

Kapitel 1: Einstieg in die Mikroprozessortechnik

Abstraktionsebenen: Schichtenmodell

Weitere Dimensionen:

SW-Anwendungen

Betriebssystem

SW

HW

Anwendung
Algorithmen
Hochsprachen
Assembler
Maschinensprache
Mikroarchitektur
Digitale Komponenten
Elektronische Bauelemente

↑
Abstraktion

↓
Hardwarenähe

Abstraktionsebenen: Schnittstelle HW/SW

- An der Oberkante “Maschinensprache” ist die sog. “Instruction Set Architecture” (ISA) des Systems definiert (\Rightarrow Kap. 2)
- Sicht der SW auf die Maschine: Maschinenbefehle, Adressierungsarten, ...
- Reale Maschinen können auch als Simulation implementiert werden
 \Rightarrow “Virtuelle Maschine”, z.B. Java VM, VirtualBox, *Simulator*

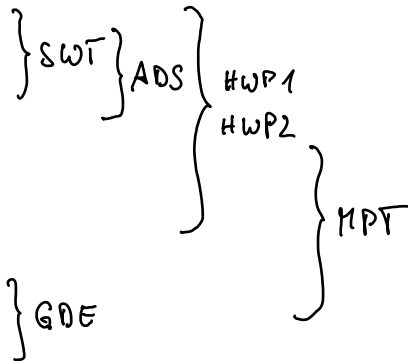
Anwendung
Algorithmen
Hochsprachen
Assembler
Maschinensprache
Mikroarchitektur
Digitale Komponenten
Elektronische Bauelemente

SW

ISA
echte oder virtuelle HW

Abstraktionsebenen: Abdeckung im Curriculum

Anwendung
Algorithmen
Hochsprachen
Assembler
Maschinensprache
Mikroarchitektur
Digitale Komponenten
Elektronische Bauelemente



Anwendung

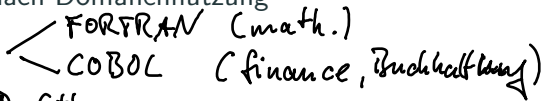
- Softwarelösung für ein Anwenderproblem
- Vereinfacht: synonym für "Programm", monolithische Anwendung
- Oft aber: Sammlung von kooperierenden Programmen+Diensten
↳ Themen von "Betriebssysteme" und "Verteilte Systeme"
- Auch Anwendungen:
Tools im Betriebssystem-Kontext (Dienstprogramme)
- Verwandt: *Bibliotheken* (Libraries) von Anwendungscode

- Abstrahierte Lösungsideen für oft wiederkehrende Teilprobleme, Bsp.:
 - ggT
 - Sieb des Eratosthenes
 - Quicksort
 - Als Teil einer Anwendung umsetzbar
 - Oft in Bibliotheken als vorgefertigter Anwendungscode
 - Gekoppelt mit Datenstrukturen, auf denen die Algorithmen ausgeführt werden
- ⇒ LV ADS
- qsort() in C*

- Abstrahieren von Maschinensprachen, besser geeignet für die Programmierung von Anwendungen und Algorithmen

- Problemorientiert statt maschinenorientiert

- Charakterisierung nach Domänennutzung

- fachspezifisch 
 - FORTRAN (math.)
 - COBOL (finance, Buchhaltung)
- generisch z.B. C++

