

# **Digitale Systeme und Computersysteme**

## *Lehrplanbereich: Computerarchitekturen*



<https://monster6502.com>

### *Auszug Lehrplan*

*Computerarchitekturen*

*Aufbau und Arbeitsweise eines Mikrocontrollers*

*das Programmiermodell eines Mikrocontrollers erklären und einfache Programme entwickeln.*

Dieses Skript wurde von **Michael Kupfner** für das Unterrichtsfach **DIC – Digitale Systeme und Computersysteme, 4AHEL, Schuljahr 2025/26** an der HTBLVA Anichstraße erstellt. Eine Weitergabe oder Vervielfältigung außerhalb dieses Unterrichts ist nicht gestattet.

## 1 Einführung und Begriffsdefinitionen

... ein **Rechner** (engl. *Computer*) besteht in seiner Grundstruktur aus drei Hauptkomponenten:

**Hardware:** Umfasst alle **mechanischen, elektrischen und elektronischen Bauteile** eines Computersystems.

Beispiele: Gehäuse, Leiterplatten, Prozessoren, Speicherchips, Tastatur, Display.

**Software:** Umfasst **alle Programme**, die auf dem Rechner ausgeführt werden. Dazu zählen:

- **Betriebssystem** (*Operating System*), z. B. Windows, Linux
- **Hilfsprogramme** (*Tools*), z. B. Compiler, Debugger
- **Anwendungsprogramme** (*Applications*), z. B. Textverarbeitung, Spiele

**Firmware:** Befindet sich in einem **Festwertspeicher** (*ROM, EPROM, EEPROM*) oder liegt als **Mikroprogramm** vor. Sie bildet die Mittelschicht zwischen Hardware und Software. Beispiele: BIOS im PC



### KI-Aufgabe: BIOS und UEFI

Das BIOS ist ein klassisches Beispiel für Firmware. Befrage eine KI, welche Aufgaben das BIOS in einem Rechner übernimmt.

Neue Systeme werden jedoch mit **UEFI** (*Unified Extensible Firmware Interface*) anstelle eines klassischen BIOS ausgeliefert. Notiere **drei wesentliche Unterschiede** zwischen BIOS und UEFI in einer kurzen Tabelle.

Computer begegnen uns heute überall – oft, ohne dass wir es merken:

- Im Smartphone, das wir täglich benutzen
- In der Waschmaschine, die automatisch das richtige Waschprogramm wählt
- Im Auto, das Bremsen, Motor und Sicherheitssysteme überwacht
- In Industrieanlagen, die Produktionsprozesse steuern

Wenn wir an einen Computer denken, haben wir meist einen PC, Laptop oder Tablet vor Augen – also ein System mit Bildschirm, Tastatur oder Touchscreen, Festplatte und Betriebssystem. Diese Geräte sind leistungsfähig, aber groß, teuer und für den Einsatz im Alltag oft unnötig komplex.

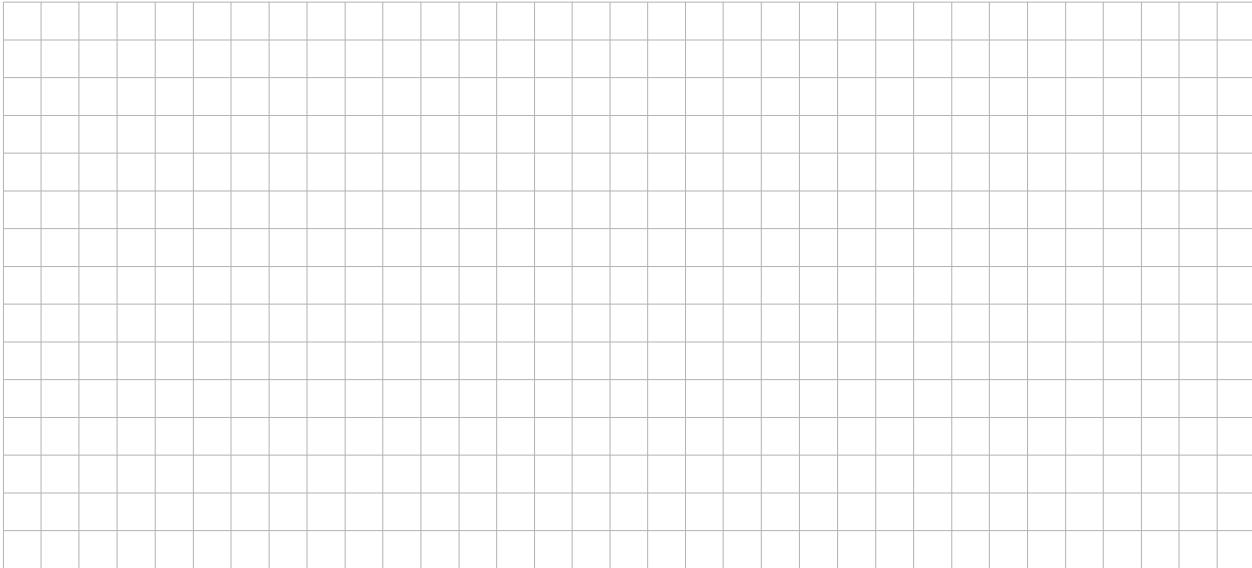
In der Elektronik arbeiten wir oft mit **kleineren Rechnern**, die speziell für **eine Aufgabe** entwickelt werden – zum Beispiel um eine Maschine zu steuern, Messwerte zu erfassen oder Daten zu übertragen.

Ein solcher **Mikrocomputer** (*Microcomputer*) enthält:

- eine **Zentraleinheit** (*CPU – Central Processing Unit*),
- **Speicher** (*Memory*),
- und **Ein-/Ausgabeschnittstellen** (*Input/Output Interfaces*)

... verbunden per **Systembus** und oft alles auf einem einzigen Chip oder einer kleinen Leiterplatte.

Blockschaltbild eines Digitalrechners / Mikrocomputers:



Die **CPU** führt die eigentliche Datenverarbeitung durch und übernimmt auch die Koordination aller internen Abläufe. Im Wesentlichen besteht die Zentraleinheit aus den Komponenten

- **Steuerwerk** (*Control Unit*)
- **Register** (*Registers*)
- **Rechenwerk** (*ALU – Arithmetic Logic Unit*)

Die **Ein-/Ausgabeeinheiten** sind die Schnittstelle zur „Außenwelt“. Dazu gehören z.B. Drucker oder Bildschirme genauso wie z.B. ein Analog-/Digital-Wandler.

Der **Speicher** wird in **Daten- und Programmspeicher** (*Data Memory, Program Memory*) unterteilt, wobei diese Unterteilung nicht zwangsmäßig unterschiedliche Hardware bedarf. Je nach Anwendungsfall steht auch nur ein gemeinsamer Speicher zur Verfügung; d.h. physisch können beide im selben Speicherbaustein liegen. Je nach **Anbindung** des **Daten- und Programmspeichers** unterscheidet man zwischen der

- **Von-Neumann-Architektur**
- **Harvard-Architektur**

Bei der **Von-Neumann-Architektur** sind Programm- und Datenspeicher **am selben Systembus** angeschlossen (Programm- und Datenspeicher können eine Speichereinheit sein). Der Vorteil liegt in einem einfacheren Aufbau, aber Befehle und Daten können nicht gleichzeitig übertragen werden.

*Von-Neumann-Architektur:*



Bei der **Harvard-Architektur** hingegen sind Programm- und Datenspeicher über **getrennte Systembusse** an die CPU angeschlossen. Dies erlaubt eine zeitgleiche Übertragung von Programm- und Dateninformationen. Damit erzielt man eine höhere Verarbeitungsleistung, jedoch mit höherem Hardwareaufwand.

*Harvard-Architektur:*



#### Merke: Programmspeicher und Datenspeicher

Der **Programmspeicher** enthält die Maschinenbefehle, die von der CPU ausgeführt werden. Damit das System nach dem Einschalten sofort lauffähig ist, wird hier **nichtflüchtiger Speicher** (z. B. ROM oder Flash) eingesetzt. Der **Datenspeicher** hingegen hält Variablen, Zwischenergebnisse und den Stack; er besteht aus **flüchtigem Speicher** (RAM), dessen Inhalt beim Ausschalten verloren geht.

Jeder Speicher besteht aus vielen Speicherzellen, die **adressiert** werden. Mit einer Adresse (z. B. 0x0000...0xFFFF) wählt die CPU genau eine Zelle aus. Über den Systembus kann der Inhalt gelesen (*Read*) oder verändert (*Write*) werden.

Physisch können Programm- und Datenspeicher in einem einzigen Baustein vereint sein (Von-Neumann-Architektur) oder getrennt vorliegen (Harvard-Architektur).

Der **Systembus** (*System Bus*) verbindet die Komponenten. Hierüber erfolgt die Datenübertragung, Synchronisation, Meldung von Ereignissen, ....

Als **Mikroprozessor (μP)** bezeichnet man die auf einem Chip realisierte CPU eines Computersystems.

Ein **Mikrocontroller (μC)** enthält neben dem μP als Zentraleinheit auch Daten- und Programmspeicher sowie Schnittstellen und Peripheriekomponenten auf einem Chip. Früher war auch der Begriff *Einplatinen-Mikrocomputer* geläufig; darunter versteht man einen μC samt Takt- und Spannungsversorgung auf einer einzigen Platine.

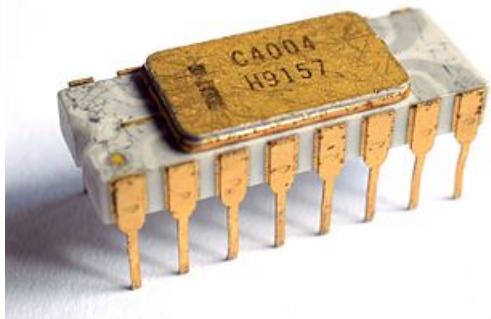
**Systems on Chip (SoC)** besitzen mehrere Verarbeitungseinheiten (CPUs), Programm- und Datenspeicher sowie vielfältige analoge und digitale Funktionseinheiten auf einem Chip.

**System in Package (SiP)** besteht aus einem oder mehreren SoC, umfangreichen Datenspeicher sowie der Spannungsversorgungskomponenten auf einem Chip. Es muss hier nur mehr noch der externe Programmspeicher (z.B. in Form einer SD-Karte) angebunden werden. Damit lassen sich sehr einfach Ein-Chip-Linux-Systeme realisieren.

**Digitale Signalprozessoren** (engl. digital signal processor, **DSP**) sind Mikroprozessoren mit speziellem Befehlssatz für die schnelle Verarbeitung komplexer mathematischer Algorithmen. DSP findet man z.B. im Bereich der Audio- und Videoverarbeitung, in Modems, ...

