

<b>Inhaltsverzeichnis</b>	
<b>1 Einführung</b>	<b>1</b>
1.1 Moore's Law . . . . .	1
1.2 Was ist ein Computer System . . . . .	1
1.3 Wissen um Assemblerprogrammierung . . . . .	1
1.4 Hardware Komponenten . . . . .	1
1.4.1 CPU . . . . .	1
1.4.2 Memory . . . . .	1
1.4.3 Memory und IO . . . . .	1
1.4.4 Systembus . . . . .	1
1.4.5 Beispiele . . . . .	1
1.5 Software-Aspekt . . . . .	1
1.5.1 Von C zu einem ausführbaren Programm . . . . .	1
1.5.2 Programmausführung auf einem Linux PC . . . . .	2
1.5.3 Programmausführung auf einem kleinen Embedded System . . . . .	2
1.6 Interaktion von Hardware und Software . . . . .	2
<b>2 Kombinatorische Logik</b>	<b>2</b>
2.1 Logische Status in einem binären System . . . . .	2
2.2 Logische Operationen . . . . .	2
2.3 Lösungsrezept . . . . .	2
2.4 Multiplexer . . . . .	2
2.5 Demultiplexer . . . . .	2
2.6 Halbaddierer . . . . .	2
2.7 1-Bit Volladdierer . . . . .	2
2.8 4-Bit Volladdierer . . . . .	2
<b>3 Sequentielle Logik</b>	<b>2</b>
3.1 Clock Signal . . . . .	2
3.2 D-Flip-Flop . . . . .	2
3.3 Counter . . . . .	2
3.4 Statusdiagramm . . . . .	2
3.5 Register . . . . .	2
3.6 Shift-Register . . . . .	2
<b>4 Cortex-M Architektur</b>	<b>3</b>
4.1 Hardware . . . . .	3
4.2 CPU Model (Hauptbestandteile M0 CPU) . . . . .	3
4.2.1 16 Core Register . . . . .	3
4.2.2 32-Bit ALU . . . . .	3
4.2.3 Flags/APSR (Application Processor Status Register) . . . . .	3
4.2.4 Control Unit with IR = Instruction Register . . . . .	3
4.2.5 Bus Interface . . . . .	3
4.3 Instruktionen Set . . . . .	3
4.4 Programmausführung . . . . .	3
4.5 Memory Map . . . . .	3
4.5.1 Memory Layout . . . . .	3
4.5.2 Adresszuweisung . . . . .	3
4.6 Integer Typen in C . . . . .	3
4.6.1 Unsigned Integers . . . . .	3
4.6.2 Signed Integers . . . . .	3
4.7 Arrays . . . . .	3
4.8 Alignment . . . . .	4
4.9 Object File Sections . . . . .	4
4.10 Memory Allozierung . . . . .	4
<b>5 Datentransfer</b>	<b>4</b>
5.1 Transfertypen . . . . .	4
5.2 EQU - Literale laden . . . . .	4
5.3 LDR - Load literal . . . . .	4
5.4 Arrays . . . . .	4
5.4.1 Byte-Array . . . . .	4
5.4.2 HalfWord-Array . . . . .	4
5.5 Memory Mapped I/O . . . . .	4
5.6 Pointer und Adressen . . . . .	4
5.7 Übungen . . . . .	4
<b>6 Arithmetische Instruktionen</b>	<b>5</b>
6.1 Flags . . . . .	5
6.2 Negative Zahlen . . . . .	5
6.2.1 Zweierkomplement . . . . .	5
6.3 Addition . . . . .	5
6.4 Subtraktion . . . . .	5
6.5 Multi-Word Arithmetik . . . . .	5
6.6 Multiplikation . . . . .	5
6.7 Übungen . . . . .	5
<b>7 Integer Casting</b>	<b>6</b>
7.1 Integergrößen in C . . . . .	6
7.2 signed ↔ unsigned . . . . .	6
7.3 Impliziter Cast von C . . . . .	6
7.4 Erweiterung . . . . .	6
7.5 Abschneiden . . . . .	6
7.6 Assembler . . . . .	6
<b>8 Logische Instruktionen, Shift, Rotate</b>	<b>6</b>
8.1 Logische Instruktionen . . . . .	6
8.2 Shift/Rotate . . . . .	6
8.2.1 Multiplizieren mit einer Konstante . . . . .	6
8.3 Übungen . . . . .	6
<b>9 Branches</b>	<b>6</b>
9.1 B . . . . .	6
9.2 BX (Register) . . . . .	6
9.3 Conditional Branches . . . . .	6
<b>10 Strukturierte Programmierung</b>	<b>7</b>
10.1 Selektion (If-else) . . . . .	7
10.2 Loop . . . . .	7
10.3 Switch . . . . .	7
<b>11 Subroutinen und Stack</b>	<b>7</b>
11.1 Struktur einer Subroutine . . . . .	7
11.2 Stack . . . . .	7
11.2.1 Implementierung . . . . .	7
11.2.2 Initialisierung . . . . .	7
11.2.3 PUSH {R0} . . . . .	7
11.2.4 POP {R0} . . . . .	7
11.2.5 ARM . . . . .	7
11.2.6 Memory hinzufügen oder entfernen . . . . .	7
11.3 Verschachtelte Subroutine . . . . .	8
11.4 Assembler Direktiven . . . . .	8
<b>12 Parameterübergabe</b>	<b>8</b>
12.1 Verschiedene Möglichkeiten . . . . .	8
12.2 Eintrittsinvarianz . . . . .	8
12.3 ARM Procedure Call Standard . . . . .	8
12.4 Stack Frame & lokale Variablen . . . . .	8
12.5 Funktionen in C . . . . .	8
12.6 Assemblercode in C nutzen . . . . .	8
<b>13 Exceptional Control Flow</b>	<b>8</b>
13.1 Exceptions . . . . .	8
13.2 Was passiert bei einem Interrupt . . . . .	8
13.3 Interrupt Control . . . . .	9
13.4 Nested Exceptions . . . . .	9
13.5 Spezielle Situationen . . . . .	9
13.6 Datenkonsistenz . . . . .	9
<b>14 Architekturen</b>	<b>9</b>
14.1 von Neumann vs. Harvard . . . . .	9
14.2 CISC vs. RISC . . . . .	9
<b>15 Pipelining</b>	<b>9</b>
<b>16 Beispiele</b>	<b>10</b>
16.1 Bits 5 und 2 setzen . . . . .	10
16.2 Bits 6 und 3 löschen . . . . .	10
16.3 Bits 6 und 4 invertieren . . . . .	10
16.4 Addition 3 Zahlen unsigned . . . . .	10
16.5 Berechnungen unsigned . . . . .	10
16.6 Komplement, 8bit . . . . .	10
16.7 Speicherbereiche berechnen . . . . .	10
16.8 IF in Assembler . . . . .	10
16.9 For-Schleife von C in Assembler . . . . .	10
16.10 Wichtige Flags bei ADD/ADDS bzw. SUB/SUBS . . . . .	10
16.11 4bit ALU - Addition / Subtraktion . . . . .	10
16.12 CMP/TST . . . . .	10
16.13 Werte in Halfword-Tabelle verdoppeln . . . . .	10
16.14 STR - Speicheradresse . . . . .	10
16.15 Interrupt / Polling . . . . .	10
16.16 Interrupt im NVIC freigeben . . . . .	10
16.17 Instruktionen pro Sekunde (sequenziell bzw. mit Pipelining) . . . . .	10
16.18 Stackpointer . . . . .	11
16.19 Sequentielle Logik . . . . .	11
16.20 C-Poiner in Assembler . . . . .	11
16.21 Kontrollstrukturen . . . . .	11
16.21.1 If-then-Else unsigned 8-bit variable . . . . .	11
16.21.2 If-then-Else signed 8-bit variable . . . . .	11
16.21.3 If-Then-Else with signed 16-bit variables . . . . .	11
16.21.4 For-loop . . . . .	11
16.22 Unterschied zwischen komb. und seq. Logik . . . . .	11
16.23 Integer Casting . . . . .	11
16.23.1 Integergrößen in C . . . . .	11
<b>17 Hilfsmittel</b>	<b>11</b>
17.1 ARM . . . . .	11
17.1.1 Little Endian / Big Endian . . . . .	11
17.1.2 Register . . . . .	11
17.1.3 Instruktionen . . . . .	11
17.2 Zer Potenz in HEX . . . . .	12
17.3 Branch Flags . . . . .	12
17.4 Wahrheitstabellen . . . . .	12

# 1 Einführung

## 1.1 Moore's Law

1965: "The number of transistors per IC doubles every year"  
somewhat slower since 1965 - doubles every 18 month

## 1.2 Was ist ein Computer System



- Ein Gerät, welches:  
 ▪ Input verarbeitet  
 ▪ Entscheidungen aufgrund des Inputs trifft  
 ▪ Verarbeitete Information wieder ausgibt

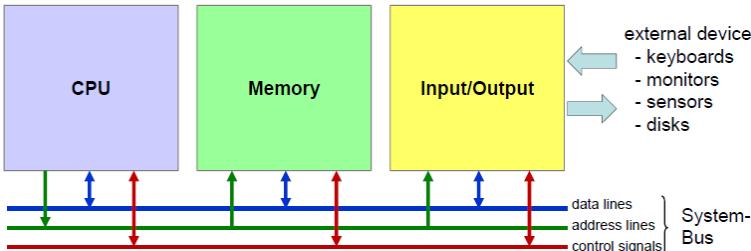
Information geht rein, wird vom Computersystem verarbeitet (Software und Hardware), wird verarbeitet ausgegeben

## 1.3 Wissen um Assemblerprogrammierung

- Mit Hilfe von Assembler können wir verstehen, was auf der Maschinenebene abläuft
- Verhalten von Programmen mit Fehlern
- Erhöhen der Performance (vorhandene und nicht vorhandene Optimierungen durch Compiler verstehen Ursachen für ineffiziente Programme finden)
- Implementieren von System Software (Boot Loader, Betriebssysteme, Interrupt Service Routinen)
- Lokalisieren und vermeiden von Sicherheitslücken z.B. Buffer Overflow

## 1.4 Hardware Komponenten

- CPU:** Zentrale Verarbeitungseinheit oder Prozessor
- Memory:** Speichert Instruktionen und Daten
- Input/Output:** Schnittstelle zu externen Geräten
- System-Bus:** Elektrische Verbindung der Blöcke



### 1.4.1 CPU

#### Datapath:

- ALU (Arithmetic and Logic Unit)
- Registers, schneller aber limitierter Speicher in der CPU, kann Ergebnisse zwischenspeichern
- 4/8/16/32/64 Bits weit

#### Control Unit:

- FSM (Finite State Machine) - liest Instruktionen und führt sie aus
- Operationstypen: Datentransfer, Arithmetische und Logische Operationen, Sprünge

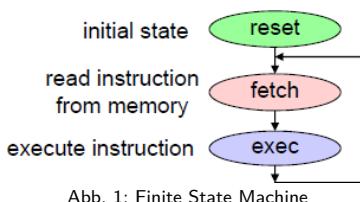


Abb. 1: Finite State Machine

### 1.4.2 Memory

- Eine Sammlung von Speicherzellen
- Kleinste adressierbare Einheit: 1 Byte; 1 Adresse pro Byte
- $2^n$  Adressen

### 1.4.3 Memory und IO

#### Main Memory - Arbeitsspeicher

- Zentraler Speicher
- Verbunden durch den Systembus
- Zugriff auf die einzelnen Bytes
- flüchtig: S(tatic)RAM, D(yamic)RAM
- nicht-flüchtig: ROM, flash

#### Secondary Storage

- langzeit oder peripherer Speicher
- Verbunden über I/O-Ports
- Zugriff auf Datenblöcke
- nicht-flüchtig
- langsamer aber kleinere Kosten

### 1.4.4 Systembus

CPU schreibt oder liest Daten vom Memory oder I/O

- adress lines:** CPU schickt die gewünschte Adresse über die Adress-Line (Von wo lesen oder wohin schreiben).  $2^n$  Adressen bei n Lines.
- control signals:** CPU sagt ob schreiben oder lesen und ob etwas gültig ist (Bus-Timing)
- data lines:** Transferiert Daten, 4/8/16/32/64 Daten-Lines

### 1.4.5 Beispiele

I/O besteht aus...