- Charakterisierung nach Sprachparadigma

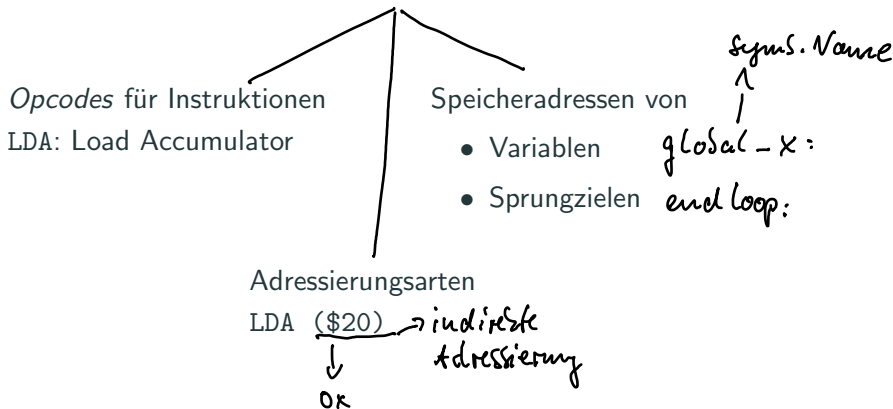
- prozedural/imperativ C, PASCAL
- objektorientiert C++, Java
- funktional LISP, Haskell

- Charakterisierung nach Abstraktionsgrad/Maschinnennähe

Abstrakt.
↑ Java
| C

- Anspruch: Maschineninstruktionen sollen auch für menschliche Programmierer lesbar sein

⇒ Symbolische, textuelle Notation der Befehle



- Instruktionen des Befehlssatzes eines Prozessors
- Konkreter: Speicherrepräsentierung der Instruktionen
 - Umfasst Opcodes (Befehl) und ggf. Operanden
 - Binäre Grundlage für die symbolische Assembler-Notation
- Meist 1:1 auf Assembler abbildbar

- AVR-Asm: LDI r17, 0xbc
 - Binär: 1110 1011 0001 1100
- Handwritten annotations:*
- Handwritten text: "immediate (nur für r16..r31)" with an arrow pointing to the "0xbc" in the AVR-Asm instruction.
 - Handwritten arrows pointing from the four bytes of the binary instruction to the fields of the AVR-Asm instruction: "1110" to "LDI", "1011" to "r17", "0001" to the comma, and "1100" to "0xbc".
 - Handwritten text: "r(16+1)" with an arrow pointing to the "1100" byte.

- Ausnahmen: *Pseudo*-Instruktionen in Assembler:
Nützliche aber nicht auf dem Prozessor verfügbare Instruktionen,
die auf andere reale Maschinenbefehle abgebildet werden
Bsp.: ARM LDR mit 32-Bit-Konstanten

- Wird charakterisiert durch:

- Instruction Set Architecture (ISA), z.B. Intel IA-32
(genauere Definition am Kapitelende)
- Struktur der prozessorinternen Systembusse
- Peripheriekomponenten (intern / im System)

↳ insbes. Mikrocontroller ↳ nicht lokale Peripherie (Drucker)

- Hauptbestandteile des Prozessors:

Steuerwerk - Rechenwerk - Speicherwerk

↓
Control
Unit

↓
ALU
Arithmetic(al)
Logic(al)
Unit

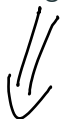
↓
Memory, M
(Unit)

- Unterste Ebene, auf der Inhalte der Informatik noch als solche erkennbar sind, z.B.
 - Logische Verknüpfungen
 - Einfache Rechenoperationen
 - Datenspeicherung und -transfer
- Auch innerhalb dieser Kategorie sind Komponenten in ihrer Komplexität gestaffelt, wie
 - Addierer
 - Logikgatter
 - Multiplexer
 - Busse

↑ Komplexität

- Hier im Fokus: Halbleiterbauelemente wie Dioden, Transistoren
- Elektrotechnische Basis der digitalen Computertechnik:

analoge Elektronik \Rightarrow digitale Elektronik



beliebige Spannungspegel

realisiert



2 Spannungspegel: 1 / 0

noch zu finden in *Analogrechnern*

digitale Rechensysteme

Begriffsfeld "Prozessor"

- "Data Processor": verarbeitet Daten
- Auf moderne Rechensysteme übertragen: führt *Programme* aus: Sequenzen von Instruktionen, die Daten lesen, verändern, schreiben (EVA-Prinzip: **E**ingabe, **V**eränderung, **A**usgabe)
- Prozessor ist die *Zentraleinheit* (*Central Processing Unit*, **CPU**) eines *Rechensystems*, abgegrenzt von
 - Peripherie
 - Speichern
 - Bussen
- Bis ca. 1970 *diskret* aufgebaut, ab dann *Mikroprozessoren* (Intel 4004, Texas Instruments TMS 1000)

