Geschichte der Rechnertechnik
II. Was sind Mikrocontroller?
III. Aufbau und Arbeitsweise von µPs und µCs
IV. Organisation des Speichersystems
V. Zugriff auf Peripherieeinheiten
VI. Programmierung von Mikroprozessoren
VII. RISC Prozessor-Architekturen

Mechanische Rechenmaschinen

- Wilhelm Schickard, 1623
  - Addition/Subtraktion von 6-stelligen Zahlen, separate Multiplikation/Division, astronomische Berechnungen
- Blaise Pascal, 1642
  - „Pascaline“, Addition/Subtraktion
- Gottfried Wilhelm von Leibniz, 1672
  - Staffelwalzenmaschine
  - vier Grundrechenarten





Babbage‘s „Analytical Engine“, 1873

- Charles Babbage, Professor für Mathematik in Cambridge
- Erster Vorschlag eines Universalrechners für allgemeine Anwendungen (General purpose):
  - programmierbar
  - separater Speicher („Store“), 1000 Worte mit 50 Dezimalstellen
  - arithmetische Einheit („Mill“), dezimale Fließkommazahlen, vier Grundrechenarten
  - Programm auf Lochkarten
  - Schleifen und bedingte Sprünge
  - Drucker und Kurvenplotter
- Maschine konnte aufgrund der Komplexität damals nicht als mechanische Maschine erbaut werden.
- Lady Ada Lovelace machte erste Vorschläge zur Programmierung der Maschine.

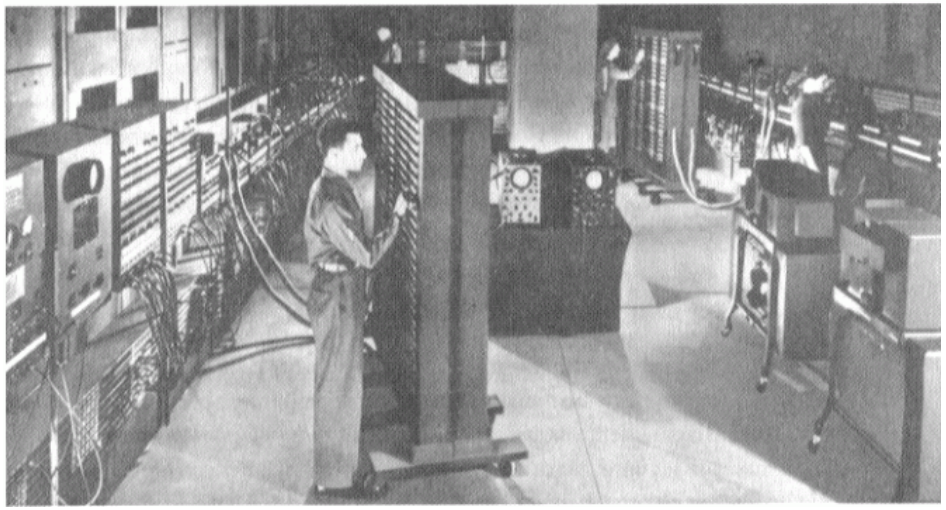
Elektromechanische Rechenmaschinen

- Konrad Zuse, 1936:
  - „Z1“: Mechanisch, programmierbar, binäre Zahlen
  - „Z3“: Elektromechanische Version der Z1 mit Relais, Speicher und arithmetischer Einheit, Programm auf Lochstreifen, 9 Instruktionen, keine bedingten Sprünge, 5.33 Hz Takt
- Howard Aiken, 1937:
  - Harvard Mark I
  - Separate Speicher für Programm und Daten (Harvard-Architektur)

Zuse Z22 im ZKM in Karlsruhe, Bildquelle: Wikipedia

Elektronische Rechner

- J.Mauchly, J.P.Eckert, 1945 (U of Pennsylvania) ENIAC (Electronic Numerical Integrator and Computer):
  - Universalrechner, diente hauptsächlich der Berechnung ballistischer Tabellen
  - bestehend aus Elektronenröhren
  - dezimales Zahlensystem, manuelle Programmierung durch Schalter
  - 18.000 Röhren, 30 Tonnen Gewicht, 1400 qm Fläche
  - 140 kW el. Leistung, Rechenleistung: 5.000 Additionen pro Sekunde



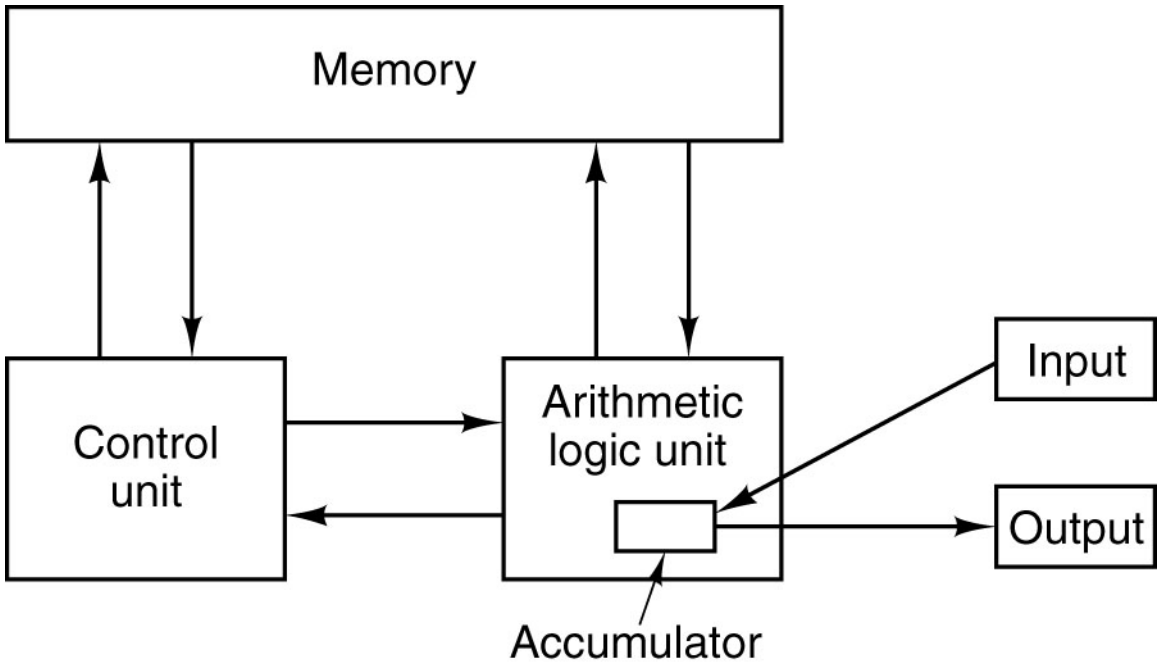
Elektronische Rechner (2)

- Moore School in Philadelphia 1944-1952, EDVAC (Electronic Discrete Variable Automatic Computer):
  - Einfachere Speicherprogrammierbarkeit: Programm im Speicherwerk (Software), „Stored Program Computer“
  - Anlehnung an Babbage:
    - Speicherwerk
    - Rechenwerk
    - Steuerwerk
  - Beeinflusst stark die Arbeiten John von Neumanns (österr.-ungarischer Mathematiker, 1903-1957), (1945 in: *First Draft of a Report on the EDVAC)*

Die Prinzipien des „von Neumann“-Rechners

- Computer: Steuereinheit (Program Control Unit), ALU, Speicher (Main Memory), I/O
- Gesteuert durch Takt
- Binäre Daten
- Speicherzugriff durch Adressen
- Programm und Daten werden im gleichen Speicher gehalten
- Serielle Abarbeitung der Befehle
- Die Steuereinheit interpretiert die Befehle und führt sie aus
- Der sequentielle Programmablauf kann durch bedingte oder unbedingte Sprünge verändert werden
- Princeton Institute for Advanced Studies (IAS) 1946, „IAS Computer“

Struktur des von-Neumann-Rechners

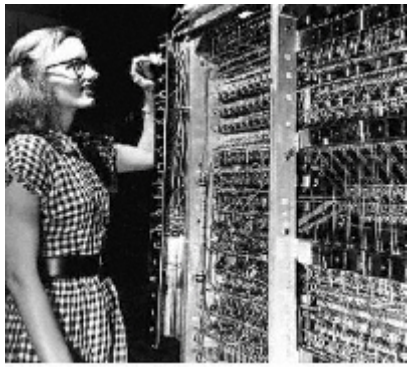


Quelle: Tanenbaum, Structured Computer Organization, (c) 2006 Pearson Education

60er und 70er Jahre: Die Ära der Großrechner

- In dieser Zeit waren Computer i.d.R. noch sehr große Rechenanlagen („mainframe“), die von Firmen oder Institutionen (Behörden, Hochschulen, Forschungsinstitute, etc.) betrieben wurden.
- Definition des Begriffs „mainframe“ (*IBM Dictionary Of Computing*):
  „A large computer, in particular one to which other computers can be connected so that they can share facilities the mainframe provides (for example, a System/370 computing system to which personal computers are attached so that they can upload and download programs and data). The term usually refers to hardware only, namely, main storage, execution circuitry and peripheral units."

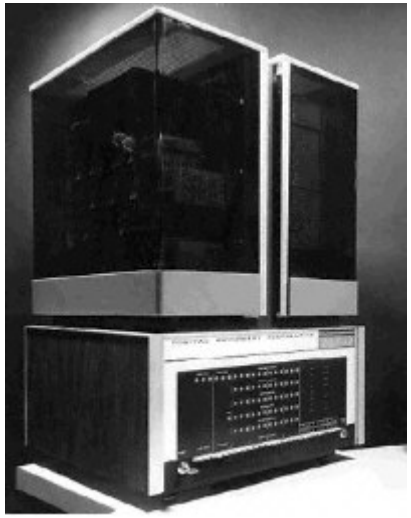
Univac



IAS

CDC6600

IBM 360

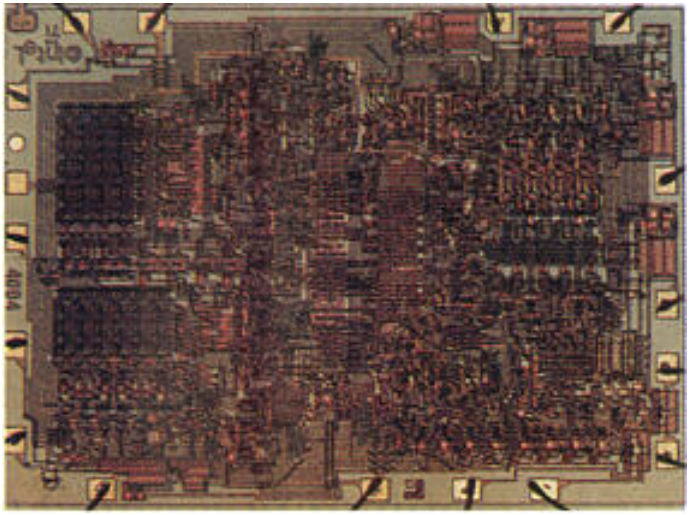


PDP-8

VAX 11/780

1970: Die Geburt des „Mikroprozessors“

- Integration wesentlicher Teile des Rechners auf einem „Chip“ (Integrierte Schaltung)
- Intel 4004/8008 (4 Bit bzw. 8 Bit Prozessor



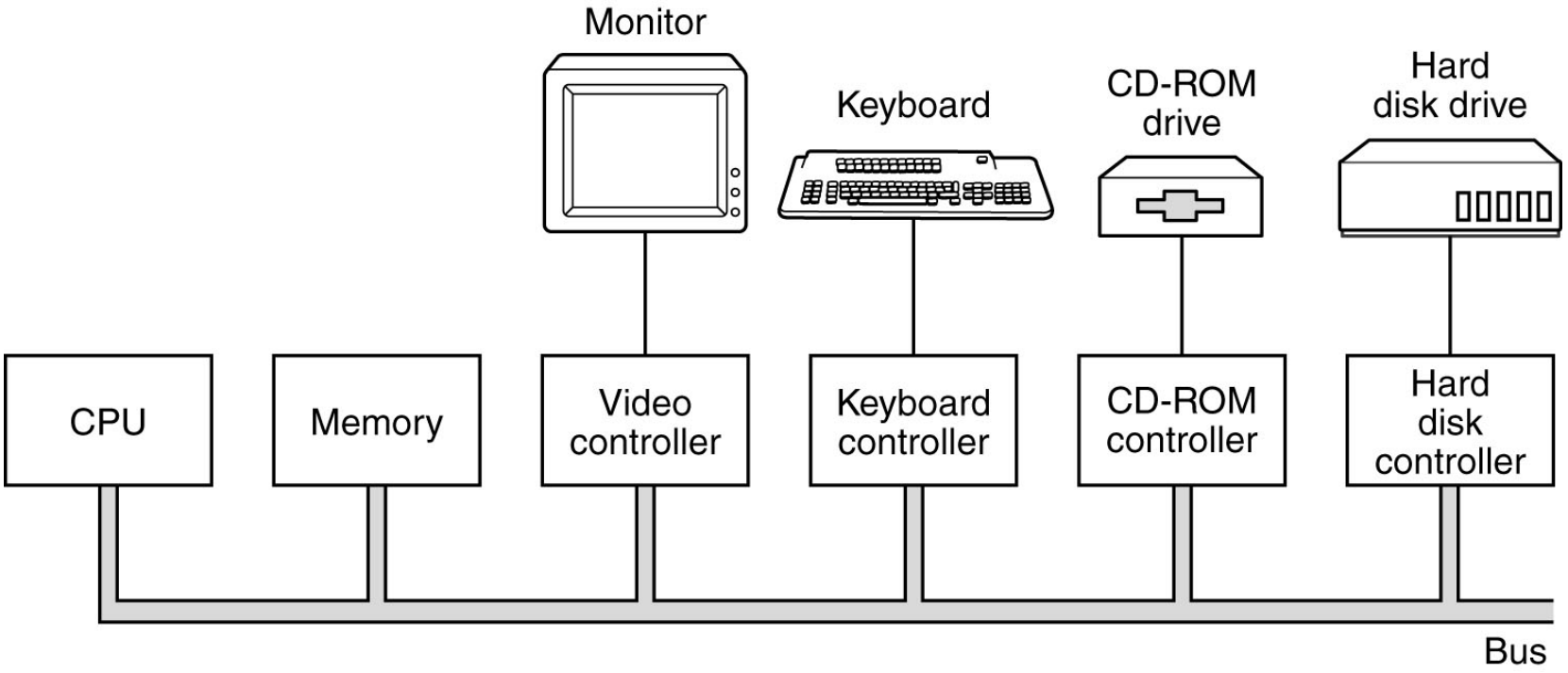
70er Jahre: Die Computer werden immer kleiner ...

