2018

Zusammenfassung CTIT1

SEP FS 2019

PASCAL BRUNNER

1 Inhaltsverzeichnis

2 Grundlegendes

- Ein **Computer System** ist ein Gerät, dass
  - Input verarbeitet
  - Entscheidungen aufgrund der Resultate trifft
  - Wiedergibt die verarbeiteten Informationen
- **Hardware** und **Software** arbeitet zusammen → **Applikation**
  - Eine gängige Hardware wird für mehrere Applikationen verwendet
  - Applikationen sind von der Software definiert

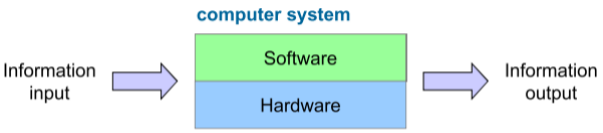
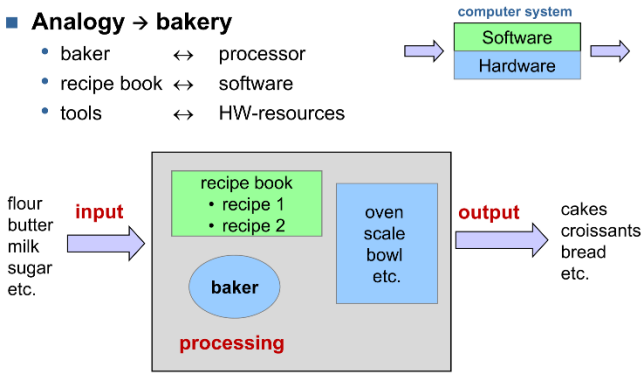


Abbildung 1 Grafik Zusammenspiel Computer System

2.1 Analogie Bäckerei



2.2 Von Neumann Architektur

Viele der heutigen Computer basieren auf der von Neumann Architektur

- **Anweisungen** und **Daten** sind im gleichen Speicher
- **Datenpfand** führt arithmetische und logische Operationen und speichert die Zwischenresultate
- **Control Unit** liest und interpretiert die Anweisungen und kontrolliert die Ausführung

Abbildung 2 Grundgedanke von Neumann

3 Hardware Komponenten

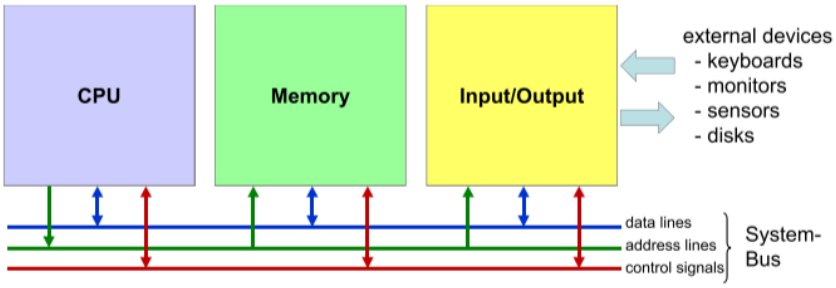
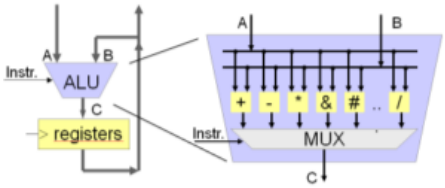


Abbildung 3 Schema der Hardware Komponenten

3.1 CPU

Der CPU besteht aus den zwei grundlegenden Komponenten **Datenpfad** und **Control Unit**

3.1.1 Datenpfad

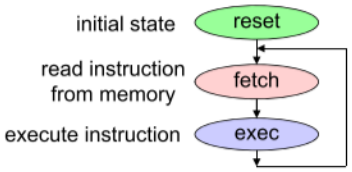


Die **ALU** führt arithmetische / logische Operationen aus. Das **Register** ist ein schneller Zwischenspeicher, welcher jedoch auf die Speichergrösse des CPUs limitert ist.

