# README

## Beschreibung

Zusammenfassung für Computer Engineering 2 auf Grundlage der Vorlesung FS 16 von Erwin Brändle
Bei Korrekturen oder Ergänzungen wendet euch an einen der Mitwirkenden.

## Modulschlussprüfung

Kompletter Stoff aus Skript, Vorlesung, Übungen und Praktikum

- Vorlesungsskript CompEng2 V1.2 komplett
  (die Kapitel 2 und 7 sind im Selbststudium individuell aufzuarbeiten)
- Korrigenda zum Skript, falls eine solche vorliegt
- Übungen im Vorlesungsskript
- Inhalt aller Praktika (inkl. Pre-/Post-Lab Übungen)
- in Vorlesungen und Praktika zusätzlich vermittelte Informationen
- Inhalt und Umgang mit dem Quick-Reference/Summary V1.2

### Die Prüfung besteht aus 2 Teilen:

| | | |
|---|---|---|
| **1.Teil** | closed Book | Theoretische Fragen zum ganzen Prüfungsinhalt |
| **2.Teil** | semi-open book | Aufgaben im Stil der Übungen, Praktika und der in den Vorlesungen gelösten Aufgaben |

### Plan und Lerninhalte

Fokus: ARM Cortex-M Architektur

- RISC-Architektur, Core-Components, Register Model, Memory Model, Exception Model, Instruction Set Architecture
- Konzept und Umsetzung der vektorisierten Interrupt Verarbeitung
- Abbildung von typischen C Programmstrukturen und Speicherklassen in das Programmiermodell der CPU
- Systembus: Address-, Daten-, Control-Bus, Adressdekodierung, Memory- und I/O-Mapping
- Speicher- und ausgesuchte Peripherieschnittstellen

## Contributors


| | |
|---|---|
| Luca Mazzoleni | luca.mazzoleni@hsr.ch |
| Stefan Reinli | stefan.reinli@hsr.ch |


## License


**Creative Commons BY-NC-SA 3.0**
Sie dürfen:

- Das Werk bzw. den Inhalt vervielfältigen, verbreiten und öffentlich zugänglich machen.
- Abwandlungen und Bearbeitungen des Werkes bzw. Inhaltes anfertigen.

Zu den folgenden Bedingungen:

- Namensnennung: Sie müssen den Namen des Autors/Rechteinhabers in der von ihm festgelegten Weise nennen.
- Keine kommerzielle Nutzung: Dieses Werk bzw. dieser Inhalt darf nicht für kommerzielle Zwecke verwendet werden.
- Weitergabe unter gleichen Bedingungen: Wenn Sie das lizenzierte Werk bzw. den lizenzierten Inhalt bearbeiten oder in anderer Weise erkennbar als Grundlage für eigenes Schaffen verwenden, dürfen Sie die daraufhin neu entstandenen Werke bzw. Inhalte nur unter Verwendung von Lizenzbedingungen weitergeben, die mit denen dieses Lizenzvertrages identisch oder vergleichbar sind.

Weitere Details: http://creativecommons.org/licenses/by-nc-sa/3.0/ch/

# ComEng2 Zusammenfassung

L. Mazzoleni S. Reinli

4. Oktober 2019

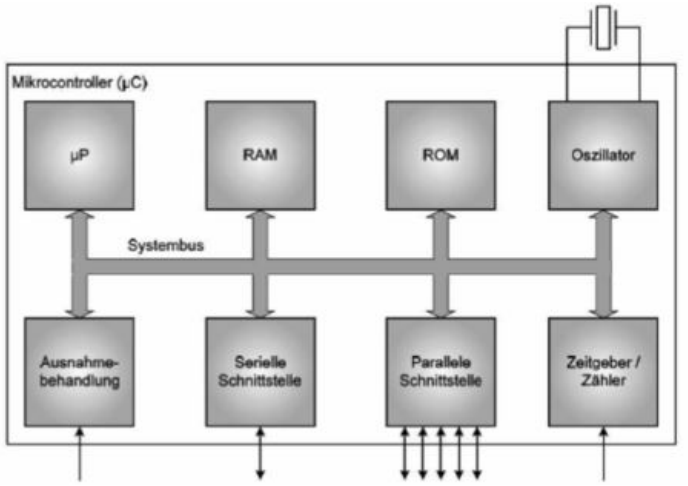
## Inhaltsverzeichnis

# 1 V1

## 1.1 Anwendung und Grundlage der uP-Technik

## 1.2 Aufbau

Verstehe die wesentlichen Systemkomponetnen des Rechnersystems auf einem IC (Integrated Circuit)

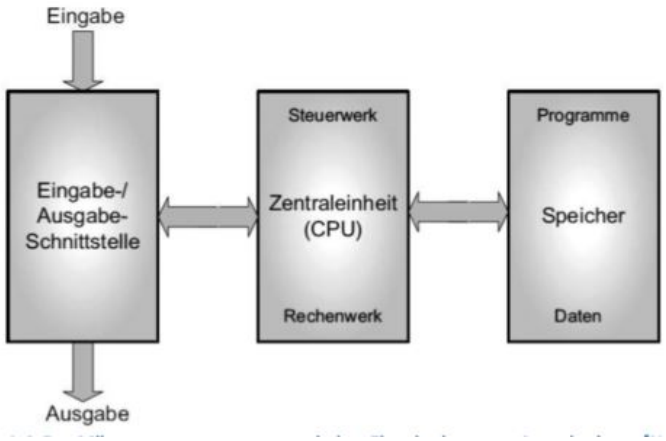



### 1.2.1 Anwendungen

- Supercomputer
- Arbeits und Server-Rechnern
- Smartphones
- Navigationssysteme
- Digitalkameras
- Drucker
- ... TEst

### 1.2.2 Aufbau von uP-basierten Systemen

- Zentraleinheit CPU mit
  - Rechenwerk ALU
  - Steuerwerk CU
  - Registersatz
- Speicher
- Eingabe-/Ausgabe-Schnittsellen



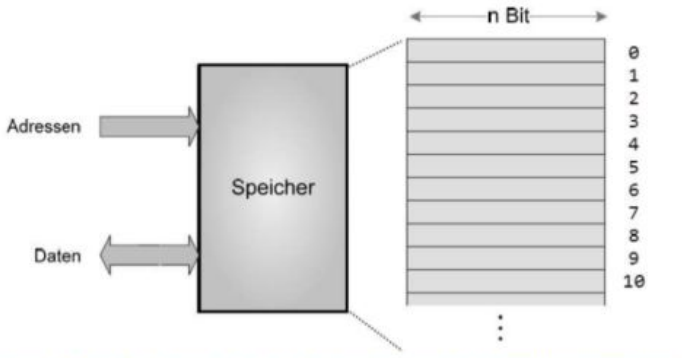
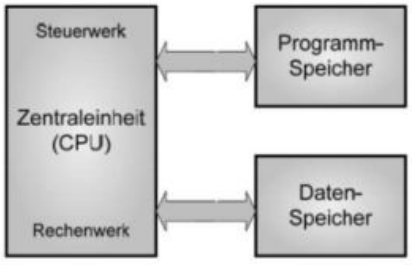