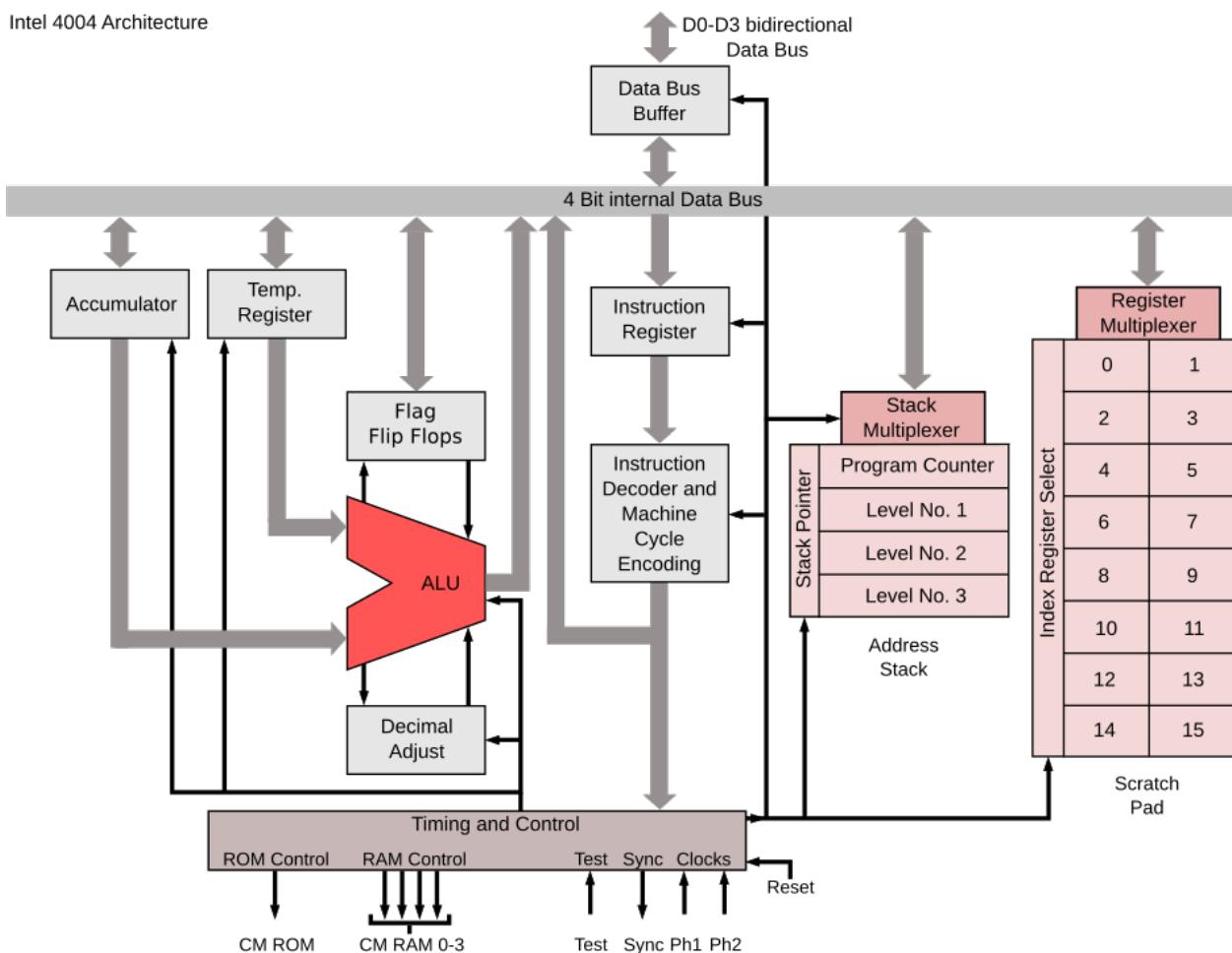
## Beispiele

- **Intel 4004** (4-Bit-Mikroprozessor des Chipherstellers Intel, kam am 15. November 1971 auf den Markt und gilt als der erste Ein-Chip-Mikroprozessor, der in Serie produziert und am freien Markt vertrieben wurde)



Produktion: 1971 bis 1981  
 Produzent: Intel  
 Prozessortakt: 500 kHz bis 740 kHz  
 Fertigung: 10 µm, PMOS  
 Befehlssatz: Intel 4 Bit  
 Sockel: 16-pin DIP

Intel 4004 Architecture



Quelle: [https://de.wikipedia.org/wiki/Intel\\_4004](https://de.wikipedia.org/wiki/Intel_4004)

- ATmega328 von Microchip  
*Quelle: Auszug Datenblatt Microchip*



## ATmega48A/PA/88A/PA/168A/PA/328/P

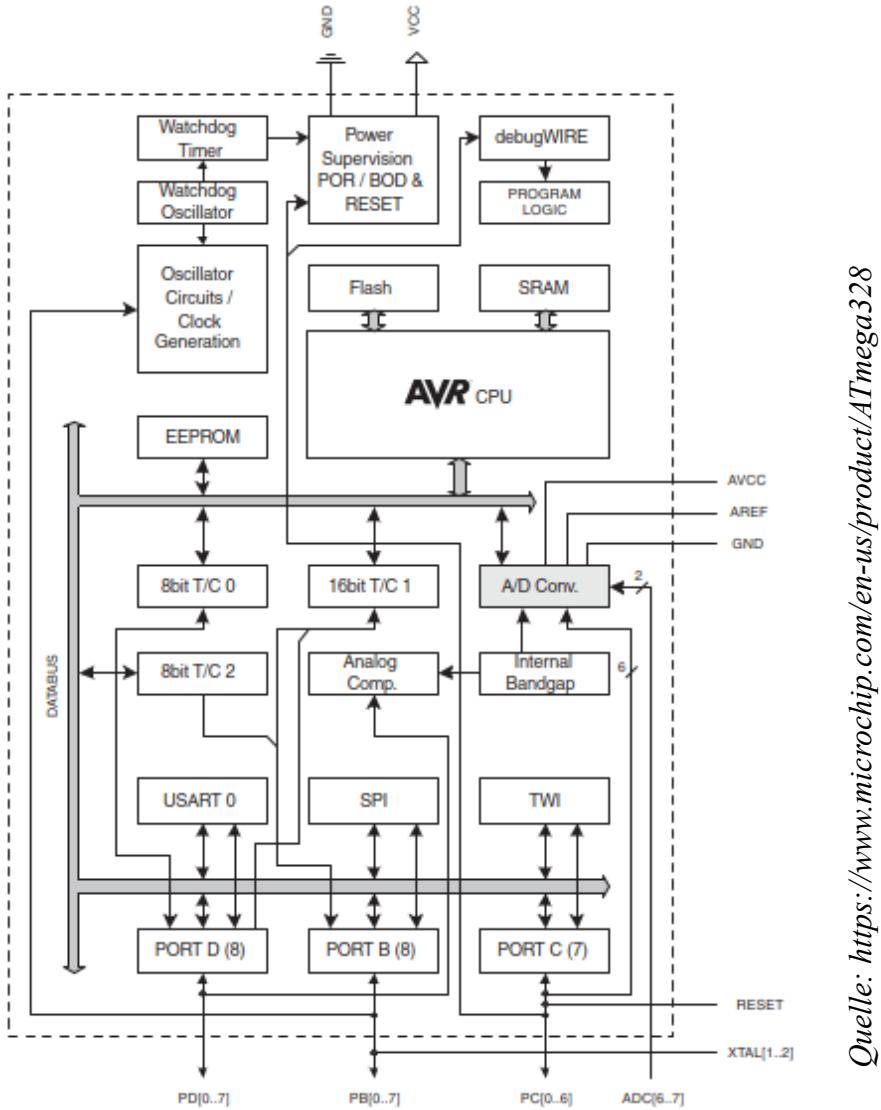
### megaAVR® Data Sheet

#### Introduction

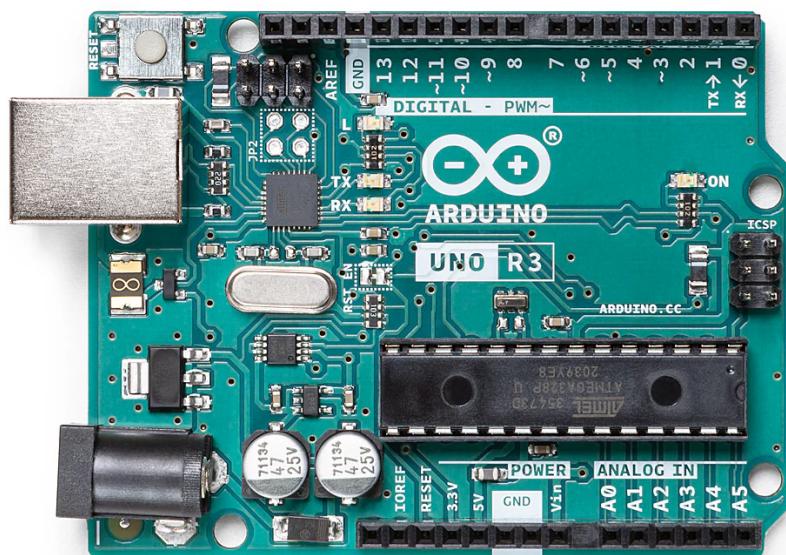
The ATmega48A/PA/88A/PA/168A/PA/328/P is a low power, CMOS 8-bit microcontrollers based on the AVR® enhanced RISC architecture. By executing instructions in a single clock cycle, the devices achieve CPU throughput approaching one million instructions per second (MIPS) per megahertz, allowing the system designer to optimize power consumption versus processing speed.

#### Features

- High Performance, Low Power AVR® 8-Bit Microcontroller Family
- Advanced RISC Architecture
  - 131 Powerful Instructions – Most Single Clock Cycle Execution
  - 32 x 8 General Purpose Working Registers
  - Fully Static Operation
  - Up to 20 MIPS Throughput at 20MHz
  - On-chip 2-cycle Multiplier
- High Endurance Non-volatile Memory Segments
  - 4/8/16/32KBytes of In-System Self-Programmable Flash program memory
  - 256/512/512/1KBytes EEPROM
  - 512/1K/1K/2KBytes Internal SRAM
  - Write/Erase Cycles: 10,000 Flash/100,000 EEPROM
  - Data retention: 20 years at 85°C/100 years at 25°C<sup>(1)</sup>
  - Optional Boot Code Section with Independent Lock Bits
    - In-System Programming by On-chip Boot Program
    - True Read-While-Write Operation
  - Programming Lock for Software Security
- QTouch® library support
  - Capacitive touch buttons, sliders and wheels
  - QTouch and QMatrix™ acquisition
  - Up to 64 sense channels
- Peripheral Features
  - Two 8-bit Timer/Counters with Separate Prescaler and Compare Mode
  - One 16-bit Timer/Counter with Separate Prescaler, Compare Mode, and Capture Mode

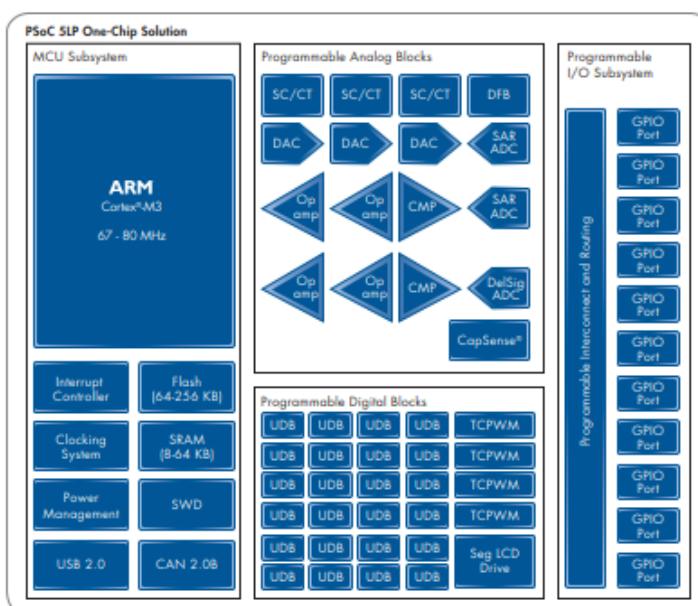


Quelle: <https://www.microchip.com/en-us/product/ATmega328>

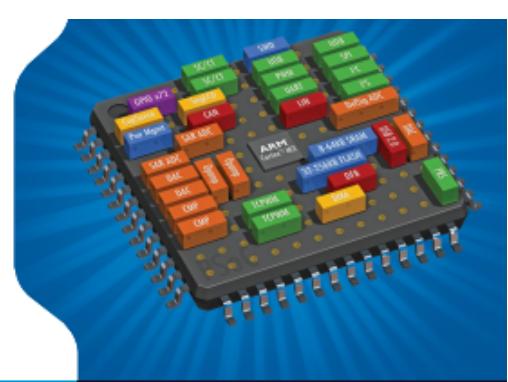


Quelle: <https://store.arduino.cc/products/arduino-uno-rev3/>

- **PSoC 5LP** von Infineon (Cypress):