➔ 4 / 8 / 16 / 32 / 64 Bit-Archiekturen

Abbildung 4 Schema Datenpfad

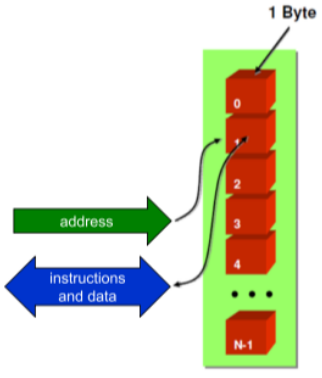
3.1.2 Control Unit



Besteht aus der Finite State Machine (FSM), welche Anweisungen liest und ausführt. Mögliche Operationen sind **Daten Transfer** (register <-> memory), **arithmetische und logische Operationen** und **jumps**

Abbildung 5 Schema FSM

3.2 Memory



Memory speichert **Anweisungen** und **Daten**. Der Speicher ist jeweils eine Aneinanderreihung von Speichereinheiten (eine Einheit = 8 Bit = 1 Byte), dabei kann man jedes einzelne Byte einzeln addressieren.

Potenzschreibweise	Effektiver Wert
2^{10}	1'024 Byte (1KB)
2^{20}	1'048'576 Byte (1MB)
2^{30}	1'073'741'824 Byte (1GB)

Memory ist ein flüchtiger Speicher und demnach sind die Information nach einem Neustart verloren.

Abbildung 6 Schema Memory

3.2.1 Main Memory / Arbeitsspeicher

Der Arbeitsspeicher ist ein zentraler Speicher, welcher über ein System-Bus verbunden ist. Auch hier lässt sich jedes einzelne Byte ansteuern. Man unterscheidet zwischen **volatile** (flüchtiger) Speicher → SRAM und DRAM und **non-volatile** (nicht-flüchtig) Speicher → ROM und flash

3.2.1 Secondary storage / zusätzlicher Speicher

Für Langzeit- oder Peripheriespeicher greift man auf den secondary storage (nicht flüchtiger) zu. Dieser ist über I/O-Ports verbunden. Der Zugriff ist hier auf Datenblöcke möglich. Der secondary storage ist zwar langsamer, dafür auch billiger. Es gibt verschiedene Arten von secondary storages:

Secondary storage-Art	Beispiele
magnetisch	Hard disk, tape, floppy
Halbleiter (Semiconductor)	SSD
Optisch	CD, DVD
Mechanisch	Punched tape / card

3.3 System-Bus

Der CPU schreibt oder liest Daten von/zum Speicher oder I/O

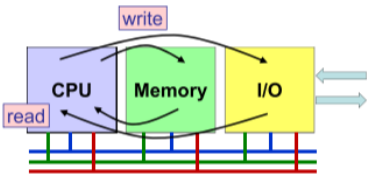


Abbildung 7 Schema System-Bus

3.3.1 Address lines

Der CPU schreibt die anzusprechende Adresse auf die Adress-Linie.
Die Anzahl von Adressen ist 2^n, wobei n = Anzahl Adress-Linien.
2^{16} = 65'536 Adresse -> 64KBytes

3.3.2 Control Signals

Der CPU teilt mit ob er schreiben oder lesen will und der CPU teilt mit wenn eine address lines oder data lines gültig sind → bus timing

3.3.3 Data Lines

Transferiert die Daten → Analogie: Der Brief, welcher im Umschlag ist. Beim **Schreiben** stellt der CPU Daten zur Verfügung und der Speicher empfängt Daten. Beim **Lesen** stellt der Speicher Daten zur Verfügung und der CPU empfängt diese. Es kann 4 / 8 / 16 / 32 / 64 Data-Lines geben.

4 Von C-Code zu einem ausführbaren Programm

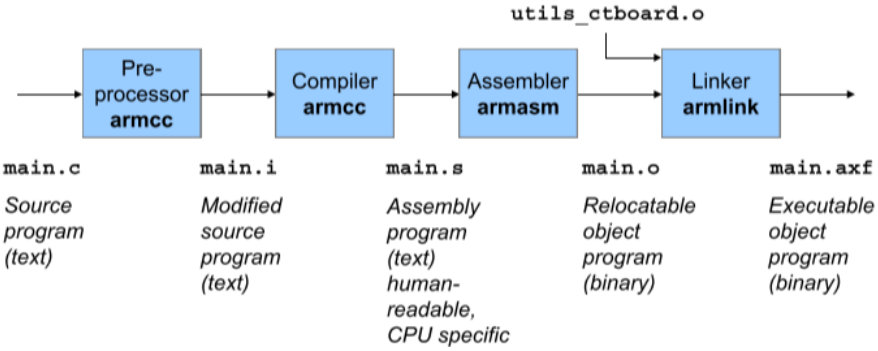


Abbildung 9 Ablauf von C zu einem ausführbaren Programm

4.1 Preprozessor

- Text wird verarbeitet
- Fügt sämtliche #include files ein
- Ersetzt sämtliche Macros (#define)

