

# 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:**

- |                |                |   |
|----------------|----------------|---|
| <b>1. Teil</b> | closed Book    | Theoretische Fragen zum ganzen Prüfungsinhalt                                       |
| <b>2. Teil</b> | 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

Flurin Arquint flurin.arquint@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 F. Arquint

30. November 2019

## Inhaltsverzeichnis

<b>1 V1</b>	<b>4</b>
1.1 Anwendung und Grundlage der uP-Technik . . . . .	4
1.2 Aufbau . . . . .	4
1.2.1 Anwendungen . . . . .	4
1.2.2 Aufbau von uP-basierten Systemen . . . . .	4
1.2.3 Havard vs Von Neumann Architektur . . . . .	4
1.2.4 Programmierung eines uP . . . . .	5
1.2.5 Befehlsformate . . . . .	5
1.3 RISC vs CISC . . . . .	5
1.3.1 RISC-Rechner . . . . .	5
1.3.2 u-Architektur . . . . .	5
1.4 Hardware . . . . .	6
1.4.1 Registersatz . . . . .	6
1.4.2 Hardware-/Software-Schnittstelle . . . . .	6
1.4.3 Taktfrequenz . . . . .	6
1.4.4 Leistungsaufnahme . . . . .	6
1.5 Software . . . . .	6
1.5.1 Ablauf . . . . .	6
<b>2 V2</b>	<b>7</b>
2.1 Compiler-Schritte . . . . .	7
2.2 Busorientierte Systeme . . . . .	7
2.2.1 Speicher . . . . .	7
2.2.2 Architektur eines uP . . . . .	8
2.3 Befehlszyklus . . . . .	8
<b>3 V3</b>	<b>9</b>
3.1 Halbleiter-Speicher . . . . .	9
3.1.1 ROM-Festwertspeicher . . . . .	9
3.1.2 RAM-Speicher-/Lese-Speicher . . . . .	9
3.2 Speicherorganisation . . . . .	9
3.2.1 Little/Big Endian . . . . .	9
3.2.2 I/O - Schnittstelle . . . . .	9
3.2.3 Speicherraumadressierung . . . . .	9
3.3 Busanschluss und Adressverwaltung . . . . .	10
3.3.1 Adresskodierung . . . . .	10
<b>4 V4</b>	<b>11</b>
4.1 Cortex-M Varianten . . . . .	11
4.1.1 Vorteile der Cortex-M-Prozessoren . . . . .	11
4.2 Cortex-M3/M4 . . . . .	12
4.3 System-Komponenten . . . . .	12

4.3.1	NVIC . . . . .	12
4.3.2	WIC (Wakeup Interrupt Controller) . . . . .	12
4.3.3	FPU - (nur Cortex M4!) . . . . .	12
4.3.4	MPU . . . . .	12
4.3.5	SYSTICK . . . . .	12
4.4	GNU-Tool-Chain Entwicklungsablauf . . . . .	13
4.4.1	SP-zugriffe (Assembler) . . . . .	13
4.5	Programm Status Register . . . . .	13
4.5.1	Q-Flag . . . . .	13
4.6	Stack . . . . .	14
4.6.1	Eigenschaften . . . . .	14
4.6.2	Main-Stack-Pointer (MSP) . . . . .	14
4.6.3	Prozessor-Stack-Pointer (PSP) . . . . .	14
4.6.4	PUSH-POP Operationen . . . . .	14
<b>5</b>	<b>V5</b>	<b>15</b>
5.1	Data Alignment . . . . .	15
5.1.1	Aligned Data . . . . .	15
5.1.2	Unaligned Data . . . . .	15
5.2	Bit-Banding . . . . .	15
5.2.1	Berechnung der Adresse . . . . .	15
<b>6</b>	<b>V6</b>	<b>16</b>
6.1	Exceptions and Interrupts . . . . .	16
6.1.1	NVIC . . . . .	16
6.2	Reset und Reset-Sequenzen . . . . .	16
6.2.1	Reset . . . . .	16
6.2.2	Reset Sequenz . . . . .	16
6.3	Fault-Handling . . . . .	17
6.4	Spezial-Register . . . . .	17
6.4.1	PRIMASK . . . . .	17
6.4.2	FAULTMASK . . . . .	17
6.4.3	BASEPRI . . . . .	17
6.4.4	Control-Register . . . . .	17
<b>7</b>	<b>V7 (ausgefallen)</b>	<b>18</b>
<b>8</b>	<b>V8</b>	<b>18</b>
8.1	Cortex M3 Instruction Set . . . . .	18
8.1.1	Thumb-2 Instruction Set . . . . .	18
8.2	Instruction Pipelining . . . . .	18
8.3	Logikstruktur des Cortex-M Prozessor . . . . .	18
8.4	Anwendungen . . . . .	18
8.4.1	Cortex-M0/M0+ / M1 . . . . .	18
8.4.2	Cortex-M3 . . . . .	18
8.4.3	Cortex-M4 . . . . .	18
8.5	Software Development Prozess . . . . .	19
8.6	Assembly-Language Syntax . . . . .	19
8.7	Unified Assembler Language (UAL) . . . . .	19
8.7.1	Register List . . . . .	19
8.8	Addressing . . . . .	19
8.8.1	Immediate Adressing . . . . .	19
8.8.2	Indirect Addressing . . . . .	19
8.8.3	Register Addressing with Displacement . . . . .	20
8.8.4	Register Indirect with Index . . . . .	20
8.8.5	Register Indirect with shifted Index . . . . .	20

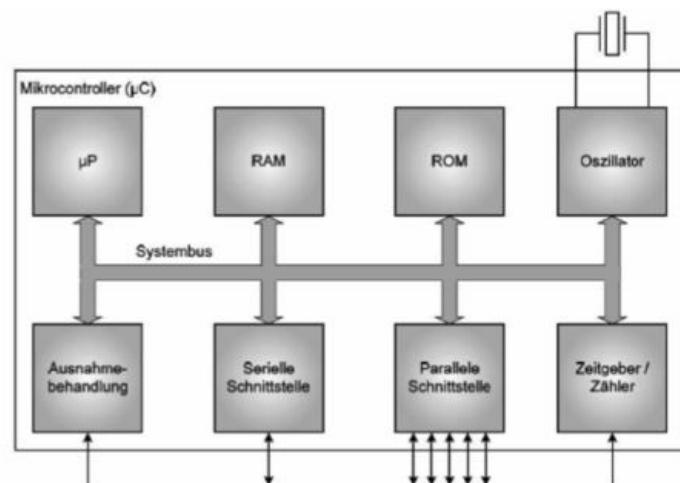
8.8.6	Register Indirect with Pre-index . . . . .	20
8.8.7	Register Indirect with Post-index . . . . .	20
8.8.8	PC-relativ . . . . .	20
8.8.9	Speicher- und I/O-Zugriffe . . . . .	20
<b>9</b>	<b>V9</b>	<b>21</b>
9.1	Interne Datenverschiebung . . . . .	21
9.2	Memory Access Instructions . . . . .	21
9.3	Stack Push und Pop . . . . .	21
9.3.1	Generelle Regeln bei der Verwendung des Stacks . . . . .	21
9.4	Shift and Rotate Instructions . . . . .	22
9.5	Bit-Field Processing Instructions . . . . .	22
9.6	Compare and Test . . . . .	22
9.7	Program Flow Control . . . . .	22
9.7.1	Unconditional Branches . . . . .	22
9.7.2	Function Calls . . . . .	22
9.7.3	Conditional Branches . . . . .	22
<b>10</b>	<b>V10</b>	<b>23</b>
10.1	C/C++ Strukturen Umsetzen . . . . .	23
10.1.1	Entscheidungen . . . . .	23
10.1.2	Analyse von Hochsprachencode und Assembly-Code . . . . .	23
10.1.3	Beispiele . . . . .	23
<b>11</b>	<b>V11</b>	<b>25</b>
11.1	Subroutinen . . . . .	25
11.2	Architecure Producer Call Standart (AAPCS) . . . . .	25
11.2.1	Regeln . . . . .	25
11.2.2	Beispiel . . . . .	25
<b>12</b>	<b>V12</b>	<b>26</b>
12.1	Registersatz . . . . .	26
12.1.1	Akkumulator . . . . .	26
12.1.2	Datenregister . . . . .	26
12.2	Assemblersprache . . . . .	26
<b>13</b>	<b>V13</b>	<b>27</b>
13.1	Allgemeiner Ablauf von Exceptions und Interrupts . . . . .	27
<b>14</b>	<b>V14</b>	<b>28</b>
14.1	Spezielle Eigenschaften des NVIC . . . . .	28
14.1.1	Tail Chaining . . . . .	28
14.1.2	Late arrival . . . . .	28
14.1.3	POP Preemption . . . . .	28

# 1 V1

## 1.1 Anwendung und Grundlage der uP-Technik

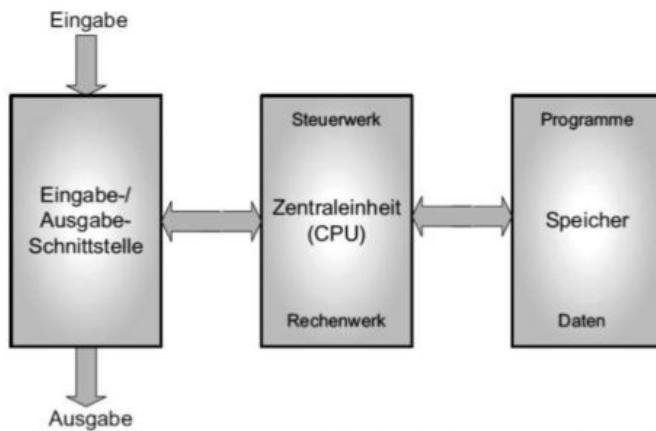
### 1.2 Aufbau

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



### 1.2.1 Anwendungen

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



### 1.2.2 Aufbau von uP-basierten Systemen

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

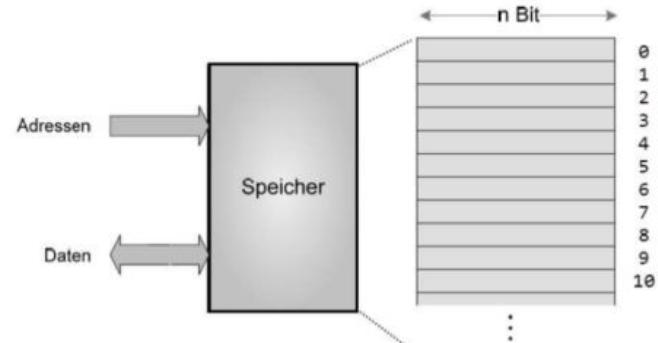
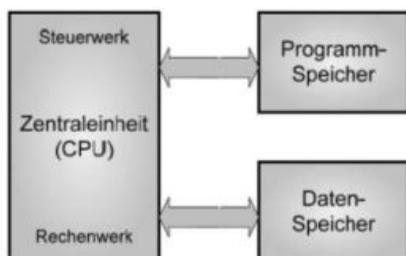


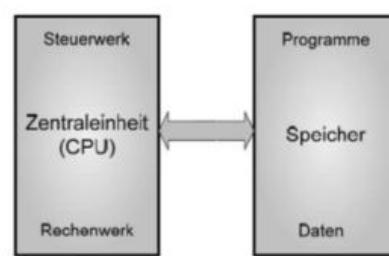
Figure 1-7 Der Speicher ist in eindeutig adressierbaren Speicherplätzen organisiert

### 1.2.3 Harvard vs Von Neumann Architektur

#### Harvard Rechnermodell



#### von Neumann Rechnermodell



## 1.2.4 Programmierung eins uP

Ein  $\mu$  P kann durch individuelle Programmierung auf ganz unterschiedliche Art angepasst werden.