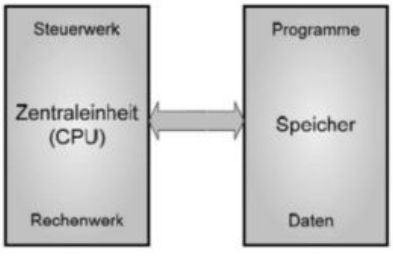

Figure 1-7 Der Speicher ist in eindeutig adressierbaren Speicherplätzen organisiert

### 1.2.3 Havard vs Von Neumann Architektur

**Harvard Rechnermodell**

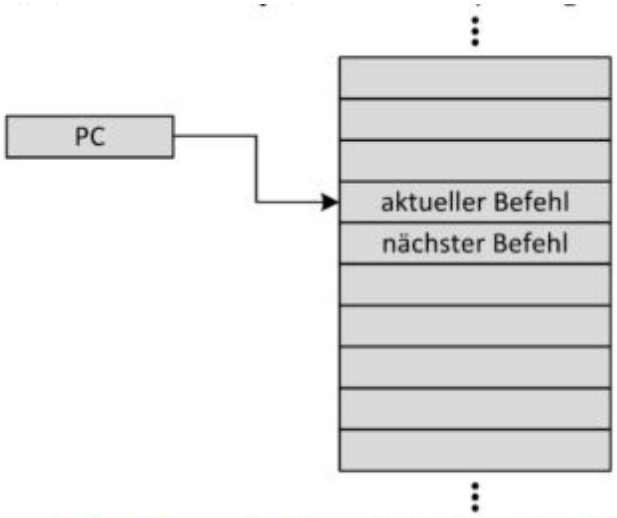



**von Neumann Rechnermodell**



### 1.2.4 Programmierung eins uP

Ein *µPkanndurchindividuelleProgrammierungauf ganzunters*
→ *entscheidend frdieDurchdringungimMarkt.*
*EinProgrammenthltinauf einander folgenderAnordnungdieM*
*Befehleoder – Instruktionen frdenµP.DieseMaschiene –*
*Befehleteilender CPUmit, welcheOperationeninwelcherReihen*
*DieBefehls folgedesProgrammswirdinnerhalbderCPUvomSte*
*DerPCenthltlauf enddieAdresseder Speicherzelledes jeweiligen*




*uellzu*

Figure 1-9 Der PC adressiert den aktuellen Maschinenbefehl

### 1.2.5 Befehlsformate

Die Art und Wirkung eines Befehls wird im Befehlswort (**OpCode**) codiert. Darin sind neben der Operation auch die Operanden spezifiziert. Die Codierung des Befehlswortes erfolgt abhängig vom *µP.DerMaschinencodesetztsichauseinemOpCodeundeinemode*

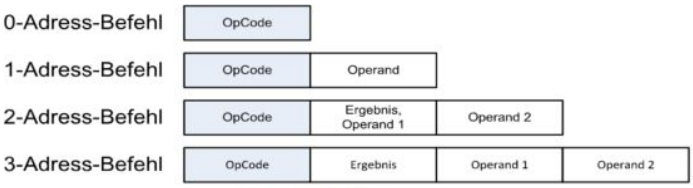



Figure 1-10 Befehlsformate

## 1.3 RISC vs CISC

**CISC**
Complex Instruction Set Computer

**RISC**
Reduced Instruction Set Computer

### 1.3.1 RISC-Rechner

effizienter als CISC-Rechner

- besteht aus einer kleinen Anz. von Befehlen mit wenigen Adressierungsarten
- Registersatz enthält eine grosse Anzahl von allg. verwendbaren Registern
  General Purpose Register (GPR)
- Speicherzugriff erfolgt über spezielle Lade- und Speicher-Befehle
  - Arithmetisch-logische Operationen arbeiten auf Registeroperanden

- Pipeline-Architecture ← *LeistungssteigerndeArchitekturEinegrossesemantischeLckeentstehtbeiderbersetzungausderHochsprac*

### 1.3.2 u Architektur

Beschreibt die architektonischen Details bei der Implementierung der *µPausSichtderProgrammierer.Diesum fasstdieBeschre*

## 1.4 Hardware

### 1.4.1 Registersatz

Register sind schnelle Zwischenspeicher für temporäre Daten im $\mu P$.

### 1.4.2 Hardware- /Software-Schnitsttelle

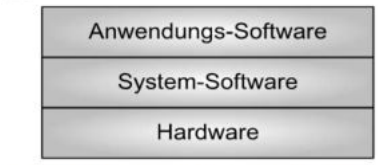


Figure 1-11 Vereinfachte HW/SW-Hierarchieebenen

### 1.4.3 Taktfrequenz

Das Taktsignal steuert die zeitliche Abfolge im $\mu P$

$\Uparrow$ Taktrate $\Leftrightarrow \Uparrow$Leistungsaufnahme
Um Energie zu sparen ist es sinnvoll die Taktrate laufend anzupassen.

### 1.4.4 Leistungsaufnahme

$$P_{Gate} = \frac{1}{2} \cdot C_{Last} \cdot V_{DD}^2 \cdot f_{Takt}$$

| | |
|---|---|
| $P_{Gate}$ | Leistung pro CMOS Gate |
| $C_{Last}$ | Lastkapazität |
| $V_{DD}$ | Versorgungsspannung |
| $f_{Takt}$ | Taktfrequenz |

## 1.5 Software

### 1.5.1 Ablauf

- Der **Präprozessor** bereitet das Quellprogramm für den Compiler vor
- Der **Compiler** übersetzt das Programm von einer Hochsprache in ein Assembly-Programm
- Der **Binder** fasst verschiedene Dateien, die verschiebbaren Maschinencode enthalten, zu einem Programm zusammen.
- Der **Loader** wandelt die verschiebbaren Adressen in absolute Adressen um und lädt sie in den Speicher des Systems.

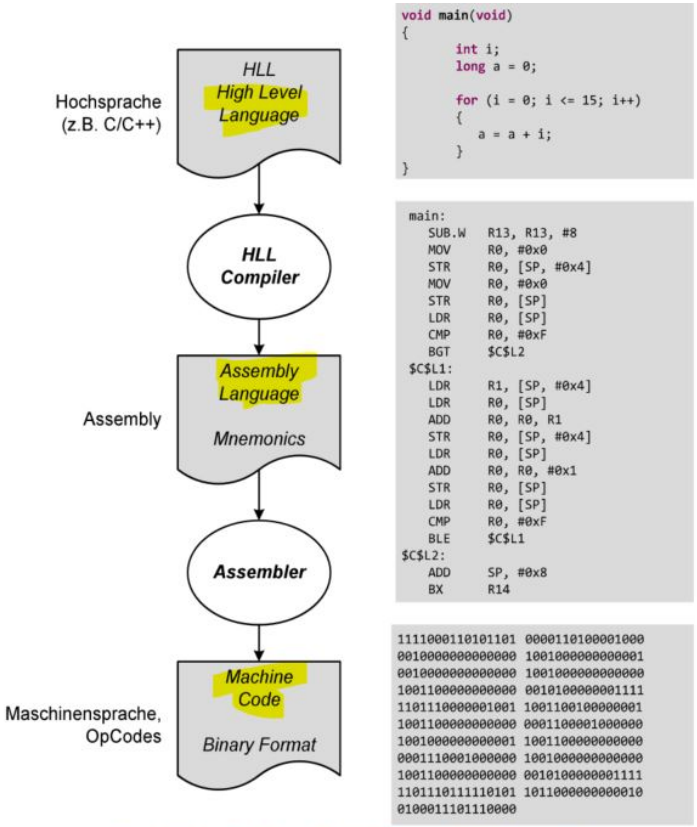


Figure 1-13 Compilier/Assemblier-Workflow am Beispiel Cortex-M3

# 2 V2

## 2.1 Compiler-Schritte

1. **Lexikalische Analyse (Scanning):**
   Die Symbole der Sprache werden erkannt und Gruppiert. Leerzeichen werden eliminiert
2. **Syntaxanalyse (Parsing):**
   Die erkannten Symbole werden in Sätzen zusammengefasst und in einem Parsbaum dargestellt
3. **Semantische Analyse:**
   Das Quellprogramm wird auf Fehler überprüft (zBsp. Typfehler) und der Parsbaum erhält Informationen über die verwendeten Bezeichner
4. **Zwischencode-Erzeugung:**
   Einige Compiler erzeugen Code in einer Zwischensprache (abstrakte Maschinen)
5. **Code-Erzeugung:**
   Erzeugen von verschiebbarem Maschinencode.