Resultat: main.i

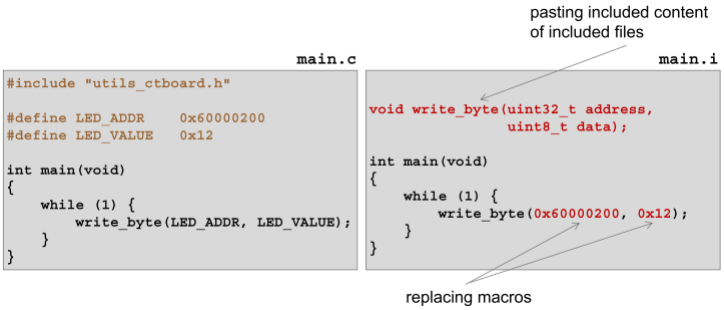


Abbildung 8 nach dem Preprozessor

4.2 Compiler

- Übersetzt den CPU unabhängigen C-Code in einen CPU-spezifischen Assembly-Code (immer noch lesbar für einen Menschen)

Resultat: main.s

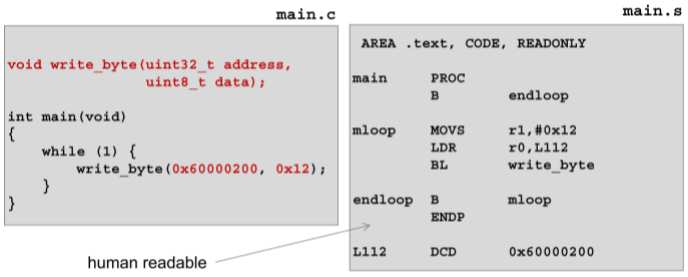


Abbildung 10 Beispiel Übersetzung in Assembly

4.3 Assembler

- Code wird zu Maschinen-Anweisungen übersetzt

Resultat: main.o → verschiebbare Objektdatei → Binäre Datei und mit einem Hex-Dump lesbar

Abbildung 11 Übersetzung zur Maschine-Anweisung

4.4 Linker

- Fügt die Objektdateien zusammen
- Löst Abhängigkeiten und Cross-Referenzen auf
- Erstellt ein executable

Resultat: main.axf

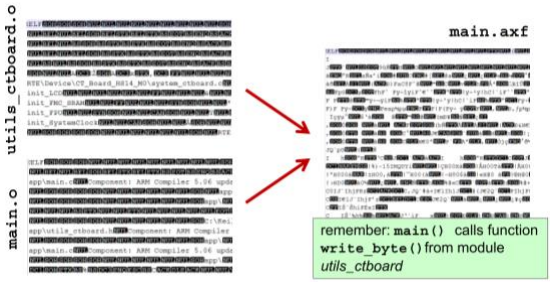


Abbildung 12 Linker-Step

Merke!

Wenn man einen Prozessor neustartet, dann beginnt der Prozessor immer bei einem bestimmten Programmpunkt

5 Host vs. Target

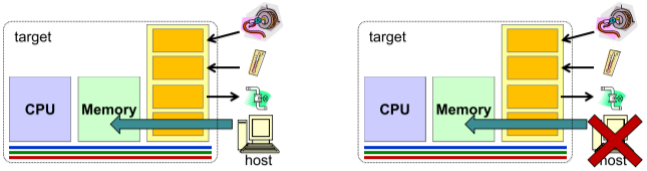


Abbildung 13 Schema Host vs. Target

5.1 Mit Host

- Der Compiler / Assembler / Linker sind auf dem Host
- Loader lädt Executable (bspw. *.axf) wird vom RAM
- Loader kopiert executable vom RAM zu einem nicht flüchtigen Speicher (FLASH)

➔ Firmware Update

5.2 Ohne Host

- Loader springt direkt zum *main()* und startet die Ausführung
- Die Fetch-Anweisung erfolgt oftmals direkt vom FLASH

6 Kombinatorische und sequentielle Logik

Kombinatorische Logik: Ist es ein Schaltjahr? Ist dividierbar durch 4 UND nicht durch 100 teilbar ODER durch 400 teilbar → Stateless

Sequentielle Logik: Je nach aktuellem Zustand und einem bestimmten Input wird entschieden, was der entsprechende Input ist. → Stateful

6.1 Kombinatorische Logik

In der kombinatorischen Logik werden die Basic-Operationen eingeführt. Der Zustand eines logischen Zustandes. Der Output ist abhängig von dem Input und der internen logischen Funktion, des Weiteren ist der Output nach einer kurzen Verzögerung stabil. Wenn man N Inputs hat, welche true or false sein können, hat man 2^N mögliche Input-Kombinationen. Das gesamte System funktioniert ohne Speicher, sprich es hat keine Speicherelemente.