- Ken Olson, Präsident und Gründer von DEC, 1970: „Es gibt keinen Grund, warum jemand einen Computer zu Hause haben wollte.“

- 1976: Apple I
- 1979: Atari 400 und 800
- 1980: Commodore VC20
- 1981: IBM PC (Personal Computer)

Bildquelle: Wikipedia

Typischer Aufbau eines PC-Systems



Kapitelübersicht

I. Kurze Geschichte der Rechnertechnik

II. Was sind Mikrocontroller?

III. Aufbau und Arbeitsweise von µPs und µCs

IV. Organisation des Speichersystems

V. Zugriff auf Peripherieeinheiten

VI. Programmierung von Mikroprozessoren

VII. RISC Prozessor-Architekturen

Die Entstehung der Mikrocontroller

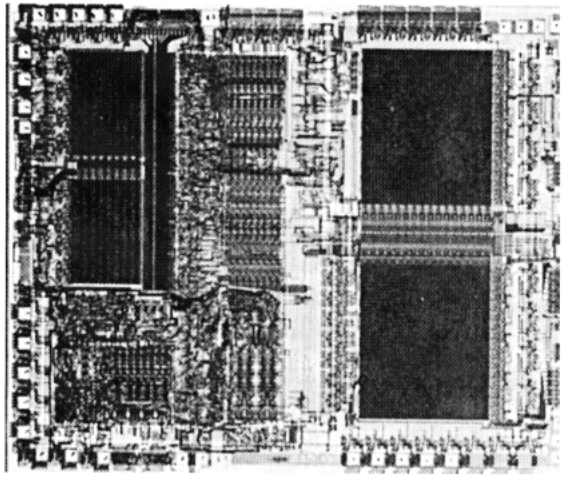
- In den 60er und 70er Jahren hält die Computer-Technologie Einzug in technische Geräte und Systeme.
  - Automobiltechnik, Luft- und Raumfahrt, Konsumelektronik, Haushalt, …
- Probleme:
  - Je mehr Bauteile ein Gesamtsystem enthält, desto größer ist die Wahrscheinlichkeit für das Auftreten von Fehlern.
  - Je größer die Anzahl der Komponenten eines Computersystems ist, desto höher ist seine Fehleranfälligkeit und Ausfallwahrscheinlichkeit.
- Die Erbauer von Mikroprozessoren begannen daher damit, externe Einheiten und die CPU gemeinsam auf einem Chip unterzubringen.
  - Verringerung der Ausfallwahrscheinlichkeit des Gesamtsystems
  - Geringerer Platzbedarf, kleinere Geräte
  - Die so entstandenen Chips wurden „Mikrocontroller“ genannt.

Mikroprozessor – Mikrocontroller

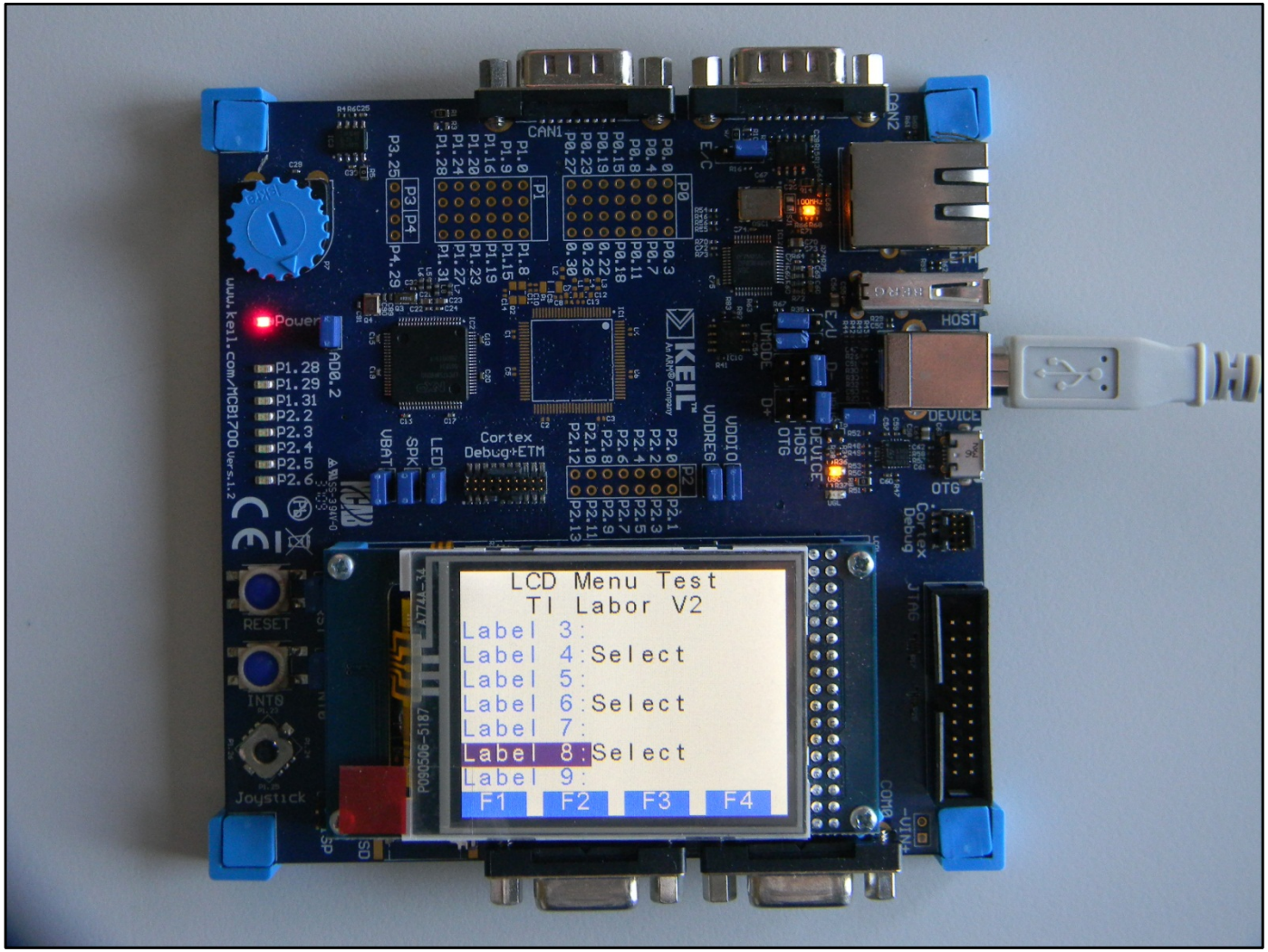
- Computer oder „Mikrocomputer“:
  - Realisiert durch Mikroprozessoren (z.B. Pentium)
  - Für allgemeine Aufgaben
  - Der Begriff „Mikrocomputer“ unterschied in den 70er Jahren diese Klasse von Rechnern von Großrechenanlagen („Mainframe“). Heute ist dieser Begriff veraltet.
- Mikrocontroller (µC):
  - Eingesetzt für spezielle „Steuerungsaufgaben“ (to control)
  - Besteht aus Mikroprozessor, Speicher und spez. Peripherie, integriert in einem „Chip“ (Integrierte Schaltung)
  - „Eingebettetes System“ (Embedded System)
  - Zentrale Komponente in vielen Anwendungen
    - Automobilelektronik (heute z.B. 70-80 µCs in Autos der Oberklasse!)
    - Unterhaltungselektronik (DVD-Player, Telefon, Fernseher, etc.)
    - Haushaltsgeräte (Kaffeemaschine, Waschmaschine, etc.)
    - Industrieelektronik
    - ....

Beispiel 8051 µC von Intel

- Intel führt 1980 den 8051 µC ein.
- Es entstand eine µC-“Familie“ (MCS-51):
  - Gleicher Prozessor-“Kern“ und Instruktionssatz
  - Unterschiedliche Konfiguration von Peripherieblöcken
  - Erweiterung des Programm- und Datenspeichers
- 8051-µC auch von anderen Herstellern:
  - Philips
  - Infineon
  - Analog Devices
  - ...



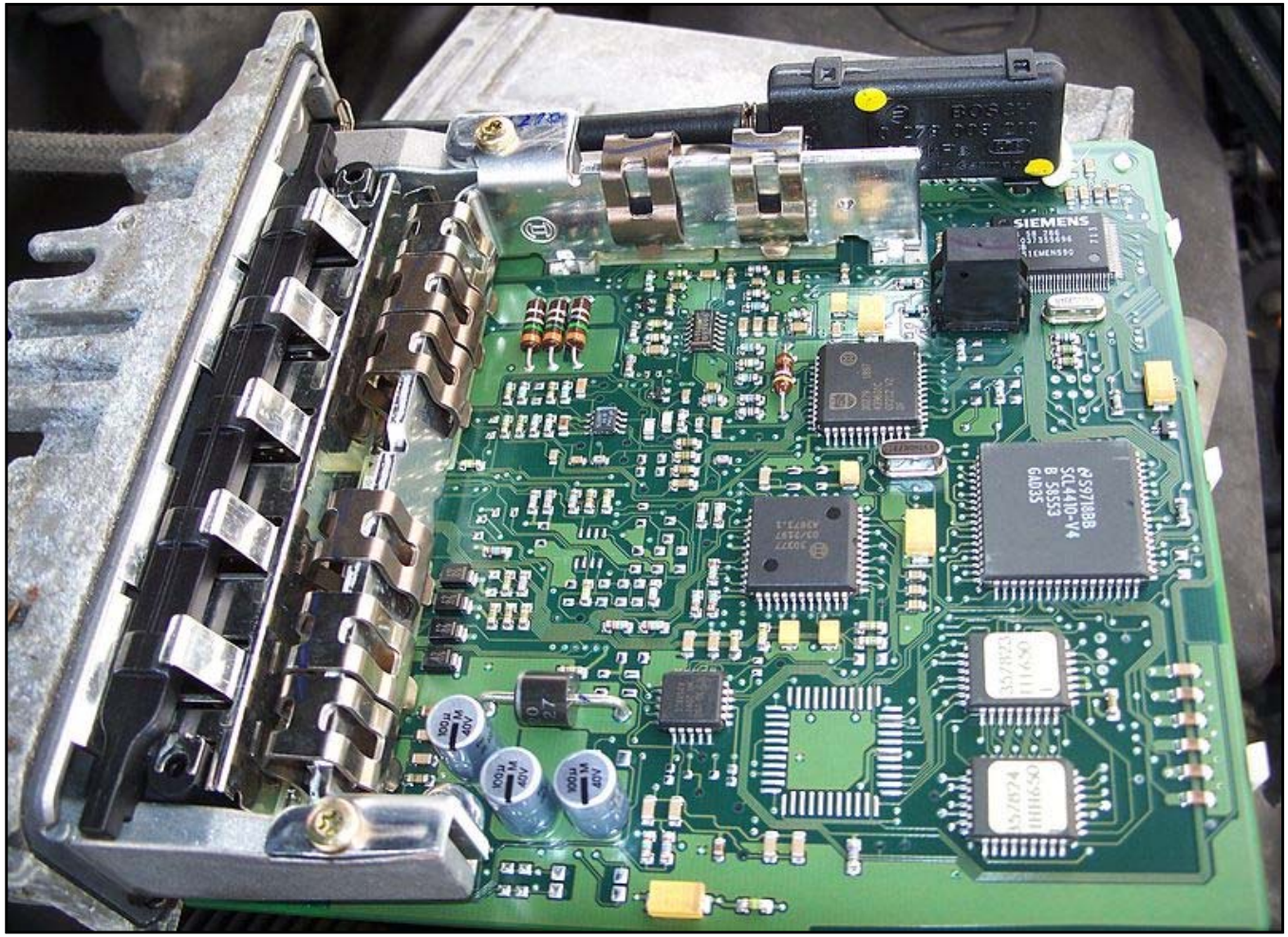
Modernes Beispiel: ARM Cortex-M3



Einsatzgebiete Mikrocontroller

- Automobilelektronik
  - von 8-Bit bis 32-Bit µC
  - aktuell bis zu 100 µC in einem Fahrzeug
- Unterhaltungselektronik (braune Ware)
  - meist 16-Bit und 32-Bit µC
  - MP3-, DVD-Player, Fernseher, Stereoanlagen
- Haushaltsgeräte (weiße Ware)
  - von 8-Bit bis 32-Bit µC, Tendenz zu 16-Bit und 32-Bit µC
  - Wasch- und Spülmaschinen, Backofen, Kochplatten, Wäschetrockner
- Kommunikationstechnik
  - von 8-Bit bis 32-Bit µC, Tendenz zu 16-Bit und 32-Bit µC
  - Handy, Telefon, Netzwerk-Baugruppen

Beispiel Motorsteuerung (Golf III)



Einsatzgebiete Mikrocontroller (2)

- Industrieelektronik
  - von 8-Bit bis 32-Bit µC, Tendenz zu 16-Bit und 32-Bit µC
  - Steuerungen und Regelungen für industrielle Anwendungen
    - z.B. SPS (Speicherprogrammierbare Steuerungen), Robotersteuerungen
- Smart Metering
  - 16- und 32-Bit µC
  - dezentrales Erfassen von Verbrauchswerten
  - sehr geringer Energieverbrauch erforderlich
- Motorsteuerungen für elektrische Motoren
  - 16-Bit und 32-Bit µC
  - Drehzahlregelung und Leistungsregelung

Kapitelübersicht

I. Kurze Geschichte der Rechnertechnik

II. Was sind Mikrocontroller?

III. Aufbau und Arbeitsweise von µPs und µCs

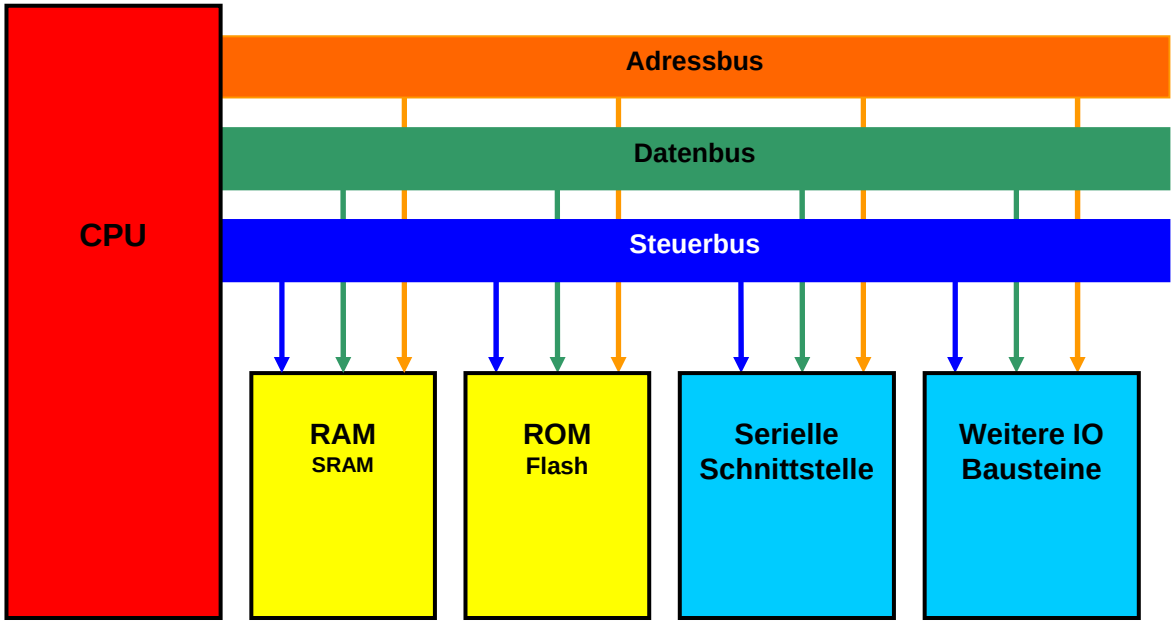
IV. Organisation des Speichersystems

V. Zugriff auf Peripherieeinheiten

VI. Programmierung von Mikroprozessoren

VII. Prozessor-Architekturen

Der Aufbau eines Mikroprozessorsystems (1)



Mikroprozessorsysteme sind i.d.R. nach dem Prinzip des Von-Neumann-Rechners aufgebaut.