→ entscheidend für die Durchdringung im Markt.

Ein Programm enthält in aufeinanderfolgender Anordnung die Maschinen-Befehle oder -Instruktionen für den  $\mu$  P. Diese Maschiene-Befehle teilen der CPU mit, welche Operationen in welcher Reihenfolge und auf welche Daten angewendet werden sollen.

Die Befehlsfolge des Programms wird innerhalb der CPU vom Steuerwerk gesteuert und schrittweise ausgeführt. Dazu wird der aktuell zur bearbeitende Befehl durch einen Programmzähler (PC) im Speicher adressiert.

Der PC enthält laufend die Adresse der Speicherzelle des jeweiligen Befehls im Speicher.

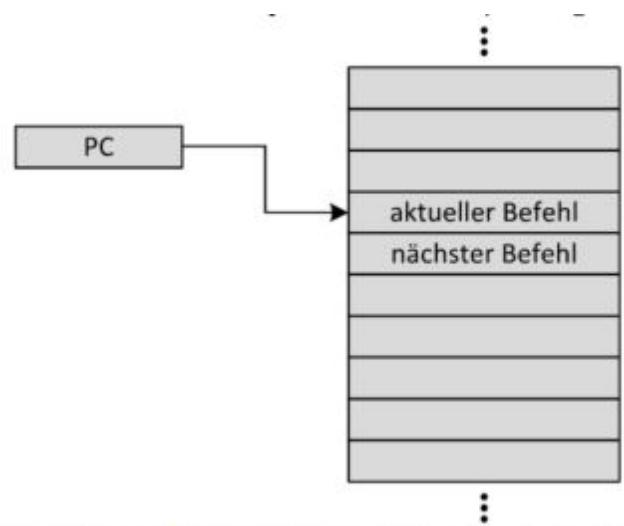


Figure 1-9 Der PC adressiert den aktuellen Maschinenbefehl

## 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  $\mu$  P. Der Maschinencode setzt sich aus einem OpCode und einem oder mehreren Operanden zusammen.

0-Adress-Befehl	OpCode			
1-Adress-Befehl	OpCode	Operand		
2-Adress-Befehl	OpCode	Ergebnis, Operand 1	Operand 2	
3-Adress-Befehl	OpCode	Ergebnis	Operand 1	Operand 2

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 ← Leistungssteigernde Architektur
- Eine grosse semantische Lücke entsteht bei der Übersetzung aus der Hochsprache

#### 1.3.2 u Architektur

Beschreibt die architektonischen Details bei der Implementierung der  $\mu$  P aus Sicht der Programmierer. Dies umfasst die Beschreibung der Zentraleinheit (CPU), des Rechenwerks (ALU) und des Steuerwerks (CU).

## 1.4 Hardware

### 1.4.1 Registersatz

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

### 1.4.2 Hardware- /Software-Schnittstelle



Figure 1-11 Vereinfachte HW/SW-Hierarchieebenen

### 1.4.3 Taktfrequenz

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

$$f_{Takt} = \frac{1}{T_{takt}}$$

$\uparrow$  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.

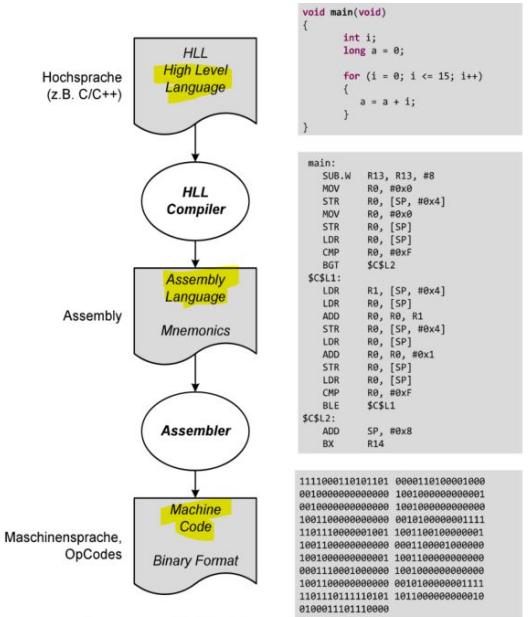


Figure 1-13 Compiler/Assemblier-Workflow am Beispiel Cortex-M

## 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.

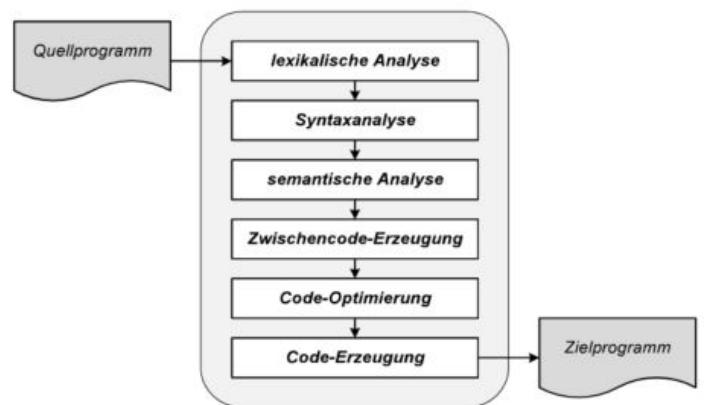


Figure 1-14 Transformation eines Programms in Maschinencode

### 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

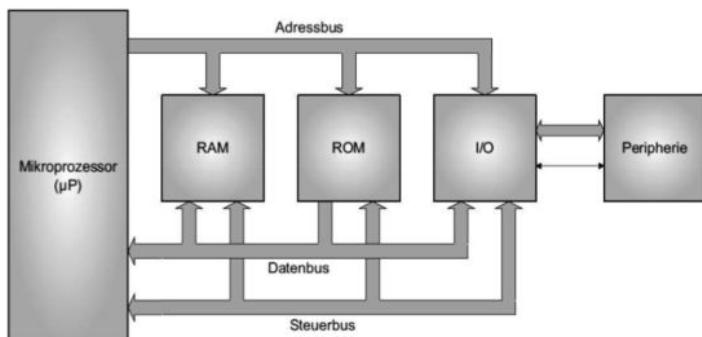


Figure 1-15 Einfaches µP-System mit Speicher und I/O-Schnittstellen [Neu07]

## 2.2.2 Architektur eines uP

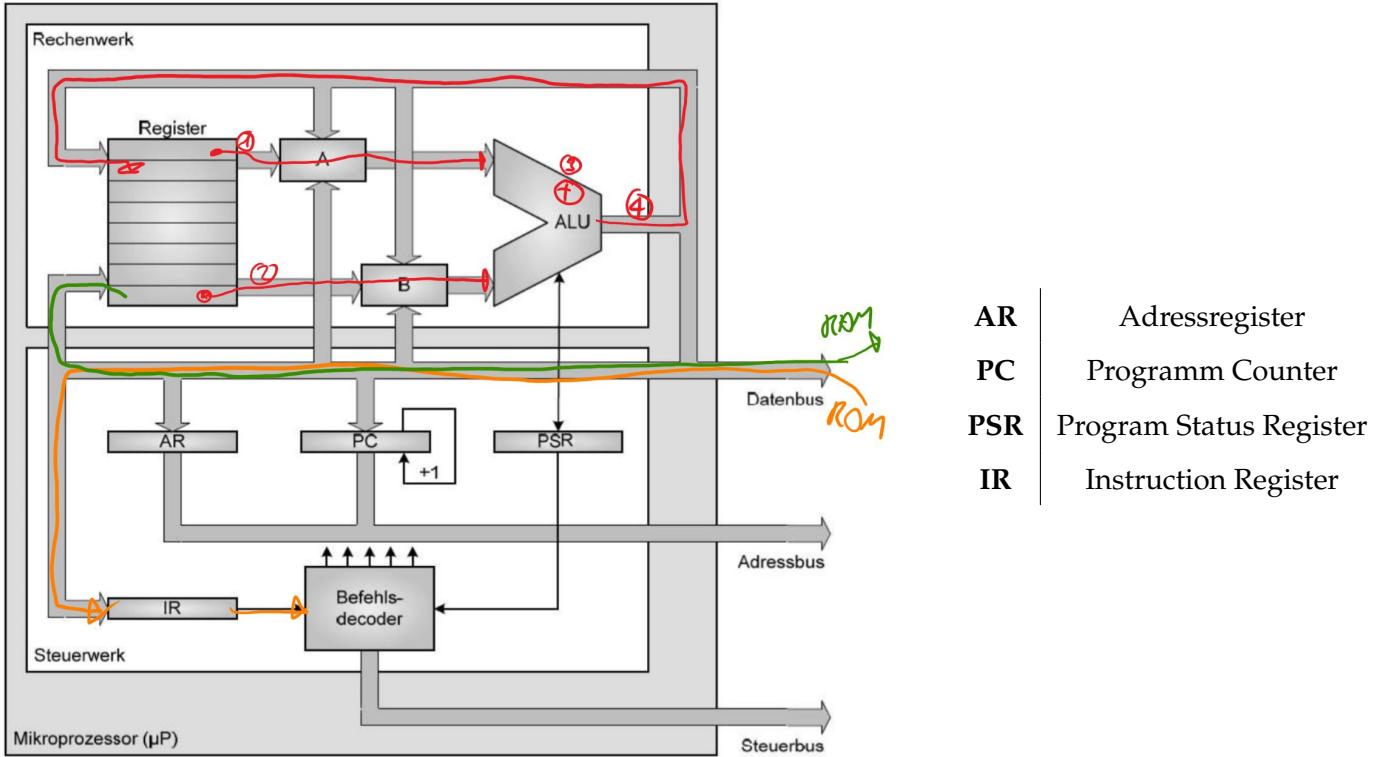


Figure 1-20 Architektur eines einfachen Mikroprozessors [Neu07]

Flags, welche sich unter anderem im PSR befinden

<b>N</b>	<b>Negative</b>	Das von der ALU berechnete Ergebniss ist negativ
<b>Z</b>	<b>Zero</b>	Das von der ALU berechnete Ergenis ist gleich 0
<b>C</b>	<b>Carry</b>	Die Berechnung der ALU hat zu einem Übertrag geführt
<b>V</b>	<b>Overflow</b>	Die Berechnung der ALU hat zu einem Overflow geführt