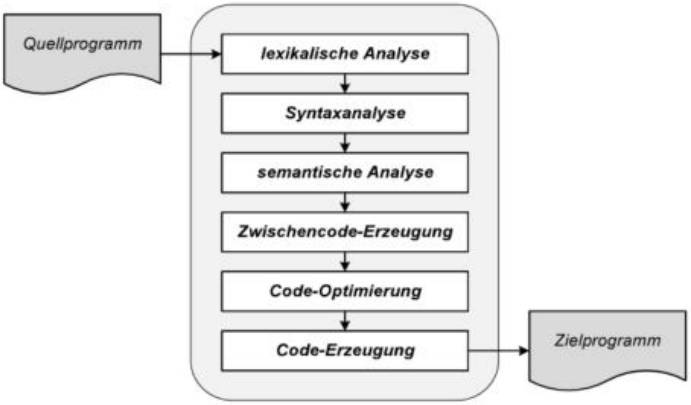



Figure 1-14 Transformation eines Programms in Maschinencode

## 2.2 Busorientierte Systeme

### 2.2.1 Speicher

**RAM**

- Random Access Memory
- Schreibe-/Lese-Speicher
- Spannungsversorgung erforderlich
- für temporäre Daten

**ROM**

- Read Only Memory
- Festwert Speicher
- auch ohne Spg. bleiben Daten erhalten

**Adressbus**

- unidirektional
- bestimmt Grösse des Adressraums

**Databus**

- bidirektional

**Steuerbus**

- kontrolliert Buszugriffe
- zeitlicher Ablauf der Signale

* Die gesammte Menge der über den Adressbus adressierbaren Speicherzellen wird **Adressraum** genannt

* Die Anzahl parallel geführten **Datenleitungen** entspricht der maximal zu übertragenden Datenbreite

* Kontrollsignale werden über den **Steuerbus** übertragen

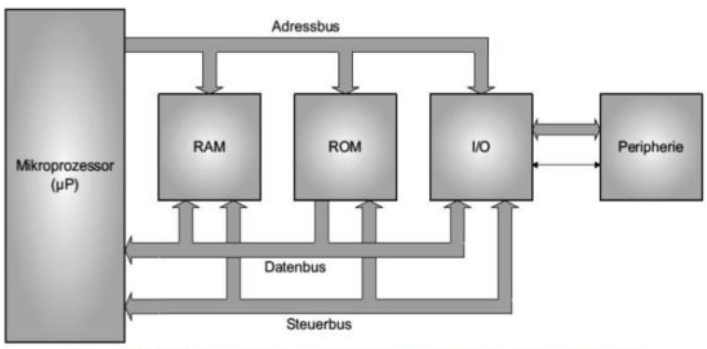



Figure 1-15 Einfaches µP-System mit Speicher und I/O-Schnittstellen [Neu07]

### 2.2.2 Architectur eines uP

| | |
|---|---|
| **AR** | Adressregister |
| **PC** | Programm Counter |
| **PSR** | Program Status Register |
| **IR** | Instruction Register |

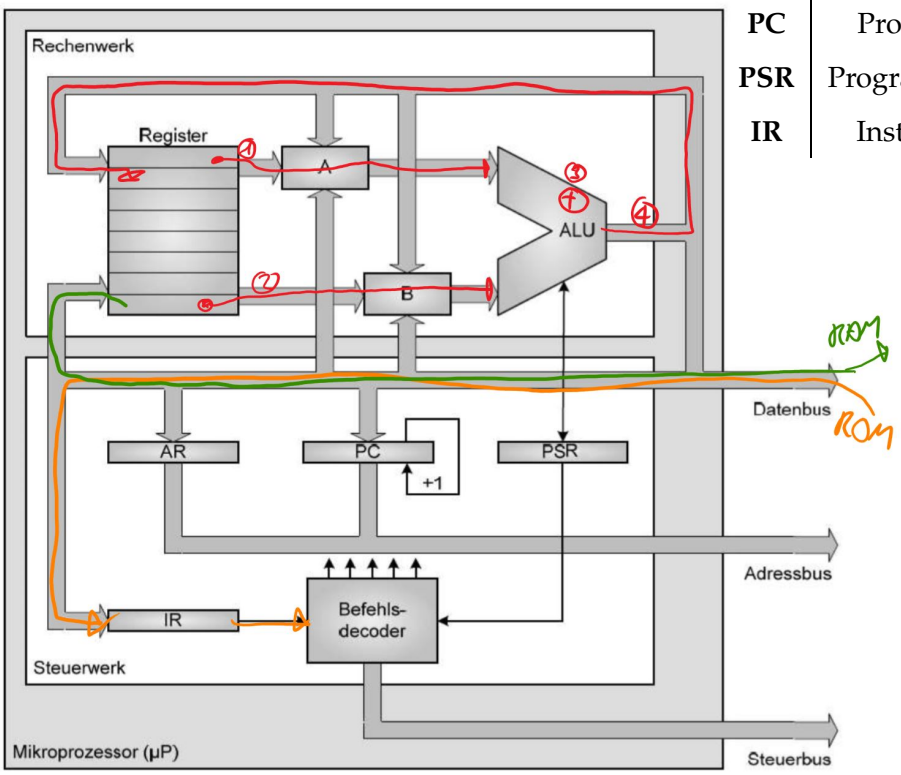



Figure 1-20 Architektur eines einfachen Mikroprozessors [Neu07]

**Flags**

| | | |
|---|---|---|
| **N** | **Negative** | Das von der ALU berechnete Ergebniss ist negativ |
| **Z** | **Zero** | Das von der ALU berechnete Ergenis ist gleich **0** |
| **C** | **Carry** | Die Berechnung der ALU hat zu einem Übertrag geführt |
| **V** | **Overflow** | Die Berechnung der ALU hat zu einem Overflow geführt |