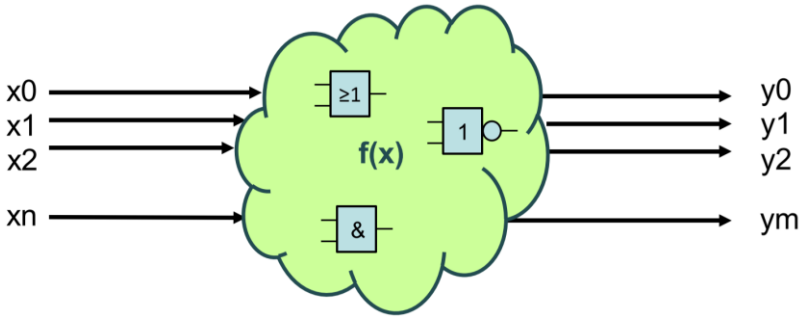
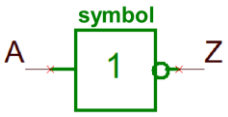


Abbildung 14 kombinatorische Logik



logic equation

Z = !A

truth table

A	Z
0	1
1	0

Abbildung 15 Inverter

Z = A

A	Z
0	0
1	1

Abbildung 16 Buffer

Z = A & B

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

Abbildung 17 AND-Verknüpfung

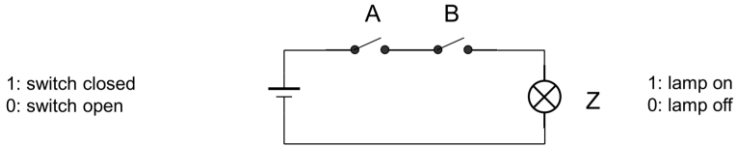


Abbildung 18 AND-Schaltung

Z = A # B

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

Abbildung 19 OR-Verknüpfung

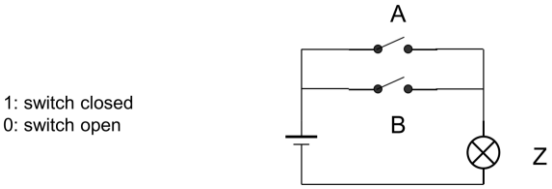
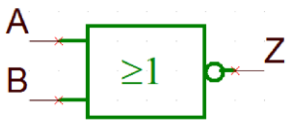


Abbildung 20 OR-Schaltung

Z = !(A & B)

A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

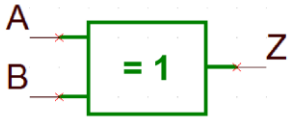
Abbildung 21 NAND



Z = !(A # B)

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

Abbildung 22 NOR



Z = A $ B

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0

Abbildung 23 EXOR

Z = !(A $ B)

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

Abbildung 24 EXNOR

Ein **Multiplexer** ist eine Schaltung mit mehreren Eingängen und einem Selektor (sagt welche Werte am Ausgang erscheinen)

0 & X = 0 **0 $ X = X**

1 & X = X **1 $ X = !X**

Ein 1-Bit Halb-Addierer hat zwei Eingänge, wobei es jeweils einen Übertrag (Carry) gibt wenn beide Eingänge eine 1 haben. Jedoch kann man da dann nicht mehrere Zahlen zusammenzählen

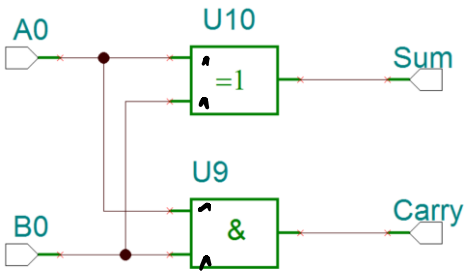


Abbildung 25 1-Bit Halb-Addierer

Aus diesem Grund gibt es den 1-Bit Voll-Addierer (FA). Das Carry des einen dient zur Summierung des anderen. Diese kann man dann durch beliebig viele n-Bit Addierer ergänzen