Der Aufbau eines Mikroprozessorsystems (2)

- CPU (Central Processing Unit):
  - Steuerwerk (Ablaufsteuerung):
    - Holen der Befehle (Instruktionen, engl.: instructions)
    - Dekodieren und Ausführen der Befehle
    - Ansteuerung von Speicher und Ein-/Ausgabeeinheiten (Peripherie)
  - Rechenwerk (ALU: Arithmetic Logic Unit):
    - Ausführen von logischen und arithmetischen Operationen
    - Erzeugen von Statusinformationen
    - Zwischenspeicherung der Operanden und Resultate
      - Häufig im so genannten „Akkumulator“ (Akku)
  - Verschiedene weitere Register (siehe später)

Der Aufbau eines Mikroprozessorsystems (3)

- Bussystem:
  - Bus: „Sammelleitung“, Ansammlung von Verbindungsleitungen
  - Besteht in der Regel aus Adress-, Daten- und Steuerbus, Datenbus ist bidirektional
  - Schreib- oder Leseoperation aus Speicher oder Peripherie:
    - CPU legt Adresse auf Adressbus
    - CPU aktiviert Steuerleitungen (Schreiben oder Lesen)
    - Lesen: Speicher legt Datum auf Datenbus, CPU speichert Datum in internen Registern
    - Schreiben: CPU legt Datum auf Datenbus, Speicher oder Peripherie speichert Daten ab

Der Aufbau eines Mikroprozessorsystems (4)

- Speicher:
  - Aneinanderreihung von Speicherplätzen, in der Regel werden Bytes (8 Bit) adressiert
  - Anwahl von Speicherplätzen über Adressen, n Bit Adressen ergibt 2^n mögliche Speicherplätze
- Ein-/Ausgabe (Input/Output, I/O), Peripherieeinheiten
  - Holen und Bereitstellen von Daten für das umgebende System (A/D- und D/A-Wandler, UART, Sensoren, Aktoren, etc.)
  - Sonderfunktionen (Timer, etc.)

Speicher und Speicherinhalt

- Die Speicherinhalte werden grundsätzlich binär gespeichert. Man unterscheidet:
  - Programm: Ein oder mehrere Bytes bilden einen Befehl oder eine „Instruktion“ für die CPU
  - Daten: Ein oder mehrere Bytes bilden ein Datum (Variable, Konstante), welche die CPU verarbeitet.
- Bei einem von-Neumann-Rechner wird allein durch die Lage/Adresse der Bytes klar, ob es sich um eine Instruktion oder ein Datum handelt:
  - Im unteren Adressbereich befindet sich das Programm
  - Im oberen Adressbereich sind die Daten

Beispiel

Adresse (8 Bit)	Speicherplatz (8 Bit)*
FFh	00h
FEh	E5h
FDh	76h
FCh	56h
FBh	55h
…	...
…	…
04h	F5h
03h	01h
02h	24h
01h	FBh
00h	E5h

Daten

Programm

*: Binäre Daten werden zur besseren Lesbarkeit meist hexadezimal dargestellt: z.B. 1110 0101b=E5h=229d

Der „Adressraum“

- Der Adressraum gibt an, welche Anzahl von Bytes ein Prozessor maximal adressieren kann.
  - z.B. mit einer Adressbreite von 8 Bit können maximal 256 Byte addressiert werden, mit 32 Bit sind es maximal 4 Gbyte
- Zu unterscheiden ist davon, welche Größe und welche Art von Speicher in einem µC-System tatsächlich eingebaut ist.
  - RAM (Random Access Memory): flüchtiger Speicher, Speicherung von Daten („Datenspeicher“)
  - ROM (Read-Only Memory): Festwertspeicher, Speicherung von Programm oder Konstanten („Programmspeicher“)

Instruktionen und Instruktionssatz

- Die Menge aller ausführbaren Instruktionen (d.h. Binärcodes, welche der Rechner „versteht“) wird als „Instruktionssatz“ bezeichnet. Dieser definiert die „Architektur“ eines Rechners („Instruktionssatzarchitektur“, engl.: „Instruction Set Architecture“, ISA)
- Das erste Byte einer Instruktion wird als „Opcode“ bezeichnet (von „operation code“)
- Neben dem „Opcode“ können noch weitere Daten zu einem Befehl dazugehören (Adressen, Konstanten, siehe Adressierungsarten).

Beispiel

Adresse (8 Bit)	Speicherplatz (8 Bit)*
01h	FBh
00h	E5h

- E5h: Opcode
  - „Hole das Datum aus dem Speicherplatz mit der nachfolgenden Adresse (im 2. Byte) und bringe dieses in den Akkumulator“.
- FBh: 2. Byte für die Instruktion
  - Wird von der CPU aufgrund des Opcodes als Adresse interpretiert

Maschinenbefehl und Assembler

- Die vom Prozessor ausführbaren Instruktionen werden auch als „Maschinenbefehl“ bezeichnet.
- Für den Programmierer wurde die so genannte „Assemblersprache“ eingeführt:
  - Jeder Maschinenbefehl wird durch symbolische „Mnemonics“ (aus dem griechischen „Mnemonik“: Gedächtniskunst) dargestellt
  - Das Assemblerprogramm erzeugt aus einem Quellcode dann das Maschinenprogramm

Wesentliche Instruktionsklassen

- Datentransport-Befehle
  - Bringen Daten von einem Speicherort zu einem anderen ohne die Daten selbst zu verändern (z.B. mov).
- ALU (Arithmetisch/Logische) Befehle
  - Verändern Daten und speichern diese ab (z.B. add, sub, mul, and)
- Steuerfluss- oder Kontroll-Befehle
  - Verändern den normalen, sequentiellen Programmablauf (z.B. branch, jump, call, return)

Beispiel (angelehnt an 8051)

Maschinenbefehl	Assembler	Bedeutung
E5 FB	`mov A, FBh`	Akku = (FBh)
24 01	`add A, #01h`	Akku = Akku + 01h
F5 FC	`mov FCh, A`	(FCh) = Akku
80 02	`jmp 02h`	PC = 02h (Sprung)

Adresse (8 Bit)	Speicherplatz (8 Bit)*
FCh	56h
FBh	55h
…	...
05h	FCh
04h	F5h
03h	01h
02h	24h
01h	FBh
00h	E5h

Was wird durch eine Instruktion spezifiziert?

- Welche Operation auszuführen ist:
  - z.B.: `add A, #01h`
- Wo die Quelloperanden zu finden sind:
  - z.B.: `mov A, FBh`
  - In CPU Registern, Speicher, I/O oder als Teil der Instruktion
- Wo das Ergebnis abzuspeichern ist (Zieloperand):
  - z.B.: `mov A, FBh`
  - Wieder Register oder Speicher
- Wo der nächste Befehl zu finden ist:
  - z.B.: `jmp 02h`

Beispiel

Programm

```
      mov A, FBh
LOOP: add A, #01h
      mov FCh, A
      jmp LOOP
```

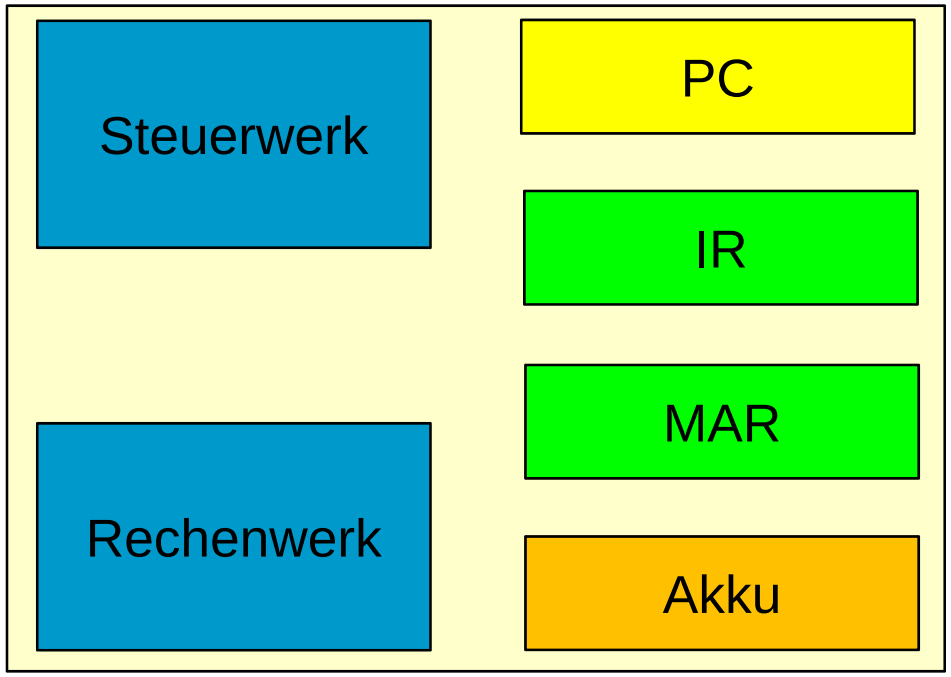
Ausführbarer Code

```
E5 FB
24 01
F5 FC
80 02
```

Speicherinhalt

Adresse	Inhalt
07	02
06	80
05	FC
04	F5
03	01
02	24
01	FB
00	E5

Die CPU



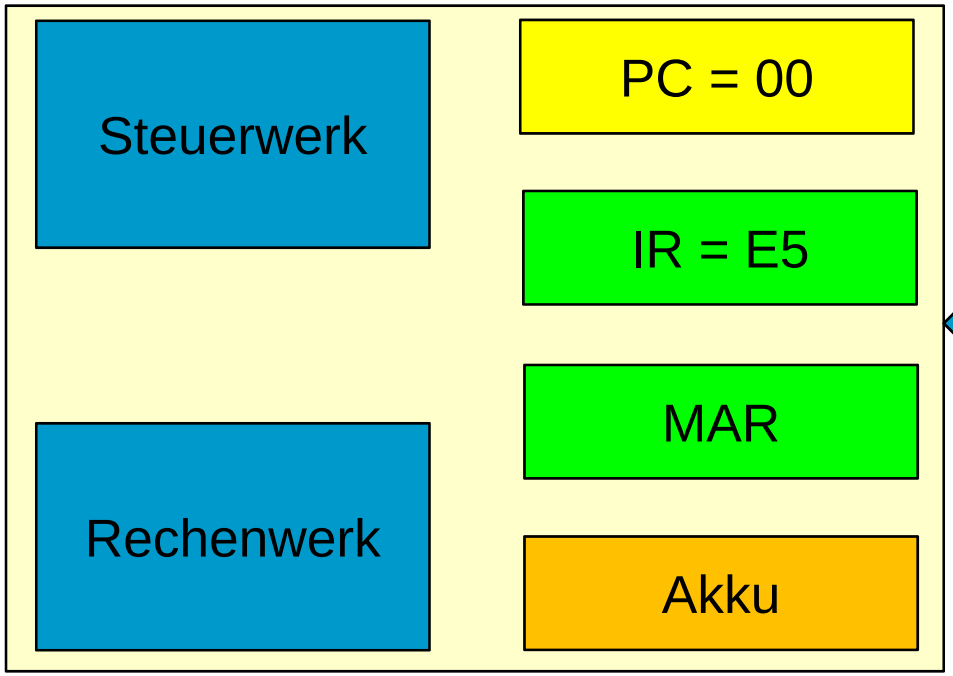
CPU-Register

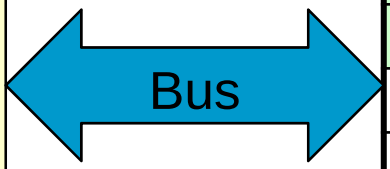
- PC (Program Counter):
  - Adressiert die Instruktionen im Programmspeicher
  - Wird bei Reset auf 0 gesetzt, damit kann die erste Instruktion geholt werden
- IR (Instruction Register):
  - Speichert eine Instruktion während der Ausführung
- MAR (Memory Adress Register):
  - Speichert eine Adresse für den Datenspeicher
- Akku (eigentlich „Akkumulator“):
  - Ergebnisregister für ALU-Ergebnisse

Abarbeitung eines Befehls (1)

- Am Anfang eines Befehlszyklus steht immer das Einlesen („Fetch“) des Opcodes
- Um den Opcode zu holen legt die CPU den Inhalt des PC (= 00) auf den Adressbus und aktiviert das „Read“-Signal (Teil des Steuerbusses).
- Der Speicher liefert daraufhin das Datum an der Adresse 0 auf dem Datenbus, was von der CPU als Opcode im IR gespeichert wird.
- Der PC wird inkrementiert (PC=PC+1=1) für den nächsten Speicherzugriff.