## 2.3 Befehlszyklus

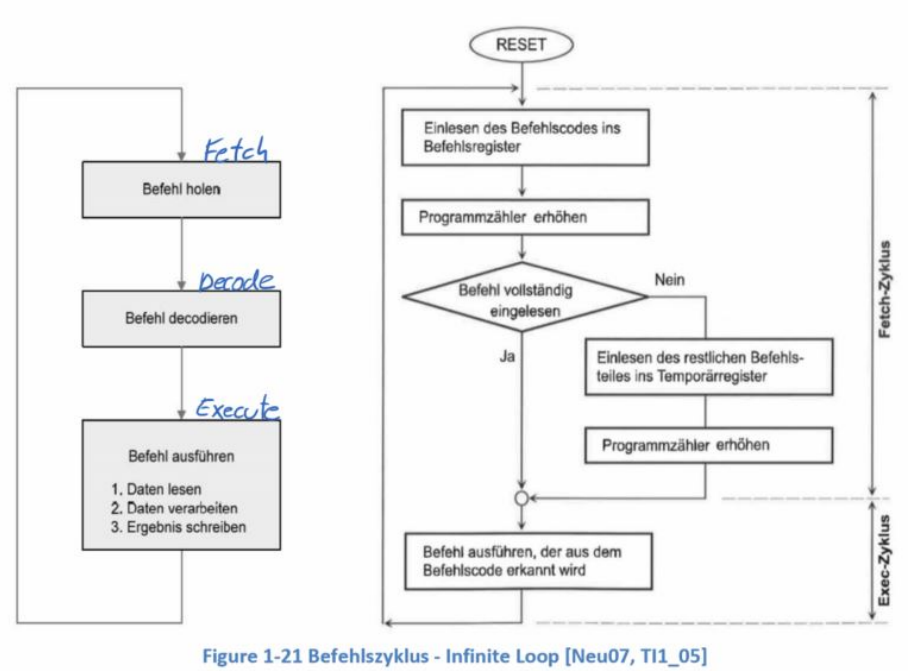



Figure 1-21 Befehlszyklus - Infinite Loop [Neu07, TI1_05]

# 3 V3

## 3.1 Halbleiter Speicher

### Zentraler Speicher

- direkt am Bussystem angeschlossen

### Peripherer Speicher

- über I/O-Schnittstelle angeschlossen

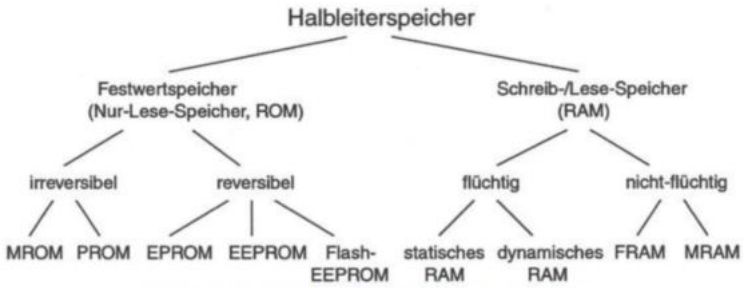



Figure 1-22 Übersicht der Halbleiter-Speicher Familien

### 3.1.1 ROM-Festwertspeicher

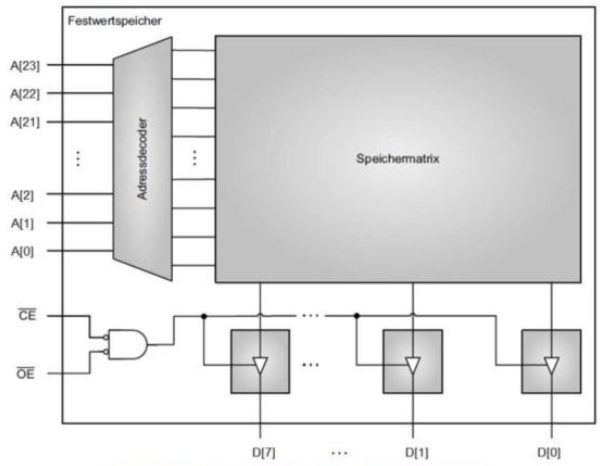



Figure 1-23 Grundaufbau ROM Festwertspeicher [Neu07]

### 3.1.2 RAM-Speicher-/Lese-Speicher

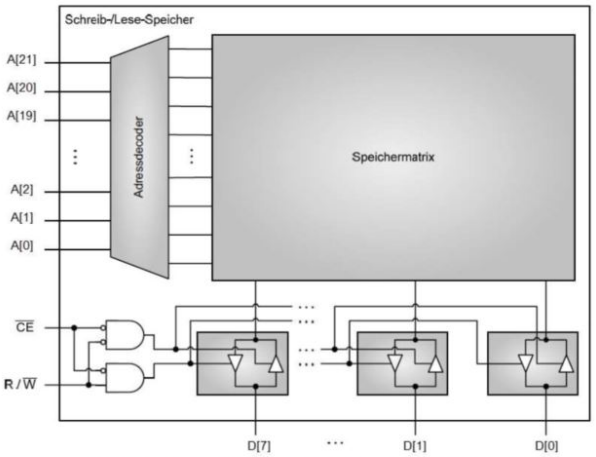



Figure 1-24 Grundaufbau RAM-Speicher mit kombinierter R/W-Leitung

## 3.2 Speicherorganisation

### 3.2.1 Little/Big Endian

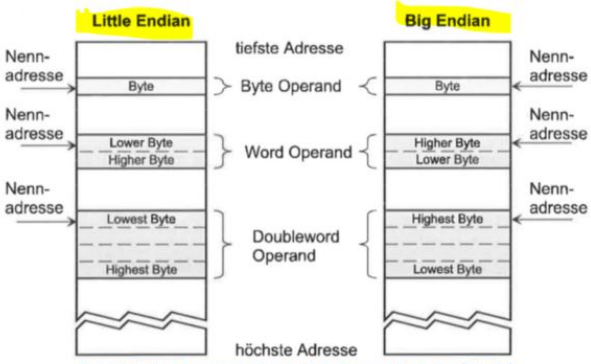



Figure 1-43 Speicherorganisation: Little Endian und Big Endian [TI1_05]

### 3.2.2 I/O - Schnittstelle

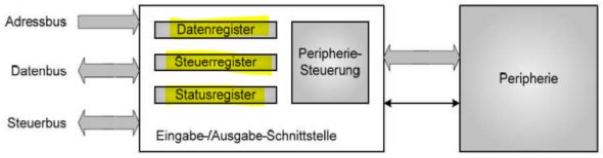
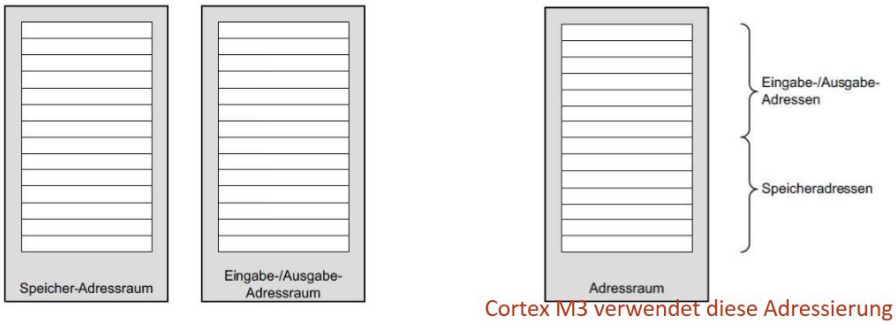



Figure 1-26 Allgemeine Struktur einer I/O-Schnittstelle [Neu07]