## 2.3 Befehlszyklus

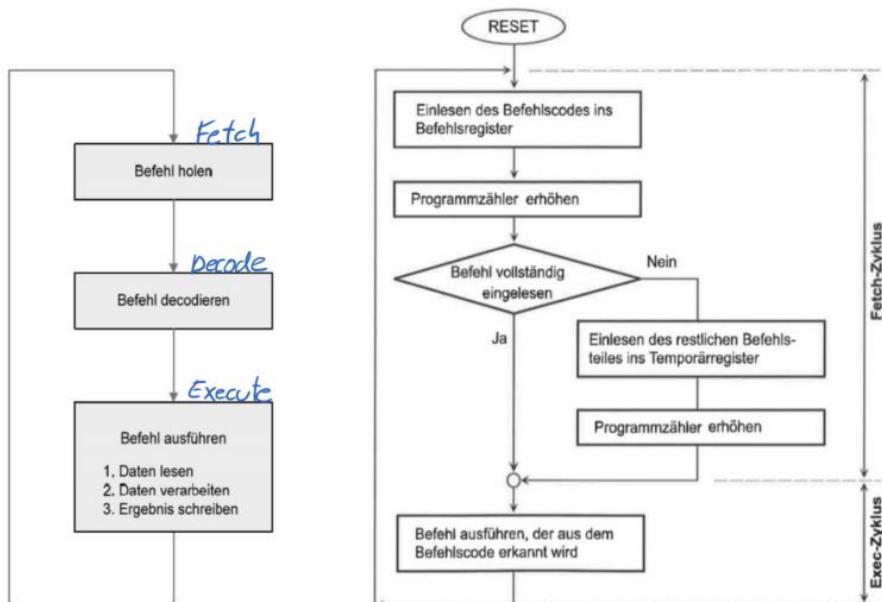


Figure 1-21 Befehlszyklus - Infinite Loop [Neu07, TI1\_05]

### 3 V3

#### 3.1 Halbleiter Speicher

##### Zentraler Speicher

- direkt am Bussystem angeschlossen

##### Peripherer Speicher

- über I/O-Schnittstelle angeschlossen

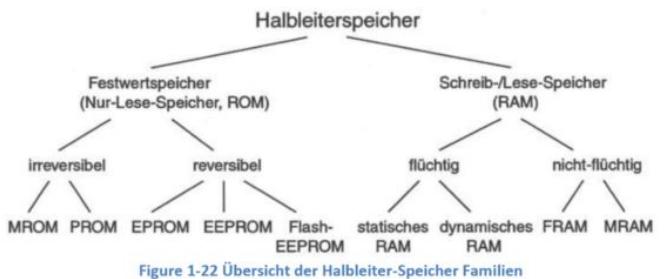


Figure 1-22 Übersicht der Halbleiter-Speicher Familien

#### 3.1.1 ROM-Festwertspeicher

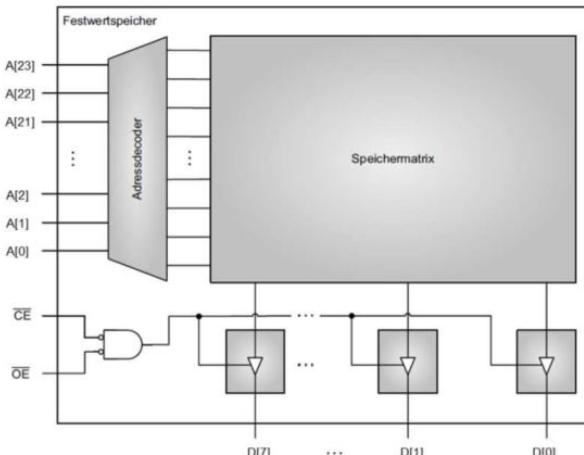


Figure 1-23 Grundaufbau ROM Festwertspeicher [Neu07]

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

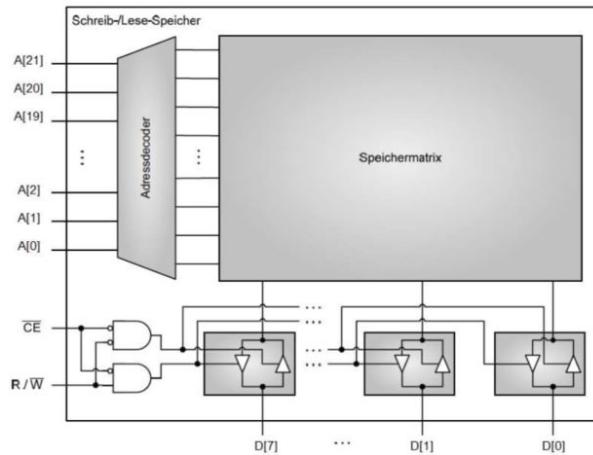


Figure 1-24 Grundaufbau RAM-Speicher mit kombinierter R/W-Leitung

### 3.2 Speicherorganisation

#### 3.2.1 Little/Big Endian

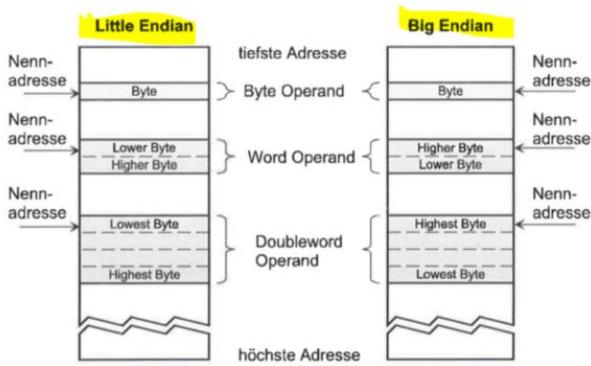
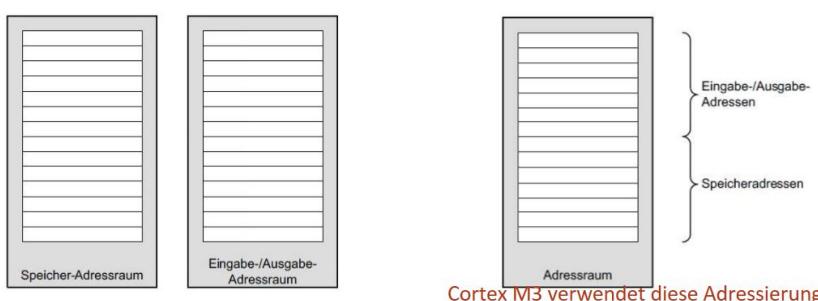


Figure 1-43 Speicherorganisation: Little Endian und Big Endian [TI1\_05]

Der ARM Cortex verwendet standardmäßig **Little Endian** für die Speicherorganisation.

#### 3.2.3 Specherraumadressierung



#### Isolierte Adressierung

- Gesamter Adressraum steht verschiedener Blöcken zur Verfügung
- Blöcke werden mit einem Steuersignal ausgewählt

#### Memory-Mapped I/O (Cortex M3)

- Verschiedene Blöcke werden in einen Adressraum eingebettet
- Benötigt eine Adresskodierung

### 3.3 Busanschluss und Adressverwaltung

Eine Adressverwaltung hat verschiedene Aufgaben zu erfüllen:

- Jede Adresse spricht nur einen einzigen Speicher- oder I/O-Baustein an.
- Adressraum möglichst gut ausnützen
- Adressräume von Speicherbausteinen müssen lückenlos aufeinander folgen.
- Jeder interne Speicherplatz bzw. jedes Register erscheint unter einer eigenen Adresse im Systemadressraum.

#### 3.3.1 Adresskodierung

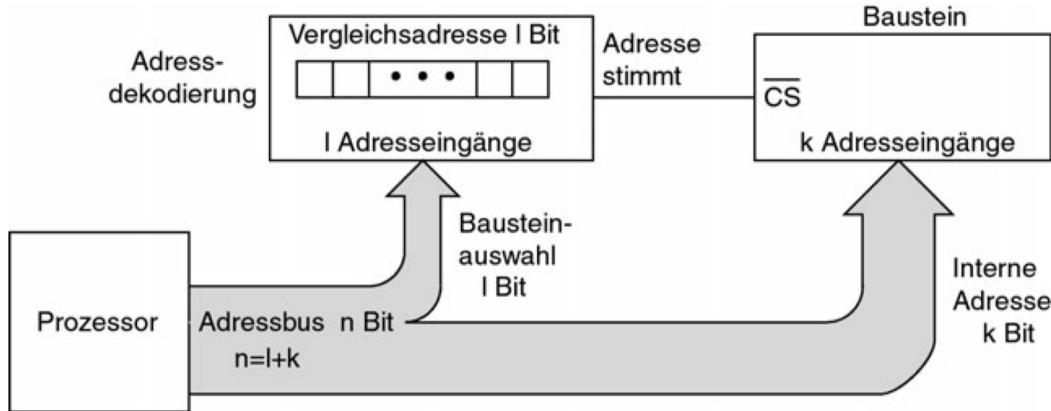


Figure 1-30 Grundprinzip der Adressdekodierung [Wüst]

Der Kernpunkt der Adresskodierung ist die Teilung des Adressbuses. Dabei werden die  $k$  niedrigsten Adressleitungen direkt an die Adresseingänge des Bausteines geführt und dienen zur Auswahl des gewünschten internen Speicherplatzes oder Registers. Die nächstfolgenden  $l$  Adressleitungen werden zur Adresskodierung auf einen Adressdekoder geführt.

## 4 V4

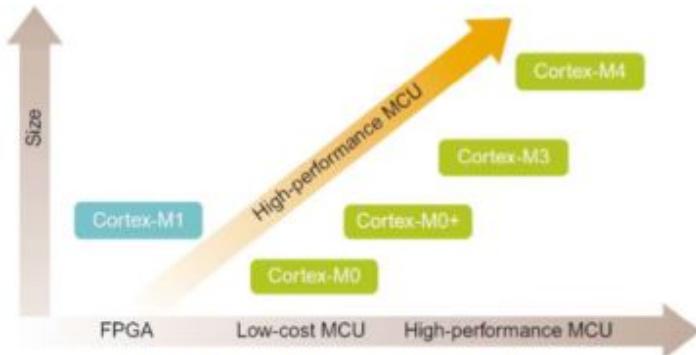
### 4.1 Cortex M Varianten

#### Cortex M0 und M0+

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

#### Cortex M3

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

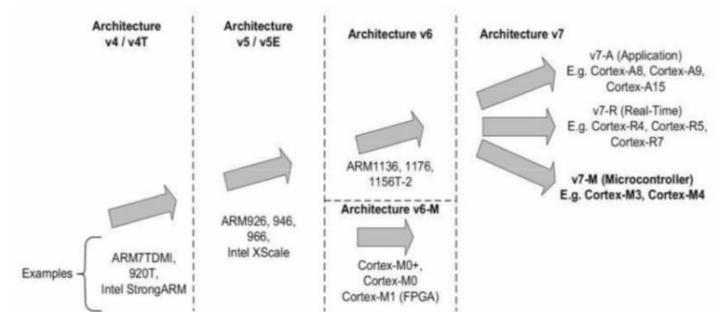


#### Cortex M1

- als Softcore Implementiert
- Vergleichbar mit Cortex-M0

#### Cortex M4

- vergleichbar mit M3 jedoch mit:
  - Digital Signal Processing (DSP)
  - Floating Point Unit (FPU)



#### Cortex-A

- HighEnd Anwendungen und Betriebssysteme
- hohe Rechenleistung
- Cache Memory

#### Cortex-R

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

#### Cortex-M

- Speziell für  $\mu$  C-Markt
- Low Cost, Low Energy
- System on Chip (SOC)

### 4.1.1 Vorteile der Cortex-M-Prozessoren

- Low Power  
 $< 200\mu\text{A} / \text{MHz}$
- Performance  
 $> 1.25 \text{ DMIPS} / \text{MHz}$
- Energy Efficiency  
low Power, high performance
- Code Density  
Thumb 2 Befehlssatz

- Interrupts  
240 Interrupts
- Easy of Use, C Friendly
- Scalability
- Debug Features
- Software portability and Reuseability
- OS Support
- Choices (Derivers, Tools, OS,...)