Abarbeitung eines Befehls (2)



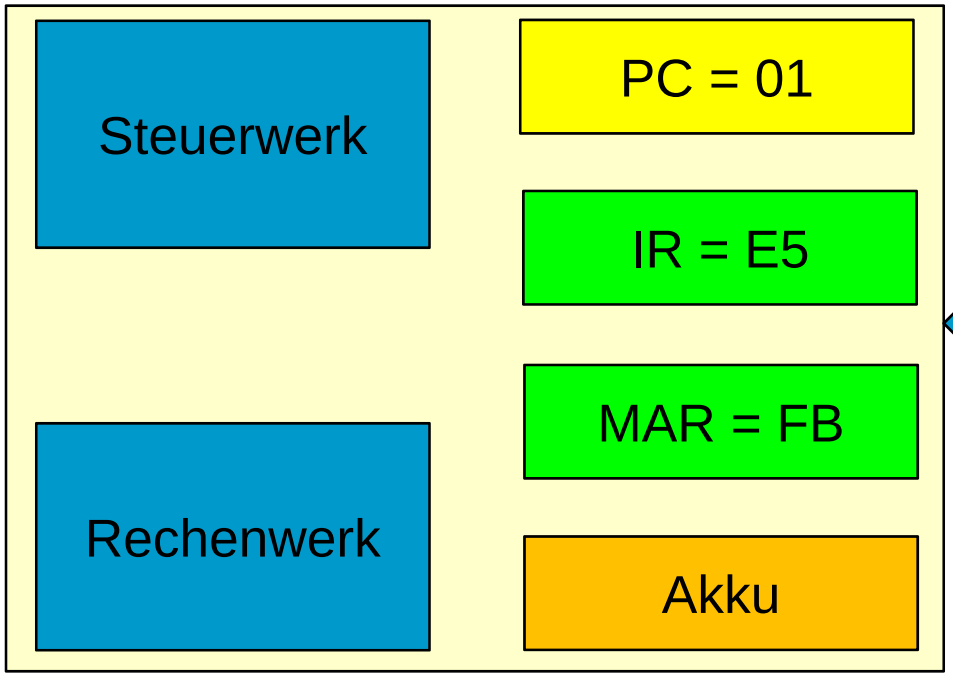


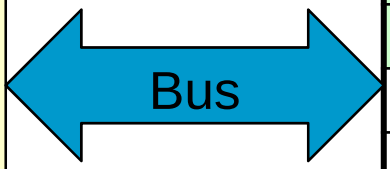
Adresse	Inhalt
FFh	00h
FEh	E5h
FDh	76h
FCh	56h
FBh	55h
…	...
…	…
04h	F5h
03h	01h
02h	24h
01h	FBh
00h	E5h

Abarbeitung eines Befehls (3)

- Der Opcode wird nun dekodiert („Decode“)
- Durch die Dekodierung des Opcodes „E5h“ wird die CPU veranlasst noch ein weiteres Byte an der Adresse 01 mit Hilfe des PC zu holen (wie bei „Fetch“). Der PC wird wiederum inkrementiert (jetzt: PC = 02).
- Dieses Byte (hier: FBh) wird als Adresse des zu holenden Datums benutzt und im MAR (Memory Adress Register) zwischengespeichert.

Abarbeitung eines Befehls (4)



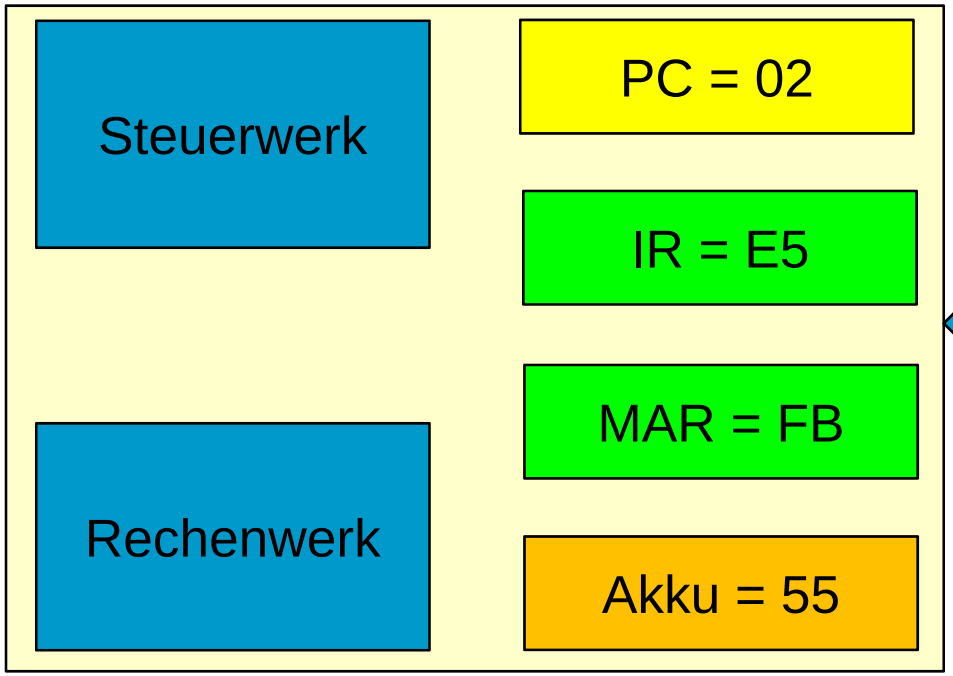


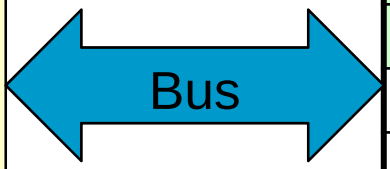
Adresse	Inhalt
FFh	00h
FEh	E5h
FDh	76h
FCh	56h
FBh	55h
…	…
…	…
04h	F5h
03h	01h
02h	24h
01h	FBh
00h	E5h

Abarbeitung eines Befehls (5)

- Die Instruktion wird nun ausgeführt („Execute“).
- Hierzu wird der Inhalt des MAR auf den Adressbus gelegt und das „Read“-Signal aktiviert.
- Der Speicher liefert das Datum an der Adresse FBh aus dem Datenspeicher auf dem Datenbus. Die CPU speichert dieses Datum im Akkumulator.
- Der nächste Befehl muss nun an Adresse 2 stehen, welcher nach Ausführen des aktuellen Befehls geholt wird (nächste „Fetch“-Phase).

Abarbeitung eines Befehls (6)





Adresse	Inhalt
FFh	00h
FEh	E5h
FDh	76h
FCh	56h
FBh	55h
…	...
…	…
04h	F5h
03h	01h
02h	24h
01h	FBh
00h	E5h

„Wortbreite“ des Prozessors

- Die „Bitbreite“ oder „Wortbreite“ des Prozessors bezeichnet die Bitbreite mit der die Daten in der CPU verarbeitet werden, d.h. die Breite der internen Register. Die Breite des externen Busses kann davon abweichen!
  - 8 Bit Prozessoren: z.B. Intel 8051, Motorola 68HC08
  - 16 Bit Prozessoren: z.B. Intel 8086, Motorola 68000, Intel 80286
  - 32 Bit Prozessoren: z.B. Intel Pentium, ARM9, MIPS, SUN Sparc, ARM Cortex
  - 64 Bit Prozessoren: z.B. SUN UltraSPARC, Intel Itanium

„Wortbreite“ des Prozessors (2)

- Die Wortbreite sagt etwas über die Leistungsfähigkeit aus:
  - Adressraum: Da für die Adressierung des Speichers Register (z.B. PC) benötigt werden, definiert die Bitbreite häufig auch den maximalen Adressraum (32 Bit = 2^{32} Byte = 4GByte, 64 Bit = 17.179.869.184 GB!)
  - Breite der Daten: Ein 8 Bit Prozessor kann auch 32 Bit Daten (z.B. „long int“ in C) verarbeiten, braucht dazu aber mehrere Instruktionen. Ein 32 Bit Prozessor kann dies in einer Instruktion tun.
- z.B. Cortex-M0: 32 Bit interne Wortbreite, 32 Bit Busbreite

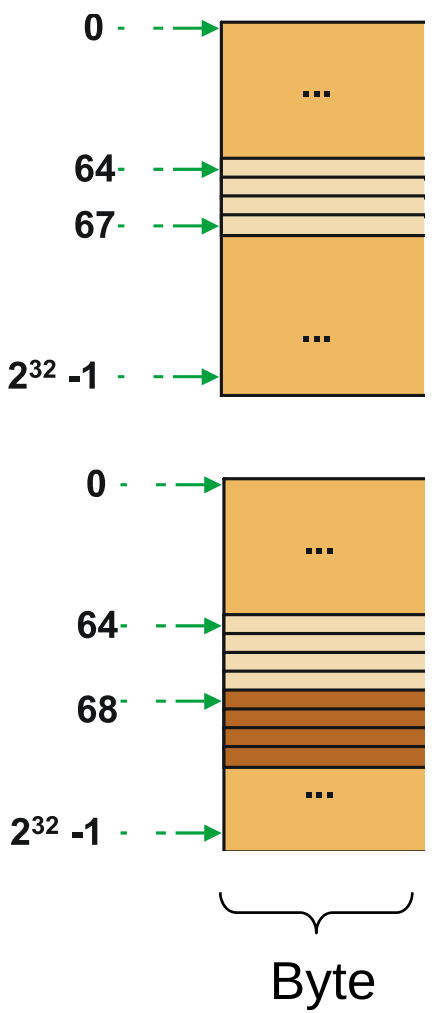
Kapitelübersicht

Organisation des Speichersystems

- Liegen Programm und Daten im gleichen Adressraum, so spricht man von einer „von-Neumann-Organisation“ des Speichers.
- Bei einigen Rechnersystemen werden Programm und Daten in getrennten Adressräumen gehalten, dies wird als „Harvard-Architektur“ bezeichnet.
- Bei leistungsfähigen Rechnersystemen befindet sich zwischen CPU und Hauptspeicher noch ein kleinerer Zwischenspeicher, dieser wird als „Cache“ bezeichnet.
- Wir gehen für die folgenden Betrachtungen von einem von-Neumann-Speichersystem ohne Cache aus.

Organisation des Speichersystems (2)

- Daten werden üblicherweise in Bytes organisiert, übliche Länge der Daten:
  - Byte, 2 Bytes, 4 Bytes und 8 Bytes
- Besteht ein Befehl oder ein Datum z.B. aus 4 Byte, dann muss die Byte-Adresse um 4 erhöht werden, um den nächsten Befehl oder das nächste Datum zu holen!



Übersicht: Wichtigste Datentypen in C

Datentyp	Anzahl Bits *	Wertebereich
(signed) char	8	-128 bis +127
unsigned char	8	0 bis +255
(signed) int	32 **	-2147483648 bis +2147483647
unsigned int	32 **	0 bis +4294967295
(signed) short	16	-32768 bis +32767
unsigned short	16	0 bis +65535
(signed) long	32	-2147483648 bis +2147483647
unsigned long	32	0 bis +4294967295
float	32	±1.175494E-38 bis ±3.402823E+38
enum	8/16	

* Benötigter Speicherplatz im Hauptspeicher
** Implementierung im ARM-C/C++-Compiler

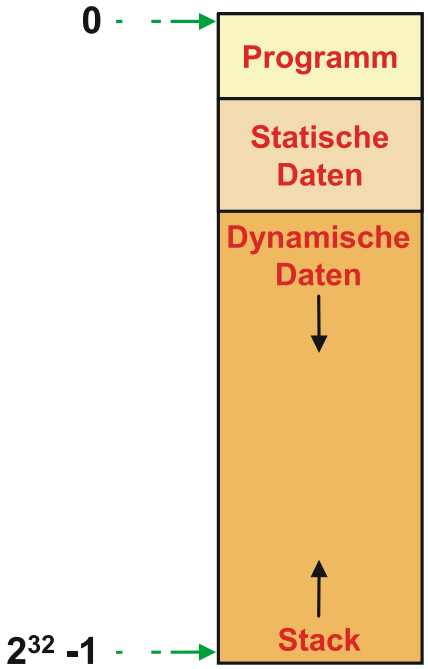
Integer-Datentypen

- Problem: Größe von „int“ ist in C nicht genau definiert und kann 16-Bit (short) oder 32-Bit (long) groß sein.
- Es empfiehlt sich daher die Verwendung von Datentypen aus „stdint.h“
  - int: signed
  - uint: unsigned
- Eindeutigkeit kann auch mit short/long erzielt werden.

Datentyp	Anzahl Bits
int8_t	8
uint8_t	8
int16_t	16
uint16_t	16
int32_t	32
uint32_t	32

Anordnung von Programm und Daten

- In einem von-Neumann-Speichersystem werden i.d.R. im unteren Bereich (also ab Adresse 0) das Programm angeordnet und nachfolgend die Daten.
- Daten:
  - Statische Daten (z.B. globale Variablen)
  - Dynamische Daten (mit „new“ oder „malloc“ im C-Programm)
  - Stack (lokale Variablen der Funktionen)



Speichersegmente

- Die Anordnung von Programm und Daten wird vom sogenannten „Linker“ automatisch vorgenommen (siehe später).
- Die Speicherbelegung kann i.d.R. in einer sogenannten MAP-Datei (Projektname.map) eingesehen werden.
- Folgende Segmente und Bezeichnungen werden verwendet:
  - .text: Programmsegment
  - .data: Globale und statische Variablen, die initialisiert werden
  - .bss: Nicht-initialisierte globale und statische Variablen

Gültigkeitsbereich (Scope) von Variablen (1)

- Bereich des Quelltextes in der die Variable definiert wurde: {}-Regel
- global (und *immer* statisch!): Variable wurde außerhalb jeglicher Funktionen definiert (auch „main“ ist eine Funktion)
  - Mit Schlüsselwort static: Nur innerhalb des Moduls/Datei gültig
  - Ohne Schlüsselwort static: Im ganzen *Programm* gültig.
  - Vorsicht: static hat hier (unglücklicherweise) eine völlig andere Bedeutung wie bei lokalen Variablen!!!

Gültigkeitsbereich (Scope) von Variablen (2)

- lokal: Variable wurde innerhalb einer Funktion definiert
- Mit Schlüsselwort static: statische Variable
  - Verhalten (Lebensdauer, Initialisierung) wie globale Variable, aber nur in dieser Funktion sichtbar.
- Ohne Schlüsselwort static: automatische Variable

Lebensdauer von statischen Variablen

- Statische Variablen: jede globale Variable oder lokale Variablen in Funktionen die mit **static** deklariert wurden
- Variable und deren Inhalt bleibt nach Verlassen der Funktion erhalten
- Variable wird zu Programmbeginn mit 0 initialisiert
- Vom Programmierer vorgegebene Initialisierung wird einmal ausgeführt, bei lokalen Variablen beim ersten Aufruf der Funktion.