**CYPRESS**  
**PSOC® 5LP**  
UNMATCHED PARALLEL PROCESSING PERFORMANCE WITH  
AN ARM® CORTEX®-M3 CPU, DIGITAL COPROCESSOR,  
HIGH PRECISION ANALOG AND CPLD-BASED LOGIC IN A  
ONE-CHIP SOLUTION



**PRODUCT OVERVIEW**

**FEATURES**

**32-BIT ARM® CORTEX®-M3 CPU**

- Up to 80-MHz Clock Speed
- Up to 256KB Flash, 64KB SRAM
- 24-Channel DMA Controller
- 24-bit, 64-tap Digital Filter Block
- Controller Area Network (CAN 2.0)
- LCD Segment Drive

**CapSense® WITH SMARTSENSE® AUTO-TUNING**

- Capacitive Sigma-Delta (CSD) Controller
- CapSense on up to 62 I/Os

**PROGRAMMABLE ANALOG BLOCKS**

- 1x 8- to 20-bit Delta-Sigma ADC
- 2x 12-bit SAR ADCs
- 4x 8-bit DACs
- 4x Comparators
- 4x Opamps
- 4x Programmable Analog Blocks (SC/CT)
- 1.024 V ±0.1% Voltage Reference

**PROGRAMMABLE DIGITAL BLOCKS**

- 4x Timer/Counter/PWM Blocks
- 24x Universal Digital Blocks (UDBs)
- Serial Communication over I2C, SPI, UART

**SUPPLY VOLTAGE**

- 1.2 - 5.5 V
- 4x Power Domains

**PACKAGES**

- 68-QFN
- 100-TQFP

PSOC 5LP Block Diagram

Quelle: <https://www.infineon.com/assets/row/public/documents/cross-divisions/45/infineon-psoc-5lp-product-overview-productbrief-en.pdf>

- STM32F437xx / STM32F439xx von STMicroelectronics  
Quelle: Datenblatt STM32F437xx/STM32F439xx DocID024244 Rev 2



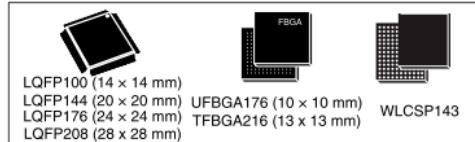
## STM32F437xx STM32F439xx

ARM Cortex-M4 32b MCU+FPU, 225DMIPS, up to 2MB Flash/256+4KB RAM, crypto, USB OTG HS/FS, Ethernet, 17 TIMs, 3 ADCs, 20 comm. interfaces, camera&LCD-TFT

Datasheet - production data

### Features

- Core: ARM 32-bit Cortex™-M4 CPU with FPU, Adaptive real-time accelerator (ART Accelerator™) allowing 0-wait state execution from Flash memory, frequency up to 180 MHz, MPU, 225 DMIPS/1.25 DMIPS/MHz (Dhrystone 2.1), and DSP instructions
- Memories
  - Up to 2 MB of Flash memory organized into two banks allowing read-while-write
  - Up to 256+4 KB of SRAM including 64-KB of CCM (core coupled memory) data RAM
  - Flexible external memory controller with up to 32-bit data bus: SRAM,PSRAM,SDRAM, Compact Flash/NOR/NAND memories
- LCD parallel interface, 8080/6800 modes
- LCD-TFT controller up to SVGA resolution with dedicated Chrom-ART Accelerator™ for enhanced graphic content creation (DMA2D)
- Clock, reset and supply management
  - 1.8 V to 3.6 V application supply and I/Os
  - POR, PDR, PVD and BOR
  - 4-to-26 MHz crystal oscillator
  - Internal 16 MHz factory-trimmed RC (1% accuracy)
  - 32 kHz oscillator for RTC with calibration
  - Internal 32 kHz RC with calibration
- Low power
  - Sleep, Stop and Standby modes
  - V<sub>BAT</sub> supply for RTC, 20×32 bit backup registers + optional 4 KB backup SRAM
- 3×12-bit, 2.4 MSPS ADC: up to 24 channels and 7.2 MSPS in triple interleaved mode
- 2×12-bit D/A converters
- General-purpose DMA: 16-stream DMA controller with FIFOs and burst support
- Up to 17 timers: up to twelve 16-bit and two 32-bit timers up to 180 MHz, each with up to 4 IC/OC/PWM or pulse counter and quadrature (incremental) encoder input
- Debug mode
  - SWD & JTAG interfaces
  - Cortex-M4 Embedded Trace Macrocell™



- Up to 168 I/O ports with interrupt capability
  - Up to 164 fast I/Os up to 90 MHz
  - Up to 166 5 V-tolerant I/Os
- Up to 21 communication interfaces
  - Up to 3 × I<sup>2</sup>C interfaces (SMBus/PMBus)
  - Up to 4 USARTs/4 UARTs (11.25 Mbit/s, ISO7816 interface, LIN, IrDA, modem control)
  - Up to 6 SPIs (45 Mbits/s), 2 with muxed full-duplex I<sup>2</sup>S for audio class accuracy via internal audio PLL or external clock
  - 1 x SAI (serial audio interface)
  - 2 × CAN (2.0B Active) and SDIO interface
- Advanced connectivity
  - USB 2.0 full-speed device/host/OTG controller with on-chip PHY
  - USB 2.0 high-speed/full-speed device/host/OTG controller with dedicated DMA, on-chip full-speed PHY and ULPI
  - 10/100 Ethernet MAC with dedicated DMA: supports IEEE 1588v2 hardware, MII/RMII
- 8- to 14-bit parallel camera interface up to 54 Mbytes/s
- Cryptographic acceleration: hardware acceleration for AES 128, 192, 256, Triple DES, HASH (MD5, SHA-1, SHA-2), and HMAC
- True random number generator
- CRC calculation unit
- 96-bit unique ID
- RTC: subsecond accuracy, hardware calendar

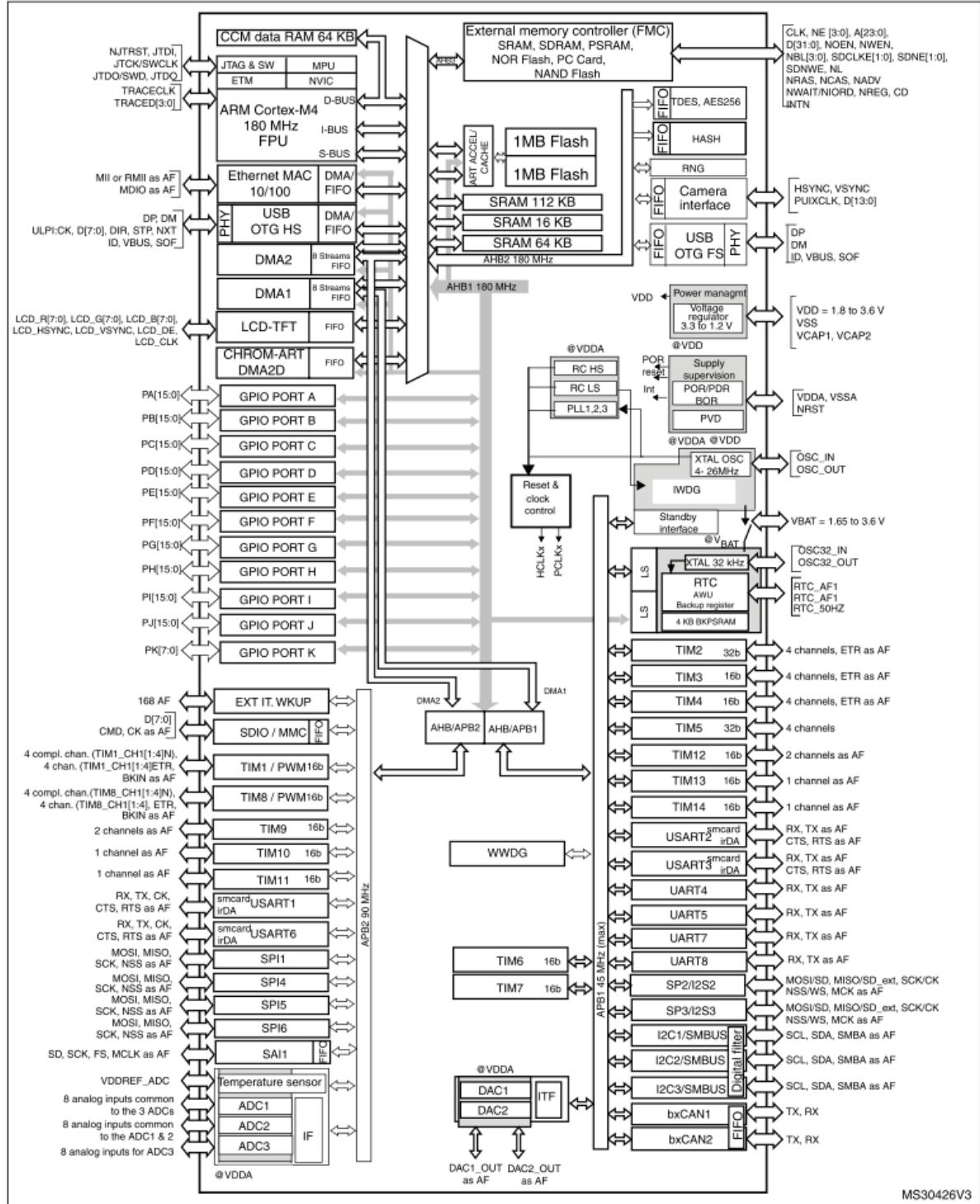
Table 1. Device summary

| Reference   | Part number  |
|-------------|--|
| STM32F437xx | STM32F437VG, STM32F437ZG, STM32F437IG, STM32F437VI, STM32F437ZI, STM32F437II   |
| STM32F439xx | STM32F439VI, STM32F439VG, STM32F439ZG, STM32F439ZI, STM32F439IG, STM32F439II, STM32F439BG, STM32F439BI, STM32F439NI, STM32F439NG |