- PC: USB-Controller, Grafikadapter, Disk-Controller, Netzwerkadapter etc.)
- Embedded System: A/D Konverte (für Sensoren), Ventil Kontroller, Serielles Interface)

## 1.5 Software-Aspekt

### 1.5.1 Von C zu einem ausführbaren Programm

#### 1. Preprocessor (hello.i, text)

Verarbeitung der Preprocessor Statements (`#define`, `#include`, ... ) mittels Textprocessing: Ersetzen und erläutern von Inhalten  
Textfile mit modifiziertem C Sourcecode (Modified source program)

#### 2. Compiler (hello.s, text)

Übersetzung von C Sourcecode in Assemblerbefehle  
Textfile mit Assemblercode (Assembly program, human-readable, CPU specific)

#### 3. Assembler (hello.o, binär)

Übersetzt Assemblercode in Maschinenbefehle  
Binäres Objectfile mit Maschinenbefehlen (Relocatable object program)

#### 4. Linker (hello, binär)

Fügt mehrere Objectfiles zu einem ausführbaren Objectfile zusammen und löst die entsprechenden Referenzen zwischen den einzelnen Objectfiles auf.  
Binäres, ausführbares Objectfile (Executable object program)

### 1.5.2 Programmausführung auf einem Linux PC

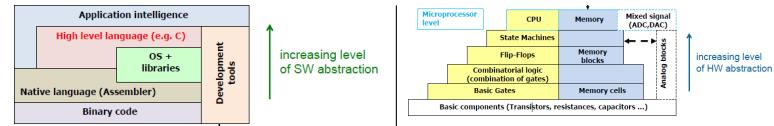
- ausführbare Datei hello ins memory laden und ausführen
- ./hello in die shell eintippen (Datei von der Disk ins Memory laden)
- OS: erstellt neuen Prozess, startet bei main()

### 1.5.3 Programmausführung auf einem kleinen Embedded System

#### Host vs.Target

- Software Entwicklung auf Host
  - Compiler/Assembler/Linker auf host
  - Loader auf Target lädt executable vom Host ins RAM
  - Loader kopiert executable vom RAM ins FLASH Memory
- System ohne Host
  - Loader springt zu main()
  - Instruktionen werden direkt vom FLASH Memory geholt

### 1.6 Interaktion von Hardware und Software



wird auf HW geladen

## 2 Kombinatorische Logik

### 2.1 Logische Status in einem binären System

- Outputs ändern sich abhängig vom Input und den internen logischen Funktionen
- Das System hat kein Memory - Es gibt kein Speicherelement
- Für  $n$  Inputs gibt es  $2^n$  mögliche Input-Kombinationen
- Der einzige Einfluss der Zeit besteht aus internen Verzögerungen (kein Clock)
- Outputs sind stabil nach einer Verzögerung

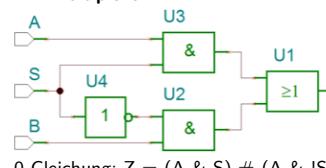
### 2.2 Logische Operationen

AND		$Z = A \& B$	NAND		$Z = \overline{A \& B}$
OR		$Z = A \# B$	NOR		$Z = \overline{A \# B}$
Buffer		$Z = A$	NOT		$Z = \overline{A}$
XOR		$Z = A \$ B$	XNOR		$Z = \overline{A \$ B}$

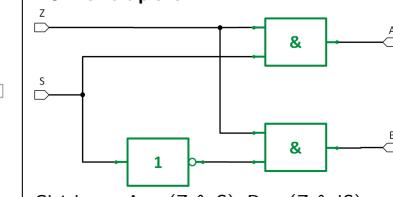
### 2.3 Lösungsrezept

1. Schaltung zeichnen
2. Wahrheitstabelle aufzeichnen
3. Gleichung aufstellen  $Z = \dots$

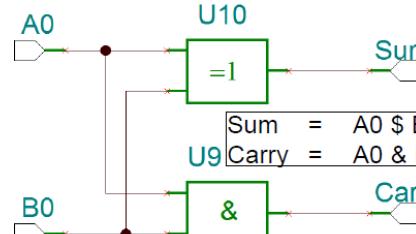
### 2.4 Multiplexer



### 2.5 Demultiplexer

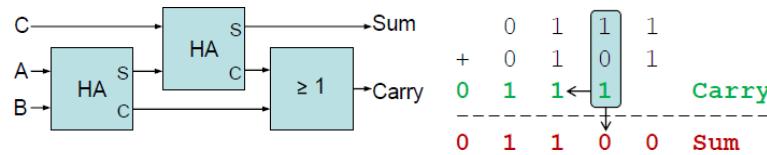


### 2.6 Halbaddierer



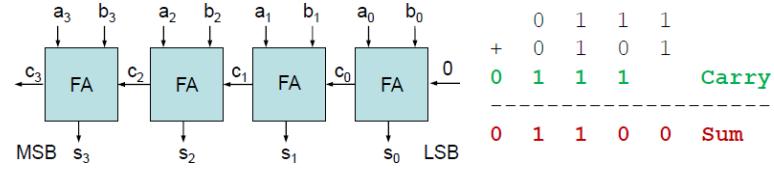
Kann zwei Bits zusammenzählen und ein Carry berechnen, kann das Carry aber nicht verarbeiten

### 2.7 1-Bit Volladdierer



Kann zwei Bits zusammenzählen und beachtet das Carry-Bit

### 2.8 4-Bit Volladdierer



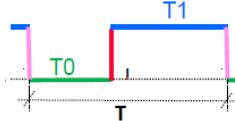
# 3 Sequenzielle Logik

Generelle Form: Finite State Machine (FSM)

- Hat Memory, Speicher vom Systemzustand
- Output ist abhängig vom Input und Internen Status
- Nächster Status hängt vom momentanen Status und den Inputs ab → Clock
- Jeder Zustand hat einen eingang und einen Ausgang

## 3.1 Clock Signal

T	f
1 s	1 Hz
1 ms = $10^{-3}$ s	1 kHz = $10^3$ Hz
1 us = $10^{-6}$ s	1 MHz = $10^6$ Hz
1 ns = $10^{-9}$ s	1 GHz = $10^9$ Hz



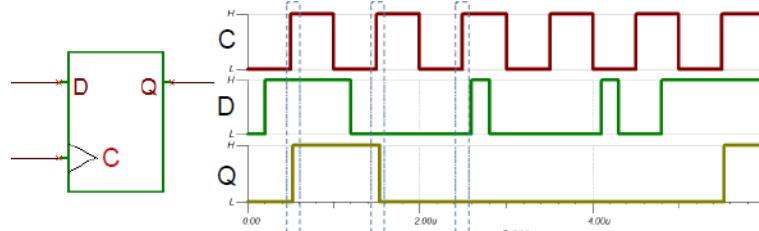
Misst die Zeit

Periode:  $T = T_0 + T_1$  [s]

Frequenz:  $f = 1/T$  [Hz] (Anzahl der Zyklen pro Sekunde)

Zyklus:  $T_1/T$  [-]

## 3.2 D-Flip-Flop



Edge Triggered Storage Element - steigende Flanke bei C → aktueller Wert von D wird an Q übertragen

sonst: keine Veränderung in Q

n Flip-Flops können  $2^n$  Status bezeichnen

## Lösungsrezept

- Von links nach rechts arbeiten
- Eine steigende Flanke um die andere
- Zuerst das Zeichnen, was von Clock abhängig ist (Status hinter Flip-Flop)
- Dann die vorderen, die diesen Status konsumieren, nachziehen

## 3.3 Counter

- Einfache Form einer FSM
- Status ändert mit steigender Flanke
- Nächster Status hängt nur vom aktuellen Status ab
- Es gibt keinen externen Input, der den Status beeinflussen könnte
- Besteht aus einem oder mehreren Flip-Flops
- Alle Flip-Flops hängen an der selben Clock

## 3.4 Statusdiagramm



## 3.5 Register

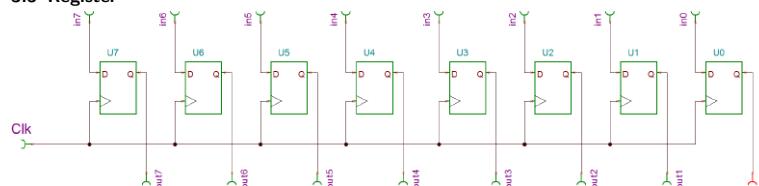
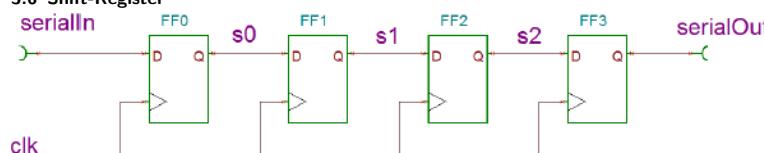


Abb. 2: Beispiel eines 8-bit Registers

## 3.6 Shift-Register



- Jeder Output ist verbunden mit dem nächsten Input
- Das erste Flip-Flop erhält den Input von aussen
- Das letzte Flip-Flop gibt den Output nach aussen
- Paralleles lesen der Daten möglich durch s0, s1 etc.
- Paralleles schreiben nur mit Multiplexer vor jedem Flip-Flop möglich

Wird genutzt für **Ethernet**, **USB**, **Smartphones**, etc (Parallele) Register; Input und Output sind parallel

# 4 Cortex-M Architektur

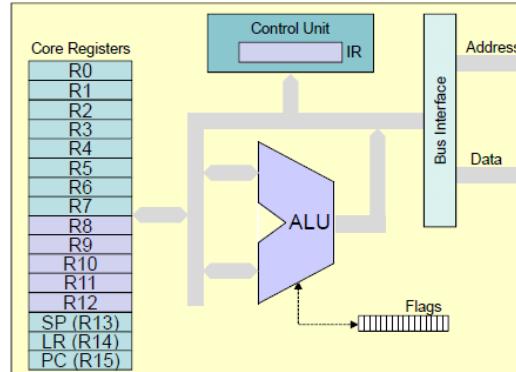
## 4.1 Hardware

### ARM Processor Portfolio

- Classic ARM7, ARM9, ARM11
- Embedded Cortex-M0, Cortex-M1, Cortex-M3, Cortex-M4, Cortex-R4 (Kosten und Energie sensitive Appl.)
- Application Cortex-A5, Cortex-A8, Cortex-A9, Cortex-A15 (Smartphones/Tablets)