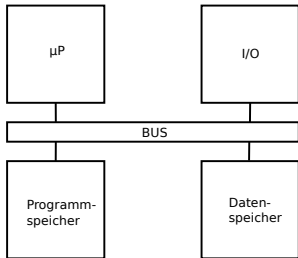
↳ eigentl. schon µController ↳ Prozessor in 1 IC

- Mikrocontroller, Microcontroller, μ Controller =
 μ Prozessor
+ eingebettete Peripherie
+ eingebetteter Speicher
- Komplettes Rechensystem auf einem IC!
- Anfangs oft als Prozessoren in Peripheriegeräten
(z.B. Beginn der PIC-Famile von Microchip)
- Prozessorkerne heutiger μ Controller aber auch mit fast allen
Features von Server-Hochleistungsprozessoren erhältlich
(Beispiel: Trend in der Handy-Entwicklung)

- **SoC: System-on-a-Chip**
- Jenseits des Mikrocontroller-Konzepts: noch mehr Teile eines Rechensystems auf einem IC
- Häufig Multi-Die
(mehrere Si-Chips in einem gemeinsamen Package)
- Oft aus separaten Coprozessoren hervorgegangen
z.B. μ Controller + Radio-IC \rightarrow Funk
- Kompaktere, effizientere Lösungen möglich
- ggf. weniger Flexibilität im Design

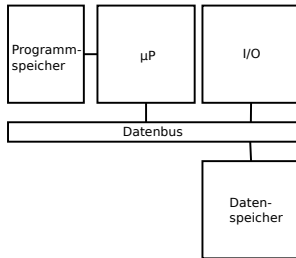
- **PSoC:** P=Programmable (Hardware)
 - μ Controller + konfigurierbarer HW-Bereich
 - SoC + FPGA \rightarrow Field Programmable Gate Array
Programmiersbare Logik-IC
- Größere Flexibilität
- ggf. Höherer Aufwand in Design und Debugging
- ggf. kompaktere, energieeffizientere oder schnellere Implementierungen
- ...wenn es richtig gemacht wird (Expertise noch rar)

Mikroprozessor im Rechengesystem



Von-Neumann-Architektur:

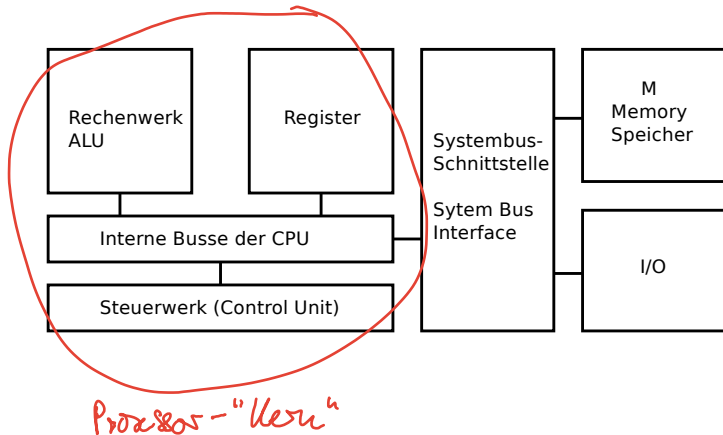
- Programm- und Datenspeicher an *demselben* Bus
- Kann auch gemeinsamer Prog.-/Datenspeicher sein