Cortex M3 verwendet diese Adressierung

# 4 V4

## 4.1 Cortex M Varianten

**Cortex M0 und M0+**

- kleinster Vertreter der CortexFam
- Ersatz von 8Bit- uC

**Cortex M1**

- als Softcore Implementiert
- Vergleichbar mit Cortex-M0

**Cortex M3**

- erster Vertreter der CortexFam
- 32 Bit Architektur
- ersetzt 8 & 16 Bit uC
- Thumb ISA (Instruction Set Architecure)
  Mix aus 16 und 32Bit langen anweisungen

**Cortex M4**

- vergleichbar mit M3 jedoch mit:

  - Digital Signal Processing (DSP)

  - Floating Point Unit (FPU)

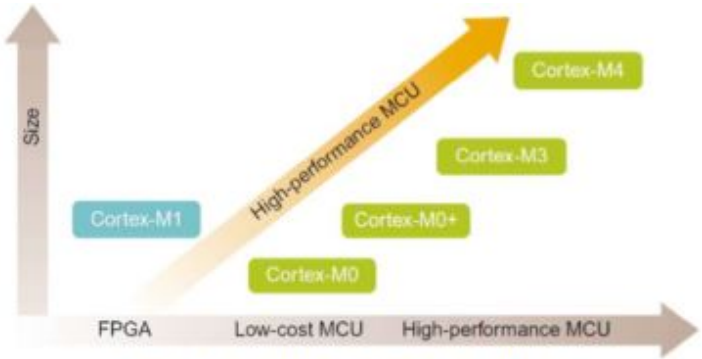


Figure 3-3 Vertreter der Cortex-M Familie [Martin]

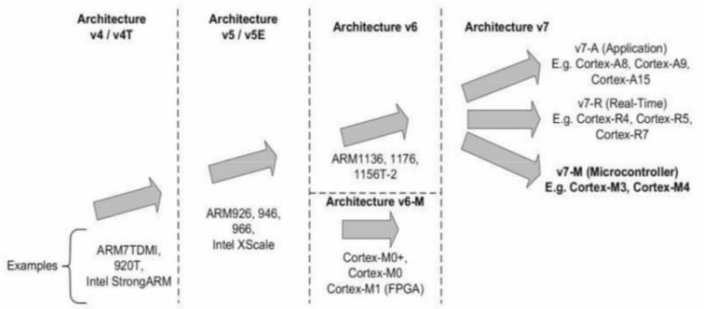


Figure 3-4 Evolution der ARM-Architekturen [Yiu3]

**Cortex-A**

- HighEnd Anwendungen und Betriebssysteme
- hohe Rechenleistung
- Chache Memory

**Cortex-R**

- Echtzeitfähigkeit
- hohe Zuverlässigkeit
- System on Chip (SOC)

**Cortex-M**

- Speziell für $\mu C$ – $MarktLowCost, LowEnergy$
- System on Chip (SOC)

### 4.1.1 Vorteile der Cortex-M-Prozessoren

- Low Power
  $< 200 \mu A/MHzPerformance$
  >1.25 DMIPS / MHz
- Energy Efficiency
  low Power, high performance
- Code Density
  Thumb 2 Befehlssatz
- Interrupts
  240 Interrupts
- Easy of Use, C Friedly
- Scalability
- Debug Features
- Software portability and Reusebility
- OS Support
- Choices (Derivers, Tools, OS,..)

## 4.2 Cortex-M3/M4

- Harvard Architecture
  → *Zugriff auf Instruktionen und Daten*
  können gleichzeitig stattfinden
- Internal Bus Interconnect
  → *mehrere Bus−Interface Nested Interrupt Controller* **(NVIC)**
- Standart Timer **(SYSTICK)**

**Optional:**

- Memory Protection Unit **(MPU)**
- Floating Point Unit **(FPU)**

## 4.3 System-Komponenten

### 4.3.1 NVIC

- Non-Maskable Interrupt (NMI)
- Bis zu 240 externe Interrupts
- 8 bis 256 Prioritätslevel

→ *ISR benötigt 12 Taktzyklen*
*Sieheauch* : **??***S***??**

### 4.3.2 WIC (Wakeup Interrupt Controller)

Für die Umsetzung von Low-Power-Modes.
Dadruch kann 99% der Cortex M3-Prozessoren im Low-Power-Bereich arbeiten.

Ist mit dem NVIC verknüpft und holt den Prozessor aus diesem Modus heraus, um auf einen Interrupt reagieren zu können

### 4.3.3 FPU - (nur Cortex M4!)

Mit der FPU lassen sich IEEE754 Signal Precision Floating-Point Operationen in sehr wenigen Takten ausführen

### 4.3.4 MPU

- ermöglicht Zugriffsregel für den Privilieged Access und User Programm Access zu definieren
- → *Wird eine Zugriffsregel verletzt erfolgt eine Exception − Regelung wo durch der Exceptrion Handler das Problem analysiert und Ausser dem ist es möglich gewisse Bereiche als read − only zu deklarieren*

### 4.3.5 SYSTICK

- 24-Bit Countdown-Timer mit automatischer Relaod-Funktion
- Wird für einen periodischen Interrupt verwendet

Wenn der Zähler den Wert 0x000000, wird dies dem NVIC signalisiert und der Reload-Wert wird aus dem Reload-Register gelesen.

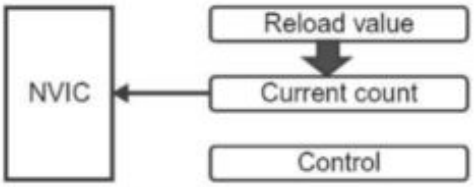


Figure 4-5 SYSTICK - Standard Timer

## 4.4 GNU-Tool-Chain Entwicklungsablauf

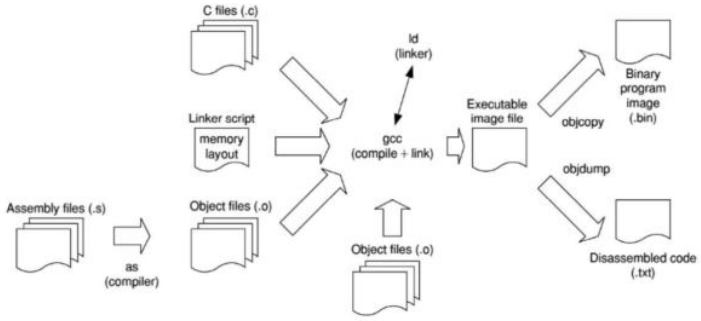



Figure 4-11 GNU Tool-Chain: Devlopment Flow (TI Code Composer Studio) [Yiu3]

| | |
|---|---|
| **APSR** | Application Program Status Register |
| **IPSR** | Interrupt Program Status Register |
| **EPSR** | Execution Program Status Register |