## 4.2 CPU Model (Hauptbestandteile M0 CPU)

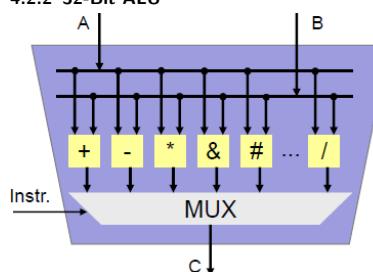
- Instruktionsset
- Prozessor (8-, 16-, 32-Bit)
- Register (Wie viele, wie gross)
- Adressarten (Wie kann Memory und IO adressiert werden)
- ARM Cortex-M (ARMv6-M = M0, ARMv7-M = M3/M4)



### 4.2.1 16 Core Register

- Jedes 32-Bit breit
- 13 allgemeine Register
  - tiefe Register (R0-R7)
  - hohe Register (R8-R12)
  - temporärer Zwischenspeicher für Daten und Adressen
- Program Counter (R15) - Adresse der nächsten Instruktion
- Stack Pointer (R13) - LIFO temporärer Datenspeicher
- Link Register (R14) - Rücksprungadresse letzter Funktionsaufruf

### 4.2.2 32-Bit ALU



- Input A & B, Resultat in C

- Integer Arithmetik - Addition/Subtraktion, Multiplikation/Division, Vorzeichenerweiterung (auffüllen mit 1 oder 0 um +/- zu bewahren)

- Logische Operatoren - AND, NOT, OR, XOR
- shift/rotate - left/right

### 4.2.3 Flags/APSR (Application Processor Status Register)

Bits werden von der CPU aufgrund der Resultate der ALU gesetzt

### 4.2.4 Control Unit with IR = Instruction Register

- Instruction Register (IR) - Maschinencode der Instruktion, die gerade ausgeführt wird
- Kontrolliert den Ablauf basierend auf der Instruktion im IR
- Generiert Kontrollsignale für alle anderen CPU Komponenten

### 4.2.5 Bus Interface

Interface zwischen dem internen CPU Bus und dem externen System-Bus; enthält Register um Adressen zu speichern

## 4.3 Instruktionen Set

Prozessoren interpretieren binär-kodierte Instruktionen

- aber binär ist schwer zu programmieren
- darum Instruktionen in menschenlesbarer Form (Assembler)
- Assembler(tool) macht die Übersetzung - assembler zu binär

### Assembler Programm

- Label (optional)
- Instruktion
- Operanden
- Kommentare (optional, mit ; eingeleitet)

Dazwischen immer min. 1 Leerzeichen

### Instruktionstypen

- Data transfer (43%) - Kopieren von einem ins andere Register, Daten vom Memory ins Register laden und andersrum
- Data processing (15%) - Arithmetische Operationen, Logische Operationen, Shift/rotate Operationen
- Control Flow (23%) - Branches, Funktionsaufrufe
- sonstiges (19%)

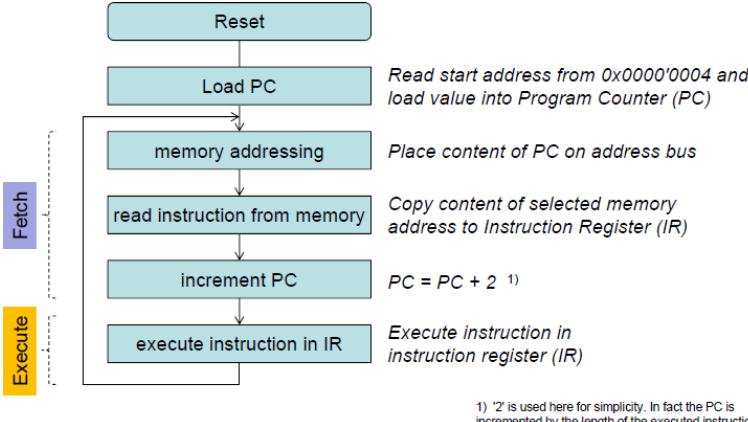
Am häufigsten (43 %) wird Datentransfer verwendet, danach Control Flow (23 %)

Die einzelnen Instruktionen und ihre Übersetzungen sind in den Beilagen

00000000	20A5	demoprg	MOVS	R0, #0xA5
00000002	2111		MOVS	R1, #0x11
00000004	1840		ADDS	R0, R0, R1
00000006	4A00		LDR	R2, =0x2000
00000008	6010		STR	R0, [R2]
0000000A	00002000			
0x0000'0000				linker adds offset of 0x0800'0000 before loading

Adressen sind immer in 2 Byte Schritten.

#### 4.4 Programmausführung



#### 4.5 Memory Map

Es ist ein grafisches Layout (Karte), das die Adressen und Größen der Elemente anzeigt, die mit der CPU kommunizieren (Speicher, Eingänge, Ausgänge). Die Memory-Map hilft dem Anwender zu wissen, wo sich die einzelnen Elemente befinden (z.B. beim Schreiben der entsprechenden Treiber).

##### 4.5.1 Memory Layout

- System Address Map
- Grafisches Layout vom Main Memory
- Linearer Bytearray
- Was ist wo gespeichert?
  - Location RAM
  - Location ROM
  - Location I/O Register

Memory Maps werden in CTIT1/2 mit der tiefsten Adresse zuunterst gezeichnet.

##### 4.5.2 Adresszuweisung

- ARM Policies (Cortex-M spezifisch, Chip-Hersteller)
- ST Designentscheide (Chipspezifisch, Anzahl und Grösse on-Chip Ram, Grösse vom Flash, Kontrollregister für Peripherie)
- CT Board Designentscheide (Boardspezifisch, LED, Switches etc.)

#### 4.6 Integer Typen in C

Integergrösse ist Plattformabhängig

8051	Cortex-Mx: Keil (ARM)	x86-64 (i7): gcc	
char	1	char	1
short	2	short	2
int	2	int	4
long int	4	long int	4
		long long int	8
char *	2	void *	4
		void *	8

Abb. 3: Integer-Größen in Bytes

##### 4.6.1 Unsigned Integers

C-Type - unsigned integers	Size	Term	inttypes.h / stdint.h
unsigned char	8 Bit	Byte	uint8_t
unsigned short	16 Bit	Half-word	uint16_t
unsigned int	32 Bit	Word	uint32_t
unsigned long	32 Bit	Word	uint32_t
unsigned long long	64 Bit	Double-word	uint64_t

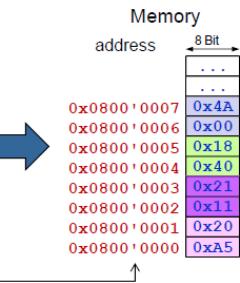
##### 4.6.2 Signed Integers

C-Type - signed integers	Size	Term	inttypes.h / stdint.h
signed char	8 Bit	Byte	int8_t
short	16 Bit	Half-word	int16_t
int	32 Bit	Word	int32_t
long	32 Bit	Word	int32_t
long long	64 Bit	Double-word	int64_t

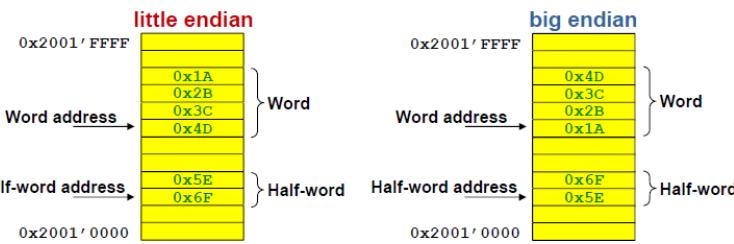
#### 4.7 Arrays

**little endian** - Least significant Byte an der tiefsten Adresse (Intel x86, Altera Nios, ST ARM(STM32))

**big endian** - Most significant Byte an der tiefsten Adress (Freescale[Motorola], PowerPC)



Examples: 0x1A2B'3C4D for Word and 0x5E6F for Half-word



Innerhalb der einzelnen Bytes ist aber immer das MSB zuvorderst

#### 4.8 Alignment

- Half-word - Variablen liegen auf geraden Adressen
- Word - Variablen liegen auf Adressen, die durch 4 teilbar sind

#### 4.9 Object File Sections

- CODE - Read-only (RAM, ROM), Instructions, Literale(Konstanten)
- DATA - Read-write (RAM), Globale Variablen, statische Variablen (C), Heap (C)
- STACK - Read-write(RAM), Funktionsaufrufe, Parameterübergabe, Lokale Variablen und lokale Konstanten

Die einzelnen Sektionen müssen im Memory nicht zusammenhängend sein.

Bsp: Berechnung Grösse, Start- und Endadresse

Falls Bereich gegeben: Zur Endadresse noch 4 Bytes hinzuzählen, dann Start - Ende (+4) = Bytes der Sektion in Hex

Falls Bytes gegeben: Bytes -4 und dann auf Basis 16 bringen. Dann Start + Basis16 Wert

#### 4.10 Memory Allozierung

- DCB - Byte
- DCW - Half-Word (HW aligned)
- DCD - Word (W aligned)
- SPACE oder % - mit Anzahl Bytes, reserviert leeren Platz

### 5 Datentransfer

Load/Store Architektur (ARM Cortex-M) - Memory wird nur durch load/store Operationen angesprochen

- Operanden vom Memory ins Register laden
- Operation ausführen
- Resultate vom Register ins Memory speichern

#### 5.1 Transfertypen

- Register zu Register
- Daten laden von Stack, Data und Code
- Literale laden nur von Code
- Daten speichern von den Register in den Stack oder Data (Code typischerweise Read-Only)

Register Memory Architektur (Intel x86) - Einer der Operanden kann im Memory liegen, Resultate können direkt ins Memory geschrieben werden

#### 5.2 EQU - Literale laden

Symbolische definition von Literalen und Konstanten, vergleichbar mit #define in C

```
MY_CONST8 EQU 0x12
MOVS R1,#MY_CONST8
```

Wird als immediate behandelt, funktioniert also nur in tiefen Registern mit genügend Immediate Platz.