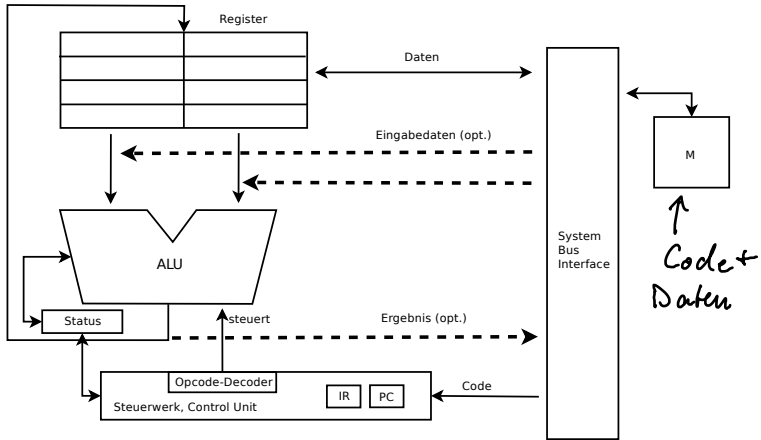
Harvard-Architektur:

- Getrennte Programm- und Datenspeicher, separate Busse
- Vermeidet '*Flaschenhals*' bei konkurrierenden Programmcode- und Datenzugriffen

Komponenten eines Mikroprozessors



Datenpfade und Verschaltung der Komponenten



Optionale Pfade (- - -) *nicht* bei Load-/Store-Architekturen!

Abarbeitung von Maschinencode: Code im Speicher

Hier gezeigt: Speicherworte (z.B. 16-Bit-Worte bei AVR-Maschinensprache)

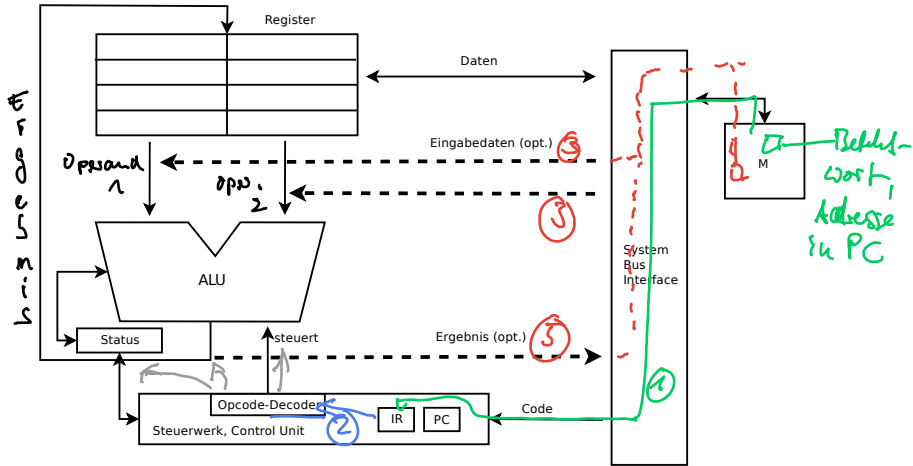
PC: 0x1000 $+2$ $+1$ $+1$ $+2$ $+2$ notequal:



Adresse	Assembler	→	Maschinencode	LDS
0x1000	(1) LDS R15, 0x2000	=>	90f0 2000	2 x Wort, Op code u. R15 - Operand 0x2000
0x1002	(2) LDI R17, 0xbc	=>	eb1c	0xbc in Opcode integriert
0x1003	(3) CMP R17, R15	=>	151f	
0x1004	(4) BREQ isequal	=>	f009	1 Wort, weiter bei (5) oder (6)
0x1005	(5) JMP notequal	=>	c001	
isequal:				
0x1006	(6) NOP	=>	0000	

Adressen zählen Worte bzw. Wörter, hier 16-Bit pro Opcode
Wortbreite für Daten kann abweichen (AVR: 8 Bit)

Abarbeitung von Maschinencode: Ablauf



(1) Instruction Fetch

(2) Instruction Decode

(3) Operand Fetch

(4) Execute

(5) Write Back

Phasen der Befehlsabarbeitung

1. Instruction Fetch

Laden der nächsten Instruktion, auf die der Program Counter (PC)¹ zeigt, aus dem Programmspeicher. Transfer in Instruction Register (IR).