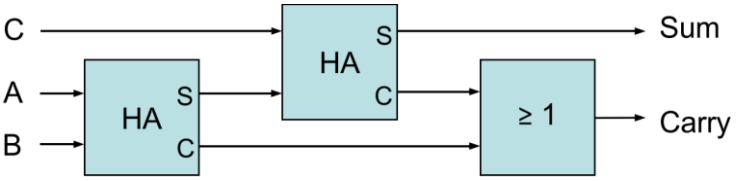


Abbildung 26 1-Bit Voll-Addierer

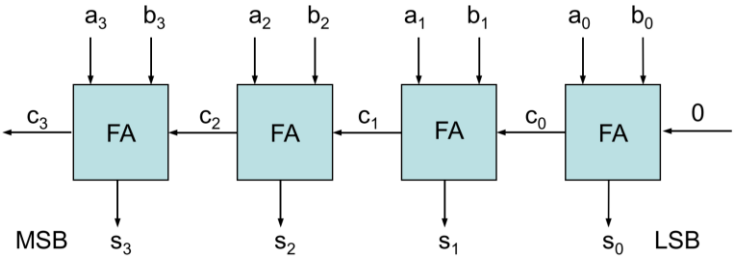


Abbildung 27 4-Bit Voll-Addierer

6.2 Sequentielle Logik

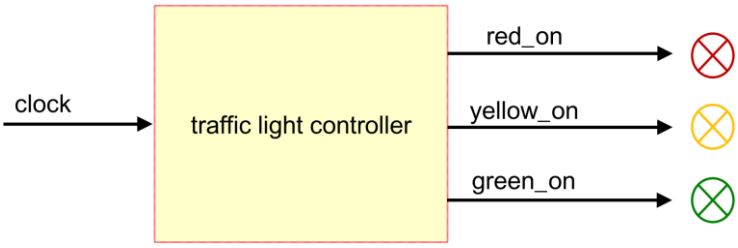


Abbildung 28 Beispiel sequentielle Logik

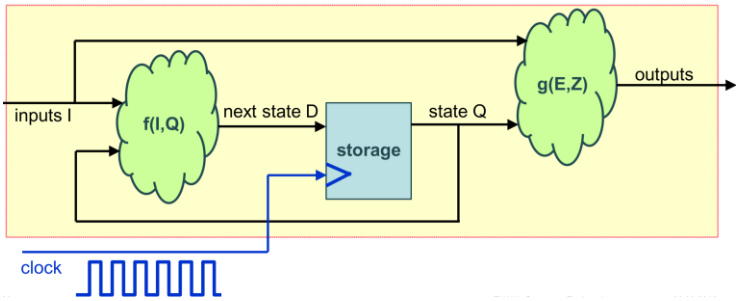
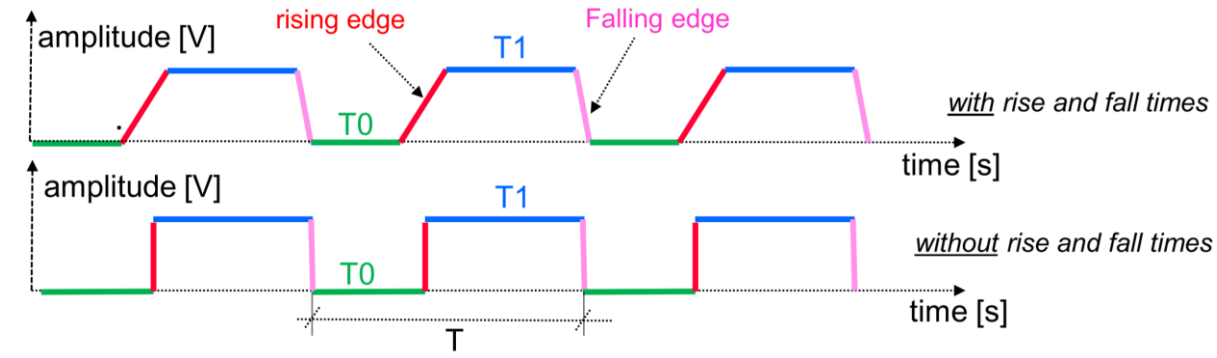


Abbildung 29 Schema sequentielle Logik

Grundsätzlich hat jede sequenzielle Logik einen Input, welche dann in einer Logik (Wolke) verarbeitet wird. Dies wird anhand einer Zustandsmaschine (Finite State Machine) verarbeitet, dieser Output wird im Speicher gespeichert. Diese Art ist stateful, da sie ein «Gedächnis» haben.