MY\_CONST8 wird während der Kompilierzeit durch 0x12 ersetzt

#### 5.3 LDR - Load literal

LDR <Rt>,[PC,#<imm>]

- Indirekter Zugriff relativ zum PC
- PC offset <imm>
- Wenn der PC nicht Word-Aligned ist wird er auf die nächst höhere Adresse aligned
- Opcode Stelle 3-4 → imm geteilt durch 4, z.B. imm=8, Opcode Stelle 3-4 = 8 / 4 → 02 (Zeile 1 unten)

LDR R1,[PC,#4]

```
1 00000014 4902 ldr_lit LDR R1,[PC,#0x08] ; hex offset
2 00000016 4A03 LDR R2,[PC,#12] ; dec offset
3 00000018 4B01 LDR R3,myLit
4 0000001A 4C01 LDR R4,myLit
5 0000001C 4D00 LDR R5,myLit
6 0000001E E003 B ldr_lit2
7 00000020 12345678 myLit DCD 0x12345678
8 00000024 9ABCDEF0 DCD 0x9ABCDEF0
```

Pseudo instruction:  
Assembler converts to  
LDR R3,[PC,#0x04]

1 Offset in Bytes wird immer um 2 nach rechts geshiftet, damit der Offset danach in Words angegeben ist

3 Wird übersetzt zu LDR R3,[PC,#0x04]

6 Branch, damit die Literale unterhalb nicht als Code gewertet werden

#### Pseudo Instruktion

LDR Rd,=literal

Für den Wert nach = wird ein Literal angelegt an einer von Assembler ausgewählten Stelle im Code. Im Code wird =XY durch einen LDR Befehl mit Immediate zum PC gesetzt.

Wir sollten die Literale nie selbst speichern, immer Assembler überlassen

Von C nach Assembler

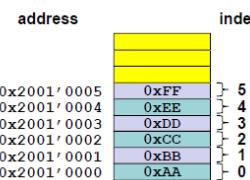
Wenn der Wert in ein MOVS Befehl passt, dann wird MOVS verwendet. Wenn der Wert grösser ist als 8 Bytes, dann wird ein Literal angelegt. Von Assembler dann natürlich ohne =, von uns müsste das = hin.

## 5.4 Arrays

### 5.4.1 Byte-Array

C-code

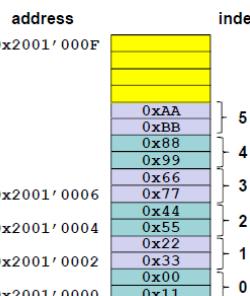
```
static uint8_t byte_array[] =
{0xAA, 0xBB, 0xCC, 0xDD,
 0xEE, 0xFF};
```



### 5.4.2 HalfWord-Array

C-code

```
static uint16_t halfword_array[] =
{0x0011, 0x2233,
 0x4455, 0x6677,
 0x8899, 0xAABB};
```



Und im gleichen Prinzip für Word-Arrays

Elementadresse = Basisadresse + Elementgrösse \* Index

// In C:

```
word_array[3] = 0xAABBCCDD;
```

; In Assembler:

```
LDR R0, =word_array
LDR R1,=0xAABBCCDD
```

```
STR R0,[R1,#0xC] (4 * 3)
```

## 5.5 Memory Mapped I/O

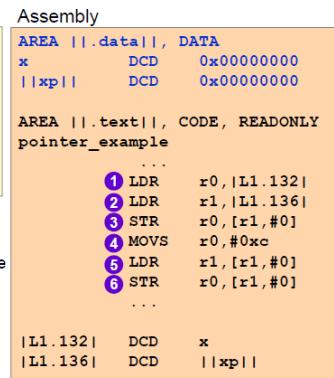
Variablen mit dem Keyword 'volatile' dürfen vom Compiler nicht wegoptimiert werden. Das bedeutet, er muss sie als Literale anlegen, anstatt dass er ihren Wert direkt in ein Register schreiben könnte. (Sie könnten sich während der Laufzeit ändern)

## 5.6 Pointer und Adressen

C-Code

```
void pointer_example(void)
{
    static uint32_t x;
    static uint32_t *xp;

    xp = &x;
    *xp = 0x0C;
}
```



Load address of x → R0

Load address of xp → R1

Store R0 (i.e. address of x) in xp variable  
(indirect memory access through R1)

Load immediate value 0x0C → R0

Load content of xp → R1  
i.e. address of x is now in R1

Store R0 at address given by R1

## 5.7 Übungen

### ▪ What is a load/store architecture?

Data processing is between registers. Transfer of data from and to the external memory is done using load (memory to register) or store (register to memory) instructions.

### ▪ What is the difference between a MOV and a MOVS instruction?

MOV Transfer does NOT affect flags (low and high registers)

MOVS Transfer affects flags (only low registers)

### ▪ Which data transfer instructions should you use if at least one high register is an operand?

MOV works for all registers (but only for reg/reg transfers)

### ▪ List different ways of initializing a low register with an immediate value. What are the advantages/disadvantages?

MOVS <Rd>, #<imm8> (value to load is in instruction) → Less memory space but limited to 8-bit values.

LDR <Rt>, [PC, #<imm>] (use PC/offset combination to point to the value to load).

→ Can be used to load larger values (up to 32-bit). Takes more space in memory.

### ▪ What is a pseudo-Instruction? Explain what is done with a <LDR Rn, = literal> pseudo-instruction.

A pseudo-Instruction does not directly translate in machine code. It is an instruction that is interpreted (or expanded) by the assembler (the tool) to generate the needed machine code instruction(s).

### ▪ Copy contents of R1 in R3 (flags unchanged):

MOV R3, R1

### ▪ Initialize R0 with 0xAA (flags unchanged):

LDR R0, =0xAA

### ▪ Initialize R1 with 234 (flags modified):

MOVS R1, #234

### ▪ Initialize R4 with 0x55AACC:

LDR R4, =0x55AACC

### ▪ Copy contents of R9 in R3:

MOV R3, R9

### ▪ Initialize R10 with 0x345678:

LDR R0, =0x345678 MOV R10, R0

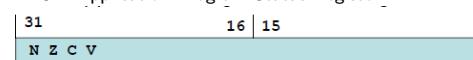
### ▪ Copy contents of R8 in R9:

MOV R9, R8

## 6 Arithmetische Instruktionen

### 6.1 Flags

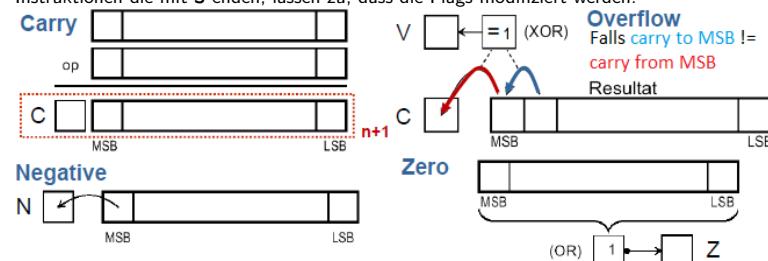
APSR: Application Program Status Register



Flag	Meaning	Action	Operands
Negative	MSB = 1	N = 1	signed
Zero	Result = 0	Z = 1	signed, unsigned
Carry	Carry	C = 1	unsigned
Overflow	Overflow	V = 1	signed

Der Prozessor weiß nicht, ob mit signed oder unsigned gearbeitet wird, er kalkuliert also immer C und V

Instruktionen die mit S enden, lassen zu, dass die Flags modifiziert werden.



Flags können mit MRS gelesen und mit MSR geschrieben werden.

### 6.2 Negative Zahlen

Zahlen mit begrenzter Anzahl an Stellen können auf einem Zahlenrad gezeigt werden. Für die Addition geht man im Uhrzeigersinn, für die Subtraktion im Gegen-Uhrzeigersinn.

Eine Negative Nummer ist somit repräsentiert als  $-a = 0-a$

#### 6.2.1 Zweierkomplement

Das Zweierkomplement bildet eine Zahl als negative Zahl ab.

Zuerst bildet man das Einerkomplement, indem man alle Ziffern umkehrt. Danach zählt man eine 1 dazu. Dann erhält man das ZK.

Instruktion RSBS - Sie erstellt das Zweierkomplement, also kehrt eine Zahl um.

Achtung: #0 gehört am Schluss immer dazu.

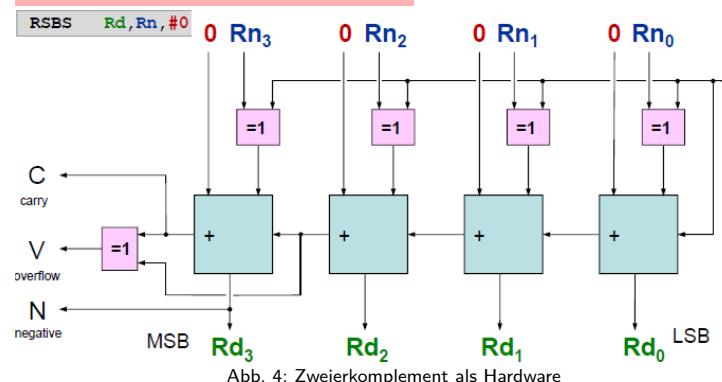


Abb. 4: Zweierkomplement als Hardware

### 6.3 Addition

#### ▪ Unsigned

- C = 1 indiziert Carry (Zahl ist über max hinaus wieder bei 0 gelandet)
- V irrelevant
- Addition von 2 grossen Zahlen → kann kleines Resultat geben

#### ▪ Signed

- V = 1 indiziert Overflow (Von positiv zu negativ gewechselt)
- kann nur passieren, wenn beide Operatoren das gleiche sign haben
- C irrelevant

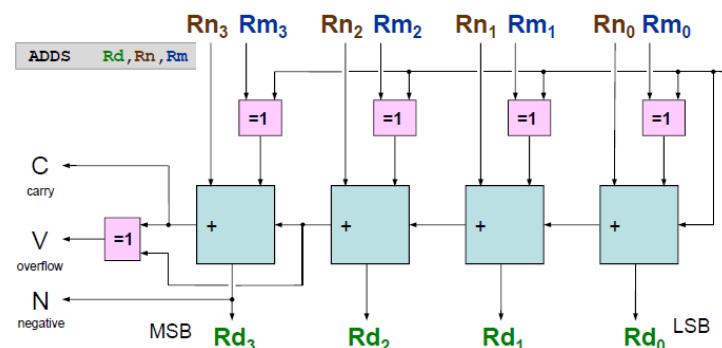


Abb. 5: Zweierkomplement als Hardware

Programm muss nach der Ausführung die Flags checken, um zu sehen ob alles geklappt hat.