Lebensdauer von automatischen Variablen

- Automatische Variable: Schlüsselwort auto bzw. keine Angabe, Variable in Funktion deklariert
- Speicherplatz der Variablen wird nach Verlassen der Funktion freigegeben.
  - Bei erneutem Aufruf der Funktion ist der alte Wert der Variablen evtl. nicht mehr vorhanden!
  - Es gibt keinen festen Speicherplatz für die Variable, sie wird normalerweise auf dem Stack gespeichert. Der Platz kann von einer anderen automatischen Variablen in einer anderen Funktion benutzt werden, daher speicherplatzsparend!
- keine Initialisierung, d.h. vor der ersten Zuweisung beliebiger Wert
- Vom Programmierer vorgegebene Initialisierung wird bei *jedem* Funktionsaufruf ausgeführt.

Speichersegmente, Heap und Stack

Stack
Heap
.bss : nicht-initialisierte globale und statische Daten
.data : initialisierte globale und statische Daten
.text : Programm-Code

```
int globalVar1;      //Globale statische Variable (.bss)
int globalVar2 = 2;  //Global, statisch, initialisiert (.data)


int main (void) {

  int localAutoVar;    //Lokale automatische Variable (Stack)

  static int localStaticVar1;      //Lokale statische Variable (.bss)
  static int localStaticVar2 = 5;  //Lokal, statisch, initialisiert (.data)

  int *ptr = malloc(sizeof(int));  //Dynamische Speicherbelegung (Heap)

  …

}
```

Anordnung der Bytes in einem Datum

- Vorsicht: Ein „word“ ist i.d.R. die native Datengröße eines Prozessors (z.B. 16 Bit oder 32 Bit), daher:
  - RISC: byte (8 bit), half word (16 bit), word (32 bit), double word (64 bit)
  - Intel: byte (8 bit), word (16 bit), double word (32 bit), quad word (64 bit)
- Die so genannte „Endianness“ definiert, wie Multi-Byte Daten im Speicher abgelegt werden („Byte Ordering“).
- “Big Endian”: Byte 0 (niedrigste Byte Adresse) ist das “most significant byte”, Byte 3 ist “least significant byte”
  - IBM 360/370, Motorola 68K, SPARC und die meisten RISCs
- “Little Endian”: Byte 0 ist “least significant byte”, Byte 3 ist “most significant byte”
  - Intel x86, VAX, Alpha

Beispiel 1: Byte-Anordnung

Adresse	Big Endian	Little Endian
184	12	78
185	34	56
186	56	34
187	78	12

Abgespeichert wird der 32-Bit Integerwert „12345678“ (hexadezimal!).
Die (Anfangs-)Adresse des Datums ist in beiden Fällen „184“.

Beispiel 2: Byte-Anordnung

```
unsigned char c[8]; // ‘a’,’b’,’c’,’d’,’e’,’f’,’g’,’h’
unsigned short s[4]; // 0x1234, 0x5678, 0x9011, 0x2233
unsigned int i[2]; // 0x98765432, 0x21098765
unsigned long long l[1]; // 0x0123456789112233
```

Address	00	01	02	03	04	05	06	07
00	a	b	c	d	e	f	g	h
08	12 34		56 78		90 11		22 33	
10	98 76 54 32				21 09 87 65			
18	01 23 45 67 89 11 22 33							

Big

Address	00	01	02	03	04	05	06	07
00	a	b	c	d	e	f	g	h
08	34 12		78 56		11 90		33 22	
10	32 54 76 98				65 87 09 21			
18	33 22 11 89 67 45 23 01							

Little

Kapitelübersicht

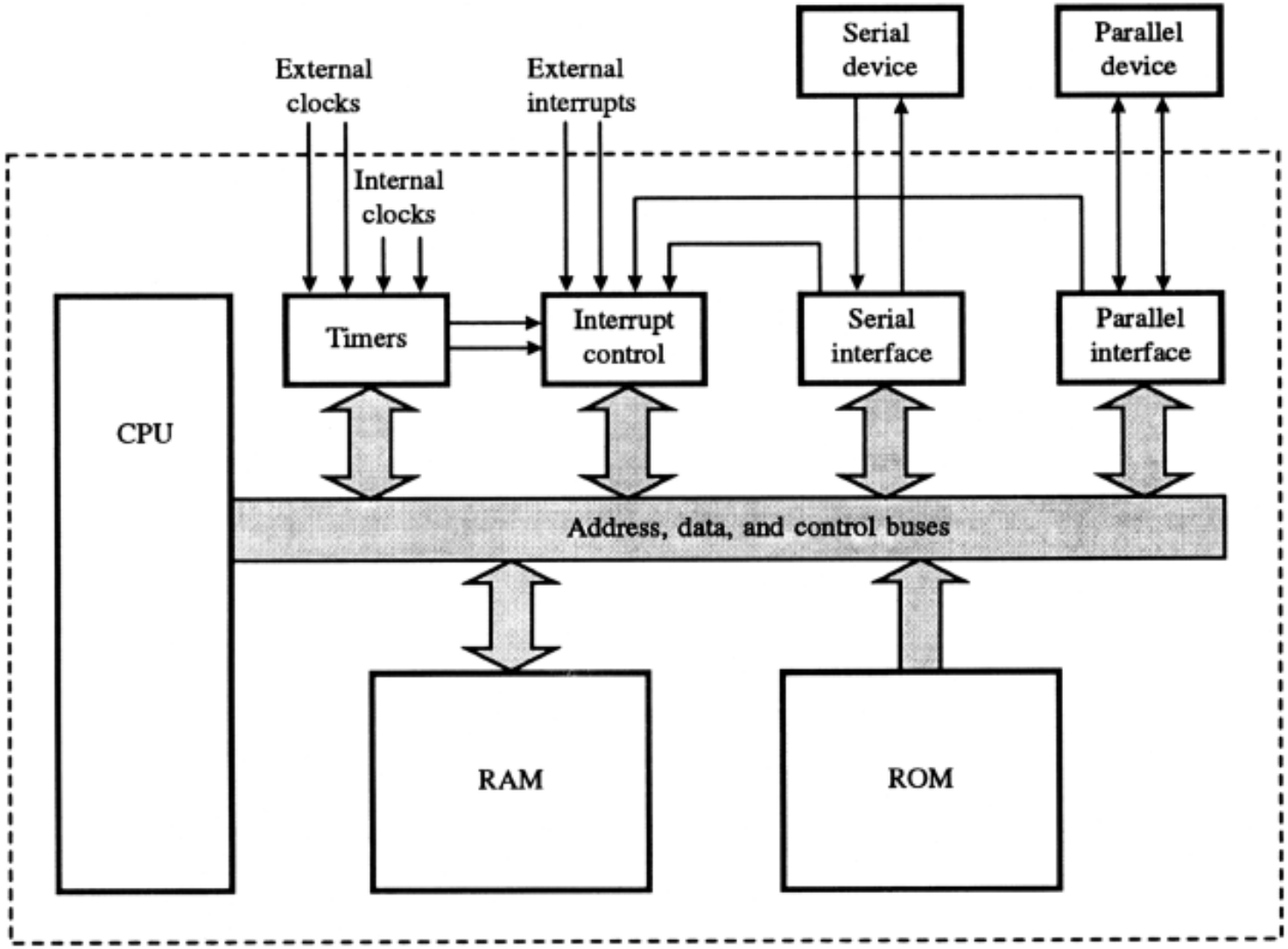
Zugriff auf Peripherieeinheiten

- In den Peripherieeinheiten sind ebenfalls Register oder Speicherzellen vorhanden. Diese dienen der Steuerung der Peripherie oder dem Datentransfer.
- Die Peripherie-Register werden i.d.R. wie Daten adressiert. Die Adressen der Register liegen daher im „Adressraum“ des Speichers und es wird ein Teil der Adressen für die Peripherieregister reserviert. Dies wird als „Memory-Mapped I/O“ bezeichnet. Der Zugriff erfolgt mit den Befehlen für das Schreiben/Lesen von Speicherzellen.

Speicheraufteilung und Memory Map

- Wenn ein Mikrocontrollersystem aufgebaut wird, dann muss die Aufteilung des Adressraums festgelegt werden (engl.: Memory Map).
- Die Aktivierung der Speicherbausteine und der Peripherieblöcke (Baustein oder „Gerät“, engl.: device) erfolgt über „Chip-Select“-Signale. Diese werden durch Dekodierung der Adressen oder eines Teils der Adressen erzeugt.
- Hierbei können auch Teile des Adressraums frei bleiben.
- Häufig werden nicht alle Adressbits dekodiert, so dass einem (I/O)-Device mehrere Adressen zugeordnet sein können.

Beispiel: Speicheraufteilung mit Peripherie



aus: MacKenzie, The 8051 Microcontroller

Beispiel: Memory Map

Beispiel: 16 Bit Adressen, daher 2^{16}=65536 mögliche Adressen (0000h-FFFFh), d.h. 64KB*.

Memory Map

Bereich	Größe (Anzahl Bytes)	Baustein
F000h – FFFFh	4096 (4 KB)	Interrupt Control
E000h – EFFFh	4096 (4 KB)	Timer
D000h – DFFFh	4096 (4 KB)	Serial Interface
C000h – CFFFh	4096 (4 KB)	Parallel Interface
8000h – BFFFh	16384 (16KB)	frei
4000h – 7FFFh	16384 (16KB)	RAM (Daten)
0000h – 3FFFh	16384 (16KB)	ROM (Programm)

*: 1 KB = 1 KiloByte = 1024 Byte

Kapitelübersicht

Programmierung von Prozessoren

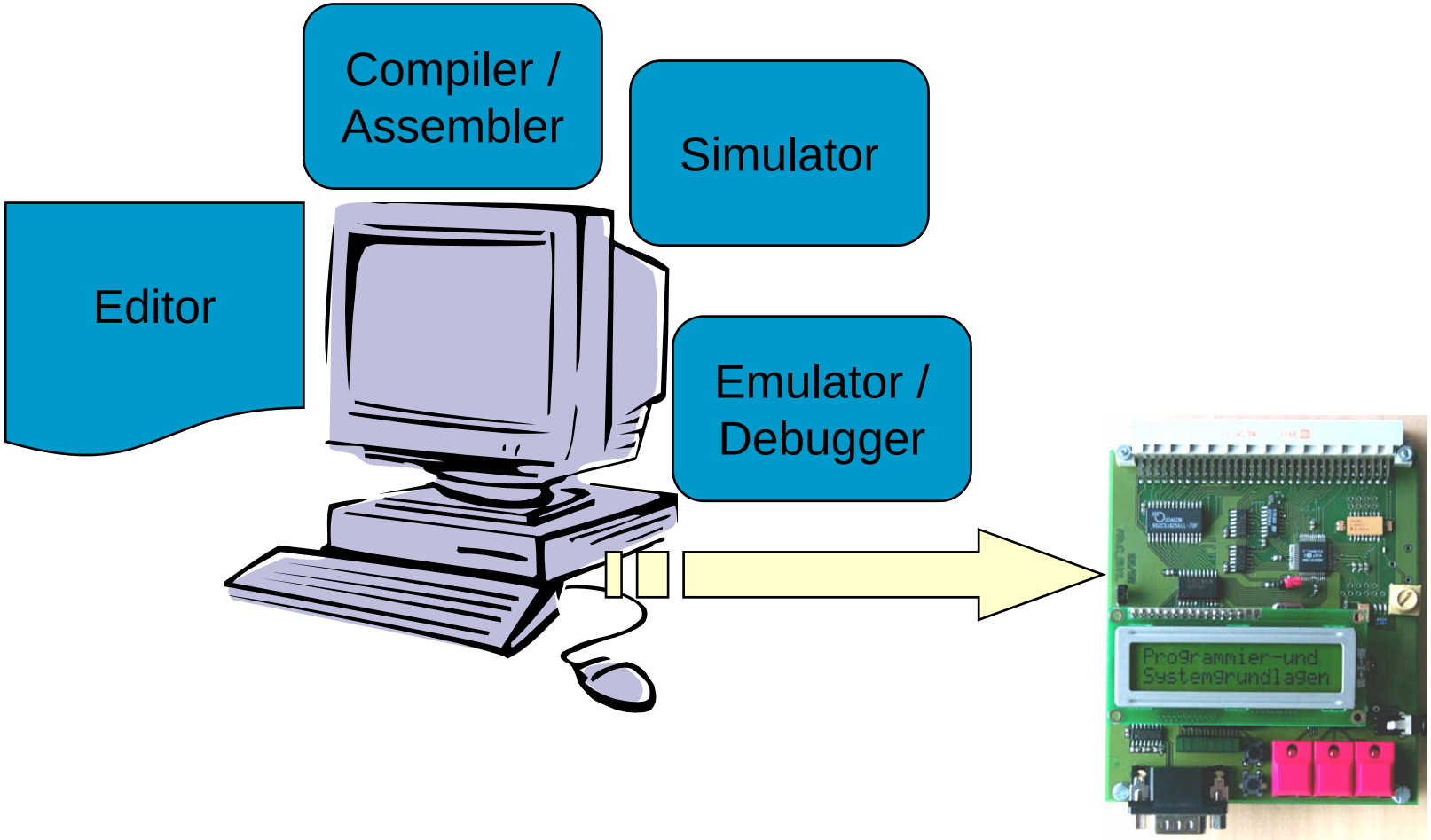
- Maschinensprache: Dies ist der direkt vom Prozessor lesbare Binärcode
- Assemblersprache: Dies ist eine direkte Abbildung des binären Maschinencodes in einen vom Menschen besser lesbaren, symbolischen Code. Der „Assembler“ setzt ein in Assemblersprache geschriebenes Programm in Maschinensprache um.
- Maschinen- und Assemblersprache sind immer spezifisch für einen Prozessor(-familie)
- Höhere Programmiersprache: Maschinenunabhängige Programmiersprache (z.B. C). Ein Compiler setzt ein Hochsprachenprogramm in ein Maschinenprogramm um.