## STM32F437xx and STM32F439xx

## Description

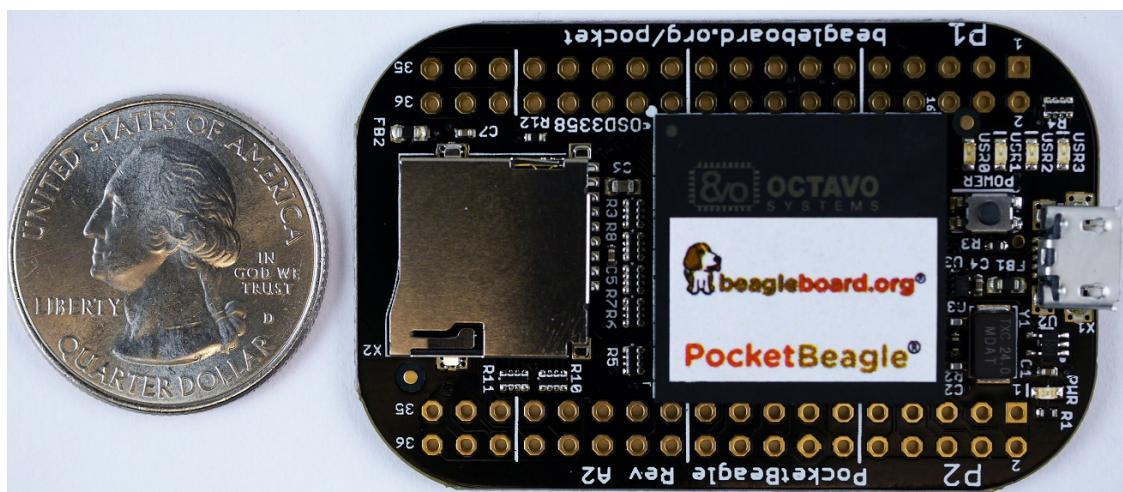
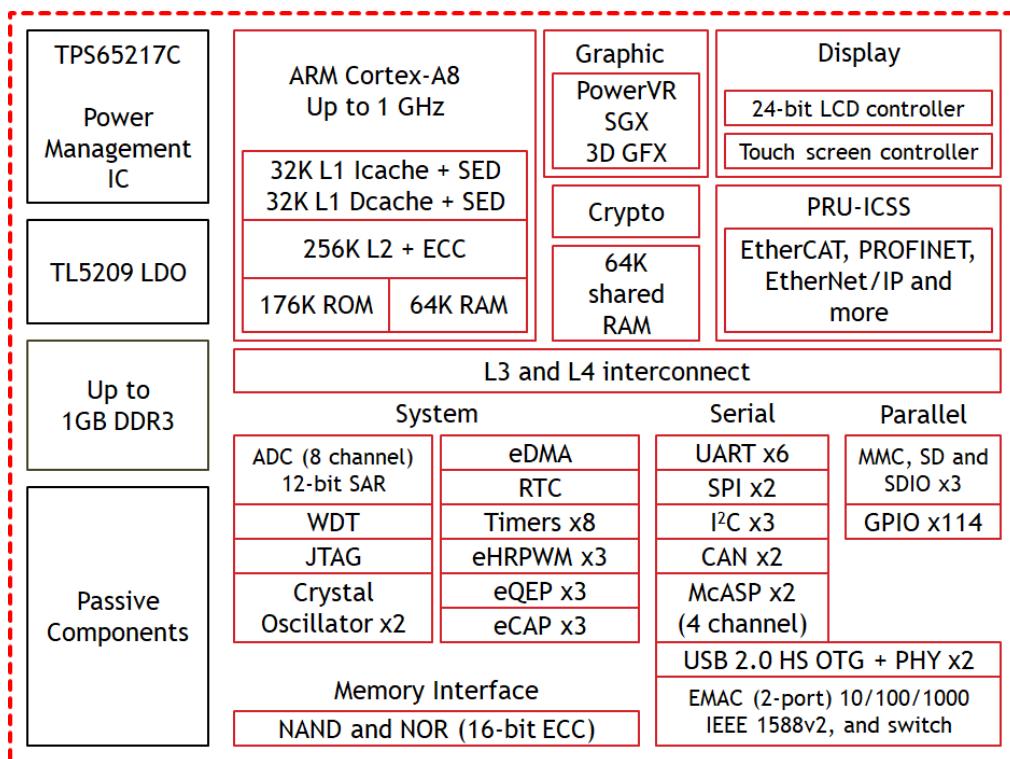
Figure 4. STM32F437xx and STM32F439xx block diagram



1. The timers connected to APB2 are clocked from TIMxCLK up to 180 MHz, while the timers connected to APB1 are clocked from TIMxCLK either up to 90 MHz or 180 MHz depending on TIMPRE bit configuration in the RCC\_DCKCFGR register.
2. The LCD-TFT is available only on STM32F439xx devices.

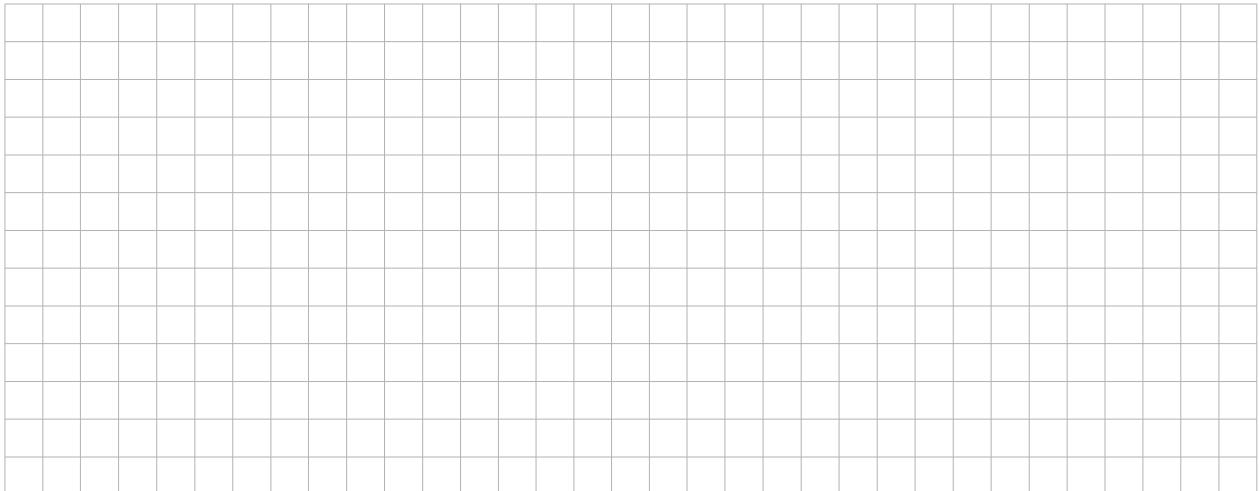
- Octavo OSD335x System-in-Package

Quelle: [https://octavosystems.com/octavo\\_products/osd335x/](https://octavosystems.com/octavo_products/osd335x/)



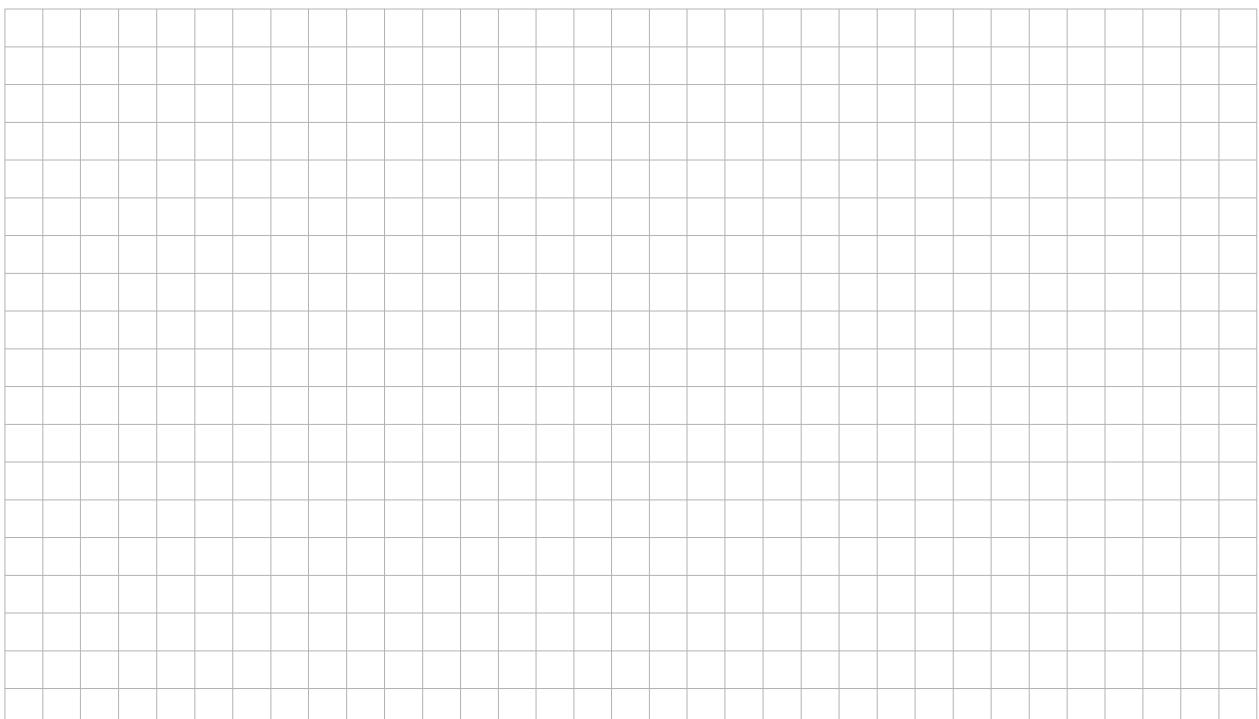
## 2 Interner Aufbau eines Mikroprozessors

Egal ob ein Mikroprozessor aus den 1970er-Jahren wie der MOS 6502 oder ein moderner ARM-Kern – alle basieren auf denselben Grundbausteinen: einem Rechenwerk (ALU), einem Steuerwerk, einem Registersatz und einem Systembus. In diesem Kapitel sehen wir uns diese Bausteine im Detail an und verstehen, wie sie zusammenspielen, um Maschinenbefehle auszuführen.



### 2.1 Rechenwerk

.. die ALU (Arithmetic Logic Unit) ist das „Rechenzentrum“ des Prozessors:



Die ALU führt

- **arithmetische Operationen** (Addition, Subtraktion, Vergleich)
- **logische Operationen** (UND-, ODER-, XOR-Verknüpfungen)
- **Schiebeoperationen**

aus.

Nicht jeder µP/µC besitzt eine ALU die multiplizieren kann (dividieren sogar seltener!). Fehlt diese, müssen solche Berechnungen in Software erfolgen. Weitere mathematische Operationen (Logarithmieren, Winkelberechnungen, ...) können meist nur von mathematischen Co-Prozessoren nativ ausgeführt werden; ansonsten sind diese Operationen ebenfalls in Software zu implementieren.

Die ALU kann nur Ganzzahloperationen oder Festkommaoperationen durchführen. Besitzt das Rechenwerk eine **FPU** (*Floating Point Unit*), so können Gleitkommaberechnungen in Hardware durchgeführt werden. Ist keine FPU vorhanden, so sind Gleitkommaberechnungen in Software nachzubilden.

Im Prinzip ist die ALU eine schnelle Schaltmatrix, deren Funktion je nach gerade auszuführendem Befehl durch Steuersignale vom Steuerwerk vorgegeben wird.

 **Merke:** Die maximale Bitbreite der ALU-Operanden legt die Einstufung des µP als 4-, 8-, 16-, 32- oder 64bit Prozessor fest.

Die ALU besitzt mindestens ein Hilfsregister. Von besonderer Bedeutung ist dabei das **Akkumulationsregister** (engl. accumulator), da hier das Ergebnis der ALU-Operation abgelegt werden kann ("Zwischenergebnis").

Verschiedene Flags werden zum sogenannten **Statusregister** zusammengefasst. Ein Flag ist eine binäre Information, die zusätzliche Ergebnisse einer ausgeführten ALU-Operation kennzeichnen. Solche Statusflags sind z.B.

**Zero Flag (ZF):** Zeigt an, ob das Ergebnis 0 ist.

**Sign Flag (SF):** Ergebnis ist negativ.

**Overflow Flag (OF):** Zeigt eine Bereichsüberschreitung im Zweierkomplement an.

**Carry Flag (CF):** Übertrag (carry) aus dem höchstwertigsten Bit

#### **KI-Aufgabe: Division ohne Divisionseinheit**

Nicht jeder Mikroprozessor besitzt eine ALU, die direkt dividieren kann. Befrage eine KI, wie eine Division durch eine reine Software-Routine umgesetzt werden kann, wenn nur die Operationen **Addition**, **Subtraktion** und **Bitverschiebungen** verfügbar sind.

Skizziere einen möglichen Algorithmus in **Pseudocode**.

Diskutiere, welche Unterschiede im Aufwand (Rechenzeit, Speicherbedarf) zwischen Hardware-Division und Software-Division bestehen.