2. Instruction Decode

Dekodieren der Instruktion: Bestimmen des auszuführenden Befehls und der Adressierungsart, Extraktion von im Opcode eingebetteten Operanden.

3. Operand Fetch

Falls erforderlich: Laden von zusätzlichen Operanden aus dem *Hauptspeicher*

4. Execute

Ausführen der Operation: Aktivierung der benötigten CPU-Komponenten und Datenpfade in der korrekten Reihenfolge.

5. Write Back

Falls erforderlich: Schreiben von Ergebnissen in den *Hauptspeicher*

¹Alternativer Name für PC: Instruction Pointer (IP)

Decodieren und Ausführen der Instruktion: festverdrahtet

- Explizit entworfene Schaltnetze und Schaltwerke, um aus den Bits der Opcodes Steuersignale für das Ansteuern der CPU-Komponenten abzubilden
- Klassischer Ansatz, schwierig zu entwickeln
- Renaissance durch RISC-Ansatz (Reduced Instruction Set Computer): weniger, einfachere Befehle ermöglichen wieder festverdrahtete Steuerwerke
- erhoffter Geschwindigkeitszuwachs

- Merkmal von CISC(Complex Instruction Set Computer)-Architekturen
- Komplexe und aufwendig zu realisierende Befehle können nicht mehr festverdrahtet umgesetzt werden
- Stattdessen: “CPU’ in der CPU” führt für jeden Befehl ein Mikroprogramm aus
- Mikrobefehle sind wiederum festverdrahtet
- flexibel, aber ggf. langsamer

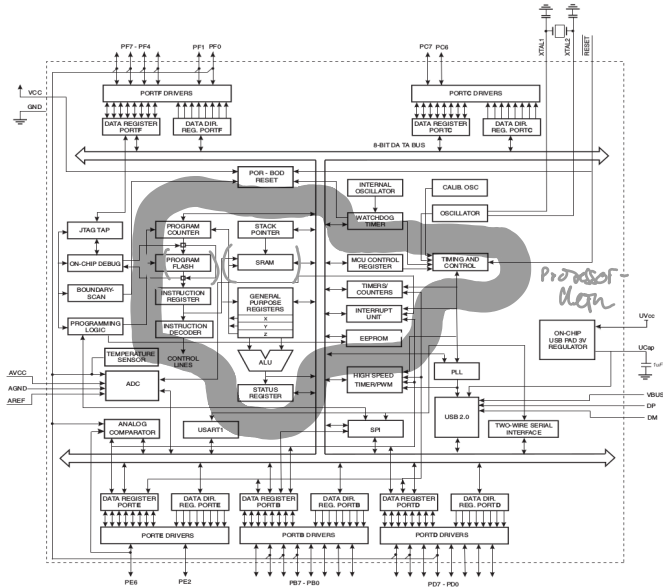
Laden von Operanden

Als Architekturvereinfachung werden *Load-Store-Architekturen* angesehen (u.a. AVR):