Höhere Programmiersprache vs. Assembler

Höhere Programmiersprache	Assemblersprache
Leicht erlern- und anwendbar (Im Vergleich zu Assembler)	Schwer erlernbar
Syntax oft an menschliche Denkgewohnheiten angepasst	Platzsparende, stark komprimierte Syntax
Maschinenunabhängig	nur auf einem bestimmten Prozessortyp lauffähig
Abstrakte, maschinenunabhängige Datentypen (Ganzzahl, Fließkommazahl)	Datentypen des Prozessors (Byte, Wort, Langwort)
Oft zahlreiche und komplexe Kontrollstrukturen	Primitive Kontrollstrukturen, oft als Makros realisiert
Datenstrukturen (Feld, Record)	Nur einfache Typen
Weitgehende semantische Analyse möglich	Nur grundlegende semantische Analyse möglich

Beispiel:
`A:=2;`
`FOR I:=1 TO 20 DO`
`  A:=A*I;`
`ENDFOR;`
`PRINT(A);`
	Beispiel:.
	`  START ST`
	`ST:  MOV R1,#2`
	`  MOV R2,#1`
	`M1:  CMP R2,#20`
	`  BGT M2`
	`  MUL R1,R2`
	`  INI R2`
	`  JMP M1`
	`M2:  JSR PRINT`
	`  .END`

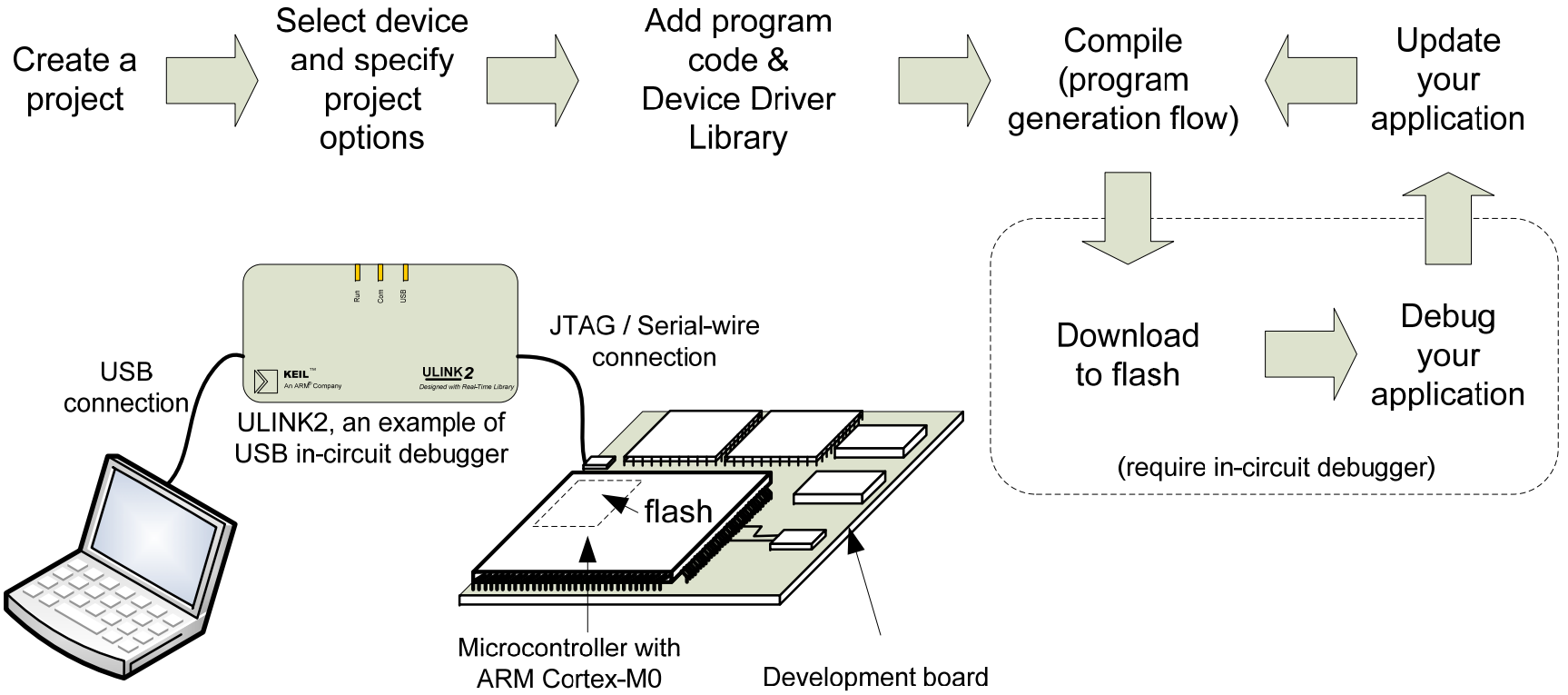
Ablauf der Softwareentwicklung



Ablauf der Softwareentwicklung (2)

- Für die Softwareentwicklung bei Mikrocontrollern benötigt man ein System, das auf einem Fremdcomputer (z.B. PC) eingesetzt wird. In einem Texteditor wird zunächst der Quelltext des erforderlichen Programms geschrieben. Hierbei kann man die typspezifische Assemblersprache oder auch eine höherer Programmiersprache wie C benutzen.
- Anschließend muss der Quelltext mit einem Cross-Assembler oder Cross-Compiler in die jeweilige Maschinensprache (Binärcode) übersetzt werden. Dieser Code kann dann auf verschiedene Weise auf die Mikrocontroller-Hardware übertragen werden.
- Ein Cross-Compiler oder Cross-Assembler ist ein Programm, welches auf einer Computerplattform (meist PC) läuft und Code für eine andere Computerplattform (µC) erzeugt.
- Für das Testen von komplexeren Programmen benötigt man zusätzliche Hilfsmittel. Im einfachsten Fall genügt ein Softwaresimulator auf einem Fremdcomputer (z.B. PC). Bei Echtzeitanwendungen benötigt man Hardware-Emulatoren mit komfortableren Debug-Möglichkeiten.
- Weitere Informationen in Kapitel 3 …

Entwurfsablauf für Cortex-M0-Systeme



Bildquelle: Joseph Yiu, The Definitive Guide to the ARM Cortex-M0

Entwurfsablauf für Cortex-M0-Systeme (2)

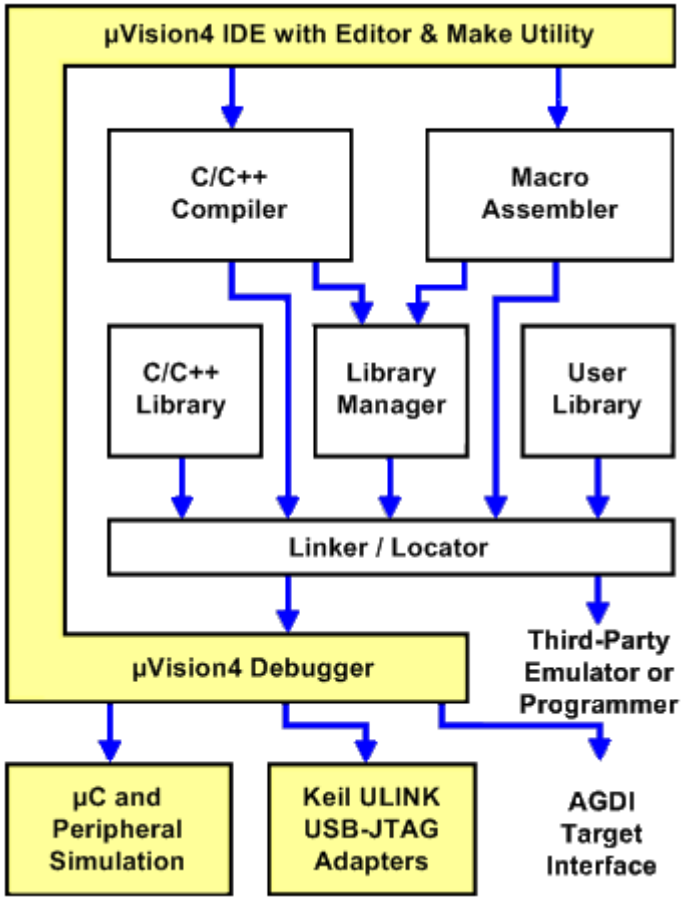
- Zu Beginn wird ein Projekt angelegt, in welchem alle Einstellungen gespeichert werden.
- Man wählt den Chip des Herstellers aus (z.B. Nuvoton NUC130VE3AN), damit sind bestimmte Voreinstellungen für Compiler und Linker schon vorgegeben.
  - Ggf. werden schon Startup-Codes hinzugefügt.
- Man kodiert den Quellcode, kompiliert und erstellt das „Image“.
  - Hierzu wird i.d.R. eine „Treiberbibliothek“ verwendet, welche die Programmierung erleichtert.

Entwurfsablauf für Cortex-M0-Systeme (2)

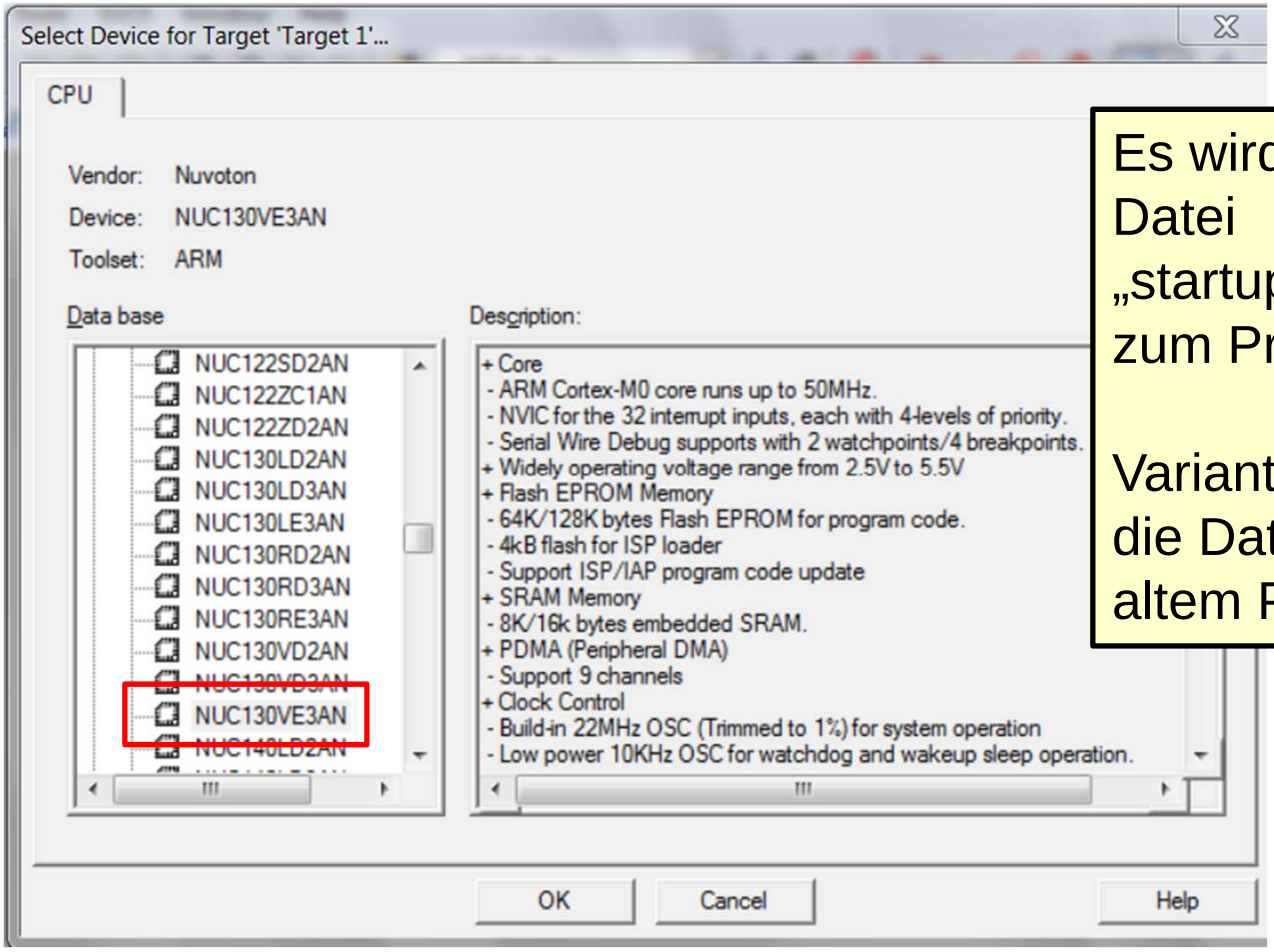
- Zum Laden des Programms auf das Zielsystem (d.h. Programmierung des Flash-Speichers) dient ein so genannter „Debug Adapter“ (auch: „In-Circuit Debugger“), welcher das Zielsystem mit dem Entwicklungsrechner verbindet.
  - z.B. ULINK2 von KEIL oder Nuvoton Nu-Link
- Der Debug Adapter ermöglicht neben dem Laden des Programms auch das so genannte „Debuggen“.
- Einfache Programme können auch ohne Zielsystem durch den Simulator simuliert werden.

Die KEIL µVision4

- µVision ist eine IDE (Integrated Development Environment), also eine Entwicklungsumgebung, die alle benötigten Werkzeuge integriert (Editor, Assembler, Compiler, Linker, Debugger, Simulator).