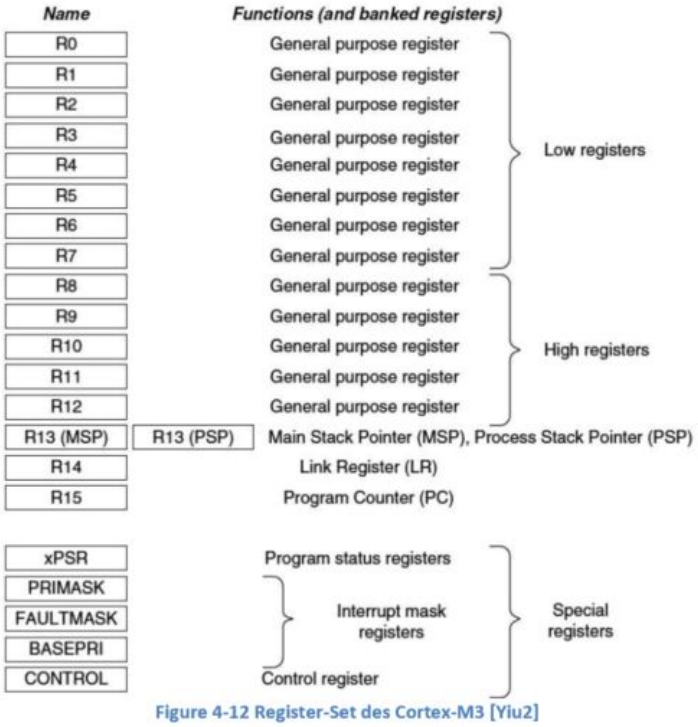



Figure 4-12 Register-Set des Cortex-M3 [Yiu2]

### 4.4.1 SP-zugriffe (Assembler)

```
MRS     r0, APSR          ; Read Flag state into R0
MRS     r0, IPSR          ; Read Exception/Interrupt state
MRS     r0, EPSR          ; Read Execution state
MSR     APSR, r0          ; Write Flag state
```

## 4.5 Programm Status Register

| | |
|---|---|
| **N** | Negativ |
| **Z** | Zero |
| **C** | Carry/borrow |
| **V** | Overflow |
| **Q** | Sticky saturation flag |
| **ICI/IT** | Interrupt-Continauble Instruction(ICE) bits<br>IF-THEN instruction status bit |
| **T** | Thumb state, always 1; Trying to clear this bit will caus a fault exception |
| **Exception number** | Indicates which exception the processor is handling |

### 4.5.1 Q-Flag

This flag is set to 1 if any of the following occurs:

- Saturation of the addition result in a *QADD or QDADD* instruction
- Saturation of the subtraction result in a *QSUB or QDSUB* instruction
- Saturation of the doubling intermediate result in a *QDADD or QDSUB* instruction
- Signed overflow during an *SMLA<x><y> or SMLAW<y>* instruction

The Q flag is sticky in that once it has been set to 1, it is not affected by whether subsequent calculations saturate and/or overflow.
**An example:**
*0x70000000 + 0x70000000* would become *0xE0000000*, but since qadd is saturating, the result is saturated to *0x7FFFFFFF* (the largest positive 32-bit integer) and the Q flag is set.

| | 31 | 30 | 29 | 28 | 27 | 26:25 | 24 | 23:20 | 19:16 | 15:10 | 9 | 8 | 7 | 6 | 5 | 4:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| APSR | N | Z | C | V | Q | | | | | | | | | | | |
| IPSR | | | | | | | | | | | | Exception number | | | | |
| EPSR | | | | | | ICI/IT | T | | | ICI/IT | | | | | | |

| | 31 | 30 | 29 | 28 | 27 | 26:25 | 24 | 23:20 | 19:16 | 15:10 | 9 | 8 | 7 | 6 | 5 | 4:0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| xPSR | N | Z | C | V | Q | ICI/IT | T | | | ICI/IT | | Exception number | | | | |

Figure 4-13 Program Status Register (oben: einzeln / unten: kombiniert)

## 4.6 Stack

- Temporäre Zwischenspeicherung von Daten während der ausführung einer Funktion
- Übergabe von Informationen an Funktionen oder Subroutinen
- Speichern von lokalen Variabeln
- Erhalten von Prozessor-Status und Register-Werten, während Exceptions oder Interrupts ausgeführt werden
- PUSH-POP-Instruktionen werden ausgeführt
- LIFO-Prinzip(Last In, First Out)

### 4.6.1 Main-Stack-Pointer (MSP)

- Standard Stack Pointer nach einem Reset
- Innerhalb von Exception-Interrupt-Handler wird immer der MSP benutzt!

### 4.6.2 Prozessor-Stack-Pointer (PSP)

- Alternativer Stackpointer
- Wird nur im Thread-Mode verwendet

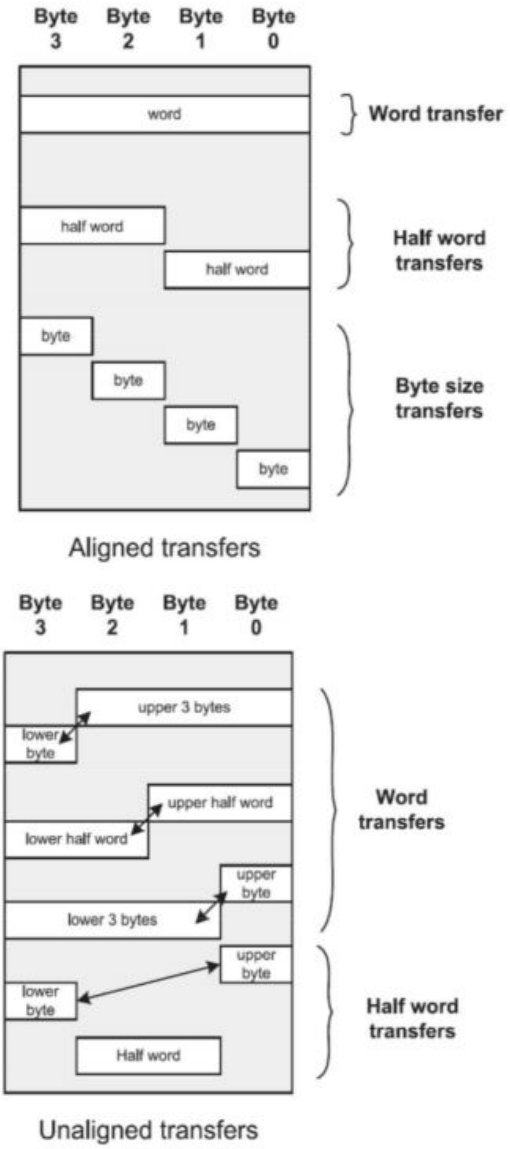
→ *beiembeddedOS – System*

# 5 V5

## 5.1 Data Alignment

### 5.1.1 Aligned-Unaligned Data

| | | |
|---|---|---|
| 1 Byte | 8 Bit | Byte |
| 2 Byte | 16 Bit | Half-Word |
| 4 Byte | 32 Bit | Word |
| 8 Byte | 64 Bit | Double-Word |