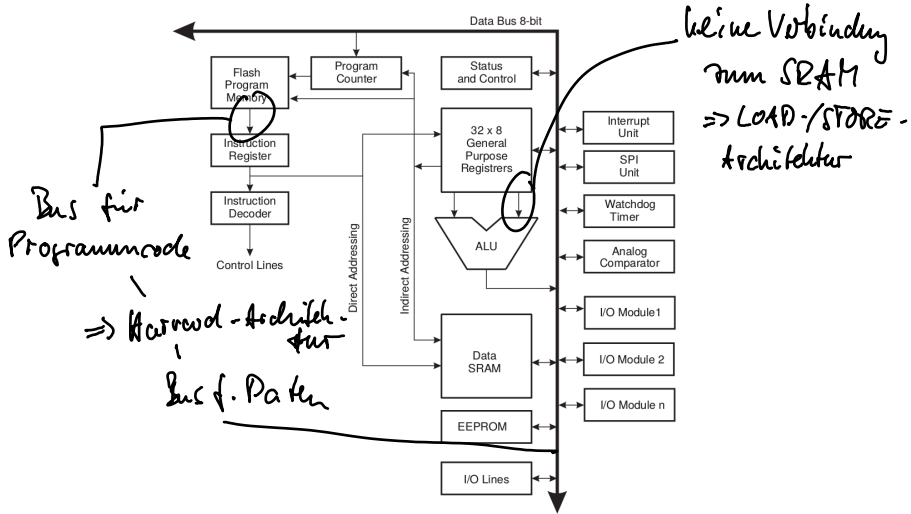
- Austauschen von Daten mit dem Hauptspeicher ist *nur* mit speziellen Transferbefehlen (z.B. LOAD/STORE)
- AVR-Instruktionen hierzu:
LD(S)/ST(S)), Load/Store (in)direct from/to Data Space
- ALU-Operationen können Ein- und Ausgangswerte dagegen *nur mit Registern* austauschen
- Die Pipeline-Stufen “Operand Fetch” und “Write Back” *entfallen* bei diesen Architekturen

Komplexere Architekturen unterstützen auch ALU-Befehle mit direktem Speichertransfer der Parameter und Ergebnisse

Blockdiagramm ATmega32U4



Blockdiagramm ATmega32U4-Core



Register

- Kleinsten, schnellsten ~~vor~~ im Programmcode direkt ansprechbarer Speicher (RAM) in der CPU
- “Notizzettel” für Rechnungen
- $t_{\text{access}} : \text{Register} < \text{Cache} < \text{Hauptspeicher} < \text{FlashROM}$
($< \text{Massenspeicher}$)

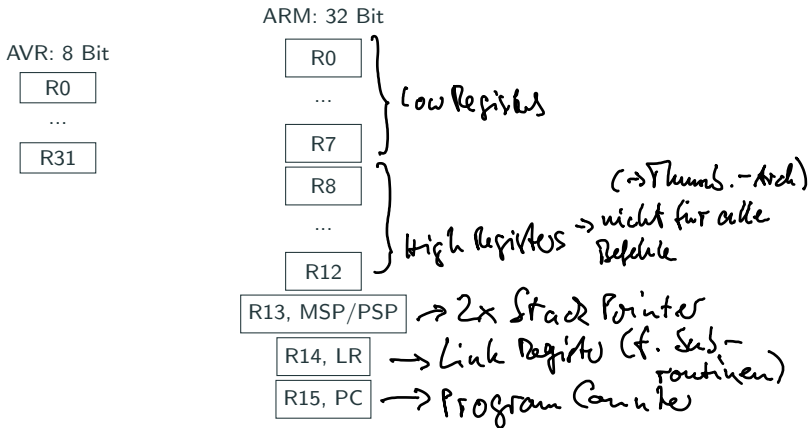
⇒ Teuer, nur wenige Bytes-KBytes

- Enge Kopplung an ALU (immer Zugriff ALU \Leftrightarrow Register, nicht unbedingt zum ALU \Leftrightarrow Hauptspeicher)

⇒ Registerbreite (Bits/Wort im Register) = Wortbreite der CPU-Architektur

Beispiele: Register AVR und ARM

- "General Purpose / Standard-Register
- Für RISC-Architekturen typisch: Einheitliche Registerbreite, Viele und gleichwertige Register



Spezialregister

Spezial:

- Besondere Funktion
- Nicht mit gängigen Instruktionen (z.B. ADD) verwendbar

AVR:

X(R26:R27) \Rightarrow Indexregister

Y (R28:R29) \Rightarrow Indexregister

Z (R30:R31) \Rightarrow Indexregister

SREG \Rightarrow Statusregister

SPH:SPL \Rightarrow Stack Pointer

PC \Rightarrow Program Counter

2 Register, weil
Stack \Rightarrow 16-bit Adresse!

ARM:

xPSR \Rightarrow Statusregister

Masks \Rightarrow Maskieren, u.a. v. Interrupts

CONTROL \Rightarrow Ausführungsmodus umschalten

PC, LR, SP sind bei ARM *keine* Spezialregister (sondern R13-R15) \Rightarrow Rechnen mit **PC** ist möglich! (z.B. Sprungtabellen)