## 8.2.1 Multiplizieren mit einer Konstante

Bsp: Multiplizieren mit 13

- Konstante als Power of 2 zeigen:  $13 = 8 + 4 + 1$
- $R0 = 13 * R1 \rightarrow R0 = (1 + 4 + 8) * R1 = R1 + (4 * R1) + (8 * R1)$

```

MOVS R0, R1 ; R0 = R1
LSLS R1, R1, #2 ; 4 * R1
ADDS R0, R0, R1 ; R0 = R0 + 4 * R1
LSLS R1, R1, #1 ; 2 * R1 -> 8 * R1
ADDS R0, R0, R1 ; R0 = R0 + 8 * R1

```

## 8.3 Übungen

- Invertieren des Inhaltes des Registers R1 (Bilden des Einerkomplements)

MOVNS R1,R1

- Verändern Sie den Inhalt des Registers R1 so, dass
  - die Bits 3..0 alle eins,
  - die Bits 7..4 alle null,
  - die Bits 17..16 alle invertiert und
  - alle übrigen Bits unverändert sind.

```

MOVS R0, #0xF          ; 3-0 -> 1
ORRS R1, R0             ; 7-4 -> 0
MOVS R0, #0xF0
BICS R1, R0             ; 17-16 inverted
MOVS R0, #0x30
LSLS R0, #12             ; 0x300000, shift 4 nibbles = 12bit
EORS R1, R0

```

- vorzeichenlosen Inhalt des Registers R7 mit der Dezimalzahl 43 multiplizieren und in R7 abspeichern

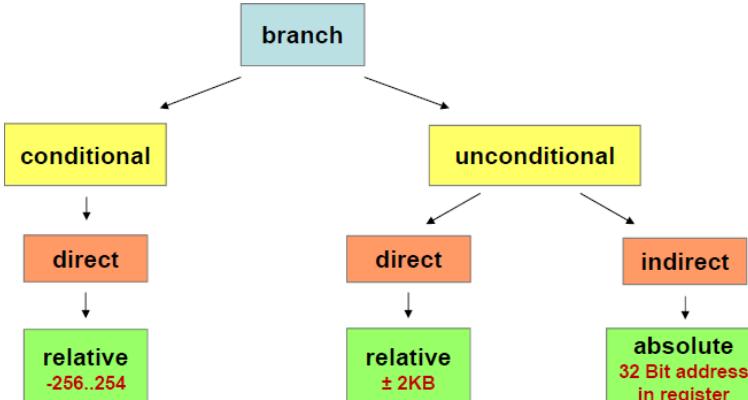
- Multiplikation mit MULS

MOVS R0, #43  
MULS R7,R0,R7

- Multiplikation mit Shift- und Addition

<code>; Multiplikator gerade Zahl ; 42 = 32 + 8 + 2 MOVS R0,R7 ; weglassen ;LSLS R0,R0,#1 ; 2 (shift) ADDS R7,R7,R0 ; 2 (addition) LSLS R0,R0,#3 ; 8 = 2^3 (shift) ADDS R7,R7,R0 ; addition LSLS R0,R0,#2 ; 32 = 8 * 2^2 ADDS R7,R7,R0 ; addition</code>	<code>; Multiplikator ungerade Zahl ; 43 = 32 + 8 + 2 + 1 MOVS R0,R7 ; 1 LSLS R0,R0,#1 ; 2 (shift) ADDS R7,R7,R0 ; addition LSLS R0,R0,#2 ; 8 (shift) ADDS R7,R7,R0 ; addition LSLS R0,R0,#2 ; 32 (shift) ADDS R7,R7,R0 ; addition</code>
--	---

## 9 Branches



- Typ - unconditional (immer); conditional (wenn condition stimmt)

- Zieladresse - relativ zum PC oder absolut

- Adressübergabe - direkt (als Teil der Instruktion), indirekt (steht in einem Register)

### 9.1 B

- Unconditional
- Direkt
- Relativ zum PC mit einem Label
- :0 hinter Instruktion nicht vergessen

### 9.2 BX (Register)

- Branch and Exchange
- Register Rm beinhaltet Zieladresse
- Unconditional
- Indirekt
- Absolut

### 9.3 Conditional Branches

Conditional Branches sind immer relative auf ARM

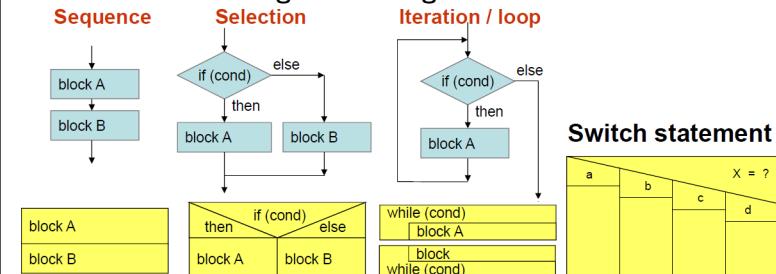
CMP - ist das gleiche wie eine Subtraktion, aber ohne dass das Resultat gespeichert wird.  
Für Größer-/Kleiner-/Gleich-Vergleiche

CMN - Ist das gleiche wie eine Addition (ohne Resultat), Prüft ob  $x = -x$  ist

TST - Prüft ob ein spezifisches Bit gesetzt ist, Logisches AND (ohne Resultat), Ändert nur N, Z und C

Abfragen ob es zutrifft mit BNE

## 10 Strukturierte Programmierung



### 10.1 Selektion (If-else)

```

int32_t nr, isPositive;
...
if (nr >= 0) {
    isPositive = 1;
} else {
    isPositive = 0;
}

```

```

CMP R1, #0x00
BLT MOVS B, R2, #1
end_if
else MOVS R2, #0
end_if
.....

```

Assume:  
nr in R1  
isPositive in R2

Achtung! Immer auf Signed oder Unsigned achten beim Vergleich!!

### 10.2 Loop

```

int32_t nr, sum;
...
sum = 0;
do {
    sum += nr;
} while (sum < 100);

```

```

MOVS R2, #0
loop ADDS R2, R2, R1
CMP R2, #100
BLT loop
.....

```

Assume: nr in R1 sum in R2

```

int32_t nr, prod;
...
prod = 1;
while (prod < 100) {
    prod *= nr;
}

```

```

B test
loop MULS R2, R1, R2
test CMP R2, #100
BLT loop
.....

```

### 10.3 Switch

```

uint32_t result, n;
switch (n) {
case 0:
    result += 17;
    break;
case 1:
    result += 13;
    //fall through
case 3: case 5:
    result += 37;
    break;
default:
    result = 0;
}

```

NR_CASES	EQU	6
case_switch	CMP R1, #NR_CASES	
	BHS R1, #2 ; * 4	
	LDR R7, =jump_table	
	LDR R7, [R7, R1]	
	BX R7	
case_0	ADD R2, R2, #17	
	B end_sw_case	
case_1	ADD R2, R2, #13	
case_3_5	ADD R2, R2, #37	
	B end_sw_case	
case_default	MOVS R2, #0	
end_sw_case	...	

Assume: n in R1  
result in R2

## 11 Subroutinen und Stack

Begriff: Subroutine, Prozedur, Funktion, Methode

- Sequenz von Instruktionen um einen Subtask zu lösen
- Werden mit einem 'Namen' aufgerufen
- Interface und Funktionalität ist bekannt
- Internes Design und Implementation ist versteckt
- Kann von irgendwo aufgerufen werden

Begriffe bei ARM

- Routine, Subroutine - Routine wird für Klarheit verwendet bei verschachtelten Calls. Routine ist der Aufrufer, Subroutine der aufgerufene
- Prozedur - Routine ohne Resultat
- Funktion - Eine Routine mit Resultat

### 11.1 Struktur einer Subroutine

```

00000050 4604 MulBy3 MOV R4,R0
00000052 0040 LSLS R0,#1
00000054 4420 ADD R0,R4
00000056 4770 BX LR

```

- Aufgerufen mit BL MulBy3
- LR = PC und PC wird auf MulBy3 gesetzt
- Am Schluss wird PC wieder auf LR gesetzt

## 11.2 Stack

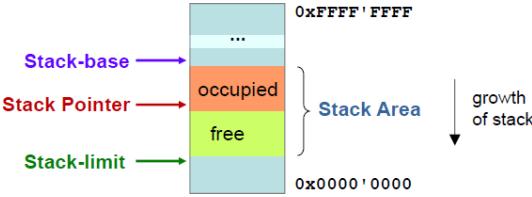
Wie kann das gelöst werden, wenn wir uns nur eine Rücksprungadresse merken können? Die Lösung ist ein Stack!

### 11.2.1 Implementierung

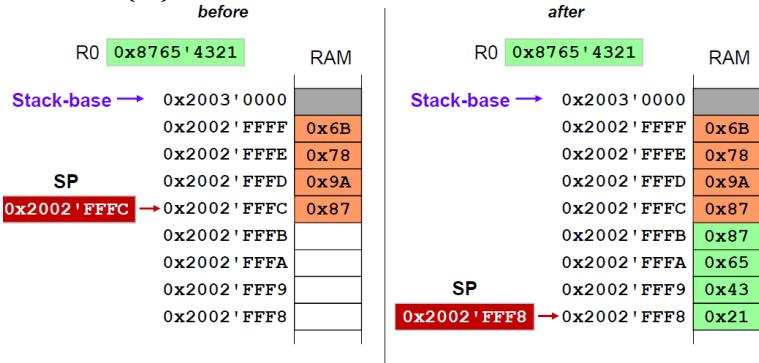
- Stack Area(Section) - Zusammenhängender Bereich im RAM
- Stack-Pointer(SP) - R13, zeigt auf den zuletzt geschriebenen Wert
- PUSH... - Dekrementiert den SP und speichert Worte
- POP... - Inkrementiert den SP und liest Wörter
- Richtung auf ARM - Der Stack wächst von der höheren, zur tieferen Adresse (full-descending)
- Alignment - Stack Operationen sind word-aligned

### 11.2.2 Initialisierung

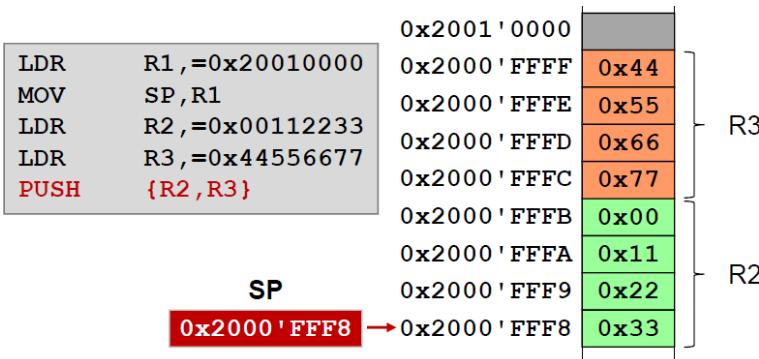
- Prozessor initialisiert SP (Stack-base) auf den Reset Wert (ausgehend von 0x0000'0000)
- Stack-base ist gleich über der Stack Area (der SP wird dekrementiert bevor das erste Wort geschrieben wird)



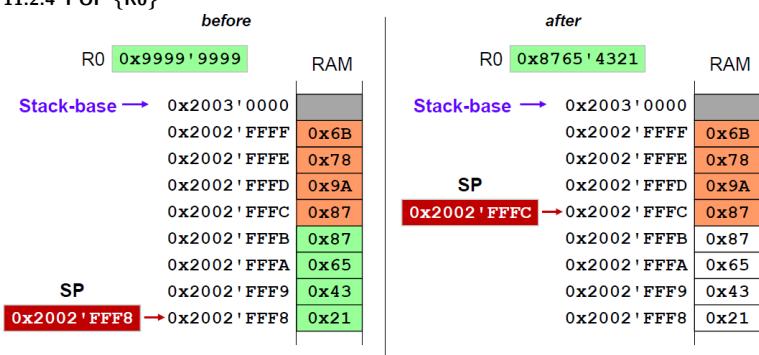
### 11.2.3 PUSH {R0}



- ein oder mehrere Register zum speichern (Macht Bitmaske für einzelne Register draus)
- Nur tiefe Register und LR (als M)
- Tiefstes Register wird zuletzt gespeichert