Gegeben sei ein fiktiver **8-Bit**-Prozessor – „EASy-8“ (*Educational Architecture Simple 8-bit*). Es gilt für die Befehlscodierung:

- 1 Byte Befehl (**Opcode**) (8 Bit)
- **Operand** = optional 1 Byte (Adresse oder Konstante)
- Adressen und Konstanten sind **8-Bit-Werte** (00–FFh)
- Alle Werte in **Hexadezimal** notiert (Suffix „.h“ weglassen, da klar ist)

#### **Register & Flags:**

- A: Akkumulator
- ZF (Zero Flag): 1, wenn Ergebnis der letzten Operation 0 war
- Speichergröße: 256 Byte (00–FF)

Dieser fiktive Prozessor verfügt über folgenden **Befehlssatz** (engl. Instruction Set):

| Opcode | Mnemonic | Beschreibung  | Zyklen |
|--------|----------|---|--------|
| 00     | NOP      | Keine Operation   | 2      |
| 10 nn  | LDA #nn  | Lade Akku mit Konstante nn; d.h. $A \leftarrow nn$  | 3      |
| 11 aa  | LDA aa   | Lade Akku aus Speicheradresse aa; d.h. $A \leftarrow M[aa]$   | 4      |
| 20 aa  | STA aa   | Speicher Akku in Adresse aa; d.h. $M[aa] \leftarrow A$  | 4      |
| 30 nn  | ADD #nn  | Addiere Konstante nn zum Akku; d.h. $A \leftarrow A + nn$   | 3      |
| 31 aa  | ADD aa   | Addiere Speicherwert aa zum Akku; d.h. $A \leftarrow A + M[aa]$   | 4      |
| 40     | NOT A    | Bitweise Negation des Akkus; d.h. $A \leftarrow \neg A$   | 2      |
| 50     | SHL A    | Verschiebe den Akkumulator um 1 Bit nach links; d.h. $A \leftarrow A \ll 1, CF \leftarrow MSB(A)$                 | 2      |
| 80 aa  | JMP aa   | Springe zu Adresse aa; d.h. relativer Sprung zu aa  | 3      |
| 8E aa  | JSR aa   | Sprung zu Unterprogramm an Adresse aa; d.h. $PC \leftarrow aa$ , Rücksprungadresse → Stack                        | 5      |
| 8F     | RTS      | Rückkehr aus Unterprogramm; d.h. $PC \leftarrow M[BASE+SP], SP \leftarrow SP+1$                                   | 5      |
| 90 dd  | BEQ dd   | Springe zu Adresse (Adresse nach BEQ) + dd , wenn ZF = 1<br>dd = 8-Bit Offset im Zweierkomplement (-128 ... +127) | 3 / 4  |
| A0     | PUSH A   | $M[BASE+SP] \leftarrow A; SP \leftarrow SP - 1$   | 3      |
| A1     | PUSH R1  | $M[BASE+SP] \leftarrow R1; SP \leftarrow SP - 1$  | 3      |
| B0     | POP A    | $SP \leftarrow SP + 1; A \leftarrow M[BASE+SP]$   | 3      |
| B1     | POP R1   | $SP \leftarrow SP + 1; R1 \leftarrow M[BASE+SP]$  | 3      |
| FF     | HLT      | Stoppt den Prozessor  | -      |

Im **Programmspeicher** steht folgender Inhalt (unser fiktiver Prozessor beginnt bei Adresse 0x00 mit der Programmabarbeitung):

| Adresse | Code |
|---------|------|
| 00      | 10   |
| 01      | 05   |
| 02      | 40   |
| 03      | 30   |
| 04      | 01   |
| 05      | 20   |
| 06      | 80   |
| 07      | FF   |

**Aufgabe:** Dekodierte die Befehle mithilfe der Befehlstabelle (disassemblieren); verwende dafür die nachfolgende Tabelle; schreibe für jeden Befehl die Mnemonik und kommentiere kurz die Wirkung. Beschreibe in eigenen Worten und im Umfang von 1-2 Sätzen die Funktionalität.

| Adresse | Code  | Mnemonik | Kommentar  |
|---------|-------|----------|--|
| 00      | 10 05 | LDA #05  | A $\leftarrow$ 0x05; d.h. Akku = 0x05 (Startwert)      |
| 02      | 40    | Not A    | = 0000 0101 $\rightarrow$ 1111 1010 $\rightarrow$ 0xFA |
| 03      | 30 01 | ADD # 01 | A $\leftarrow$ 0xFA + 01 = 0xFB                        |
| 05      | 20 80 | STA 80   | A(0xFB) nach 80 speichern                              |
| 07      | FF    | HLT      | stop   |

### Was macht das Programm?

|  |
|--|
| gibt 0x05 in den Startwert, negiert diesen wert $\rightarrow$ 0xFA, zählt 1 dazu (+1) = (0xFB) $\rightarrow$ speichert diesen Wert nach Speicherstelle 80 $\rightarrow$ stop |
|--|

**Aufgabe:** Im Programmspeicher steht folgender Inhalt (unser fiktiver Prozessor beginnt bei Adresse 0x00 mit der Programmausführung):

- 10 10 01  $\rightarrow$  speichert 0x01 in A
- 01 20 80  $\rightarrow$  speichert A (0x01) in adresse 80
- 20 00  $\rightarrow$  nichts verändert sich
- 80 50 verschiebt A um 1 bit nach links  $\rightarrow$  A = 0x02
- 00 80 02  $\rightarrow$  springt zur Adresse 02
- 50 20 80  $\rightarrow$  speichert A(0x02) in adresse 80
- 80 ... Schleife
- 02

Disassembliere den Programmcode mithilfe der Befehlstabelle. Schreibe für jede Instruktion die Mnemonik auf und kommentiere kurz ihre Wirkung. Beschreibe im Umfang von 1 – 2 Sätzen anschließend was das Programm macht.

**Aufgabe:** Gegeben ist folgendes EASy-8 Assemblerprogramm:

```

loop:    LDA #0F 10 0F ; Lade Startmuster in den Akku      A  $\rightarrow$  0x0F
          STA 80 20 80 ; Muster an LED-Port ausgeben      Adresse 0x80  $\rightarrow$  0x0F
          NOT A 40      ; Akku invertieren                 A  $\rightarrow$  0000 1111  $\rightarrow$  1111 0000  $\rightarrow$  0xF0
          STA 80 20 80 ; invertiertes Muster an LED-Port A zu Adresse 0x80  $\rightarrow$  0xF0

```

JMP loop 80 02 ; Endlosschleife springt zu Adresse 0x00

Ergänze jede Assemblerzeile um einen Kommentar – was macht dieses Programm?

Übersetze jede Instruktion in die entsprechenden Maschinenbefehle (Hexwerte). Schreibe die resultierenden Codes in der Reihenfolge auf, wie sie im Programmspeicher stehen (beginnend ab Adresse 0x00).

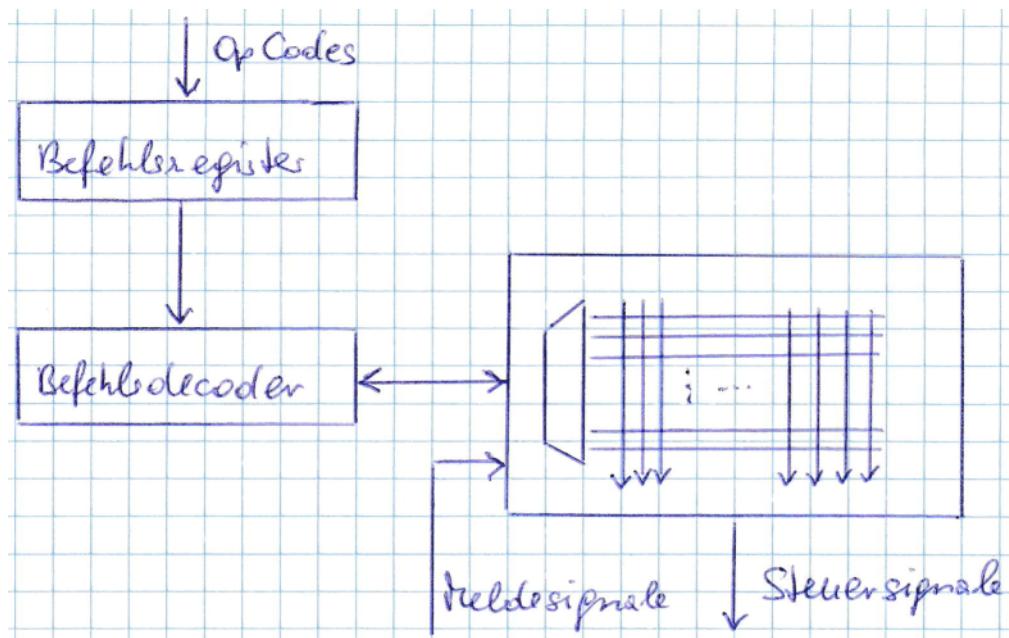
## **2.2 Steuerwerk**

... engl. *Control Unit* und steuert alle prozessorinternen Abläufe:

- Datentransport zwischen **Registern**, ALU und Speicher
  - Steuerung von **Lese-/Schreibzugriffen** auf Speicher oder I/O
  - Behandlung von **Interrupts** und Resets
  - Verwaltung von **Power-Modi** (z. B. Sleep, Low Power)

Außerdem stellt es **Adressinformationen** bereit und koordiniert beim Zugriff auf Speicher oder Ein-/Ausgabekomponenten die **externen Abläufe** (Bus-Steuerung).

Prinzipiell besteht ein einfaches Steuerwerk aus dem Befehlsregister, den Befehlsdecodierer sowie ggf. dem Mikrocodespeicher:



Ein Programm für einen µP besteht aus einer Folge von Maschinenbefehlen. Diese werden gemäß der Abarbeitungsreihenfolge aus dem Programmspeicher in den Prozessor geladen. Dabei wird jener Teil des Maschinenbefehls in das **Befehlsregister** (*Instruction Register, IR*) geladen, der die gewünschte Operation und die dazu benötigten Prozessorkomponenten spezifiziert. Dieser Befehlsteil wird **Operationscode** (engl. *Opcode*) genannt.

### Prinzipieller Aufbau eines Maschinenbefehls:

Der Opcode wird vom **Befehlsdecodierer** (*Instruction Decoder*) entschlüsselt. Dabei wird auch festgestellt, ob und wieviele Operanden der Befehl benötigt, wo diese sich befinden und welche Register benötigt werden. Im Mikrocodespeicher sind die, für jeden Befehl nötigen, intern abzuarbeitenden Bitmuster abgespeichert – wobei ein Befehl intern auch aus mehreren Mikrobefehlen aufgebaut sein kann.

Das Steuerwerk behandelt auch Unterbrechungsanforderungen (z.B. Interrupts), Reset, Zuteilung auf den Systembus (wenn andere Komponenten auf den Systembus zugreifen wollen – z.B. CoProzessor oder DMA-Controller), Fehlermeldungen (z.B. bus error) oder spezielle PowerModi ("Schlafmodus", Low Power Modi, ...).

Bei **CISC-Prozessoren** (engl. *Complex Instruction Set Computer*) wird jeder Befehl durch eine Sequenz von Mikrobefehlen realisiert. Einfache Befehle benötigen weniger Mikrobefehle, komplexere Anweisungen erfordern eine längere abzuarbeitende Befehlskette. Bei **RISC-Prozessoren** (engl. *Reduced Instruction Set Computer*) wird prinzipiell auf eine Mikroprogrammsteuerung verzichtet. Jede Anweisung wird in Hardware realisiert – es gibt keine komplexen Anweisungen mehr; der Großteil der Befehle braucht nur einen CPU Taktzyklus für die Umsetzung.

RISC/CISC-Architekturen haben unterschiedliche Vor-/Nachteile. **CISC Prozessoren** verfügen über einen **umfangreichen Befehlssatz**, **viele Adressierungsarten** und benötigen nur **wenige interne Register**. Für die Umsetzung eines Algorithmus in den Maschinencode ist weniger Programmcode nötig (im Vergleich zu RISC). Die Ausführung der Programmabarbeitung benötigt jedoch **mehrere CPU-Taktzyklen** (aufgrund der Mikrobefehlssequenzen). Da bei **RISC** auf die Mikroprogrammsteuerung verzichtet wird, wird **weniger Chipfläche** benötigt => **kostengünstiger** zu produzieren. Da es weniger Anweisungen/Befehle gibt, müssen komplexere Abläufe in mehreren Schritten selbst ausprogrammiert werden, was einen **längeren Programmcode** zur Folge hat (Vergleich zu CISC). Da hierbei öfters Ergebnisse zwischenzuspeichern sind, sind **mehr interne Register** vorhanden.