(LIFO)

Einfuhr / ADS: Stack-Konzept? (alt. „Keller“)

PUSH: Wert auf Stack legen } als Assembler-ops verfügbar
POP: „ von Stack holen }

Technisch offen, je nach Systemarchitektur festgelegt:

↓ ↑ ? AVR und AVR: Stack wächst nach unten!
(Anfangs SP-Adresse am höchsten, PUSH verringert SP usw.)

	POP	PUSH	
AVR	PRE- inkr.	post- dekr.	⇒ SP zeigt auf leeres Element (nächstes)
AVR	post- inkr.	PRE- dekr.	⇒ SP zeigt auf letztes belegtes E.

Geben Zustand des Prozessors wieder (hauptsächlich ALU)

Ergebnis^{des} d. letzten ALU-
Operation (z. B. ADD)

Load/Store-Instruktionen verändern Statusbits meist nicht.

- oder gezielt: ARM MOV → MOVS

↳ Status verändern

Status ist Basis für Conditionals: Bedingungen für Sprünge
u. andere Operationen

Weitere wichtige Statusinformationen:
Interrupt-Verarbeitung, Betriebsmodus

Statusregister AVR

- N : Negative : höchstes Bit (jeweils 1 Bit in AVR SREG-Register pro Buchstabe)
- Z : Zero : Ergebnis war 0
- C : Carry : Übertrag bei Operation (auch shift & co.)
- V : Overflow : Überlauf
- S : Sign : $N \oplus V$: korrektes Vorzeichen auch nach Überlauf arith.
- H : Half-Carry : Übertrag Bit 3 \rightarrow 4
- T : Transfer : Hilfsbit, um Bitwerte zu kopieren (BLD + BST)
- I : Interrupt : Enable/Disable

N	V	\oplus	
0	0	0	\rightarrow positiv
0	1	1	\rightarrow neg. trotz $N=0$
1	0	1	\rightarrow neg.
1	1	0	\rightarrow positiv trotz N-Flag

(basiert auf letzte ALU-Op.-Ergebnis wenn nicht anders angegeben)

Statusregister ARM

N : Negative

Z : Zero

C : Carry

V : overflow

Q :: Saturation Bit (QADD : Addition m. 7 max-Ergebnis, z.B.

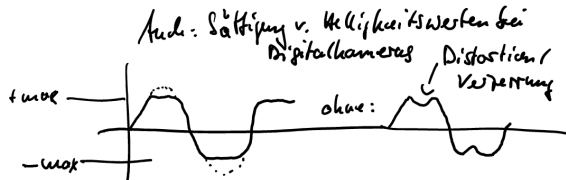
$$\max(8, 3+7)=8$$

|C|/IT:4 :

T; Phantasie-kraft.

Exception No.: 8

) → Interrupts



Arithmetic(al) Logic(al) Unit

⇒ Rechenoperationen, Bitmanipulation inkl. Shift + Rotate

In Load/Store-Arch: keine Anbindung an Hauptspeicher (HS)