6.2.1 Clock Signal

Das Timing-Diagramm wird in Form «without rise and fall times" gezeichnet.

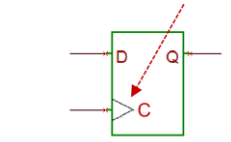


period T = T0 + T1 [s] **frequency** f = 1/T [Hz] **duty cycle** = T1/T [-]

Abbildung 30 abstract timing respresentation

6.2.2 D-Flip-Flop

Wenn mein der Clock-Flanke eine Steigung kommt, dann wird dieser Zustand, welcher aktuell im D ist für Q übernommen. Bei allen anderen wird keine Änderung von Q vorgenommen



value at input D is stored and transferred to output Q, if C changes from 0 to 1 (0 → 1)

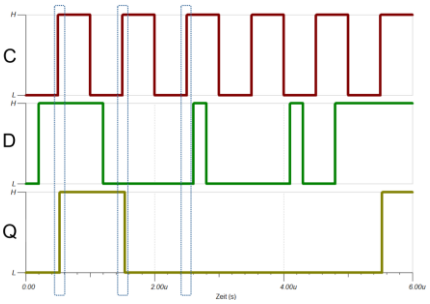


Abbildung 31 D-Flip-Flop

Ein Flip-Flop kann zwei Status repräsentieren, zwei Flip-Flops können vier Status repräsentieren, n-Flip-Flops können 2^n Status repräsentieren.

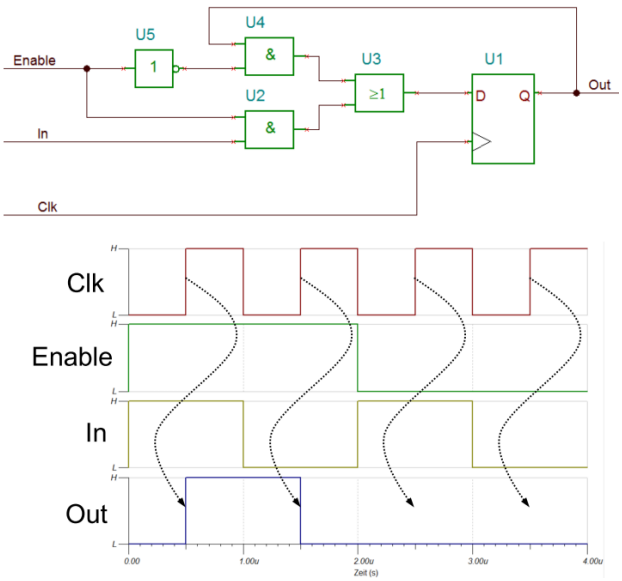


Abbildung 32 Flip-Flop mit Multiplexer

6.2.3 Counter

Counter sind spezielle sequenzielle Schaltungen. Der nächste Schritt ist nur vom aktuellen Status abhängig → Input ist irrelevant

Gray Counter

Wir vor allem für die Kommunikationstechnik verwendet und dient zur Fehlererkennung. Denn der Gray-Counter wechselt immer nur ein Signal → Wenn zwei Änderungen vorgenommen wurden, so ist ein Fehler aufgetreten