## 4.2 Cortex-M3/M4

- Harvard Architecture
  - Zugriffe 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

Siehe auch: Spezielle Eigenschaften des NVIC S28

### 4.3.2 WIC (Wakeup Interrupt Controller)

Für die Umsetzung von Low-Power-Modes.

Dadurch 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

### 4.3.5 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.

### 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 Priviliegied Access und User Programm Access zu definieren
- → Wird eine Zugriffsrege verletzt erfolgt eine Exception-Regelung wodurch der Exceptrion Handler das Problem analysiert und ggf. beheben kann
- → Ausserdem ist es möglich gewisse Bereiche als read-only zu deklarieren

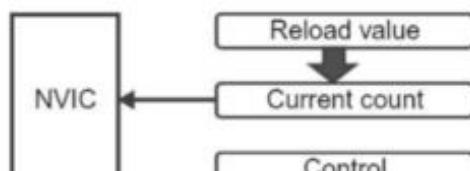


Figure 4-5 SYSTICK - Standard Timer

## 4.4 GNU-Tool-Chain Entwicklungsablauf

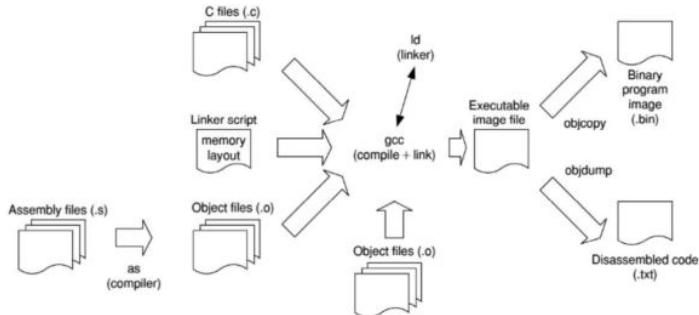


Figure 4-11 GNU Tool-Chain: Development Flow [TI Code Composer Studio] [Yiu3]

<b>APSR</b>	Application Program Status Register
<b>IPSR</b>	Interrupt Program Status Register
<b>EPSR</b>	Execution Program Status Register