### 5.1.2 Bit-Banding

**BitBandAliasAddress = BitBandAliasBase + (MemoryAddres - BitbandRegionBase)* 32 + 4*BitNumber**

$$BNr = [mod_{32}(BBAA - BBAB)] \cdot 2^{-2} \quad \rightarrow mod_{32} = 5 \textit{ Stellen von LSB in binr}$$

$$MA = (BBAA - BBAB) \cdot 2^{-5} + BBRB$$

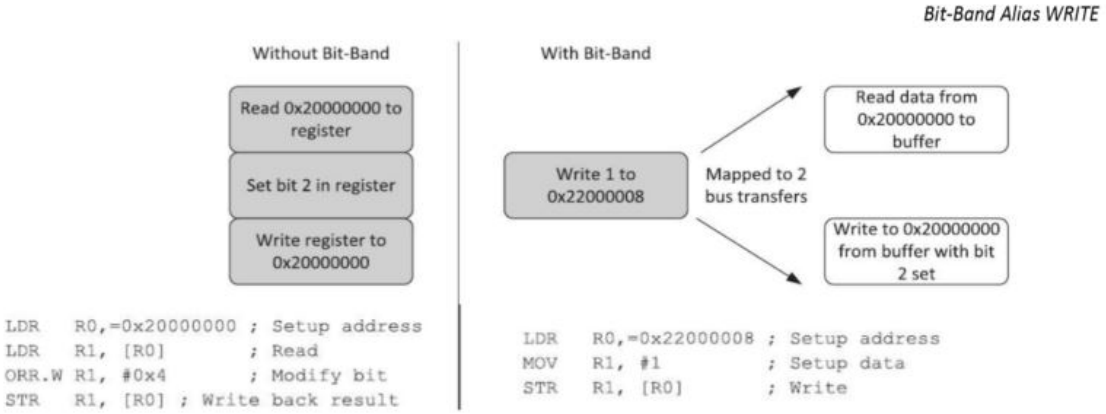



Figure 4-36 Bit-Band Alias WRITE

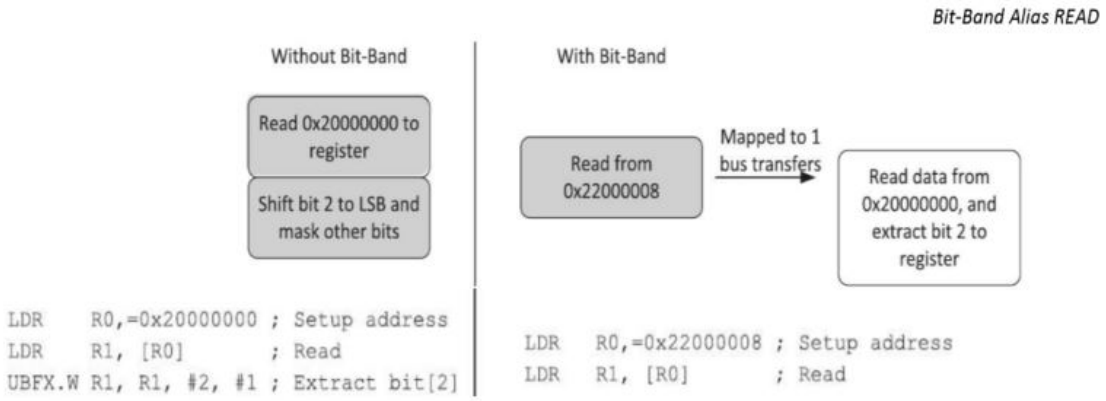



Figure 4-37 Bit-Band Alias READ

# 6 V6

## 6.1 Exceptions and Interrupts

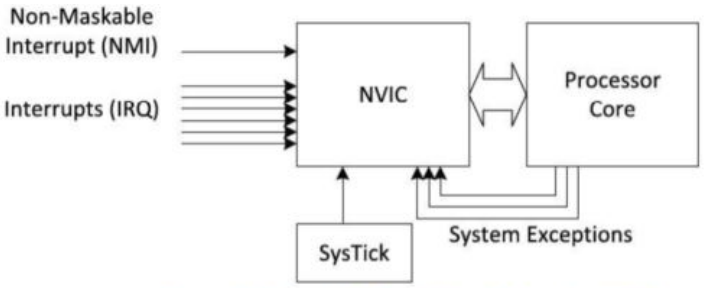


Figure 4-38 Unterschiedliche Quellen für Exception [Yiu3]

Der NVIC verarbeitet bis 240 IRQ und einen NMI
normaler Programmablauf → *Background*
Exception-Handler → *Foreground*

Exception-Handler für Interrupts werden als Interrupt Service Routine (ISR) bezeichnet.

## 6.2 Reset und Reset-Sequenzen

### 6.2.1 Reset

Es gibt 3 Arten von Reset:

**Power-on Reset** Resettet den gesammten $\mu C, auchallePeripherienundDebug - Komponenten$
**System Reset** Resettet nur den Prozessor und die Peripherien, aber nicht die Debug-Komponenten
**Processer Reset** Resettet nur den Prozessor

### 6.2.2 Reset Sequenz

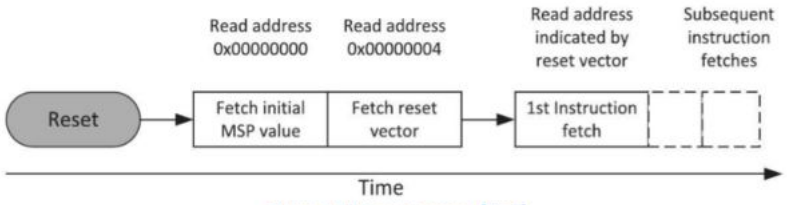


Figure 4-40 Reset-Sequenz [Yiu3]

## 6.3 Spezial-Register

Register um Exceptions ein oder auszuschalten:
→ *PRIMASK, FAULTMASK, BASEPRI*

### 6.3.3 BASEPRI

- Register das bis zu 8 Bits enthalten kann
- definiert eine Prioritätsstufe

- Hohe Stufe = Hohe Priorität
- Wenn das gesetzt wird, werden alle Interrupts mit gleicher oder tieferer Stufe deaktiviert

### 6.3.4 Control-Register

Das Kontroll-Register definiert:

1. Die Auswahl zwischen MSP (Main-SP) und PSP (Process-SP)
2. Die Zugriffsstufe und Thread-Mode
   (Ob Privilegd oder unprivilegd)

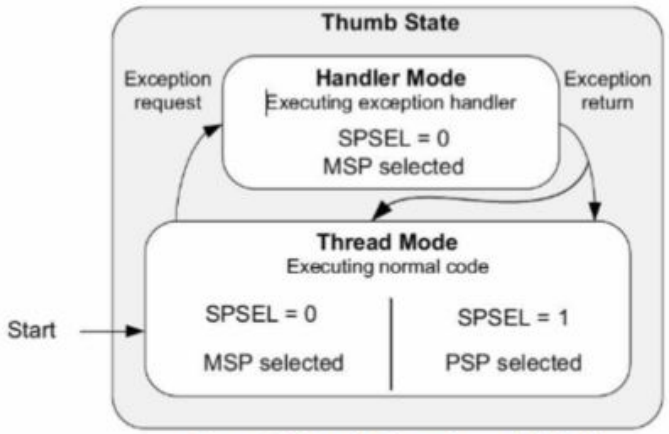


Figure 4-16 Stack Pointer Auswahl [Yiu3]

| | 31:3 | 2 | 1 | 0 |
|---|---|---|---|---|
| CONTROL | | | SPSEL | nPRIV |

Figure 4-15 CONTROL Register beim Cortex-M3

# 7 V7

## 7.1 Cortex M3 Instruction Set

### 7.1.1 Thumb-2 Instruction Set

| | |
|---|---|
| Ziel | -Erhöht die Code-Dichte |
| | -Mehr Leistung |
| Cortex M3 Processor | 1.25 DMIPS / MHz |

## 7.2 Logikstruktur des Cortex-M Prozessor

Sourceoperanden: Rn, Rm
Destinationsoperand: Rd
Ein *Barrel-Shifter* vereinfacht Berechnungen,
da Multiplikationen einfacher realisiert werden können.