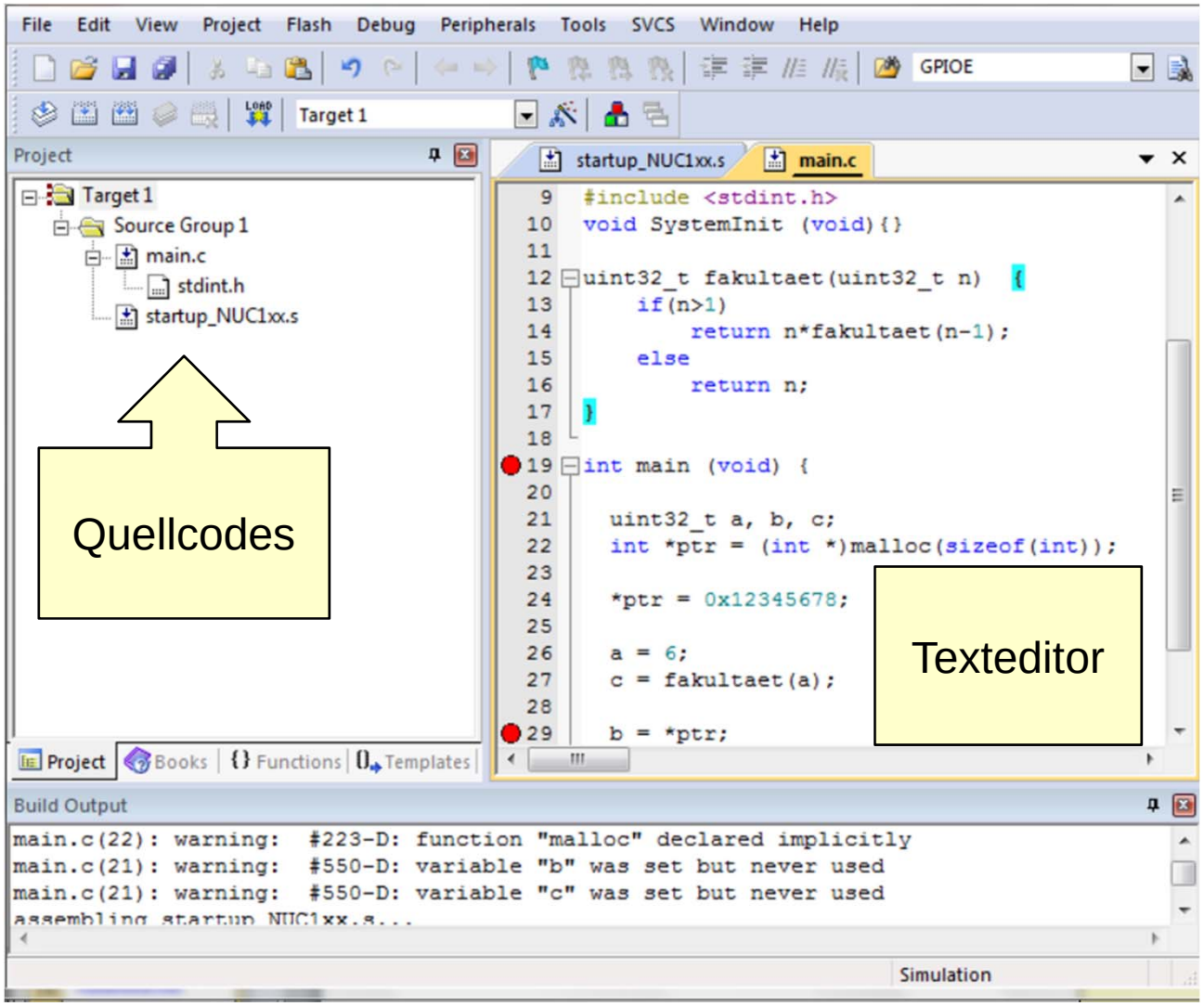
Anlegen eines Projektes: Auswahl des Chips



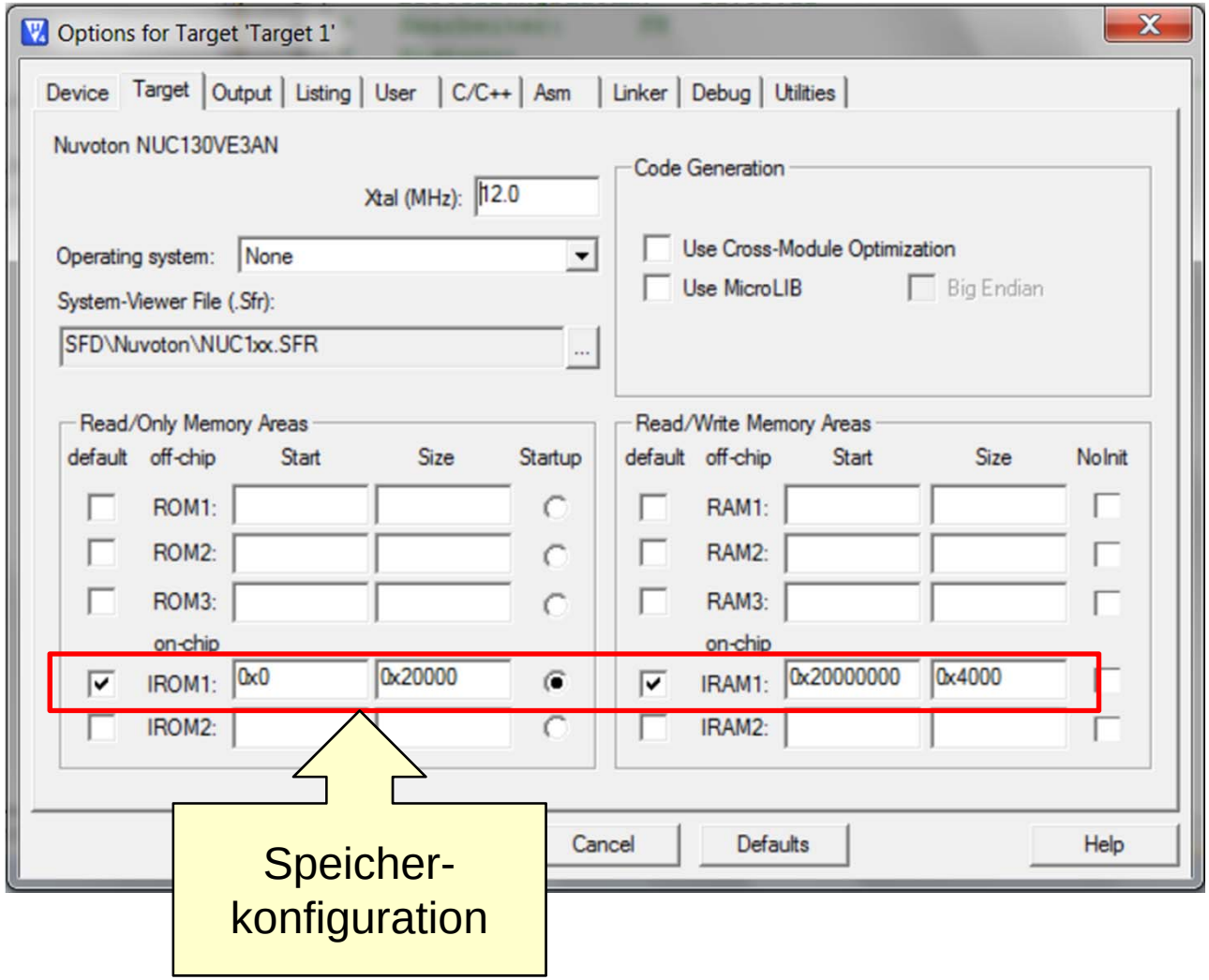
Es wird eine Datei „startup_NUC1xx.s“ zum Projekt hinzugefügt.

Variante: Man kopiert die Datei aus altem Projekt hinzu.

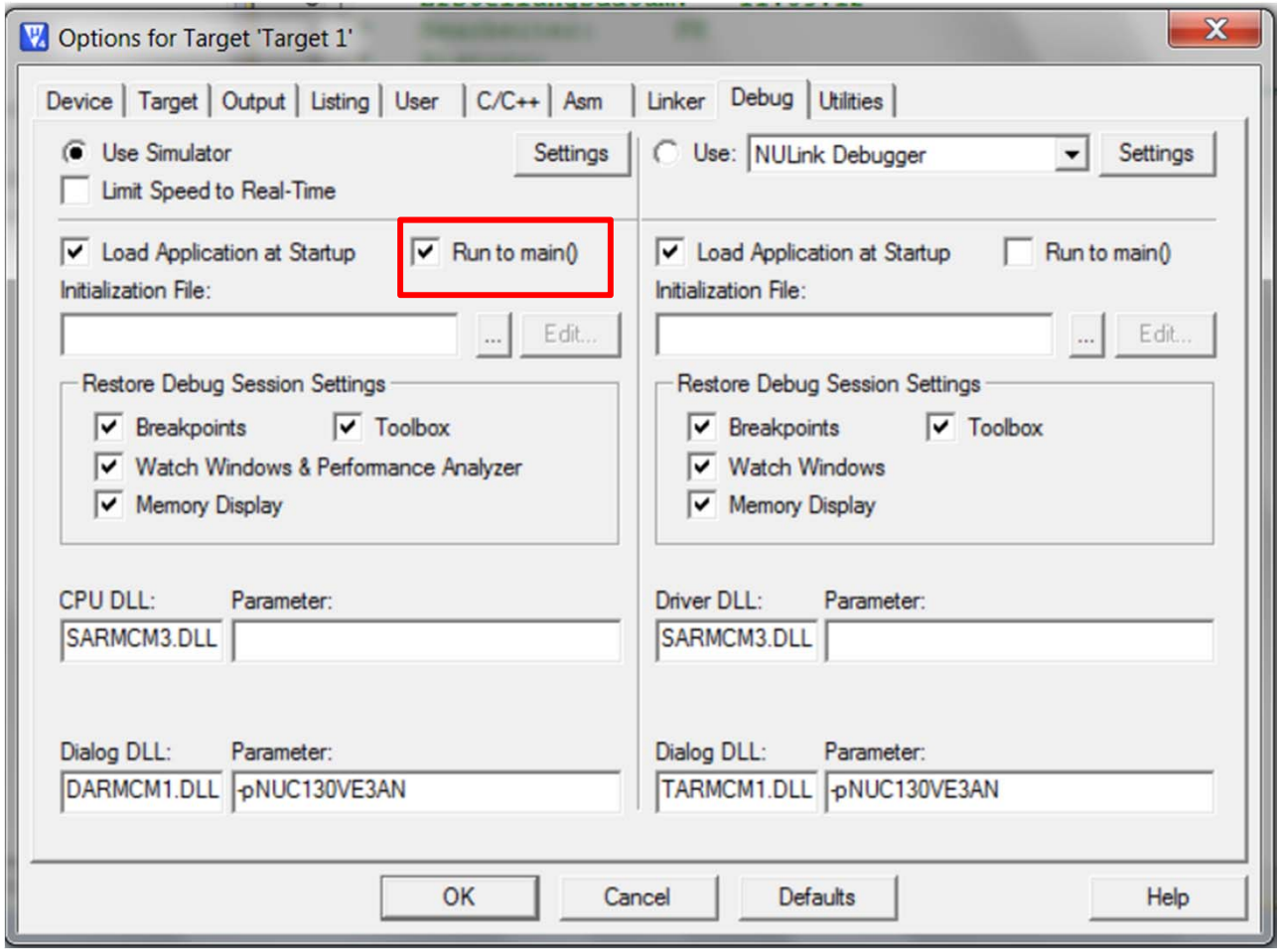
Projektoberfläche der µVision



Einstellen von Projektoptionen



Auswahl von Simulator oder Debugger



Programmierfehler und ihre Ursachen

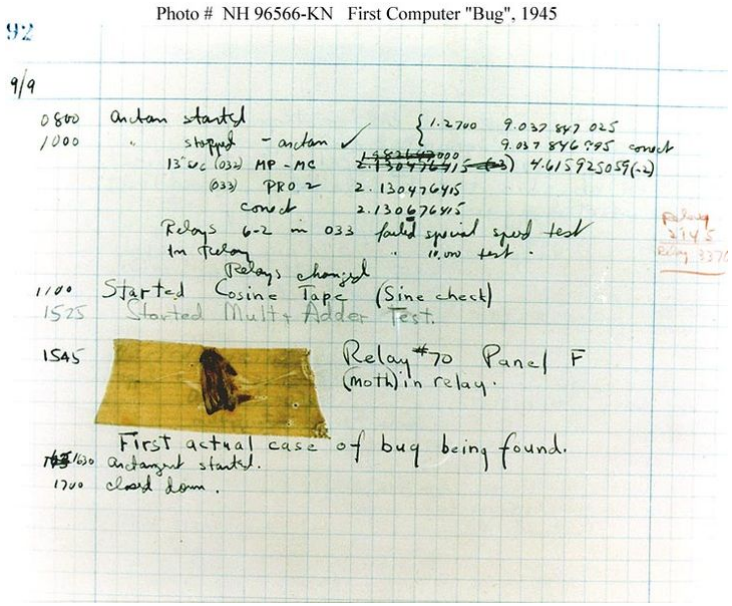
- Syntaxfehler: Werden vom Compiler oder Assembler entdeckt.
- Laufzeitfehler: Hier handelt es sich um Funktionsfehler oder „semantische“ Fehler des Programms.
  - Bestimmte Zustände oder Eingaben des Programms wurden nicht berücksichtigt.
  - Die Spezifikation des Programms ist mehrdeutig, unvollständig, ungenau oder fehlerhaft oder wurde vom Programmierer falsch verstanden.
  - Die Mikrocontroller-Hardware wurde nicht vollständig verstanden (z.B. Speicherzugriff, Stacküberlauf).
  - Interrupts zum „falschen“ Zeitpunkt sind oft Fehlerquellen.
  - ...

Programmierfehler und ihre Ursachen (2)

- Trotz sorgfältigster Programmierung lassen sich Fehler nicht vermeiden. Ein umfangreicher Test der Software unter verschiedensten Betriebsbedingungen des Systems ist daher unerlässlich.
- Fehler werden unter Programmierern als „Bugs“ bezeichnet (engl. Käfer, Wanze, Insekt).

Herkunft des Wortes „Bug“

- Das Wort „Bug“ wurde schon im 19. Jahrhundert für kleine Fehler in mechanischen und elektrischen Teilen verwendet. Knistern und Rauschen in der Telefonleitung würde z. B. daher rühren, dass kleine Tiere („Bugs“) an der Leitung knabbern. Thomas Edison hat 1878 an seinen Freund Tivadar Puskás einen Brief über die Entwicklung seiner Erfindungen geschrieben, in dem er kleine Störungen und Schwierigkeiten als „Bugs“ bezeichnete („*… that ‚Bugs‘ – as such little faults and difficulties are called – show themselves…*“).
- Einer modernen Legende zufolge ist die Bezeichnung in der Anfangszeit der Computer entstanden, als Insekten in den großen Maschinen die Funktionsweise der Relais störten und Kurzschlüsse verursachten. Die Erfindung des Begriffs wird oft der Computerpionierin Grace Hopper zugesprochen. Sie verbreitete die Geschichte, dass am 9. September 1945 eine Motte in einem Relais des Computers Mark II Aiken Relay Calculator zu einer Fehlfunktion führte. Die Motte wurde entfernt und in das Logbuch mit den Worten „First actual case of bug being found.“ („Das erste Mal, dass ein Bug auch wirklich gefunden wurde.“) geklebt.