Ggf. auch Berechnung von Sprungzielen

- + flexible, mächtige Kontrollanweisungen

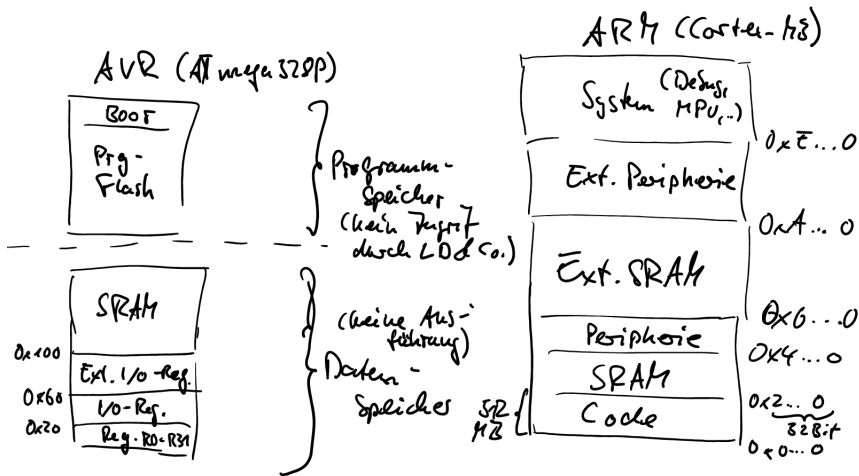
- Verknüpfung von ALU mit Steuerwerk

Schnittstelle zw. Prozessoren und integriertem Speicher +
Peripherie
μController

bzw. externem Sp. + Periph
μ Prozessor + klassische Systemarch.

- ggf. mehrere Schnittstellen (log. Sicht)
 - └ RAM
 - └ Code-Speicher
 - └ I/O
 - └ DMA (Direct Memory Access)

Speicher



Von Neumann-/Harvard-Architektur

Unterschied: Programm + Daten in gemeinsamen Speicher
⇒ v. Neumann

in separaten Speichern
⇒ Harvard

Von-Neumann-Arch. legt auch fest:

- Codierung von Befehlen in Opcodes ⇒ Nutzung des HS wie für Daten
- Ausführungsphasen ⇒ Modularisierung d. Mikroarchitektur

Ausführungsreihenfolge sequentiell, aber die Sprünge wandelbar
Realisierbar im Program Counter Register:
 Inkrement: lineare Progr. Ausführung
 Sprung: PC-Wert neu laden

Extras der Mikrocontroller

- Speicher
 - Programm/Code: (Flash-) ROM
 - HS: RAM (meist statisch)
 - Maschinen-/Zwischenspeicher, z.B. EEPROM, F(ER)AM
 - I/O
 - analog: D/A / A/D - Umsetzer
 - digital
 - parallel
 - seriell
 - Synchron (USART)^{zsp.}
 - asynchron (UART)
 - (Feld) serielle Busse
 - SPI
 - IIC/I²C
 - One-Wire-Busse
 - drahtlos
 - Timer/Counter
 - LPWM
 - LWDT
 - Capture/Compare
 - Memory Management:
 - kein MPU
 - MPU
 - MMU
- MPU = Memory Protection Unit
MMU = Memory Management U.
- PWM = Pulse Width Modulation
WDT = Watchdog Timer

- Nützliche Kommandos:
 - layout
 - layout src: (C-)Quelltext anzeigen
 - layout ~~asm~~ ^{asm} Assembler-Quelltext anzeigen
 - layout regs: letzte src/asm-Einstellung + Registeranzeige
 - continue (c) führt bis zum nächsten Breakpoint aus
- C-Programme auf Assemblerebene debuggen:
 - n bzw. s überspringen *alle* Assembler-Instruktionen eines C-Statements
 - nexti bzw stepi führen jew. eine Assembler-Instruktion aus
 - Optimierung des Compilers mit -O0 ausschalten!

- Nützliche Befehle auf der *qemu-Konsole* (es gibt eine!)
 - `system_reset`: Prozessor (und damit Programm) neu starten (gdb kann offen bleiben)
 - `info ...` (\Rightarrow `help info`), z.B. `info registers`
- Wenn UART-Emulation gewünscht: `-serial none` beim Start weglassen
- Start ohne `-nographic`: öffnet neues Fenster, dort mit `Ctrl-ALT-1..3` zwischen Grafik, Konsole und serieller Ausgabe umschalten

Generell: *Emulation* versucht, das Verhalten eines Systems nach außen korrekt wiederzugeben, während *Simulation* versucht, innere Mechanismen des Systems nachzubilden.