Gegenüberstellung CISC ↔ RISC:

|   | CISC         | RISC  |
|---|--------------|-------|
| Komplexe Befehle                          | ✓            | ✗     |
| ALU-Operationen mit Operanden im Speicher | ✓            | ✗     |
| Anzahl der Anweisungen                    | >200         | <150  |
| Anzahl CPU-Takte je Anweisung             | typ 2 bis 10 | typ 1 |
| Registeranzahl                            | <20          | >=32  |

## 2.3 Adresswerk

Das Adresswerk berechnet –nach Vorgaben des **Steuerwerks**– die **effektive Adresse** (*effective address, EA*) eines **Befehls** oder **Operanden**. In der minimalen Ausprägung besteht es lediglich aus einem **Adressaddierer** (engl. *address adder*), der Adressen inkrementiert oder mit Offsets verrechnet.

Typische Grundfälle sind:

- **Sequentieller Ablauf:** Die nächste Adresse wird **um 1 (oder Befehlslänge)** erhöht.
- **Sprung/Zweig:** Es wird eine **neue Adresse** vorgegeben (absolute Adresse) **oder** aus einer

aktuellen Adresse plus **Offset** gebildet (relative Adresse).

- **Direkter Zugriff auf Daten:** Eine **feste Adresse** wird verwendet (z.B.  $0x80$  für Speicher oder *memory-mapped I/O*).



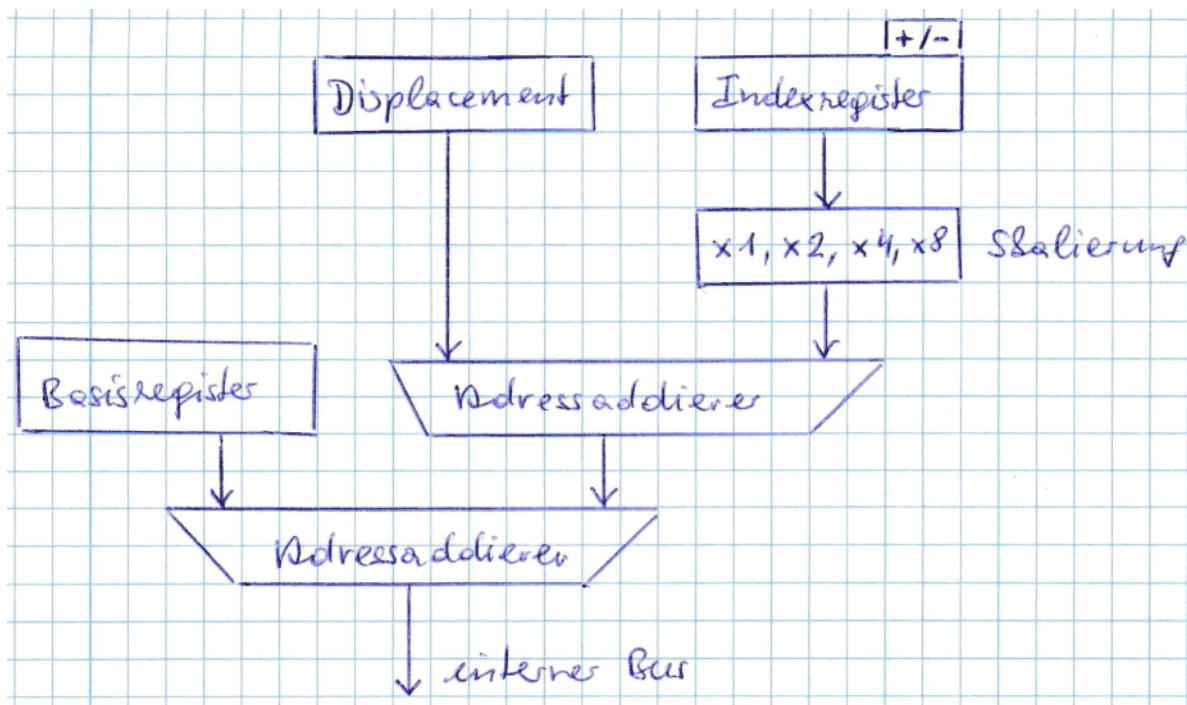
**Merke:** Das **Adresswerk** beantwortet stets die Frage: „Wo liegt der nächste Befehl/Operand?“

**Adressierungsarten** legen fest, wie aus **Befehl** und **Registerinhalten** die **effektive Adresse (EA)** entsteht. Im nachfolgenden wird ein kleiner Einblick in sehr einfache Adressierungsarten gegeben (es gibt noch weitere, auch komplexere Adressierungsarten):

- **Immediate (unmittelbar):** Der Operand selbst ist eine Konstante. Er steht unmittelbar hinter dem Opcode im Programmspeicher.  
*Beispiel:* LDA #03 → „03“ ist der Wert; die EA ist die auf dem Opcode folgende.
- **Direkt / Absolut:** Der Operand des Befehls enthält die Adresse der Speicherzelle, auf die zugegriffen wird.  
*Beispiel:* LDA 80 → Akku erhält den Inhalt von Speicherzelle  $0x80$ .  
*EA-Formel:* EA = addr.
- **Relativ (für Sprünge):** Neue Adresse = **aktuelle Befehlsadresse + Offset**.  
*Beispiel:* „Springe +10 Anweisungen (vor)“ oder „-6 Anweisungen (zurück)“.  
*EA-Formel:* EA = aktuelle\_Adresse + Offset (Offset oft vorzeichenbehaftet).
- **Indiziert / Indirekt:** Adresse ergibt sich aus **Basis + Index** bzw. aus einem **Zeiger** im Speicher.
- **Implizit:** Keine explizite Adressangabe; die Operanden sind durch den Befehl festgelegt

Das Adresswerk verfügt meist über zusätzliche interne Register, wie das

- **Basisregister (base register)** – zeigt auf den **Beginn** einer Datenstruktur (z. B. Tabelle)
- **Indexregister (index register)** – enthält den **Laufindex** (z. B. Element-Nr.)
- **Displacement/Offset** – kleiner **Feldversatz** (z. B. um ein Struktur-Feld zu adressieren)



sodass der Adressaddierer die **effektive Adresse**

$$EA = Basis + Index \times Skalierung + Displacement$$

berechnet (wobei die Skalierung die Elementgröße in Byte ist). Dies ist vor allem bei der Arbeit mit Arrays / Tabellen sehr vorteilhaft.

#### Aufgabe: Adressierungsarten

Ordne den folgenden EASy-8 Befehlen die jeweilige Adressierungsart zu:

ADD 50; NOT A; BEQ 02; RTS; JMP 90

Ergänze in einem kurzen Satz, warum du dich jeweils dafür entschieden hast.

## 2.4 Registersatz

 **Merke:** Register sind extrem **schnell** ansprechbare, **interne Speicherplätze**. Sie dienen der CPU zur temporären Ablage von Operanden und Zwischenergebnissen und ermöglichen dadurch einen wesentlich schnelleren Zugriff als auf den Hauptspeicher.

Je nach Anwendungsfall unterscheidet man zwischen...

- **Datenregister:** dienen zur Zwischenspeicherung von Operanden
- **Adressregister:** enthalten die Adresse oder Teile davon, wobei hier zwischen Basis- und Indexregister unterschieden wird. Im **Indexregister** (engl. *Pointer*) wird meist eine Adressdistanz (engl. *offset, displacement*) abgelegt.
- **Spezialregister:** diese können durch Maschinenbefehle direkt adressiert werden. Dazu zählen z.B.
  - das **Statusregister** (Flags)
  - der **Programmzähler** (engl. *Program Counter, PC*): enthält die Speicheradresse des als nächsten einzulesenden Datums
  - **Stackregister** (mit *Stackpointer, SP*)

Der **Stack** („**Kellerspeicher**“) arbeitet nach dem **LIFO** (*Last In – First Out*) Prinzip und wird für die Speicherung von **Rücksprungadressen**, **Zwischenwerte**, **Unterprogramme** und **Interrupts** verwendet. Bei vielen CPU-Architekturen **wächst der Stack zu kleineren Adressen** ( $SP \downarrow$ ), das ist aber architekturabhängig.

#### i Info: Der Stack in Informatik und IT-Security

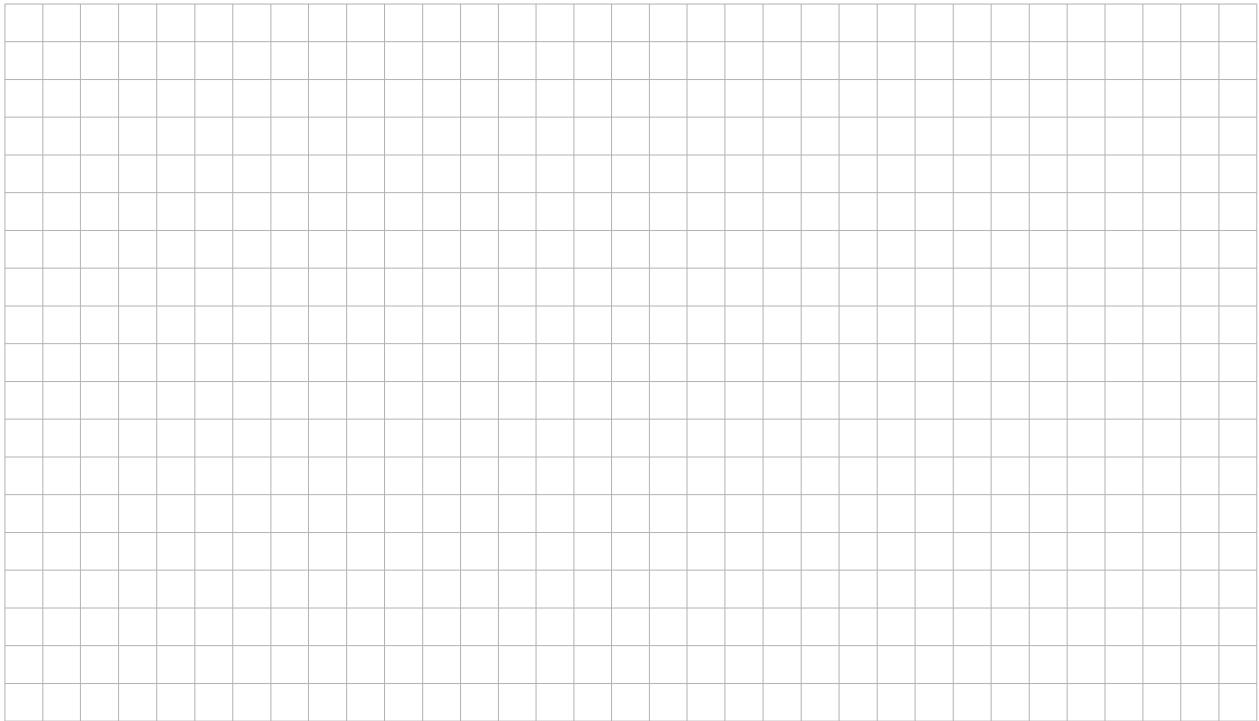
Der Stack ist ein **fundamentales Konzept** der Informatik und wurde schon in frühen CPU-Architekturen für **Unterprogrammaufrufe** und **Rücksprungadressen** eingesetzt; später kam die Nutzung für Interrupt-Verarbeitung hinzu. Die Instruktionen **PUSH** und **POP** gehören zu den grundlegendsten Befehlen in der Assembler-Programmierung und finden sich in praktisch allen Architekturen – von klassischen 8-Bit-Prozessoren bis zu modernen x86-64 Systemen.

Auch in der **IT-Security** spielt der Stack eine wichtige Rolle: **Stack Overflow** ist nicht nur ein Entwicklerforum, sondern auch der Name einer bekannten **Angriffsmethode**. Eine besonders gefährliche Technik ist **Return-Oriented Programming (ROP)**, bei der Angreifer gezielt die Rücksprungadressen im Stack manipulieren (→ *Details sind nicht prüfungsrelevant*).