### 11.2.4 POP {R0}



- ein oder mehrere Register zum Speichern (Bitmaske)
- Nur tiefe Register und PC (als P)
- Das tiefste Register wird zuerst entladen

### 11.2.5 ARM

- Nur Worte (32-Bit)
- Es können keine Bytes oder Half-Words gepusht/-poppt werden
- SP mod 4 = 0 → Word aligned
- Anzahl Pushes = Anzahl Pops
- Stack-limit < SP < stack-base

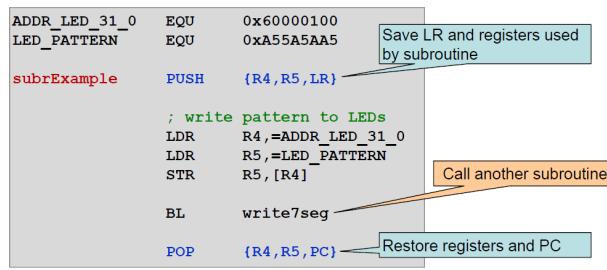
### 11.2.6 Memory hinzufügen oder entfernen

- ADD <RD>, SP, #<imm>; ADD <RD>, SP, <Rdm>, Inhalt vom Stack mit Offset in ein Register laden
- ADD SP, SP, #<imm>; ADD SP, SP, <Rm> Memory dealloziieren
- SUB SP, SP, #<imm> Memory alloziieren

Es gibt auch eine LDR und STR Methode mit dem SP

Um PUSH oder POP selber zu machen, muss einfach genügend Memory alloziert werden (Anzahl Register \* 4) und dann nach und nach mit aufsteigendem Offset (index \* 4) geschrieben werden (der kleinste zuerst). Beim Laden mit aufsteigendem Offset zurückladen und dann memory dealloziieren.

### 11.3 Verschachtelte Subroutine



### 11.4 Assembler Direktiven

PROC/ENDP und FUNCTION/ENDFUNC

werden auf der ersten und letzten Zeile der Funktion geschrieben, für den Debugger. Es ist nicht nötig BX LR aufzurufen, da der PC ja zurückgesetzt wird.

## 12 Parameterübergabe

### 12.1 Verschiedene Möglichkeiten

#### By value

- Werte in abgemachten Registern (R1 - Parameter, R0 - Rückgabewert)
- Effizient und Simpel
- Begrenzte Anzahl von Register

#### By Reference

- Über gibt Referenz der Datenstruktur im Register
- Erlaubt Übergabe von grossen Strukturen
- Bsp. in R0 - Adresse einer Tabelle und in R1 - Länge der Tabelle (by value)

#### Globale Variablen

- Gemeinsame Variablen in Data Area Bsp:

```
param1 SPACE 1
result SPACE 1
```

- Hoher Overhead, da Caller und Callee die Variablen lesen müssen
- Fehleranfällig, nicht wartbar (keine Kapselung, viele Abhängigkeiten, Braucht unique Variablennamen)

### 12.2 Eintrittsinvarianz

Wenn eine Funktion aufgerufen wird und dann eine Funktion aufruft, die wieder die vorherige Funktion aufruft, dann sind die Input-Werte in den Registern oder globalen Variablen für die erste Funktion nicht mehr vorhanden.

**Lösung:** Register und Stack verwenden → ARM Procedure Call Standard

### 12.3 ARM Procedure Call Standard

- teil des ABI für die ARM Architektur
- ABI - Application Binary Interface (Spezifikation zu Funktionsaufrufen, Parameterübergabe, Binäres Format)
- EABI - ABI für Embedded Applications

#### AAPCS spezifiziert:

- Datenlayout - Grösse, Alignment, fundamentale Datentypen
- Registerverwendung - für was ist welches Register
- Memory Sektionen & Stack - Code, read-only data, read-write data, stack, heap
- Stack - full descending, word-aligned
- Subroutine Calls - Mechanismus der LR und PC verwendet
- Rückgabewerte - Rückgabewerte durch R0 (und R1-R3)
- Parameterübergabe - Übergabe in R0-R3 und im Stack

#### Verwendung der Register

Register	Synonym	Role
r0	a1	Argument / result / scratch register 1
r1	a2	Argument / result / scratch register 2
r2	a3	Argument / scratch register 3
r3	a4	Argument / scratch register 4
r4	v1	Variable register 1
r5	v2	Variable register 2
r6	v3	Variable register 3
r7	v4	Variable register 4
r8	v5	Variable register 5
r9	v6	Variable register 6
r10	v7	Variable register 7
r11	v8	Variable register 8
r12	IP	Intra-Procedure-call scratch register <sup>1)</sup>
r13	SP	
r14	LR	
r15	PC	

Register contents might be modified by callee

Callee must preserve contents of these registers (Callee saved)

- Scratch Register - Für Zwischenberechnungen, Sind oftmals nicht im Programmcode erwähnt, haben limitierte Lebenszeit
- Variablen Register - Register, das den Wert einer Variable beinhaltet, oftmals im Source Code erwähnt. (R8-R11) oftmals ungenutzt
- Argument, Parameter - Sind austauschbar, Formale Parameter einer Subroutine
- Parameter - Caller kopiert Argumente in R0 bis R3, weitere in den Stack
- Rückgabe fundamentaler Datentypen
  - kleiner als ein Wort - Zero oder Sign extend zu einem Word und in R0
  - Word - R0
  - Double Word - R0(LSB)/R1
  - 128-Bit - R0(LSB)-R3
- Rückgabe zusammengesetzter Datentypen (structs, arrays..)
  - Bis zu 4 Bytes - in R0
  - Größer als 4 Bytes - Gespeichert in data area, Adresse als extra argument übergeben

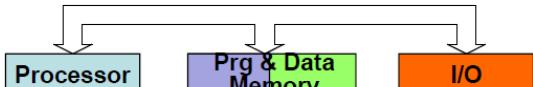


# 14 Architekturen

## 14.1 von Neumann vs. Harvard

von Neumann

### Systembus



Harvard



## 14.2 CISC vs. RISC

Complex Instruction Set Computer   Reduced Instruction Set Computer (RISC)

- |   |  |
|---|--|
| <ul style="list-style-type: none"> <li>▪ Traditioneller Memory Zugriff</li> <li>▪ Komplexe Adressierung</li> <li>▪ Hohe Codedichte</li> <li>▪ Die meisten Compiler können nur Sub-set der Instruktionen</li> <li>▪ Braucht oftmals manuelle Optimierung für embedded systems</li> <li>▪ Programm muss auf externes Memory warten</li> </ul> | <ul style="list-style-type: none"> <li>▪ Load-Store Architektur</li> <li>▪ Nur einfache Adressierung</li> <li>▪ Mehr Lines of code</li> <li>▪ Reduziertes Instruktionssatz (weniger HW, höhere Clockrate)</li> <li>▪ Erlaubt effektive Compiler Optimierung mit limitierten generischen Instruktionen</li> <li>▪ Programm arbeitet nur mit Register bei Vollgas</li> </ul> |
|---|--|

### Vorteile RISC

- Einfache HW: Mehr Register/Cache auf silicon area → Weniger Zugriffe aufs Memory
- Mehrere Datenpfade möglich
- Einfacheres Pipelining
- Höhere Clock Frequenzen

### Vorteile CISC

- Weniger Programm-Memory benötigt mit komplexen Instruktionen
- Kurze Programme können schneller und mit weniger Memory Zugriff sein

	Von Neumann	Harvard
RISC	ARM Cortex-M0 Series	ARM Cortex-M3/M4 Series (Harvard Cache Architecture)
CISC	x86	Specialized Processors (e.g. DSP)

# 15 Pipelining

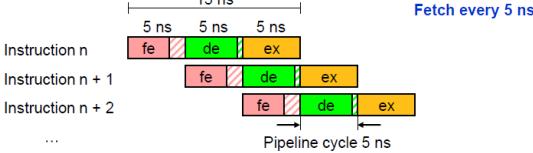
## Instruktionsverarbeitung

- fe: fetch (Read instruction) - 3ns
- de: decode (Decode instruction, read register or memory) - 4ns
- ex: execute (Execute Instruction, write back result) - 5ns

Eine Instruktion dauert so ungefähr 12 ns. Erst dann wird mit der nächsten begonnen, also alle 12ns kann eine Instruktion sequenziell durchgeführt werden.

Wie auch schon in der Industrie, hat man herausgefunden, dass man schneller ist, wenn man die verschiedenen Aktionen pro Operation auf verschiedene Stellen auslagert, anstatt dass man alle von einer Stelle erledigen lässt. Dazu braucht es:

- Die gleiche Ausführungszeit bei allen Stellen
- Keine Ressourcen Abhängigkeit untereinander
- Branch Instruktionen können die Pipeline leeren



### Vorteile

- Alle Stationen haben die gleiche Ausführungszeit
- Massiver Performance-Gewinn
- Jede Station braucht einfache Hardware und lässt höhere Clock-Rate zu

### Latenz

Ohne Pipelining:

Instructions per second = 1 / Instruction delay

Mit Pipelining (Pipeline muss zuerst gefüllt sein):

Instructions per second = 1 / Max stage delay

### Optimales Pipelining

- Alle Operationen sind auf Registern (single cycle execution)
- Es braucht dann 6 Clock cycles um 6 Instruktionen auszuführen (nachdem Pipeline mal gefüllt)
- Clock cycles pro Instruction (CPI) = 1

Wenn eine LDR oder STR Operation vorkommt, müssen gewisse Leerzeiten eingebaut werden, da sie mehrere Clock-Cycles brauchen.

### Kontrollrisiko

Passiert dann, wenn gebranzt wird. Der Compiler weiß vorher noch nicht, wohin er dann springt. Deshalb muss er Leerzeiten einfügen und warten, bis die Entscheidung gefallen ist damit er weitermachen kann.

**Standardlösung:** Der Compiler merkt sich eine Statistik und bereitet anhand dieser schonmal vor. Wenn er zweimal falsch gelegen hat, ändert er die Statistik.