### 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
```

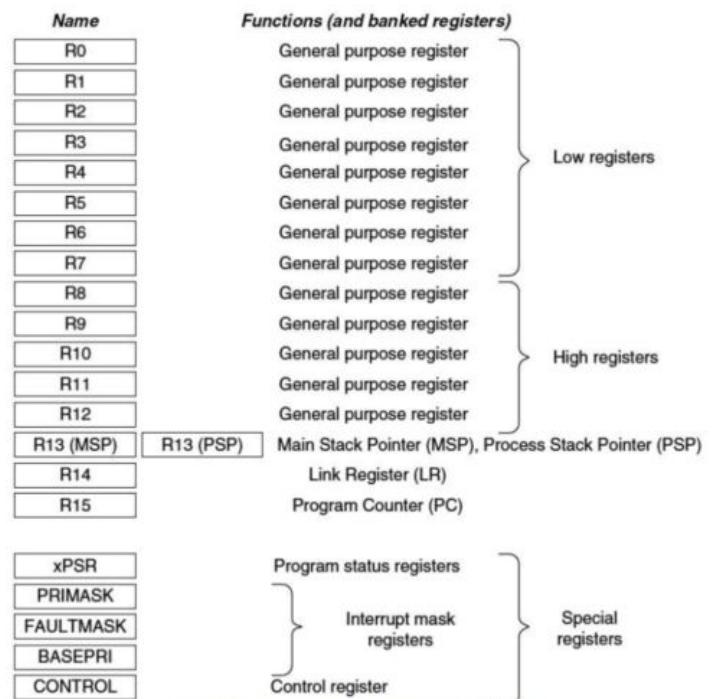


Figure 4-12 Register-Set des Cortex-M3 [Yiu2]

## 4.5 Programm Status Register

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



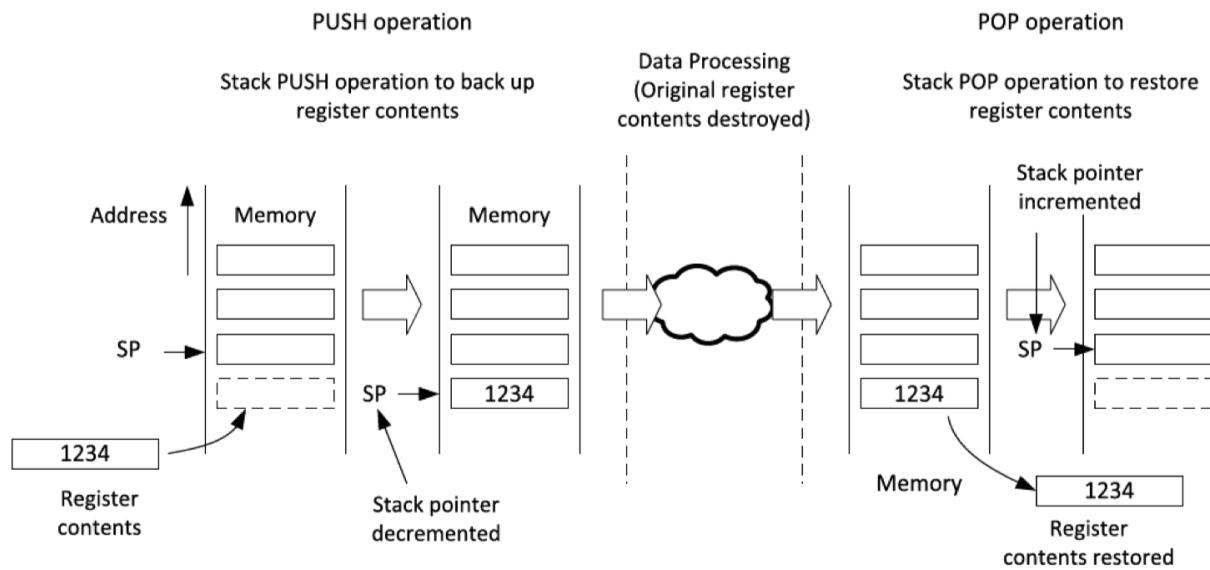
Figure 4-13 Program Status Register (oben: einzeln / unten: kombiniert)

## 4.6 Stack

### 4.6.1 Eigenschaften

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

### 4.6.4 PUSH-POP Operationen



Der Stack wird von der höchsten Adresse aus gefüllt, das heisst das bei einer PUSH-Operation der Stack-Pointer dekrementiert wird. Es gibt auch kombinierte PUSH/POP Operationen. Die wohl wichtigste ist die POP/Return Operation, bei welcher das Link-Register (LR) beim PUSH auf den Stack gelegt wird und bei der POP-Operation direkt in den Programm-Counter (PC) geschrieben wird. Da alle Register in der Registerbank 32-Bit breit sind, wird der Stack-Pointer immer eine *4-Byte aligned Adresse* enthalten.

## 5 V5

### 5.1 Data Alignment

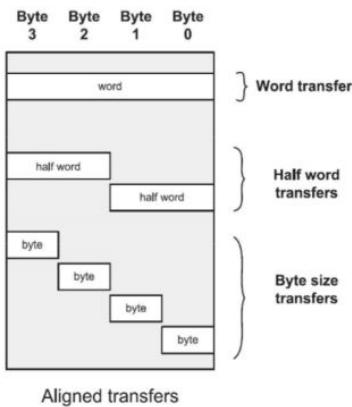
In der Cortex-M Nomenklatur werden direkt aufeinanderfolgende Bytes wie folgt bezeichnet:

<b>2 Byte</b>	16-Bit <b>Half-Word</b>	Halbwort
<b>4 Byte</b>	32-Bit <b>Word</b>	Wort
<b>8 Byte</b>	64-Bit <b>Double-Word</b>	Doppelwort

Pro Operation (read oder write) können auf dem internen 32-Bit breiten Datenbuses 4 Byte gemeinsam übertragen werden. Sehr oft sind aber kleinere Dateneinheiten (Half-Word oder Byte) zu übertragen. Zudem kommt es recht häufig vor, dass die Daten nicht auf die 32-Bit Wortgrenze ausgerichtet sind, also als **misaligned data** im Speicher abgelegt sind. Misaligned Half-Word, Word oder Double-Word Operanden im Speicher benötigen somit mehr als einen *Memory-Cycle*. Wenn man zum Beispiel einen Operanden von der Adresse 0x102 lesen will, wird im ersten *Memory-Cycle* ein Word von der Adresse 0x100 gelesen und dann im zweiten *Memory-Cycle* wird ein Word von der Adresse 0x104 gelesen. Der Operand wird dann aus den beiden gelesenen Word zusammengesetzt.

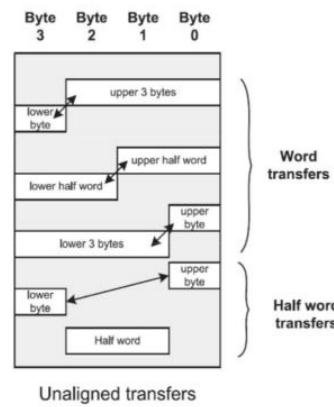
#### 5.1.1 Aligned Data

Operanden können mit einem *Memory-Cycle* ausgesehen/geschrieben werden.



#### 5.1.2 Unaligned Data

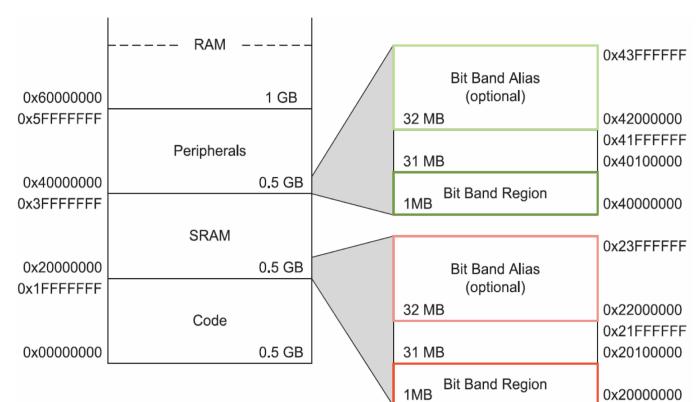
Operanden müssen aus mehreren *Memory-Cycle* zusammengesetzt werden.



## 5.2 Bit-Banding

Bit-Banding verwendet zwei verschiedene Regionen im Adressraum, um effektiv auf dieselben physischen Daten zu verweisen:

- Im *Bit-Band Region* Speicherbereich referenziert jede Adresse genau ein einzelnes Byte an Daten, bestehend aus 8 Bits
- Im *Bit-Band Alias* Speicherbereich wird mit jeder Adresse nur ein einzelnes Bit in der primären *Bit-Band Region* angesprochen.



### 5.2.1 Berechnung der Adresse

$$\text{BBAA} = \text{BBAB} + (\text{MA} - \text{BBRB}) \cdot 2^5 + 4 \cdot \text{BNr}$$

$$\text{BNr} = [0x0000001F \& (\text{BBAA} - \text{BBAB})] \cdot 2^{-2}$$

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

Bei der zurückrechnung ist **MA** die Byte-Adresse und **BNr** immer zwischen 0-7. Wenn nun die aligned Half-Word oder aligned Word Adresse gefragt ist muss die Adresse und die Bitnummer noch dementsprechend angepasst werden.

Legende:

BBAA	Bit-Band Alias Address
BBAB	Bit-Band Alias Base
MA	Memory-Address
BBRB	Bit-Band Region Base
BNr	Bit-Number

## 6 V6

### 6.1 Exceptions and Interrupts

Exceptions sind Ereignisse, die den sequenziellen Programmablauf verändern. Der Prozessor unterbricht den normal laufenden Programmablauf (Background) und führt einen Exception-Handler (Foreground) aus. Exceptions werden vom NVIC (nested vectored interrupt controller) verarbeitet. Der NVIC kann eine Reihe von *Interrupt Request (IRQs)* und einen *Non-Maskable Interrupt (NMI)* verarbeiten, wobei der *NMI* beispielsweise von einem Watchdog-Timer aktiviert werden kann.

Der Prozessor selbst ist auch eine Quelle von Exceptions.

Jede Exception-Quelle hat eine zugehörige Exception-Number:

- Exception-Number 1-15 gelten als *System-Exceptions*
- Exception-Number 16-255 sind *Interrupts*

Der NVIC kann bis zu 240 IRQs verarbeiten, in der Praxis sind es aber oft weniger.

Exception-Handler für Interrupts werden als Interrupt Service Routine (ISR) bezeichnet, wobei die Interrupt-Latenzzeit (*Interrupt Latency*) gerade mal 12 Clockzyklen beträgt.

#### 6.1.1 NVIC

Jeder Interrupt kann individuell aktiviert/deaktiviert werden, außerdem kann sein *Pending State* durch die Software gesetzt/gelöscht werden. Den Exceptions können *Priority-Level* zugeordnet werden und somit ausgelöste ISRs mit tieferer Priorität durch einen Interrupt höherer Priorität unterbrochen werden.

Es gibt 8 verschiedene Prioritätsstufen, wobei diese noch in Prioritätsgruppen aufgeteilt werden können. Daraus entstehen dann *Preemptive levels* und *Subpriority levels*. Wenn nun zum Beispiel zwei Interrupts das gleiche *Preemptive level* besitzen, kann mit dem *Subpriority level* bestimmt werden, welcher Interrupt beim gleichzeitigen Auftreten zuerst ausgeführt wird. Ein Interrupt mit einer höheren *Subpriority* kann jedoch keinen einer tiefen *Subpriority* unterbrechen, wenn sie das gleiche *Preemptive level* besitzen.

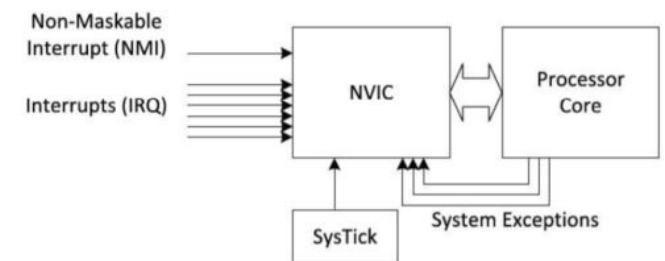


Figure 4-38 Unterschiedliche Quellen für Exception [Yiu3]

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Preemptive priority	Sub prio.						not used

Priority Value

## 6.2 Reset und Reset-Sequenzen

### 6.2.1 Reset

Es gibt 3 Arten von Reset:

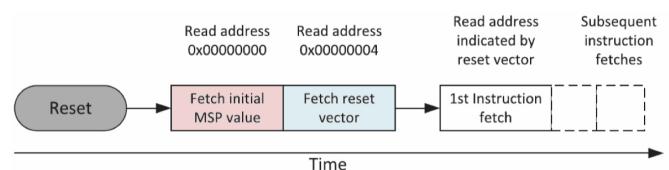
**Power-on Reset** Resettet den gesamten  $\mu$  C, auch alle Peripherien und Debug-Komponenten

**System Reset** Resettet nur den Prozessor und die Peripherien, aber nicht die Debug-Komponenten

**Processor Reset** Resettet nur den Prozessor

### 6.2.2 Reset Sequenz

Nach einem Reset und bevor der Cortex-M Prozessor mit der eigentlichen Programmausführung startet, liest die CPU die ersten beiden 32-Bit Word aus dem Speicher. Am Anfang in der Vektor-Tabelle steht der **Initial Main-Stack-Pointer (MSP)** gefolgt vom **Initial Program Counter (PC)**. Das Setup des MSP ist notwendig, um von Beginn weg einen gültigen Stack zu haben.



## 6.3 Fault-Handling

Der Fault-Exception Mechanismus erlaubt eine schnelle Reaktion auf Systemfehler und gibt der Software die Möglichkeit, Notfallszenarien einzuleiten. Standardmäßig sind die Exceptions *Bus Fault*, *Usage Fault* und *Memory Management Fault* deaktiviert, stattdessen triggern alle drei Faults die *Hard Fault Exception*.