Quelle: Wikipedia

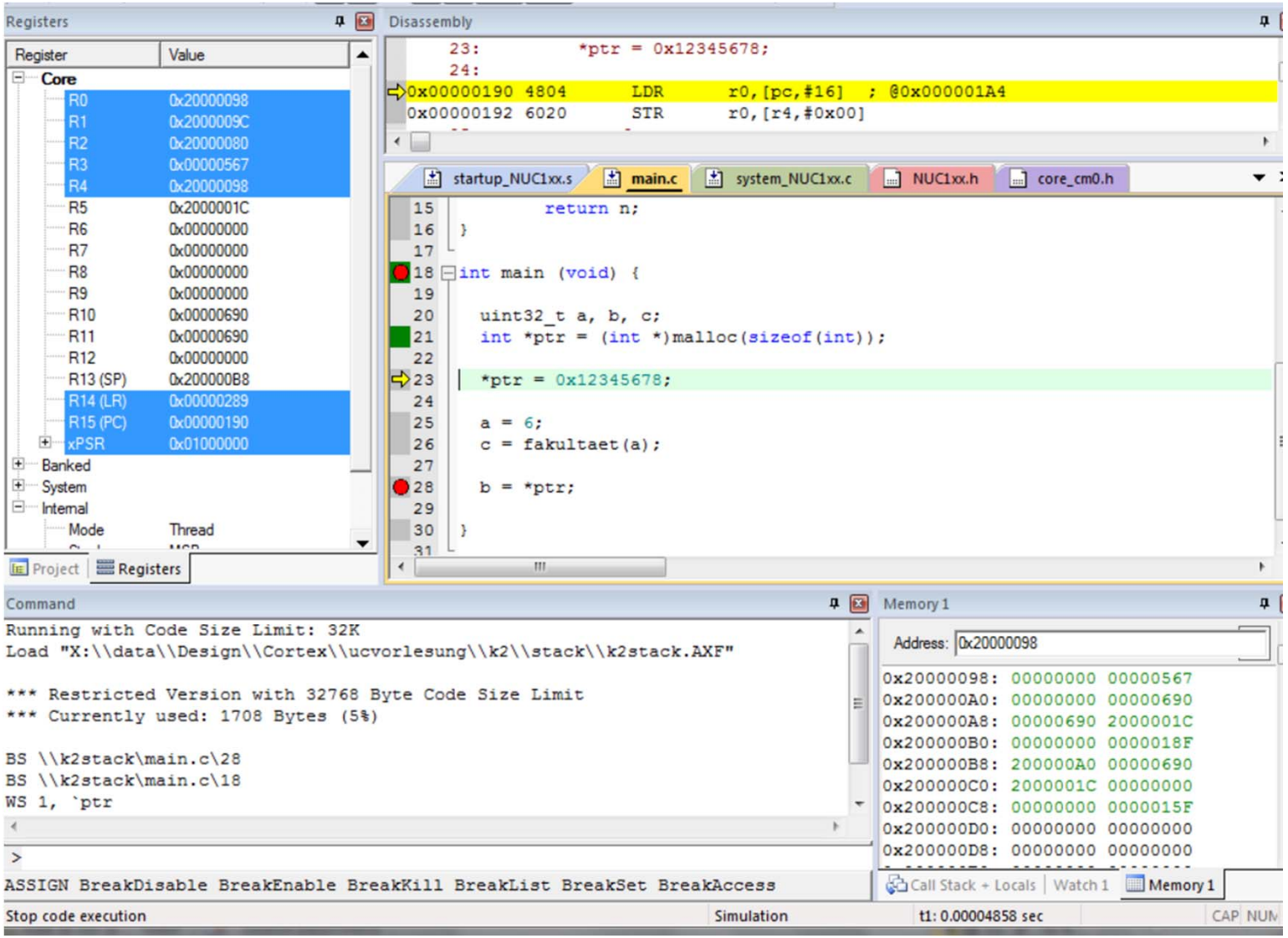
Simulator und Debugger

- Wenn der Quellcode erstellt und syntaktisch richtig ist, kann die Funktionalität des Programms getestet werden.
- Das Testen des Programms und das Aufspüren von Fehlern zählt zu den zeitaufwändigsten Arbeiten im Software-Entwicklungszyklus.
- Daher werden Werkzeuge, die zum Lokalisieren der Fehler im Quellcode benutzt werden, häufig als (Source-Level-)„Debugger“ bezeichnet:
  - **Simulator**: Die zu testende Software läuft nicht auf der Zielhardware, sondern der Zielprozessor (und teils auch die Peripherie) wird auf dem Entwicklungsrechner simuliert (Instruktionssatz-Simulator).
  - **Debugger**: Die Zielhardware besitzt bestimmte Einrichtungen, die ein Debuggen im System erlauben und die mit dem Debugger auf dem Entwicklungsrechner kommunizieren.
  - In der Regel bietet der Simulator die gleiche Bedienoberfläche mit identischen Funktionen wie der Debugger an. Das Problem beim Simulator ist die unzureichende Modellierung der Peripherie.

Debugger-Funktionen

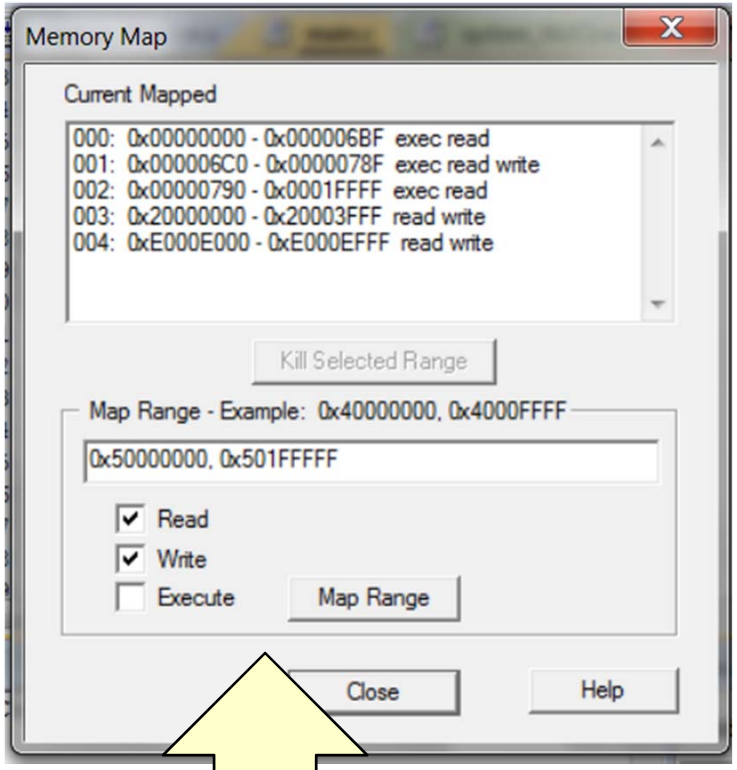
- Die Funktionen, die ein Debugger anbietet, hängen hauptsächlich von den Debug-Einrichtungen der Zielhardware ab. Folgende Funktionen sind zumeist vorhanden (auch beim Simulator):
  - Laden des Anwenderprogramms (Loader) und Start (F5)
  - Einzelschrittausführung (single-stepping)
    - Ausführen eines Befehls (step, F11) oder von Funktionen (step over, F10)
  - Setzen von Haltepunkten (Breakpoints)
    - Programmhaltepunkte (im Quelltext)
    - Datenhaltepunkte (Referenzierung von Variablen)
    - Bedingte Haltepunkte
  - Auslesen und Verändern von Register- und Speicherinhalten

Der Simulator / Debugger der µVision



Simulator und Memory Map

- Beim NUC130 ist die Peripherie nicht modelliert.
- Daher ist auch die Adress-Map nicht vollständig. Der Simulator prüft aber Zugriffe auf nicht spezifizierte Bereiche.
  - Fehler:
    ```
    *** error 65: access
    violation at
    0x50000100 : no
    'write' permission
    ```



Angabe zusätzlicher Bereiche im Simulator (Debug->Memory Map)

Initialisierungsdatei für Simulator / Debugger

- Einstellungen von Debugger / Simulator können über eine Initialisierungsdatei vorgegeben werden („Initialization File“).

Inhalt von z.B. debug.ini:

```
MAP 0x50000000, 0x501FFFFF READ WRITE
```

Kapitelübersicht

Von CISC zu RISC

- Performance-Untersuchungen in den 70er Jahren (z.B. von IBM) an Großrechnern (mainframes) brachten folgende Ergebnisse:
  - Compiler konnten die Vielzahl von Instruktionen und unterschiedlichen Adressierungsarten nicht verwenden, nur wenige einfache Befehle wurden verwendet.
  - Die Komplexität der Maschinen führte zu langsamen Ausführungszeiten der Befehle (CPI_i hoch, f niedrig).
- IBM Studie des IBM370-Rechners (200 Befehle):
  - 10 Instruktionen für 80% des Programms
  - 21 Instruktionen für 90% des Programms
  - 30 Instruktionen für 99% des Programms
- Ziel der RISC-Projekte Anfang der 80er: Entwicklung von leistungsfähigen Rechnerarchitekturen

CISC vs. RISC

- Complex Instruction Set Computer
  - Einige hundert Instruktionen
  - Dutzende von Adressierungsarten
  - Viele Datentypen
  - Komplexe Steuerlogik
- Reduced Instruction Set Computer
  - weniger und einfachere Instruktionen
  - weniger Adressierungsarten
  - weniger Datentypen
  - Load/Store-Architektur, viele Arbeitsregister
  - einfache Steuerlogik
  - schnellere Zykluszeit
  - Single-Cycle-Execution (CPI = 1) durch Pipelining

Protagonisten der RISC-Philosophie

- IBM, 1975: IBM 801
- University of California, Berkeley:
  - David Patterson und Carlo H. Sequin, ab 1980 RISC Projekt
  - 1982: RISC-I, 1983: RISC-II
  - Vorfahre der SUN-SPARC-Rechner
- Stanford University:
  - John L. Hennessy, ab 1981 MIPS Projekt (Microprocessor without Interlocked Pipeline Stages)
  - Vorfahre der MIPS-Rechner

Einige wichtige RISC CPUs

Year	Company	Model	Data Width
1986	MIPS	R2000	32
1986	Hewlett-Packard	PA-RISC	32
1986	Advanced RISC Machines	ARM	32
1987	SUN	Sparc	32
1988	Motorola	88000	32
1992	Hitachi	SuperH	32
1991	MIPS	R4000	64
2000	Compaq (DEC)	Alpha	64
2000	HP	8800	64
2000	IBM	Power4	64
2000	SUN	UltraSparc3	64
2000	Intel	IA-64 (Itanium)	64

RISC – CISC: Situation heute

- In den 80er Jahren ein Glaubenskrieg, Unterscheidung RISC-CISC heute nicht mehr so scharf.
- Viele leistungssteigernde RISC-Maßnahmen sind auch in CISC-Architekturen eingeflossen, siehe z.B. Pentium.
- Leistungsstarke Architekturen sind heute aber im Kern RISC-Architekturen

Was ist eine „Prozessor-Architektur“?

- Eine Architektur (auch: Instruction Set Architecture, ISA) definiert folgendes:
  - Instruktionssatz:
    - Maschinen/Assemblerbefehle
    - Instruktionsformat der Befehle (32-Bit, 16-Bit, Bedeutung der einzelnen Bits in einer Instruktion)
  - Programmiermodell (engl.: programmers model):
    - Prozessor-Modi
    - Register
    - Organisation des Speichers und der Peripherieeinheiten
  - Exceptions und Interrupts
  - Debugging-Einrichtungen

Load-Store-Architekturen

- Leistungsfähige Prozessoren sind RISC-Rechner, die auch als „Load-Store“-Architekturen bezeichnet werden.
  - Separate Instruktionen für Speicherzugriff (load, store)
  - Arithmetische Instruktionen arbeiten nur mit Registern
  - Die Maschine verfügt über viele Arbeitsregister (typisch 16 bis 32 Register)
  - Fixes Instruktionsformat, z.B. 32 Bit
- Die höhere Leistungsfähigkeit wird erreicht durch:
  - Weniger Speicherzugriffe durch viele Register
  - Effizientes „Pipelining“ (Fließbandverarbeitung) der Befehle

Beispiel: Einfache Load-Store-Architektur

- Instruktionssatz:

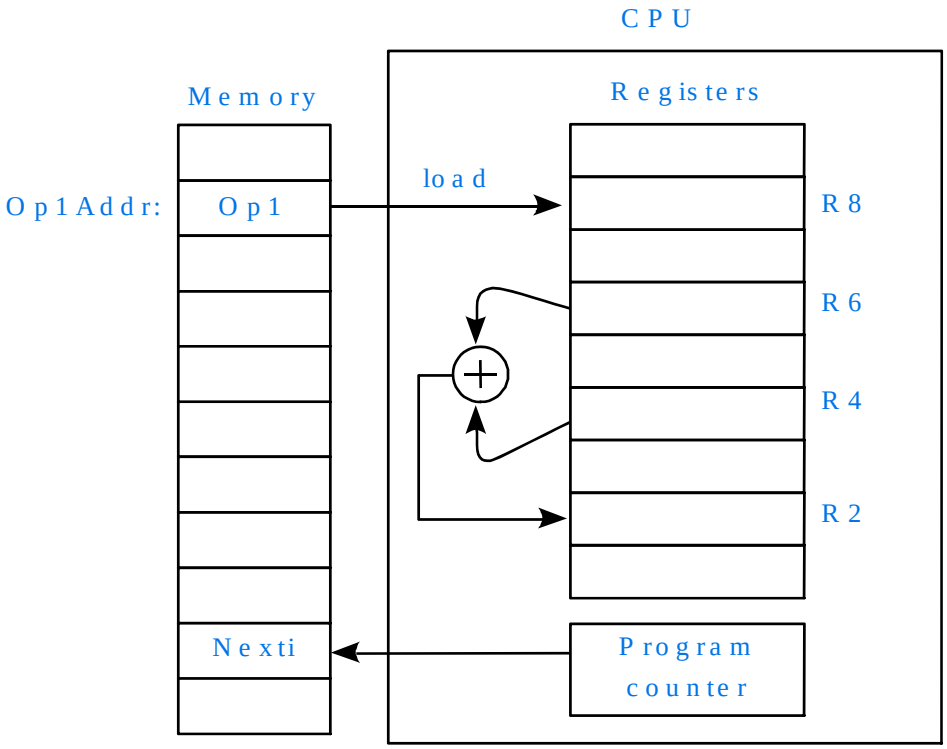
  add *Rx, Ry, Rz* sub *Rx, Ry, Rz* mul *Rx, Ry, Rz*
  load *Rx, M* store *M, Rx*

- Beispiel C-Code: E = A*B - (A+C*B)
  (A, B, C und E sind die Adressen von Variablen im Hauptspeicher,
  R1 bis R7 sind Arbeitsregister)

  Assembler/Maschinencode:

```
load R1, A
load R2, B
load R3, C
mul R4, R3, R2          ;      C*B
add R5, R1, R4          ;      A + C*B
mul R6, R1, R2          ;      A*B
sub R7, R6, R5          ;      A*B - (A+C*B)
store E, R7
```

Schema einer Load-Store-Maschine



Instruction formats

load R8, Op1 (R8 <- Op1)

load	R8	Op1Addr

add R2, R4, R6 (R2 <- R4 + R6)

add	R2	R4	R6

8051: Akkumulator-Maschine