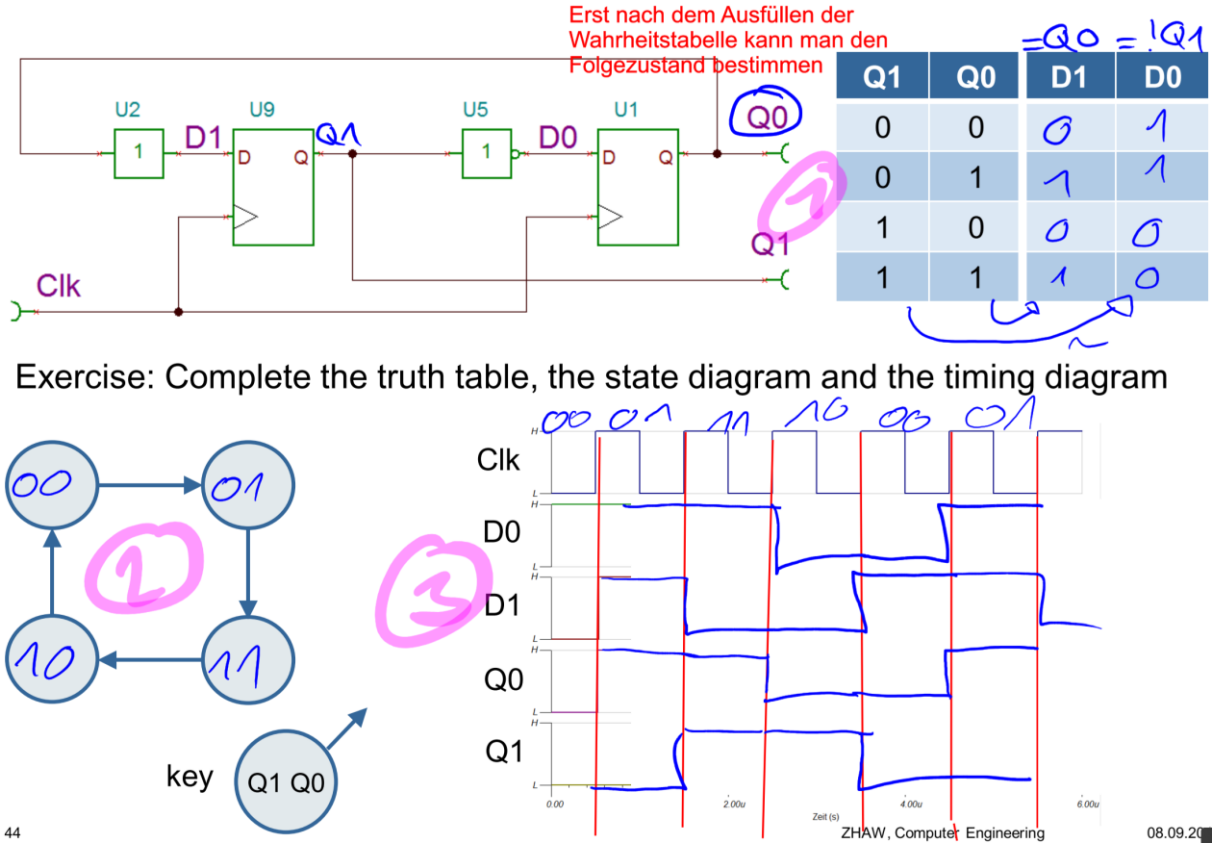


Abbildung 33 Schulbeispiel

6.2.4 Register

Ein Register ist ein Speicherbaustein

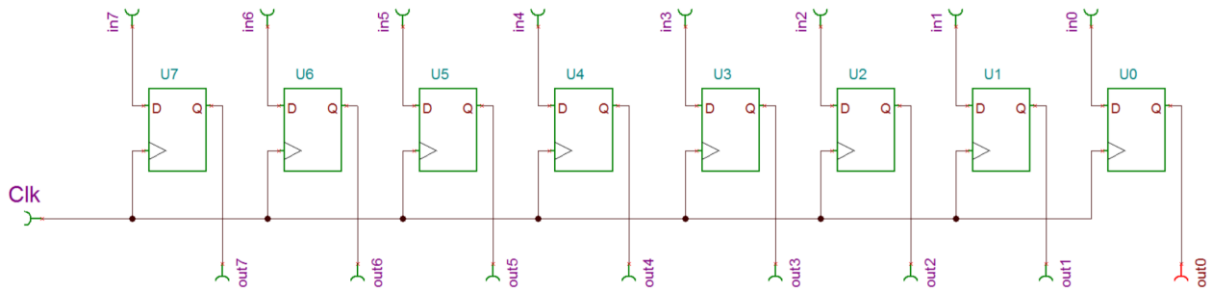


Abbildung 34 Beispiel 8-Bit Register

Schieberegister

Wird oftmals parallel gelesen und alles seriell ausgegeben. Wird sehr oft in der seriellen Kommunikation verwendet.