## 6.4 Spezial-Register

Register um Exceptions ein oder auszuschalten:

→ **PRIMASK,FAULTMASK,BASEPRI**

### 6.4.1 PRIMASK

- 1-bit Register
- Wenn das aktiv ist, werden NMI-Interrupts erlaubt
  - alle anderen Interrupts werden überdeckt
  - Default-Wert = 0, also deaktiviert

### 6.4.3 BASEPRI

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

### 6.4.4 Control-Register

Das Kontroll-Register definiert:

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

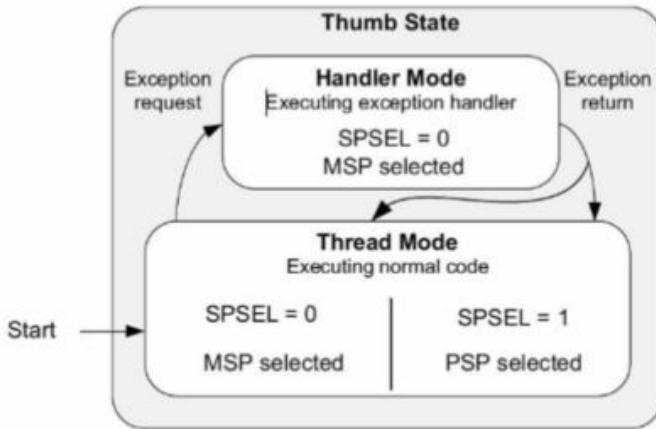
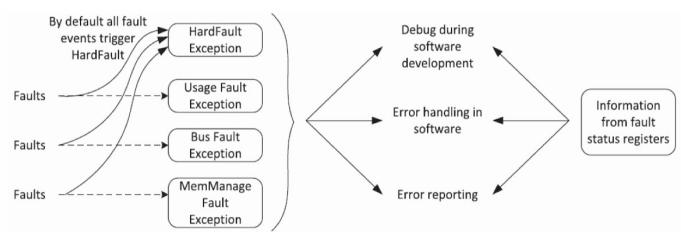


Figure 4-16 Stack Pointer Auswahl [Yiu3]



### 6.4.2 FAULTMASK

- 1-bit Register
- Wenn das aktiv ist, werden nur noch NMI-Interrupts akzeptiert.  
Alle anderen Interrupts und Exception-Handlings werden deaktiviert  
→ Default-Wert = 0

CONTROL	31:3	2	1	0
		SPSEL	nPRIV	

Figure 4-15 CONTROL Register beim Cortex-M3

## 7 V7 (ausgefallen)

## 8 V8

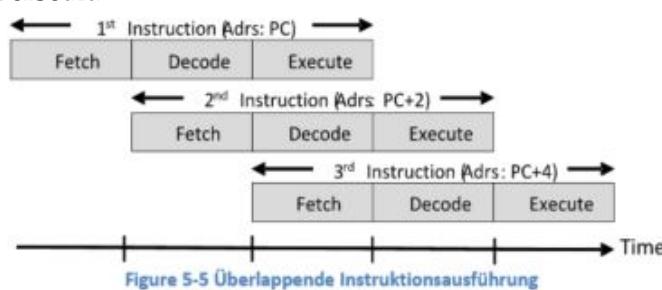
### 8.1 Cortex M3 Instruction Set

Ziel: -Erhöht die Code-Dichte  
-Mehr Leistung

Cortex M3 Processor: 1.25 DMIPS / MHz

### 8.2 Instruction Pipelining

Mit Pipelining bearbeitet der Cortex-M Prozessor zur gleichen Zeit drei Befehle - jeweils um einen Takt versetzt.



Fetch: 16-Bit Instruktion aus Programmspeicher laden

Decode: Vorangehende Instruktion decodieren

Execute: Nochmals vorangehende Instruktion ausführen

Bei einem Sprung im Programmcode muss eine Pipeline komplett geleert werden, da die sequentielle Reihenfolge je nach Konstellation nicht korrekt ist.

### 8.4 Anwendungen

#### 8.4.1 Cortex-M0/M0+ / M1

- einfaches I/O Handling

#### 8.4.2 Cortex-M3

- Komplexe Datenverarbeitung
- anspruchsvolle Applikationen

#### 8.4.3 Cortex-M4

- DSP-Funktionalität
- Floating Point Support

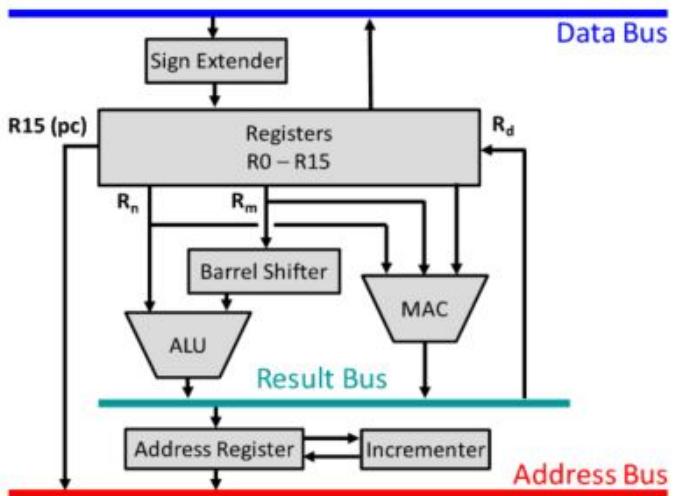
### 8.3 Logikstruktur des Cortex-M Prozessor

Sourceoperanden: Rn, Rm

Destinationsoperand: Rd

MAC: Memory Access Calculator

Ein Barrel-Shifter vereinfacht Berechnungen, da Multiplikationen einfacher realisiert werden können.

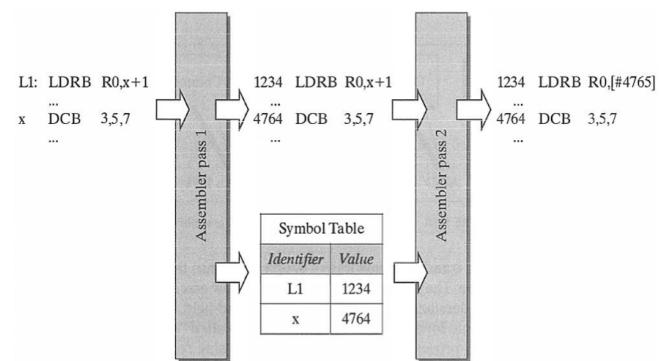


Das 32-Bit breite Programm-Status-Register (xPSR) enthält die Flags. Das xPSR gliedert sich in drei verschiedenen Darstellungen, je nach aktuellem Prozessorstatus als: *Application-Program Register (APSR)*, *Interrupt-Program Status Register (IPSR)* oder *Execution-Program Status Register (EPSR)*.

Die wichtigsten Flags (N, Z, C, V) sind die vier höchsten Bits des APSR, welche die Basis für bedingte Verzweigungen darstellen.

## 8.5 Software Development Prozess

Die Umsetzung von Hochsprachen Source-Code in Assembly-Language und weiter in den binären OpCode des jeweiligen Prozessors ist ein komplexer, mehrstufiger Prozess. Im ersten Durchgang baut der Assembler eine Symboltabelle auf, welche Informationen über sogenannte *programmer-defined Identifiers* enthält (z.B. Adressen von Sprungmarken, Subroutinen, Variablen, I/O-Port Registern, usw.). Während des zweiten Durchgangs benutzt der Assembler diese Informationen, um die einzelnen Assembler-Instruktionen zu vervollständigen. Erst in einem weiteren Teilschritt der Assemblierung werden die binären OpCode ermittelt.



## 8.6 Assembly-Language Syntax

Label	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

Label	optional
OpCode	spezifiziert den Befehl
Operand	Parameter
Comment	optionale Beschreibung

## 8.7 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

### 8.7.1 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

## 8.8 Addressing

### 8.8.1 Immediate Addressing

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

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

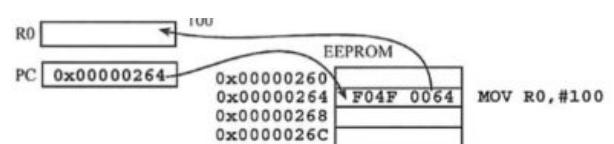


Figure 5-10 Beispiel: Immediate Addressing [Val1]

### 8.8.2 Indirect Addressing

Bei der *Indirect Addressing Modes* sind die Daten im Memory. Ein Register enthält irgendwie einen Zeiger auf diese Daten. Nach der *Fetch-Phase*, bei welcher die Instruktion aus dem Programmspeicher gelesen wird, sind noch einer oder mehrere Speicherzugriffe erforderlich um die Daten zu lesen oder zu schreiben.

Form: [Rn]

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

R1 wird nicht verändert

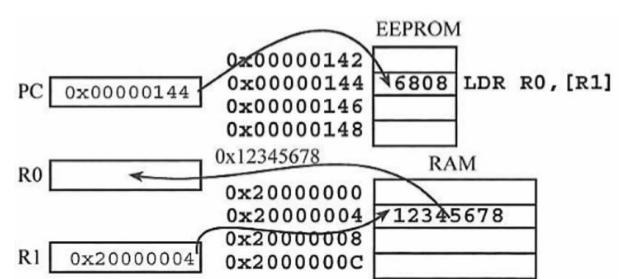


Figure 5-11 Beispiel: Register Indirect Addressing [Val1]

### 8.8.3 Register Addressing with Displacement

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

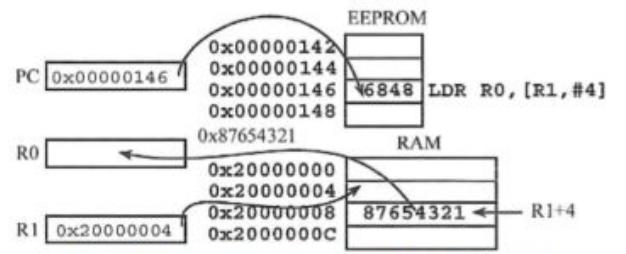


Figure 5-12 Beispiel: Register Indirect with Displacement [Val1]

### 8.8.4 Register Indirect with Index

Form: [Rn,Rm]

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

### 8.8.5 Register Indirect with shifted Index

Form: [Rn,Rm,LSL # imm]

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

### 8.8.6 Register Indirect with Pre-index

Form: [Rn,# offset]!

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

### 8.8.7 Register Indirect with Post-index

Form: [Rn],# offset

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

### 8.8.8 PC-relativ

PC wird als Pointer verwendet. Form: lable

B Location ;jump to Location

BL Subroutine ;call Subroutine, Rücksprungadresse wird gespeichert

### 8.8.9 Speicher- und I/O-Zugriffe

Es benötigt immer zwei Instruktionen um auf Daten im RAM oder I/O zuzugreifen. → PC-Relative Addressierung wird verwendet

1. Erstellt Zeiger auf das Objekt
2. Greift über den Zeiger Indirekt auf den Speicher zu

LDR R1,Count ;R1 points to variable Count  
LDR R0,[R1] ;R0= value pointed to by R1

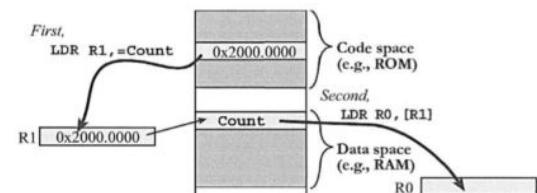


Figure 5-13 Indexed Addressing über ein Register für den Zugriff auf eine RAM-Speicherstelle [Val1]

## 9 V9

### 9.1 Interne Datenverschiebung

Für die Verschiebung von Daten von Register zu Register oder der Beschreibung eines Registers mit einer Konstanten kann die Operation **MOV** verwendet werden. Diese folgend angewendet werden:

Instructions for Transferring Data within the Processor			
Instruction	Dest	Source	Operation
MOV	R4,	R0	;Copy value from R0 to R4
MOVS	R4,	R0	;Copy value from R0 to R4 w. APSR (flags) update
MRS	R7,	PRIMASK	;Copy value of PRIMASK (special register) to R7
MSR	CONTROL,	R2	;Copy value of R2 into CONTROL (special register)
MOV	R3,	#0x34	;Set R3 value to 0x34
MOVS	R3,	#0x34	;Set R3 value to 0x34 with APSR update
MOVW	R6,	#0x1234	;Set R6 to a 16-bit constant 0x1234