Der Stack arbeitet nach dem **LIFO-Prinzip** (Last In – First Out).

- PUSH legt einen Wert oben auf den Stack.
- POP holt den zuletzt abgelegten Wert wieder zurück.  
Viele CPU-Architekturen verwenden den Stack für **Funktionsaufrufe, Rücksprungadressen und Interrupts**.

Funktionsweise:



Unsere fiktive EASy-8 CPU besitzt die Stack-Instruktionen

- PUSH: legt den Inhalt eines Registers auf dem Stack ab ( $M[SP] \leftarrow Register; SP \leftarrow SP - 1$ )
- POP: holt den obersten Stack-Wert ins Register ( $SP \leftarrow SP + 1; Register \leftarrow M[SP]$ )

**Aufgabe:** Es sollen nun die Inhalte von Akku A und Register R1 getauscht werden, ohne ein drittes Register zu benötigen.

Dies kann mit Hilfe des Stack sehr elegant gelöst werden - **Pseudo-Assemblercode (EASy-8):**

```
; Vorher: A = 0x12, R1 = 0x34, SP = 0xFF
PUSH A
PUSH R1
POP A
POP R1
; Nachher: A = 0x34, R1 = 0x12 → erfolgreich getauscht
```

| Befehl  | A  | R1 | SP | Speicher[SP] |
|---------|----|----|----|--------------|
| Initial | 12 | 34 | FF | -            |
| PUSH A  |    |    |    |              |
| PUSH R1 |    |    |    |              |

|        |  |  |  |  |
|--------|--|--|--|--|
| POP A  |  |  |  |  |
| POP R1 |  |  |  |  |

Der **zuletzt** gepushte Wert (von R1 bzw. 0x34) wird **zuerst** wieder geholt → landet in A. Danach kommt der zuvor gepushte Wert (von A bzw. 0x12) zurück → landet in R1.

**Merk:** Die Anweisung **POP** löscht den Wert am Stack nicht, sondern verschiebt lediglich den Stackpointer. Der alte Wert bleibt im Speicher erhalten, wird jedoch beim nächsten PUSH überschrieben.

PUSH/POP ändern in der Regel **keine ALU-Flags** (ZF/CF/...). Das hilft, den Unterschied zwischen **Datenbewegung** und **Berechnung** klar zu halten.

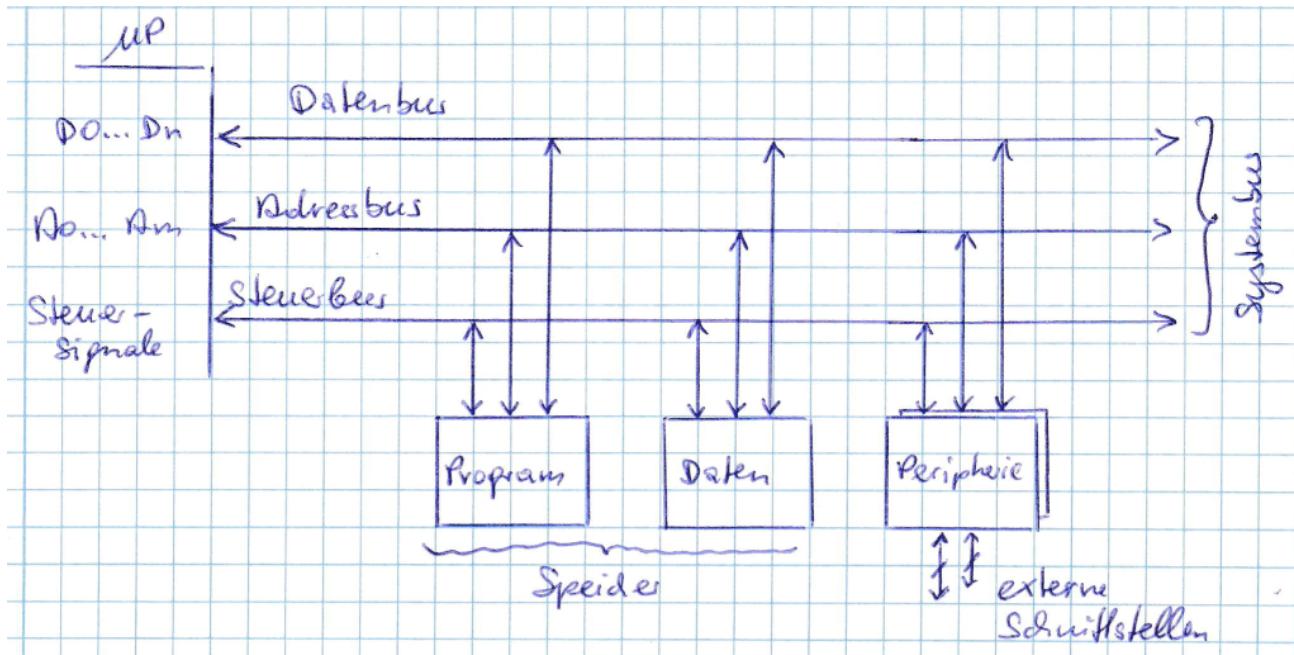
Anmerkung: Es ist architektur-spezifisch, ob zuerst der Wert auf dem Stack abgelegt wird und dann die SP-Adresse dekrementiert wird (Post-Decrement PUSH) ODER zuerst die SP-Adresse dekrementiert und dann der Wert abgespeichert wird (Pre-Decrement PUSH). Entsprechend verhält sich dann POP (pre-inc vs. post-inc).

## 2.5 Systembus

... verbindet die Komponenten des Mikrocomputers, stellt also ein gemeinsames Bussystem dar. Über diesen Systembus werden Informationen ausgetauscht.

Der Systembus besteht aus

- **Adressbus**
- **Datenbus**
- **Steuerbus**



Bei vielen μP-Systemen ist der **Adressbus (A0 ... Am)** unidirektional (μP gibt die Adresse vor),

der **Datenbus (D0 ... Dn)** und **Steuerbus bidirektional** ausgeführt. Signale des Steuerbus sind z.B.

**R/W**: log.1 bedeutet Leseoperation, log. 0 bedeutet Schreiboperation

**READY**: dem µP wird über dieses Signal mitgeteilt, ob Wartezyklen einzulegen sind oder der Systembuszugriff beendet wurde

**MREQ**: Memory Request – es soll auf Speicher und nicht auf Peripheriekomponenten zugegriffen werden

### 3 Befehlsabarbeitung

Prinzipiell werden Befehle in lückenlos aufsteigender Folge dem Speicher entnommen und durch die CPU abgearbeitet. Dieser lineare Ablauf wird durch **Sprünge (jump)**, **(bedingte) Verzweigungen (branch)**, **Unterprogramm-Aufrufe (call)** und **Rückkehrbefehle (return/ret)** sowie durch **Ausnahmen (exceptions**, z. B. Division durch 0) und **asynchrone Unterbrechungen (Interrupts)** unterbrochen.

Unter dem Begriff **Befehlszyklus** (engl. *instruction cycle*, oft auch *fetch-decode-execute*) versteht man den unendlichen, sich stetig wiederholenden Ablauf von

- Befehlscode **lesen** (engl. *instruction fetch*)
- Befehlscode **decodieren** (engl. *instruction decode*)
- Befehl **ausführen** (engl. *execute*)

Die nötigen Einzelschritte sind dabei (im Prinzip) ...

#### Befehlscode lesen:

1. Programmzähler (PC) wird auf den Adressbus geschaltet.
2. Der Inhalt der damit adressierten Speicherstelle (= Opcode) wird in das Befehlsregister (IR) geladen.
3. PC wird inkrementiert (gelesene(n) Byte(s); ist architekturspezifisch).

#### Befehlscode decodieren:

Der **Befehlsdecoder** interpretiert den **Opcode**.

Welche Operation? Welche **Adressierungsart**? Welche **Register/Einheiten** werden benötigt?

Ergebnis der Dekodierung sind interne **Steuersignale** (bei CISC ggf. via **Mikrocode**, bei RISC meist **hartverdrahtet**).

#### Befehl ausführen:

1. Werden weitere Operanden benötigt?

Solange dies der Fall ist:

- a. PC auf Adressbus schalten
- b. Der Inhalt der damit adressierten Speicherstelle (= Operand) in das jeweilige Register

übertragen.

- c. PC inkrementieren
- 2. ALU rechnet (arithmetisch/logisch) **oder** es erfolgt ein Speicher-/I/O-Zugriff
- 3. Ergebnis ablegen (Write-Back) - Zielregister/Speicher schreiben.
- 4. Flags aktualisieren: z. B. ZF, CF, SF, OF.
- 5. PC anpassen (falls Steuerfluss-Befehl): Bei Sprung/Call/Ret/... erhält der PC eine neue Adresse (absolut oder relativ).

Generell gilt: Das Steuerwerk sorgt dafür, **wer wann welche Daten wohin** bewegt – der PC gibt die **Position** vor, die ALU erzeugt **Ergebnisse**, Flags beschreiben deren **Eigenschaften**.

Moderne CPUs nutzen Optimierungen (z. B. **Prefetch**, **Pipelines**, **Caches**).

### KI-Aufgabe: CPU-Pipelining

Befrage eine KI, was unter **Pipelining** bei Prozessoren verstanden wird. Notiere ein kurzes Beispiel (z. B. mit den Phasen *Fetch*, *Decode*, *Execute*) und vergleiche es mit einer Fabrik- oder Fließband-Analogie. Überlege, welche Probleme beim Pipelining auftreten können.

Erkläre deinem Banknachbarn, wie du das Pipelining-Prinzip verstanden hast, und hört euch gegenseitig eure Erklärungen an.

### Merke: Befehlszyklus

Jede CPU arbeitet in einem fortlaufenden Zyklus aus **fetch – decode – execute**:

- **Fetch:** Der nächste Befehl wird anhand des Programmzählers (PC) aus dem Speicher geholt.
- **Decode:** Das Steuerwerk interpretiert den Opcode und steuert die beteiligten Einheiten an.
- **Execute:** Der Befehl wird ausgeführt.

Dieser Zyklus läuft ununterbrochen ab – solange die CPU nicht angehalten oder die Spannungsversorgung abgeschaltet wird.

## 4 Speichermodell

Ein Mikroprozessor arbeitet mit **Adressen** und **Daten**. Das **Speichermodell** beschreibt, wo Programme und Daten liegen, wie darauf zugegriffen wird und welche Bereiche für Peripherie reserviert sind. Dabei unterscheidet man zwischen folgenden **Speicherarten**:

- **Programmspeicher** (*Program Memory*, z. B. ROM/Flash): enthält Maschinenbefehle.
- **Datenspeicher** (*Data Memory*, z. B. RAM): hält veränderliche Daten (Variablen), Stack, Zwischenergebnisse.
- **Nichtflüchtiger Datenspeicher** (z. B. EEPROM/Flash-NVM): für Parameter/Kalibrierwerte

Der **Adressraum** ist die Menge aller gültigen Adressen (z. B. 0x00...0xFF bei unserem EASy-8-Demoprozessor, oder eben 0x00000000...0xFFFFFFFF als 32-Bit-Adresse bei ARM).

Eine **Memory Map** teilt den Adressraum in Bereiche auf, z. B. in ...