**Was kann man selbst tun:** Weniger Loops und alles in einem Loop erledigen.

# 16 Beispiele

## 16.1 Bits 5 und 2 setzen

```
MOVS R2,#0x24 ; 0010'0100 in binär
ORRS R1,R1,R2
```

## 16.2 Bits 6 und 3 löschen

```
MOVS R2,#0x48 ; 0100'1000 in binär
BICS R1,R1,R2
```

```
MOVS R2,#0xB7 ; 1011'0111 in binär
ANDS R1,R1,R2
```

## 16.3 Bits 6 und 4 invertieren

```
MOVS R2,#0x50 ; 0101'0000 in binär
EORS R1,R1,R2
```

## 16.4 Addition 3 Zahlen unsigned

```
; Zahlen in R2, R3, R4 (32 bit)
; -> Ergebnis R1:R0
```

```
MOVS R1,#0
MOVS R5,#0
ADD S R0,R2,R3 ; R2 + R3
ADCS R1,R5 ; Carry addieren
ADD S R0,R0,R4 ; R0 + R4
ADCS R1,R5 ; Carry addieren
```

## 16.5 Berechnungen unsigned

```
Zahl1 DCB ?
```

```
Zahl2 DCB ?
```

```
; Zahl1 = Zahl1 - Zahl2
```

```
LDR R7,=Zahl1
```

```
LDRB R1,[R7]
```

```
LDR R2,=Zahl2
```

```
LDRB R2,[R2]
```

```
SUBS R1,R1,R2 ; Alternativ : SUBS
```

```
→ R1,R2
```

```
STRB R1,[R7]
```

```
; Zahl1 = Zahl1 + 42
```

```
LDR R7,=Zahl1
```

```
LDRB R1,[R7]
```

```
ADDS R1,#42
```

```
STRB R1,[R7]
```

## 16.6 Komplement, 8bit

```
MOVS R1,#0xC4
```

```
MOV R2,R1
```

```
MVNS R1,R1
```

```
RSBS R2,R2,#0
```

R1 = 3Bh

binär 1100'0100 -> 0011'1011 = 3B

R2 = 3Ch ; NOT

AL+1 (Zweierkomplement)

-> 0011'1011 -> 0011'1100 = 3C

## 16.7 Speicherbereiche berechnen

CODE Beginn der Adresse 0x08001000

Länge 1024 Bytes

DATA Beginn der Adresse 0x20030400

Länge 512 Bytes

CONST Beginn lückenlos nach DATA

Länge 256 Bytes

## Lösung:

High 0x2003FFFF

CODE 0x08001000 - 0x080013FF

0x08001000 + 3FF (1023)

DATA 0x20030400 - 0x200305FF

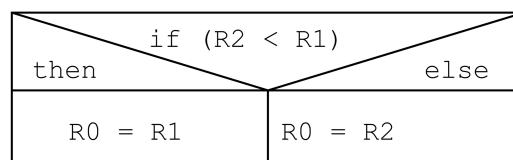
0x20030400 + 1FF (511)

CONST 0x20030600 - 0x200306FF

0x20030600 + FF (255)

Low 0x08000000

## 16.8 IF in Assembler



## Code in Assembler

```
CMP R2, R1
BGE else
then MOV R0, R1
B endif
else MOV R0, R2
endif
```

## 16.9 For-Schleife von C in Assembler

C-Code:

```
int32_t i, count;
```

```
for(i=0;i<10;i++) {
    count++;
}
```

Assembler: R0 = i, R1 = count

```
MOVS R0,#0
```

```
B testCond
```

```
loopStart ADDS R1,#1
```

```
ADDS R0,#1
```

```
testCond CMP R0,#10
```

```
BLT loopStart
```

## 16.10 Wichtige Flags bei ADD/ADDS bzw. SUB/SUBS

- Zwei Flags sind für die Interpretation des Resultates wichtig. Wie heißen die beiden Flags und wann verwendet man welches zur Interpretation des Resultates?

- Unsigned: N!=V (BLT)
- Signed: C==0 (LO)

## 16.11 4bit ALU - Addition / Subtraktion

op1	op2	Addition: op1 + op2			Subtraktion: op1 - op2		
		Resultat	Carry	Overflow	Resultat	Borrow	Overflow
0x8	0xB	0x3	1	1	0xD	1	0

Add signed (-8)+(-5)=(-13) -> nicht darstellbar -> overflow = 1

Add unsigned 8+11=19 -> nicht darstellbar -> carry = 1

Sub signed (-8)-(-5)=(-8)+5=-3 -> overflow = 0

Sub unsigned 8-11 = -3 -> nicht darstellbar als US -> borrow = 1

Anmerkung: auf dem ARM gilt C=0 -> borrow = 1

## 16.12 CMP/TST

- Falls der signed Wert im Register R1 grösser als 37d ist, so soll das Register R1 auf den Wert 20d gesetzt werden. Andernfalls soll es den bestehenden Wert behalten.

```
CMP R1,#37
```

```
BLE endcmp
```

```
MOV R1,#20
```

```
endcmp
```

▪ Falls Bit 3 im Register R1 gesetzt ist (=1'), soll der Inhalt des Registers R0 um eins nach links verschoben werden. Andernfalls soll nichts geschehen.

```
MOVS R2,#0x008
TST R1,R2
BEQ endtest
LSLS R0,#1
endtest
```

### 16.13 Werte in Halfword-Tabelle verdoppeln

TABLELENGTH	EQU 32
AREA	MyAsmVar, DATA, READWRITE
srcTable	SPACE TABLELENGTH
destTable	SPACE TABLELENGTH

- Schreiben Sie ein Assemblerfragment, welches in einer Schleife alle Werte aus der Tabelle srcTable liest, verdoppelt und an der gleichen Position in destTable abspeichert.

```
MOVS R0,#0
LDR R7,=srcTable
LDR R6,=destTable
loop
    LDRH R2,[R7,R0]
    LSLS R2,R2,#1
    STRH R2,[R6,R0]
    ADDS R0,#2
    CMP R0, TABLELENGTH
    BNE loop
```

### 16.14 STR - Speicheradresse

- Die Register R1, R7 und R0 enthalten folgende Datenwerte:

- R1: 0x23B107A4
- R7: 0x200048D0
- R0: 0x0000000C

- Nun wird folgender Befehl ausgeführt:  
- **STR R1, [R7, R0]**

- Geben Sie an, auf welcher Speicheradresse, welcher Datenwert abgelegt wird:

Speicheradresse	Datenwert
0x200048DC	0xA4
0x200048DD	0x07
0x200048DE	0xB1
0x200048DF	0x23

Bemerkung: Speicheradresse 0x200048D0 (R7) mit Offset C (R0)

### 16.15 Interrupt / Polling

- Wozu wird das Interruptverfahren angewendet? Beschreiben Sie die wesentlichen Vorteile gegenüber dem Pollingverfahren mit wenigen Worten.
  - CPU-Last reduzieren, geringere Leistungsaufnahme
  - CPU kann andere Aufgaben ausführen
  - Antwortzeit verkürzen und Jitter verringern

Schreiben Sie einen Interrupt Handler **EXTI3\_IRQHandler**, welcher die gegebene Variable **counter** bei jedem Aufruf um 8 erhöht.

Die Variable ist wie folgt definiert:

```
counter DCD 0 ; 4 Byte
```

Die Funktion **reset\_EXTI3\_int()** für den Reset von EXTI3 ist extern gegeben.

```
IMPORT reset_EXTI3_int
```

**EXTI3\_IRQHandler**

```
PROC
    EXPORT EXTI3_IRQHandler
    PUSH {R0-R3, LR}
    LDR R2, =counter
    LDR R3, =8

    LDR R1, [R2]
    ADDS R1, R1, R3 ; oder ADDS R1,#8
    STR R1, [R2] ; Increment counter

    BL reset_EXTI2_int
    POP {R0-R3, PC}
    ENDP
```

Anmerkung: Register R0-R3 müssen nicht gepushed werden

### 16.16 Interrupt im NVIC freigeben

Die Adresse des benötigten Registers ist wie folgt gegeben:

```
REG_SETENA0 EQU 0xe000e100 ;Address of Interrupt Enable Reg
;Ext. Interrupts IRQ31 IRQ0
```

Code:

```
LDR R1, =REG_SETENA0 ; load SETENA0 address
LDR R0, =0x200 ; enable EXTI3
STR R0, [R1] ; Store register value
```

### 16.17 Instruktionen pro Sekunde (sequenziell bzw. mit Pipelining)

- |                |   |      |
|----------------|---|------|
| 1. fe: fetch   | Read instruction                            | 4ns  |
| 2. de: decode  | Decode instruction, read register or memory | 6ns  |
| 3. ex: execute | Execute instruction, write back result      | 10ns |

#### Bei sequentieller Ausführung der Instruktionen

Instructions per second = 1 / Instruction delay

Instruction delay =  $(4 + 6 + 10) * 10^{-9} \text{ s}$

Instructions per second =  $1/(20 * 10^{-9}) = 50 \text{ Mio.}$

#### Mit Pipelining

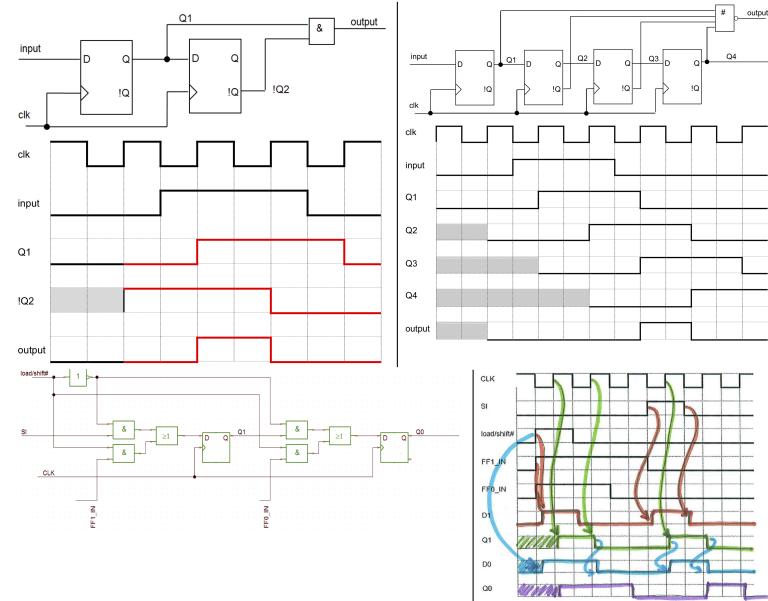
Instructions per second = 1 / max stage delay

Instructions per second =  $1/(10 * 10^{-9}) = 100 \text{ Mio.}$

### 16.18 Stackpointer

Adresse	Inhalt(Byte)	Stackpointer	Inhalt(Register)
0x200001F0	F8	SP3	R2/SP1
0x200001F1	01		
0x200001F2	00		
0x200001F3	20		R3
0x200001F4	04	SP2	
0x200001F5	03		
0x200001F6	02		R0
0x200001F7	01		
0x200001F8	11	SP1/SP4	
0x200001F9	11		R1
0x200001FA	34		
0x200001FB	12		
0x200001FC	01		R1
0x200001FD	10		
0x200001FE	12		
0x200001FF	00		R0
0x20000200		SP0	

### 16.19 Sequenzielle Logik



### 16.20 C-Pointer in Assembler

```
uint32_t x;
uint32_t y;
uint32_t *xp;

void main(void) {
    x = 3;
    xp = &x;
    y = *xp;
}

Statements to code
x = 3;
xp = &x;
y = *xp;
```

```
AREA myvar, DATA, READWRITE
X DCD 0
Y DCD 0
XP DCD 0
AREA myCode, CODE, READONLY
THUMB
main PROC
EXPORT main
; X=3
LDR R0,=0x03. ; X = 3
STR R0,[R3] ; Now store 3 in X
;XP = &X
STR R3,[R5]
;Y = *XP
LDR R2,[R5]
LDR R1,[R2]; Read contents of X in R1
STR R1,[R4]; Store R1 in Y (pointed to by R4)
```