MOVN	R6,	#0x8765	;Set the upper 16-bit of R6 to 0x8765
MVN	R3,	R7	;Move negative value of R7 into R3

### 9.2 Memory Access Instructions

Der Cortex-M3 verfügt über viele verschiedene Instruktionen für den Speicherzugriff. Dies wegen den verschiedenen Adressierungsarten und Datengrößen. Für normale Datentransfers, sind folgenden Instruktionen vorhanden:

Memory Access Instructions for Various Data Sizes		
Data Type	LOAD (Read from Memory)	STORE (Write to Memory)
8-bit unsigned	LDRB	STRB
8-bit signed	LDRSB	STRB
16-bit unsigned	LDRH	STRH
16-bit signed	LDRSH	STRH
32-bit	LDR	STR
Multiple 32-bit	LDM	STM
Double-word (64-bit)	LDRD	STRD
Stack operations (32-bit)	POP	PUSH

Mit der **LDM** bzw. **STM** Instruktion ist es möglich gerade eine ganze Registerliste zu Laden bzw. zu Speichern.

### 9.3 Stack Push und Pop

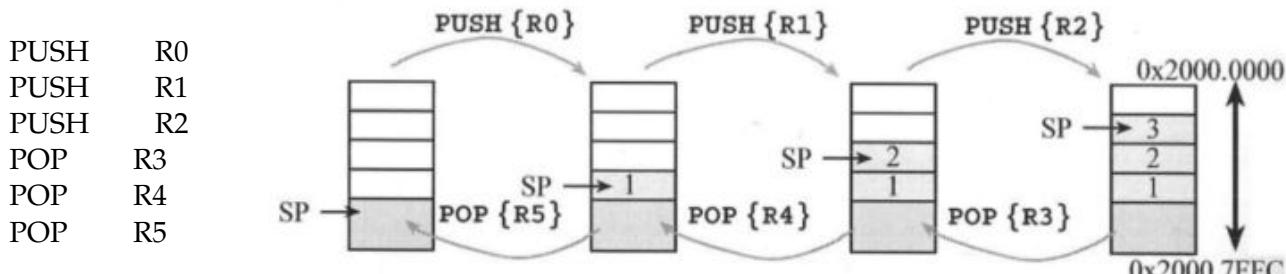


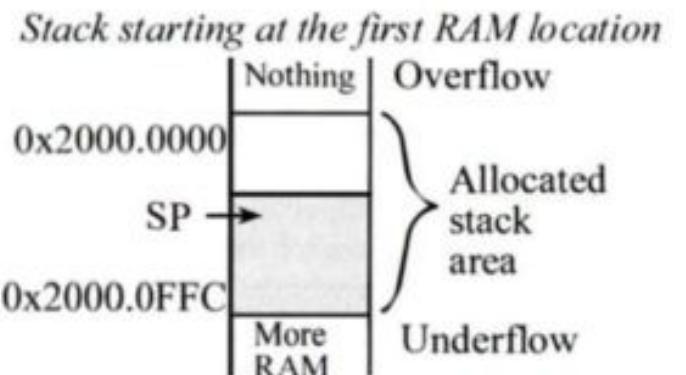
Figure 5-15 Push/Pop Stack-Operations [Val1]

**PUSH** und **POP** können auch Registerlisten mitgegeben werden, wie folgendes Beispiel zeigt:

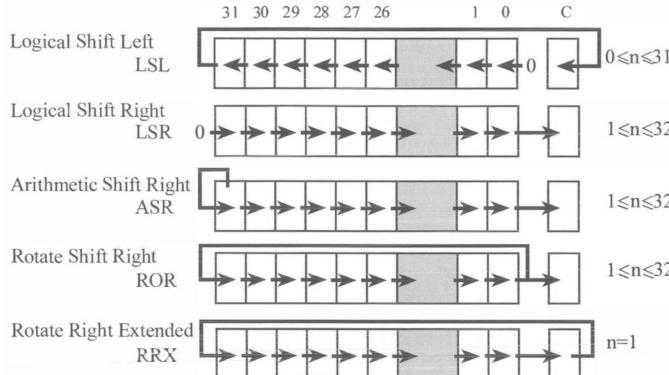
```
PUSH {R4-R6,LR} ;Save R4 to R6 an LinkRegister at the beginning of a subroutine.  
; LR contains the return address  
... ;processing the subroutine  
POP {R4-R6,PC} ;POP R4 to R6, and return address from stack. The return address  
;is stored into PC directly, this triggers a branch (subroutine return)
```

#### 9.3.1 Generelle Regeln bei der Verwendung des Stacks

1. Funktionen sollten die gleiche Anzahl Push und Pop Befehle aufweisen.
2. Stackzugriff nur innerhalb des allozierten Bereichs
3. Es sollte nicht über den SP auf den Stack geschrieben oder gelesen werden.
4. Stack sollte zuerst den SP dekrementieren und erst dann die Daten ablegen.
5. Stack sollte die Daten zuerst lesen und erst dann den SP inkrementieren.



## 9.4 Shift and Rotate Instructions



LSL: Signed, unsigned Multiplikation mit  $2^n$

LSR: Unsigned Division mit  $2^n$

## 9.6 Compare and Test

Die **COMPARE** und **TEST** Instruktionen aktualisieren die Flags im **APSR**, welche für den *Conditional Branch* oder *Conditional Execution* benötigt werden.

Instructions for Compare and Test	
Instructions	Description
CMP Rn, Rm	Compare: Calculate Rn-Rm. APSR is updated but the result is not stored.
CMP Rn, #immed	Compare: Calculate Rn-immediate data. APSR is updated but the result is not stored.
CMN Rn, Rm	Compare negative: Calculate Rn+Rm. APSR is updated but the result is not stored.
CMN Rn, #immed	Compare negative: Calculate Rn+immediate data. APSR is updated but the result is not stored.
TST Rn, Rm	Test (bitwise AND): Calculate AND result between Rn and Rm. N bit and Z bit in APSR are updated but the AND result is not stored. C bit can be updated if barrel shifter is used.
TST Rn, #immed	Test (bitwise AND): Calculate AND result between Rn and immediate data. N bit and Z bit in APSR are updated but the AND result is not stored.
TEQ Rn, Rm	Test (bitwise XOR): Calculate XOR result between Rn and Rm. N bit and Z bit in APSR are updated but the XOR result is not stored. C bit can be updated if barrel shifter is used.
TEQ Rn, #immed	Test (bitwise XOR): Calculate XOR result between Rn and immediate data. N bit and Z bit in APSR are updated but the EXOR result is not stored.

## 9.7 Program Flow Control

### 9.5 Bit-Field Processing Instructions

Der Cortex-M3/M4 Prozessor verfügt über viele verschiedene *Bit-Field Processing Operations*-Instruktionen, einzelne sind folgend aufgelistet:

Instructions for Bit-Field Processing	
Instructions	Description
CLZ Rd, Rm	Count leading zero
RBIT Rd, Rn	Reverse bit order in register
BFC Rd, #lsb, #width	Clear bit field within a register
BFI Rd, Rn, #lsb, #width	Insert bit field to a register
SBFX Rd, Rn, #lsb, #width	Copy bit field from source and sign extend it
UBFX Rd, Rn, #lsb, #width	Copy bit field from source register

### 9.7.1 Unconditional Branches

*Unconditional Branches* sind Sprünge zu einem Label, welche wie folgt implementiert werden können.

Unconditional Branch Instructions	
Instructions	Description
B label	Branch to label. If a branch range of over +/- 2KB is needed, you might need to specify B.W to use 32-bit version of branch instruction for wider range.
B.W label	Branch and exchange. Branch to an address value stored in Rm, and set the execution state of the processor (T-bit) based on bit 0 of Rm. <i>(Bit 0 of Rm must be 1 because Cortex-M processor only supports Thumb state.)</i>
BX Rm	Branch and exchange. Branch to an address value stored in Rm, and set the execution state of the processor (T-bit) based on bit 0 of Rm. <i>(Bit 0 of Rm must be 1 because Cortex-M processor only supports Thumb state.)</i>

### 9.7.2 Function Calls

Für den Funktionsaufruf wird die Instruktion *Branch and Link* verwendet.

Instructions for Calling a Function	
Instructions	Description
BL label	Branch to a labeled address and save the return address in LR.
BLX Rm	Branch to an address specified by Rm, save the return address in LR, and update T-bit in EPSR with LSB of Rm.

Flags (status bits) in APSR, for Controlling Conditional Branch	
Suffix {cond}	Meaning, Branch Condition
EQ Equal	Z = 1
NE Not equal	Z = 0
CS or HS	Carry set, Unsigned > carry OR borrow
CC or LO	Carry clear, Unsigned ≤ carry OR borrow
MI Min us/negative	C = 1
PL Plus/positive or zero (non-negative)	N = 1
VS Overflow	N = 0
VC No overflow	V = 1
HI Unsigned > ("Higher")	V = 0
LS Unsigned ≤ ("Lower or Same")	C = 1 && Z = 0
GE Signed ≥ ("Greater than or Equal")	C = 0   Z = 1
LT Signed < ("Less Than")	N = V
GT Signed > ("Greater Than")	N ≠ V
LE Signed ≤ ("Less than or Equal")	Z = 0 && N = V
AL Always (unconditional)	Z = 1    N ≠ V only used with IT instruction

### 9.7.3 Conditional Branches

*Conditional Branches* sind bedingte Sprünge, bei welchen nur zur angegeben Adresse gesprungen wird, wenn die Flags die Bedingung erfüllen. Die Flags (APSR) werden immer vor einem *Conditional Branch* mit einer **CMP**-Instruktion evaluiert.

Instructions for Conditional Branch	
Instructions	Description
B{cond} label	Branch to label if condition is true. E.g.,
B{cond}.W label	Branch to label if condition is true. E.g., CMP R0, #1 BEQ p2 ;if Equal, then go to p2 MOV R3, #1 ;R3=1 B p3 ;go to p3 ;label p2
B{cond} loop	If a branch range of more than +/-254 Bytes is needed, B.W can be used as a 32-bit version of branch instruction for wider range.

Ein Beispiel:

```
CMP R0, #1 ;compare R0 to 1
BEQ p2 ;if Equal, then go to p2
MOV R3, #1 ;R3=1
B p3 ;go to p3
;label p2
p2: MOV R3, #2
p3: ;label p3
     ;other subsequence operation
```

## 10 V10

### 10.1 C/C++ Strukturen Umsetzen

#### 10.1.1 Entscheidungen

In Assembler-Sprache ist eine Entscheidung praktisch immer in einem 2-Stufigen Ablauf umgesetzt.

- Benötigte Flags ermitteln
- Zugehörige bedingte Sprünge ausführen

Dabei wird nach folgendem Ablauf gearbeitet:

1. Vergleich: Zwei Werte werden subtrahiert, dabei wird nur auf die Flags geschaut.
2. Anhand der Flags werden dann die bedingten Sprünge ausgeführt

#### 10.1.2 Analyse von Hochsprachencode und Assembly-Code

Für die Bestimmung der richtigen *Conditional Branch* Instruktion kann wie am folgendem Beispiel gezeigt werden vorgegangen werden:

if (...)	
then	else
foo_one();	foo_two();

Thumb-2 Assembly Code	C Source Code
<pre> LDR    R0, G2 LDR    R1, G1 LDRSH.W R0, [R0, #0] LDR    R1, [R1] CMP    R0, R1 ;G2-G1 BGE    else_if then_if: BL    foo_one B     end_if else_if: BL    foo_two end_if: </pre>	<pre> signed Long G1; //signed 32-bit signed short G2; //signed 16-bit ... if (G1 &gt; G2)      //0 &gt; G2-G1 {     foo_one(); } else {     foo_two(); } </pre>

1. Bestimmung des Datenformates der Variablen: *signed* oder *unsigned*  
⇒ Beispiel: signed
2. Gleichung für den *Conditional Branch* aufstellen: **CMP Rn, Rm** entspricht Flags für  $Rn - Rm$   
⇒ Beispiel:  $G1 > G2 \Leftrightarrow 0 > G2 - G1$

#### if-Block

Für einen Sprung in den if-Block wäre nun der Operator  $\textcircled{>}$  entscheidend.  
⇒ Conditional Branch Instruction: **BLT**

Im Beispiel folgt jedoch der if-Block nach dem else-Sprung.

#### 10.1.3 Beispiele

##### IT-Instruktion

Thumb-2 Assembly Code	C Source Code
<pre> ;A=R0, B=R1, C=R2 ... CMP    R0, #0x0    ;A=0 ITTEE GT          ;4 instr. ADDGT R1, #0x1    ;B += 1 MOVGT R2, #0x0    ;C = 0 ADDLE R2, R1      ;C += B MOVLE R1, #0x64   ;B = 100 </pre>	<pre> register signed long A,B,C; ... if (A &gt; 0) {     B += 1;     C = 0; } else {     C += B;     B = 100; } </pre>

#### else-if-Block

Die Gleichung wird zusätzlich negiert:

$$0 > G2 - G1 = 0 \leq G2 - G1$$

Für einen Sprung in den else-if-Block wäre nun der Operator  $\textcircled{\leq}$  entscheidend.  
⇒ Conditional Branch Instruction: **BGE**

⇒ Conditional Branch Instruction: **BGE**

Wenn eine if..then..else Struktur nur wenige Sequenzen von Anweisungen enthält und keine *Branches* kann dies auch mit der *IT-Instruktion* umgesetzt werden, wobei diese maximal vier nachfolgende Anweisungen zu kontrollieren vermag.

Die erste Instruktion im *IT-Block* ist aktiviert, wenn der Condition-Code im Operandenfeld erfüllt ist. Die weiteren Anweisungen im Block werden durch anhängen von eins bis drei Buchstaben mnemonisch gesteuert: T ⇒ then; E ⇒ else

Allen Anweisungen im *IT-Block* muss der entsprechende Condition-Code angehängt werden. Es werden **immer** gleich viele Taktzyklen ausgeführt, egal welcher Pfad gefahren wird.

## FOR-Schleife

Thumb-2 Assembly Code	C Source Code
<pre>;XYZ=R0 ... <b>MOV R0, #0x0</b>           ;check loop <b>CMP R0, #0x5</b>           ;endup loop <b>BHS end_for</b> start_for:     BL foo_one ... <b>ADD R0, R0, #0x1</b>       ;YXZ++ <b>CMP R0, #0x5</b>           ;check loop <b>BLO start_for</b>          ;loop again end_for: ...     BL foo_two ... ;</pre>	<pre>register uint32_t XYZ; ... for (XYZ=0; XYZ&lt;5; XYZ++) {     foo_one(); } ... //BLO ≡ BCC foo_two(); ...</pre>
<pre>;XYZ=R0 ... <b>MOVS R0, #0xA</b>          ;check loop <b>BEQ end_for</b>             ;endup loop start_for:     BL foo_one ... <b>SUBS R0, R0, #0x1</b> <b>BNE start_for</b> end_for: ...     BL foo_two ...</pre>	<pre>register uint32_t XYZ; ... for (XYZ=10; XYZ!=0; XYZ--) {     foo_one(); } ... foo_two(); ...</pre>

Eine universelle Schleife (Iteration) besteht im Allgemeinen aus vier Teilen:

- Initialisierung
- Test auf Abbruch oder Fortsetzung
- Update der Loop-Control Variable
- Loop-Body (zu wiederholender Programmcode)

Bei FOR-Schleifen, welche die Laufvariable gegen Null vergleichen, können so manche Instruktionen erspart werden. So können statt der CMP-Instruktion die **MOVS-/SUBS-Instruktionen** verwendet werden, welche gerade das *Zero-Flag* evaluieren. Solange die Laufvariable nicht im Body verwendet wird, kann jede FOR-Schleife damit optimiert werden.

# 11 V11

## 11.1 Subroutinen

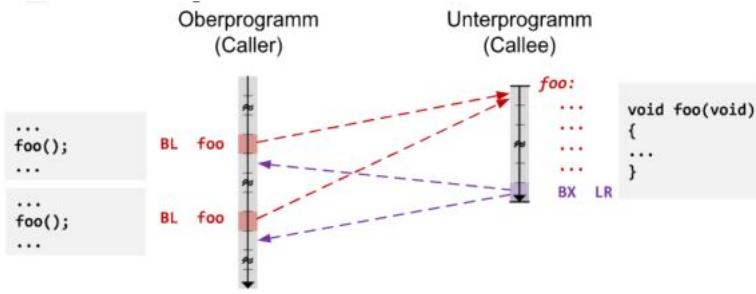


Figure 6-16 Unterprogramm-Aufruf und -Rücksprung

## 11.2 Architecture Producer Call Standard (AAPCS)

### 11.2.1 Regeln

- Bei Funktionsaufrufen werden die Register **R0-R3** als **Parameter** an eine C-Funktion verwendet
- Werden mehr als vier Funktionsparameter benötigt, werden diese vom Caller auf den Stack gelegt und auch wiedervom Caller entfernt.
- Die Funktionen müssen die Inhalte der Register **R4-R11** (falls benutzt) während der Ausführung sichern, um sie am Ende wieder rekonstruieren
- Der **Rückgabewert** einer Subroutine (8-bit, 16-bit, 32-bit) wird in den **Registern R0** übertragen. Handelt es sich um einen 64-bit Rückgabewert, so sind die unteren 32-bit im Register R0 und die oberen 32-bit im Register R1 übertragen
- Mit PUSH und POP wird immer eine **gerade Anzahl von Registern auf dem Stack** gelegt bzw. vom Stack eine **8-byte Alignment** auf dem Stack einzuhalten

### 11.2.2 Beispiel

Die Parameterübergabe bei Funktionen soll mit dem nachfolgenden Beispiel erklärt werden.

Die Funktion besitzt einen unsigned 8-Bit Rückgabewert und sechs signed 8-Bit Parameter.

(→ 2 Parameter werden über den Stack übergeben)

- Stackpointer SP (R13)**  
wird um 8 nach unten verschoben

- Variablen *e* und *f* werden vom Caller auf den Stack gelegt.

- R0 - R3* werden Parameter zugewiesen

- Sprung zum Label *foo\_4*

- R2, R3, R7, LR* werden auf den Stack gelegt (*SP* wird um 16 nach unten verschoben)

- R7* wird für Zugriff auf *e* und *f* als Framepointer gesetzt.

- R0 - R3* werden auf den Stack gelegt

Thumb-2 Assembly Code		C Source Code	Erweiterte Parameterübergabe via Stack
⑤ PUSHL {R2, R3, R7, LR}		uint8_t foo_4 (int8_t a, int8_t b, int8_t c, int8_t d, int8_t e, int8_t f);	
⑥ ADD R7, SP, #0x10			
⑦ STRB.W R3, [R13, #3]			
STRB.W R2, [R13, #2]			
STRB.W R1, [R13, #1]			
STRB.W R0, [R13, #0]			
	← Momentaufnahme Stack	{ return (a ^ e ^ f); }	
LDRSB.W R0, [R7, #0]			
LDRSB.W R1, [R7, #4]			
LDRSB.W R2, [R13, #0]			
EOR R0, R2			
EOR R1, R0			
UXTB R0, R1			
POP {R2, R3, R7, PC}			
	;	uint8_t value8;	
;	value8 auf dem Stack	value8 = foo_4(0x12, 85,	
① SUB SP, #8		0x77, 63,	
② MVN.W R0, #95		0xA0, value8);	
STR R0, [SP]		...	
LDRSB.W R0, [R13, #8]			
③ MOV R1, #0x55			
MOV R2, #0x77			
MOV R3, #0x3F			
STR R0, [SP, #0x4]			
MOV R0, #0x12			
④ BL foo_4			
STRB.W R0, [R13, #8]			
ADD SP, #8			
...			

Call stack diagram showing parameter passing:

- Caller stack frame (before call):
  - SP points to value8 (on stack)
  - value8 (on stack) contains 0xA0, 0xFF, 0xFF, 0x??, 0x??, 0x??, 0x00, 0x00, 0x00
- Callee stack frame (after call):
  - SP points to R0
  - R0 contains 0x12, 85, 0x77, 63, 0xA0, value8

## 12 V12

### 12.1 Registersatz

#### 12.1.1 Akkumulator

Bei einfachen Prozessoren oft das einzige Register

#### 12.1.2 Datenregister

→ General Purpose Register(GPR) bzw Universalregister

### 12.2 Assemblersprache

Assembling      Übersetzung von Assemblersprache in Maschinensprache

Disassembling    Übersetzung von Maschinencode in Assemblersprache

## 13 V13

### 13.1 Allgemeiner Ablauf von Exceptions und Interrupts

Interrupts werden in der Regel von der umgebenen Peripherie oder externen Input-Pins generiert und als Ereignis der CPU-Infrastruktur signalisiert, welche dann eine Handler-Routine einschalten.

Siehe Exceptions and Interrupts Seite: 16

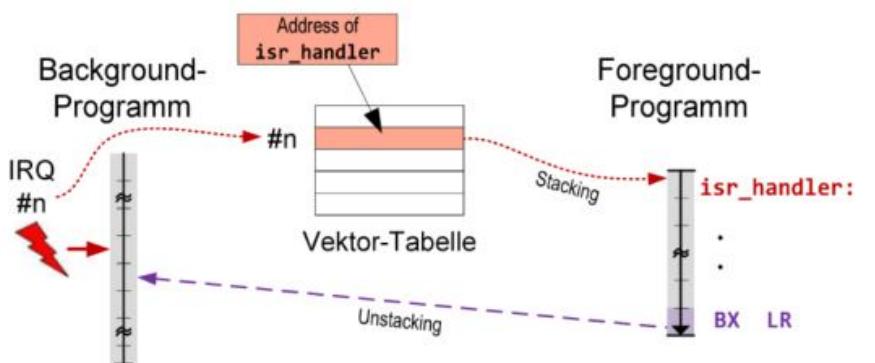


Figure 7-1 Allgemeiner Interrupt-Ablauf

Table 7-1 Cortex-M3 Exceptions und Priority-Levels [Martin]

	Exception	Name	Priority	Descriptions
Fault Mode and Startup Handlers				(Initial Stack Pointer)
	1	Reset	-3 <sup>-(Highest)</sup>	Reset
	2	NMI	-2	Nonmaskable Interrupt
	3	Hard fault	-1	Default fault if other handler not implemented
	4	Memory manage fault	Programmable	MPU violation or access to illegal locations
	5	Bus fault	Programmable	Fault if AHB interface receives error
	6	Usage fault	Programmable	Exceptions due to program errors
System Handlers	11	SVCall	Programmable	System service call
	12	Debug monitor	Programmable	Breakpoints, watch points, external debug
	14	PendSV	Programmable	Pendable service request for System Device
	15	Systick	Programmable	System Tick Timer
Custom Handlers	16	Interrupt #0	Programmable	External interrupt #0
	...	...	...	...
	...	...	...	...
	255	Interrupt #239	Programmable	External interrupt #239

## 14 V14

### 14.1 Spezielle Eigenschaften des NVIC

#### 14.1.1 Tail Chaining

Wenn eine Exception auftritt während bereits eine andere Exception-Behandlung mit gleicher oder höherer Priorität läuft, so wird die neue Exception hintenangestellt. Nach Abschluss des laufenden Exception Handlers, kann die CPU sofort den neuen Exception Request behandeln

#### 14.1.2 Late arrival

Wenn der Prozessor einen auftretenden Exceptionrequest akzeptiert, dann startet er die Stacking-Sequenz. Kommt während dem stacking eine weitere Exception mit höherer Priorität hinzu, so kann diese Late-Arrival-Exception noch bevorzugt behandelt werden.

#### 14.1.3 POP Preemption

Diese Funktion stellt gewissermassen eine Umkehrung des Late-Arrivals dar. Wenn eine Exception Request während dem Unstacking auftritt, so wird das Unstacking abgebrochen, und sofort VectorFetch und Instruction Fetch für den neuen Request durchgeführt. → Geschwindigkeitsoptimierung

## Anhang

### Glossar, Abkürzung

µVision	ARM Keil™ µVision® IDE
AAPCS	ARM Architecture Procedure Call Standard
ALU	Arithmetic Logic Unit, Rechenwerk
AMBA®	Advanced Microcontroller Bus Architecture
AR	Address Register, Adressregister
CCStudio, CCS	Texas Instruments Code Composer Studio™ IDE
CISC	Complex Instruction Set Computer
CMOS	Complementary Metal Oxide Semiconductor
CMSIS	Cortex Microcontroller Software Interface Standard
CPU	Central Processing Unit, synonym für Prozessor oder Zentraleinheit
CU	Control Unit, Steuerwerk
DAP / DWT	Debug Access Port, Data Watchpoint and Trace
DSP	Digitaler Signal Prozessor
FIFO	First-In-First-Out (Buffer, Speicher)
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
GPR	General-Purpose Register
IDE	Integrated Development Environment, Integrierte SW-Entwicklungsumgebung
IR	Instruction Register, Instruktionsregister, Befehlsregister
IRQ	Interrupt Request
ISA	Instruction Set Architecture, Befehlssatz-Architektur
ISR	Interrupt Service Routine
ITM	Instrumentation Trace Macrocell
LIFO	Last-In-First-Out (Buffer, Speicher), Stack
LSB	Least Significant Byte
LSBit	Least Significant Bit (rechts)
MAC	Memory Access Calculator or Multiply Accumulate Instruction
MIPS	Million Instructions Per Second
MMU	Memory Management Unit
MPU	Memory Protection Unit
MSB	Most Significant Byte
MSBit	Most Significant Bit (links)
NVIC	Nested Vectored Interrupt Controller
OpCode	Operation Code, Befehlscode
PC	Program Counter, Programmzähler
RAM	Random Access Memory (Schreib-/Lese-Speicher)
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory (Festwertspeicher)
SoC, SoPC	Systems-on-Chip, System-on-Programmable-Chip
SP, MSP, PSP	Stack Pointer, Main Stack Pointer, Process Stack Pointer
SR, PSR, xPSR	Statusregister, Program Status Register
SYSTICK	Standard Timer, SYSTICK
UAL	Unified Assembler Language, common syntax for ARM and Thumb instructions
uC, µC	Mikrocontroller
uP, µP	Mikroprozessor
WIC	Wakeup Interrupt Controller