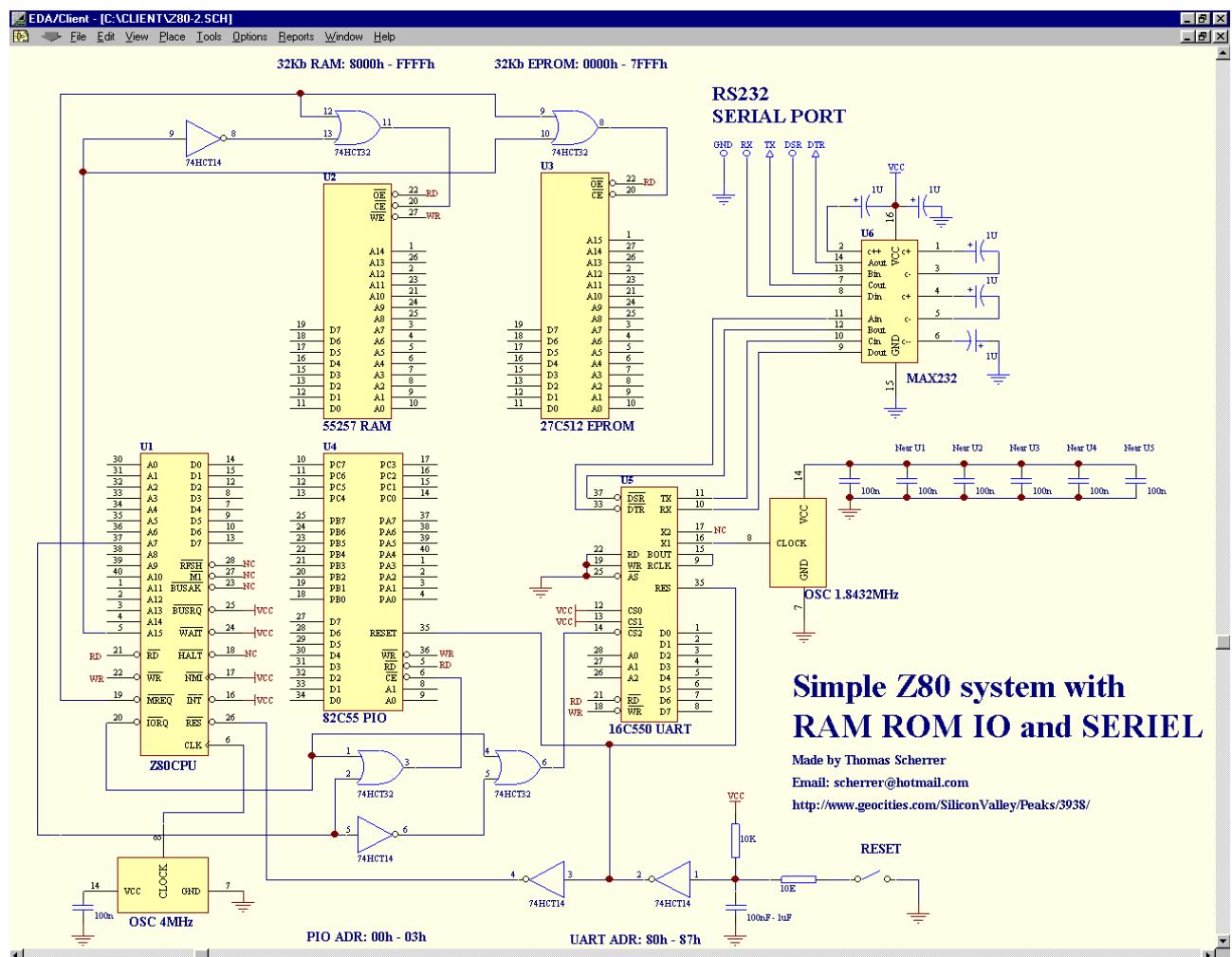
- **Code-Bereich** (nur lesbar/ausführbar)
- **RAM-Bereich** (lesbar/schreibbar)
- **Speicher-gemappte Peripherie** (*memory-mapped I/O*, z. B. LED-Port bei 0x80)
- **Systemregister ("Fuse Bits") / Boot-ROM**

Je nach Prozessorentwurf kann es **getrennte Adressräume für Speicher und I/O-Komponenten** geben oder **Speicher und I/O-Komponenten teilen sich einen gesamten Adressraum ("Shared Memory")**.

#### Getrennte Adressräume für Speicher und I/O-Komponenten:

Die Unterscheidung, ob auf Speicher oder auf I/O-Komponenten zugegriffen wird, erfolgt über **dedizierte Signale des Steuerbus** sowie anhand des **Befehlssatzes**. Für den Zugriff auf I/O-Komponenten gibt es z.B. spezielle IN- und OUT-Befehle.

Beispiel: Z80                    MEMREQ, IOREQ



Quelle <http://www.z80.info/>

Gemeinsame Adressräume für Speicher und I/O-Komponenten:

Zur **Unterscheidung** werden **verschiedene Adressen** eingesetzt. Unabhängig ob Speicher oder I/O-Komponente – es können alle Operationen ohne Einschränkung angewendet werden.

*Beispiel: PSoC5LP - Auszug aus dem Architecture Technical Reference Manual*

Table 13-1. PSoC 5LP Memory Map

| Address Range           | Size   | Use   |
|-------------------------|--------|---|
| 0x00000000 – 0x1FFFFFFF | 0.5 GB | Program code. Includes the exception vector table at power up, which starts at address 0  |
| 0x20000000 – 0x3FFFFFFF | 0.5 GB | SRAM. This includes a 1 MByte bit-band region starting at 0x20000000, and a 32 Mbyte bit-band alias region starting at 0x22000000.        |
| 0x40000000 – 0x5FFFFFFF | 0.5 GB | Peripherals. This includes a 1 MByte bit-band region starting at 0x40000000, and a 32 Mbyte bit-band alias region starting at 0x42000000. |
| 0x60000000 – 0x9FFFFFFF | 1 GB   | External RAM  |
| 0xA0000000 – 0xDFFFFFFF | 1 GB   | External peripherals  |
| 0xE0000000 – 0xFFFFFFFF | 0.5 GB | Internal peripherals, including the NVIC and debug and trace modules  |

Table 13-2. PSoC 5LP Address Map

| Address Range             | Purpose   |
|---------------------------|---|
| 0x0000 0000 – 0x0003 FFFF | Up to 256 KB Flash  |
| 0x1FFF 8000 – 0x1FFF FFFF | Up to 32 KB SRAM in code region                                     |
| 0x2000 0000 – 0x2000 7FFF | Up to 32 KB SRAM in SRAM region                                     |
| 0x2000 8000 – 0x2000 FFFF | Alias of address range 0x1FFF 8000 – 0x1FFF FFFF, accessible by DMA |
| 0x4000 4000 – 0x4000 42FF | Clocking, PLLs, and oscillators                                     |
| 0x4000 4300 – 0x4000 43FF | Power management  |
| 0x4000 4500 – 0x4000 45FF | Ports interrupt control   |
| 0x4000 4700 – 0x4000 47FF | Flash programming interface   |
| 0x4000 4900 – 0x4000 49FF | I <sup>2</sup> C controller   |
| 0x4000 4E00 – 0x4000 4EFF | Decimator   |
| 0x4000 4F00 – 0x4000 4FFF | Fixed timer/counter/PWMs  |
| 0x4000 5000 – 0x4000 51FF | General purpose I/Os  |
| 0x4000 5300 – 0x4000 530F | Output port select register   |
| 0x4000 5400 – 0x4000 54FF | External memory interface control registers                         |
| 0x4000 5800 – 0x4000 5FFF | Analog subsystem interface  |
| 0x4000 6000 – 0x4000 6OFF | USB controller  |
| 0x4000 6400 – 0x4000 6FFF | UDB configuration   |
| 0x4000 7000 – 0x4000 7FFF | PHUB configuration  |
| 0x4000 8000 – 0x4000 87FF | EEPROM  |
| 0x4000 A000 – 0x4000 A400 | CAN   |
| 0x4000 C000 – 0x4000 C800 | Digital filter block  |
| 0x4001 0000 – 0x4001 FFFF | Digital interconnect configuration                                  |
| 0x4800 0000 – 0x4800 7FFF | Flash ECC bytes   |
| 0x6000 0000 – 0x60FF FFFF | External Memory Interface (EMIF)                                    |
| 0xE000 0000 – 0xE00F FFFF | Cortex-M3 PPB registers, including NVIC, debug, and trace           |

Viele Speicher und/oder I/O-Komponenten sind nach wie vor in Byte (8bit) organisiert. Beim Austausch (Schreiben / Lesen) eines Bytes gibt es kein Problem. Beim Austausch von z.B. 16 Bit über einen 8-Bit Bus sind zu erst

- die oberen 8 Bit ("High Byte") gefolgt von den unteren 8 Bit ("Low Byte") **ODER**
- die unteren 8 Bit ("Low Byte") gefolgt von den oberen 8 Bit ("High Byte")

zu übertragen. Selbige Überlegung gilt auch, wenn ein 16 Bit Datum in einen -als 8 Bit organisierten Speicher- abgelegt werden muss. 16 Bit benötigen 2 Speicherstellen – dabei kann

entweder

- das High Byte auf der 1. Speicherstelle und das Low Byte auf der 2. Speicherstelle
  - das Low Byte auf der 1. Speicherstelle und das High Byte auf der 2. Speicherstelle

abgelegt werden.

Wird zuerst das Low Byte abgelegt, dann spricht man vom sog. **Little-Endian-Format** ("niederwertiges Byte an niedriger Adresse"). Wird zuerst das High Byte abgelegt, dann vom **Big-Endian-Format**. Analog gilt dies für 32-Bit-Werte (4 Byte), 64-Bit-Werte (8 Byte), ...

## Aufgabe: Speicherformat Little-Endian vs. Big-Endian

1. Wie wird der 16-Bit-Wert `0x8013` in einem System abgelegt, das gemäß
    - **Little-Endian-Format** arbeitet?
    - **Big-Endian-Format** arbeitet?
  2. Gegeben sei folgender Speicherinhalt:

| <b>Adresse</b> | <b>Daten</b> |
|----------------|--------------|
| 0x00           | 0x12         |
| 0x01           | 0x34         |
| 0x02           | 0x56         |
| 0x03           | 0x78         |

Welchen 32-Bit-Wert liest eine CPU, die gemäß

- Little-Endian-Format arbeitet?
  - Big-Endian-Format arbeitet?

## 5 Befehlssatz

Der **Befehlssatz** (*Instruction Set*) umfasst alle **Maschinenbefehle**, die eine CPU ausführen kann. In der Praxis gliedert man ihn in:

- **Transportbefehle** (*data transfer*): Datentransport zwischen Registern, Speicherstellen und I/O-Komponenten; z. B. MOV, LDA/STA, PUSH/POP
- **Arithmetische und logische Befehle** (*ALU ops*): z. B. ADD, SUB, CMP, AND/OR/XOR, sowie Schiebe-/Rotierbefehle
- **Bitmanipulationsbefehle** (*bit manipulation*): Setzen, Rücksetzen und negieren einzelner Bits/Felder
- **Programmsteuerbefehle** (*control flow*): Steuerung des Programmflusses (Sprünge und (bedingte) Verzweigungen, Unterprogramme (CALL/RET))
- **Prozessorsteuerbefehle** (*system/control*): beeinflussen die CPU-Arbeitsweise; z. B. Interrupt-Enable/Disable, Traps, Breakpoint/SVC, Power-Modi

Viele Befehle ändern **Statusflags** (Zero, Carry, Sign, Overflow). Diese **Seiteneffekte** steuern oft den weiteren Programmfluss (bedingte Verzweigungen).

 **Merke:** Statusflags (ZF, CF, SF, OF) steuern den Programmfluss → Basis für IF/ELSE.

Beispiele für ...

... Transport:

```
LDA #05      ; A ← 5
STA $80      ; M[$80] ← A   (z. B. LED-Port)
PUSH A       ; A auf Stack sichern
POP A        ; A vom Stack zurückholen
```

... ALU & Flags:

```
ADD #03      ; A ← A + 3   (ZF/CF/SF/OF werden gesetzt)
AND #F0      ; High-Nibble maskieren
CMP #00      ; vergleicht A mit 0, setzt Flags (A unverändert)
```

... Programmfluss:

```
BEQ 02       ; wenn ZF=1 → springe (relativ) um +2 Adressstellen weiter
JMP loop     ; bedingungslos springen
CALL subr    ; Unterprogramm
RET          ; Rückkehr
```

... System/Control:

```
SEI          ; Interrupts sperren
CLI          ; Interrupts erlauben
HLT          ; CPU anhalten
```