### 16.21 Kontrollstrukturen

#### 16.21.1 If-then-Else unsigned 8-bit variable

```
uint8_t nrX = ...;
uint8_t nrY = ...;

if (nrX > nrY) {
    nrX = nrY;
} else {
    nrY = nrX;
}
```

```
LDR R6,=nrX ; R6 = address of nrX
LDRB R0,[R6] ; R0 = byte stored at nrX
LDR R7,=nrY ; R7 = address of nrY
LDRB R1,[R7] ; R1 = byte stored at nrY
CMP R0, R1 ; *unsigned* comparison
BLS else ; store nrY value at nrX
STRB R1, [R6]
B endif
else
STRB R0,[R7] ; store nrX value at nrY
```

#### 16.21.2 If-then-Else signed 8-bit variable

```
int8_t varA = ...;
int8_t varB = ...;

if (varA < -17 && varB > 36)
{
    varA = -varB;
} else {
    varA = varB;
}
```

```
LDR R6,=varA ; R6=address of varA
LDRB R0,[R6] ; R0=byte stored at varA
SXTB R0,R0 ; extend signed varA
LDR R7,=varB ; R7=address of varB
LDRB R1,[R7] ; R1=byte stored at varB
SXTB R1,R1
MOVS R2,#17 ; +17
RSBS R2,R2 ; -17
CMP R0,R2
BGE else ; *signed* comparison
CMP R1,#36
BLE else ; *signed*
RSBS R1,R1,#0 ; R1=-R1
STRB R1,[R6] ; varA = -varB
B endif
else
STRB R1,[R6]
```

### 16.21.3 If-Then-Else with signed 16-bit variables

```

int16_t varC = ...;
if (varC == 2344 || varC > 6788) {
    varC = varC / 4;
} else {
    varC = varC / 2;
}

LDR R7,=varC ; R7=address of varC
LDRH R0,[R7] ; R0=value stored at varC
SXTB R0,R0 ; extend signed varC
LDR R2,=2344
CMP R0,R2
BEQ then ; *signed* comparison
LDR R2,=6788
CMP R0,R2
BGT then ; *signed* comparison
ASRS R0,R0,#1 ; R0 = R0 / 2
STRH R0,[R7] ; varC = varC / 2
B endif
then
ASRS R0,R0,#2 ; R0 = R0 / 4
STRH R0,[R7] ; varC = varC / 4
endif

```

### 16.21.4 For-loop

```

int32_t = 0;
int32_t count = 0;

for(i=0 ; i < 10; i++) {
    count++;
}

LDR R6,i ; R6=address of i
LDR R0,[R6] ; R0=value at i
LDR R7,count ; R7=address of count
LDR R1,[R7] ; R1=value at count
B cond
loop
ADDS R0,R0,#1
ADDS R1,R1,#1
cond
CMP R0, #10
BLT loop ; *signed* comparison
STR R0,[R6] ; store final i
STR R1,[R7] ; store final count

```

### 16.22 Unterschied zwischen komb. und seq. Logik

#### Kombinatorische Logik:

- Ausgänge ändern sich nur in Abhängigkeit mit den Eingängen und den internen logischen Verknüpfungen.
- Keine Zustände, kein Speicher

#### Sequentielle Logik:

- Enthält Speicher, Systeme haben einen Zustand
- Ausgänge hängen von Eingängen und vom Zustand des Systems ab
- Der nächste Zustand des Systems hängt ab von den Eingangssignalen und aktuellem Zustand

### 16.23 Integer Casting

#### 16.23.1 Integergrößen in C

- 8-Bit
  - unsigned:** 0 – 255    **signed:** 0 – 127, (-128) – (-1)
- 16-Bit
  - unsigned:** 0 – 65'535    **signed:** 0 – 32'767, (-32'768) – (-1)
- 32-Bit
  - unsigned:** 0-4'294'967'295    **signed:** 0-2'147'483'647, (-2'147'483'648) – (-1)

## 17 Hilfsmittel

### 17.1 ARM

#### 17.1.1 Little Endian / Big Endian

- Little Endian:** LSB at lower address
- Big Endian:** MSB at lower address

#### 17.1.2 Register

**Low Registers (R0-R7):** fully accessible

**High Registers (R8-R12):** only accessible with MOV, ADD, CMP; only CMP sets flags

#### 17.1.3 Instruktionen

Arithmetic	ADDcdS <sup>†</sup>	reg, reg, arg	add
	SUBcdS	reg, reg, arg	subtract
	RSBcdS	reg, reg, arg	subtract reversed operands
	ADCcdS	reg, reg, arg	add both operands and carry flag
	SBCcdS	reg, reg, arg	subtract both operands and adds carry flag – 1
	RSCcdS	reg, reg, arg	reverse subtract both operands and adds carry flag – 1
	MULcdS	reg <sub>d</sub> , reg <sub>m</sub> , reg <sub>s</sub>	multiply reg <sub>m</sub> and reg <sub>s</sub> , places lower 32 bits into reg <sub>d</sub>
Bitwise logic	ANDcdS	reg, reg, arg	bitwise AND
	ORcdS	reg, reg, arg	bitwise OR
	eorcdS	reg, reg, arg	bitwise exclusive-OR
	BICcdS	reg, reg <sub>a</sub> , arg <sub>b</sub>	bitwise reg <sub>a</sub> AND (NOT arg <sub>b</sub> )
Comparison	CMPcd	reg, arg	update flags based on subtraction
	CMNcd	reg, arg	update flags based on addition
	TSTcd	reg, arg	update flags based on bitwise AND
	TEQcd	reg, arg	update flags based on bitwise exclusive-OR
MOV	MOVcdS	reg, arg	copy argument
	MVNcdS	reg, arg	copy bitwise NOT of argument
Memory access	LDRcdB <sup>‡</sup>	reg, mem	loads word/byte/half from memory into a register
	STRcdB	reg, mem	stores word/byte/half to memory from a register
	LDMcdum	reg!, mreg	loads into multiple registers
	STMcdum	reg!, mreg	stores multiple registers
Branch	SWPcdB	reg <sub>d</sub> , reg <sub>m</sub> , [reg <sub>n</sub> ]	copies reg <sub>m</sub> to memory at reg <sub>n</sub> , old value at address reg <sub>n</sub> to reg <sub>d</sub>
	Bcd	imm <sub>24</sub>	branch to imm <sub>24</sub> words away
	BLcd	imm <sub>24</sub>	copy PC to LR, then branch
	BXcd	reg	copy reg to PC, and exchange instruction sets (T flag := reg[0])
	SWIcd	imm <sub>24</sub>	software interrupt

<sup>†</sup> S = set condition flags

<sup>‡</sup> B = byte, can be replaced by H for half word(2 bytes)

shift: shift register value	mem: memory address
LSL #imm5	shift left 0 to 31
LSR #imm5	logical shift right 1 to 32
ASR #imm5	arithmetic shift right 1 to 32
ROR #imm5	rotate right 1 to 31
RRX	rotate carry bit into top bit
LSL reg	shift left by register
LSR reg	logical shift right by register
ASR reg	arithmetic shift right by register
ROR reg	rotate right by register
[reg, ±imm <sub>12</sub> ]	reg offset by constant
[reg, ±reg]	reg offset by variable bytes
[reg, ±reg, shift]	reg, offset by shifted variable reg <sub>1</sub> <sup>†</sup>
[reg, ±imm <sub>12</sub> !]	update reg by constant, then access memory
[reg, ±reg, shift!]	update reg by variable bytes, then access memory
[reg, ±imm <sub>12</sub> !]	update reg by shifted variable, then access memory
[reg, ±reg, shift!]	access address reg, then update reg by offset
[reg, ±reg, shift]	access address reg, then update reg by variable
[reg, ±reg, shift]	access address reg, then update reg by shifted variable

<sup>†</sup> shift distance must be by constant

### 17.2 2er Potenz in HEX

2er-Potenz	Dezimal	Hex	Hex - 1	kByte
2 <sup>0</sup>	1	0x1	0	
2 <sup>1</sup>	2	0x2	1	
2 <sup>2</sup>	4	0x4	3	
2 <sup>3</sup>	8	0x8	7	
2 <sup>4</sup>	16	0x10	F	
2 <sup>5</sup>	32	0x20	1F	
2 <sup>6</sup>	64	0x40	3F	
2 <sup>7</sup>	128	0x80	7F	
2 <sup>8</sup>	256	0x100	FF	
2 <sup>9</sup>	512	0x200	1FF	
2 <sup>10</sup>	1'024	0x400	3FF	1
2 <sup>11</sup>	2'048	0x800	7FF	2
2 <sup>12</sup>	4'096	0x1000	FFF	4
2 <sup>13</sup>	8'192	0x2000	1FFF	8
2 <sup>14</sup>	16'384	0x4000	3FFF	16
2 <sup>15</sup>	32'768	0x8000	7FFF	32
2 <sup>16</sup>	65'536	0x10000	FFFF	64
2 <sup>17</sup>	131'072	0x20000	1FFFF	128
2 <sup>18</sup>	262'144	0x40000	3FFFF	256
2 <sup>19</sup>	524'288	0x80000	7FFFF	512
2 <sup>20</sup>	1'048'576	0x100000	FFFFFF	1024
2 <sup>21</sup>	2'097'152	0x200000	1FFFFFF	2048
2 <sup>22</sup>	4'194'304	0x400000	3FFFFFF	4096
2 <sup>23</sup>	8'388'608	0x800000	7FFFFFF	8192
2 <sup>24</sup>	16'777'216	0x1000000	FFFFFFF	16'384
2 <sup>25</sup>	33'554'432	0x2000000	1FFFFFFF	32'768
2 <sup>26</sup>	67'108'864	0x4000000	3FFFFFFF	65'536
2 <sup>27</sup>	134'217'728	0x8000000	7FFFFFFF	131'072
2 <sup>28</sup>	268'435'456	0x10000000	FFFFFFF	262'144
2 <sup>29</sup>	536'870'912	0x20000000	1FFFFFFF	524'288
2 <sup>30</sup>	1'073'741'824	0x40000000	3FFFFFFF	1'048'576
2 <sup>31</sup>	2'147'483'648	0x80000000	7FFFFFFF	2'097'152
2 <sup>32</sup>	4'294'967'296	0x100000000	FFFFFFF	4'194'304

### 17.3 Branch Flags

Code	Meaning	Flag(s)
AL	Always	-
EQ	Equal	Z==1
NE	Not equal	Z==0
CS	Carry set	C==1
CC	Carry clear	C==0
MI	Minus/negative	N==1
PL	Plus/positive or zero	N==0
VS	Overflow	V==1
VC	No overflow	V==0
<b>Unsigned</b>		
HS	Higher or same	C==1
LO	Lower	C==0
HI	Higher	C==1 and Z==0
LS	Lower or Same	C==0 and Z==1
<b>Signed</b>		
GE	Greater than or equal	N==V
LT	Less than	N!=V
GT	Greater than	Z==0 and N==V
LE	Less than or equal	Z==1 and N!=V

### 17.4 Wahrheitstabellen

AND	OR	EXOR	NAND	NOR	XNOR
A	B	Z	A	B	Z
0	0	0	1	0	1
0	1	1	0	1	0
1	0	1	1	0	1
1	1	1	1	1	1