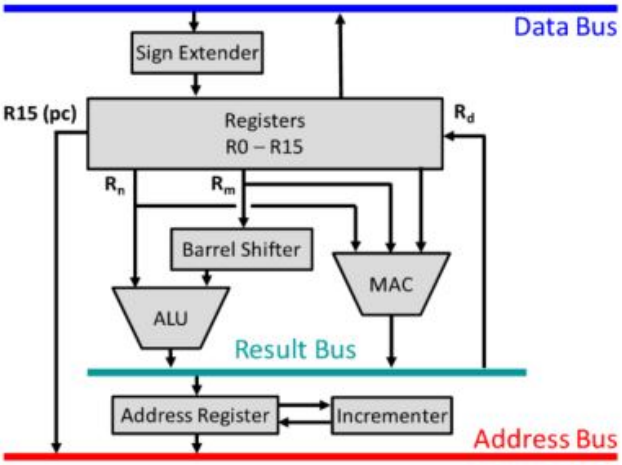


Figure 5-2 Logik-Struktur des Cortex-M Prozessors

## 7.3 Instruction Pipelining

MAC = Memory Access Calculator
Load Store Architektur

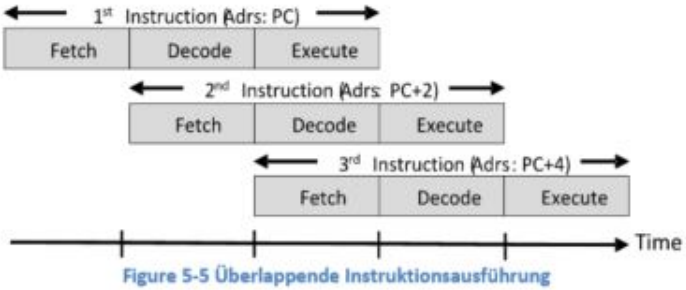


Figure 5-5 Überlappende Instruktionsausführung

## 7.4 Anwendungen

### 7.4.1 Cortex-M0/M0+ / M1

- einfaches I/O Handling

### 7.4.2 Cortex-M3

- Komplexe Datenverarbeitung
- anspruchsvolle Applikationenen

### 7.4.3 Cortex-M4

- DSP-Funktionalität
- Floating Point Support

## 7.5 Assembly-Language Syntax

| Lable | OpCode | Operand | Comment |
|---|---|---|---|
| L1 | ADD | R0,R1,#5 | Replace R0 by sum of R1 and 5 |
| FUNC | MOV | R0,#100 | this sets R0 to value 100 |
| | BX | LR | this is a function return |

| | |
|---|---|
| **Lable** | optional |
| **OpCode** | spezifiziert den Befehl |
| **Operand** | Parameter |
| **Comment** | optionale Beschriebung |

## 7.6 Unified Assembler Language (UAL)

Syntax für ARM und Thumb Instructionen.
Die meisten Instruktionen arbeiten mit Registern
**BSP**

```
MOV   R2,#100   ;R2=100,Direkte Zuweisung
LDR   R2,[R1]   ;R2= den Wert von R1
ADD   R2,R0     ;R2=R2+R0
ADD   R2,R0,R   ;R2=R0+R1
```

### 7.6.1 Register List

```
Norm. Form  reglist   ;R1,R2...Rn
PUSH        LR        ;save LR on stack
POP         LR        ;remove from stack; place in LR
PUSH        R1-R3,LR  ;save R1,R2,R3; return address
POP         R1-R3,PC  ;restore R1,R2,R3 and return
```

## 7.7 Addressing

### 7.7.1 Immediate Adressing

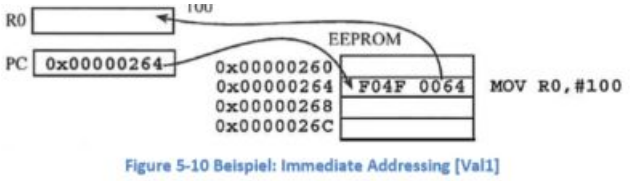


Figure 5-10 Beispiel: Immediate Addressing [Val1]

Der Datenwert ist unmittelbar in der Instruktion erhalten. Daher kein zusätzlicher Speicherzugriff erforderlich
Form: # imm

```
MOV   R0,# 100   ;R0=100, immediate addressing
```

### 7.7.2 Indirect Addressing

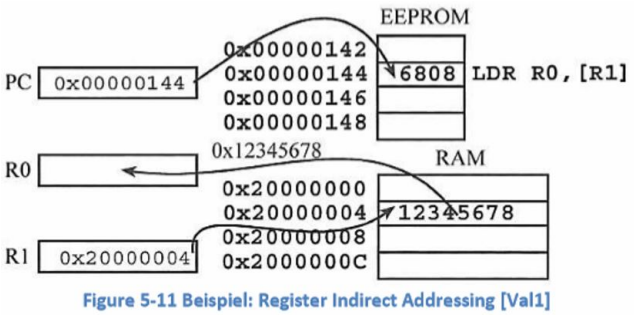


Figure 5-11 Beispiel: Register Indirect Addressing [Val1]

Bei der indirekten Adressierung sind mehrere Speicherzugriffe erforderlich.
Form: [Rn]

```
LDR   R0,[R1]   ;R0=value pointed to by R1
```

Ein Register enthält irgendwie einen Zeiger auf dieses Register
**R1 wird nicht verändert**

### 7.7.3 Register Addressing with Displacment

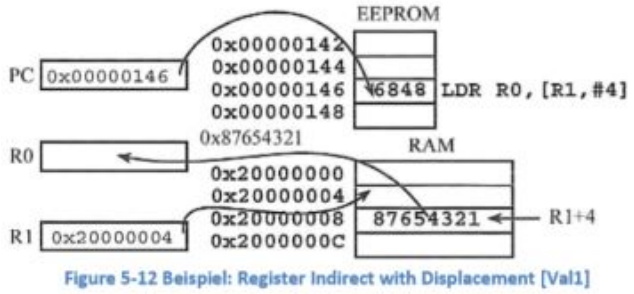


Figure 5-12 Beispiel: Register Indirect with Displacement [Val1]

Dasselbe, nur wird hier dem Wert R0 noch # 4 hinzugefügt
R1 bleibt weiterhin unverändert.
Form: [Rn,# imm]

```
LDR   R0,[R1,# 4]   ;R0=word pointed to by R1+4
```

### 7.7.4 Register Indirect with Index

Form: [Rn,Rm]

```
LDR   R0,[R1,R2]   ;R0= word pointed to by R1+R2
```

### 7.7.5 Register Indirect with shifted Index

Form: [Rn,Rm,LSL # imm]

```
LDR   R0,[R1,R2;LSL #2]   ;R0= word pointed to by R1+4*R2
```

### 7.7.6 Register Indirect with Pre-index

Form: [Rn,# offset]!

```
LDR   R0,[R1,#4]!   ;first R1=R1+4, then R0= word pointed to by R1
```

### 7.7.7 Register Indirect with Post-index

Form: [Rn],# offset

```
LDR   R0,[R1],#4   ;R0= word pointed to by R1, then R1=R1+4
```

### 7.7.8 PC-relativ

PC wird als Pointer verwendet. Form: lable

```
B     Location      ;jump to Location
BL    Subroutine    ;call Subroutine, Rücksprungadresse wird gespeichert
```