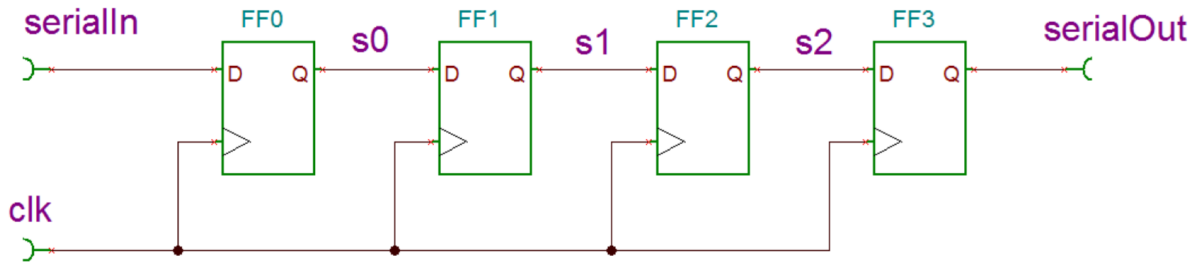


Abbildung 35 Beispiel 4-Bit Schieberegister

7 Architektur

7.1 CPU Model

7.1.1 Core Register

- 16 Core Register à 32 Bit → 13 können für den allgemeinen Gebrauch verwendet werden
  - R15 → Program Counter zeigt auf die nächste Instruktion / Befehl
  - R13 → Stack Pointer LIFO Prinzip
  - R14 → Link Register speichert die Rücksprungadresse
- Speicherstellen, welche nicht direkt addressierbar sind

7.1.2 ALU

- 32 Bit grosse Process Unit A & B als Input Result C
- Integer Artihmetik möglich
  - Addition / Subtraktion
  - Multiplikation / Division
- Logische Operationen
- Bitshifting / Rotationen

7.1.3 Control Unit

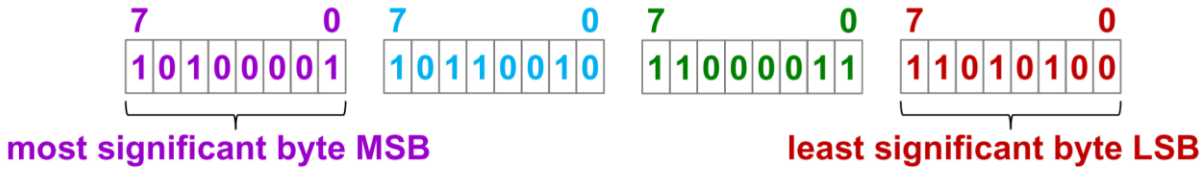
- Zusätzliches Register

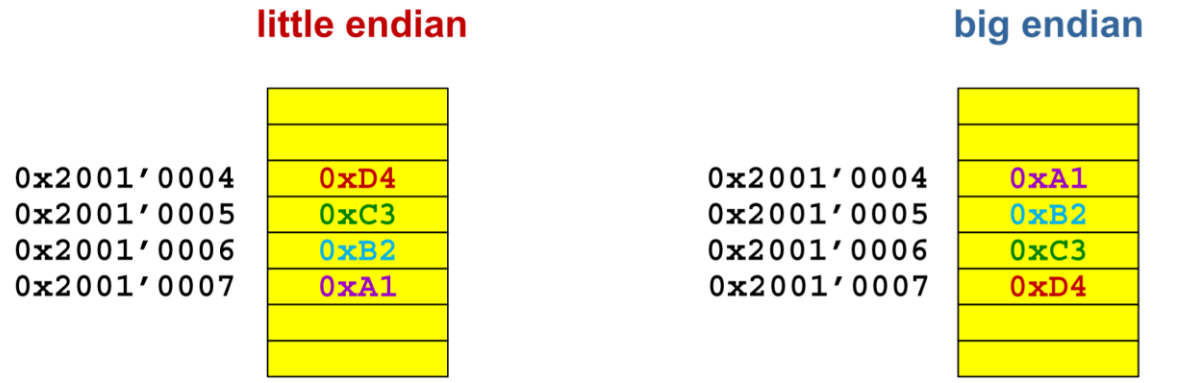
7.2 Assembler Program

Label	Instr.	Operands	Comments
demoprg	MOVS	R0,#0xA5	; copy 0xA5 into register R0
	MOVS	R1,#0x11	; copy 0x11 into register R1
	ADDS	R0,R0,R1	; add contents of R0 and R1
			; store result in R0
	LDR	R2,=0x2000	; load 0x2000 into R2
	STR	R0,[R2]	; store content of R0 at
			; the address given by R2

Abbildung 36 einfaches Assembler Program

- Store Word `0xA1B2'C3D4` at Address `0x2001'0004`





7.3 Program Execution

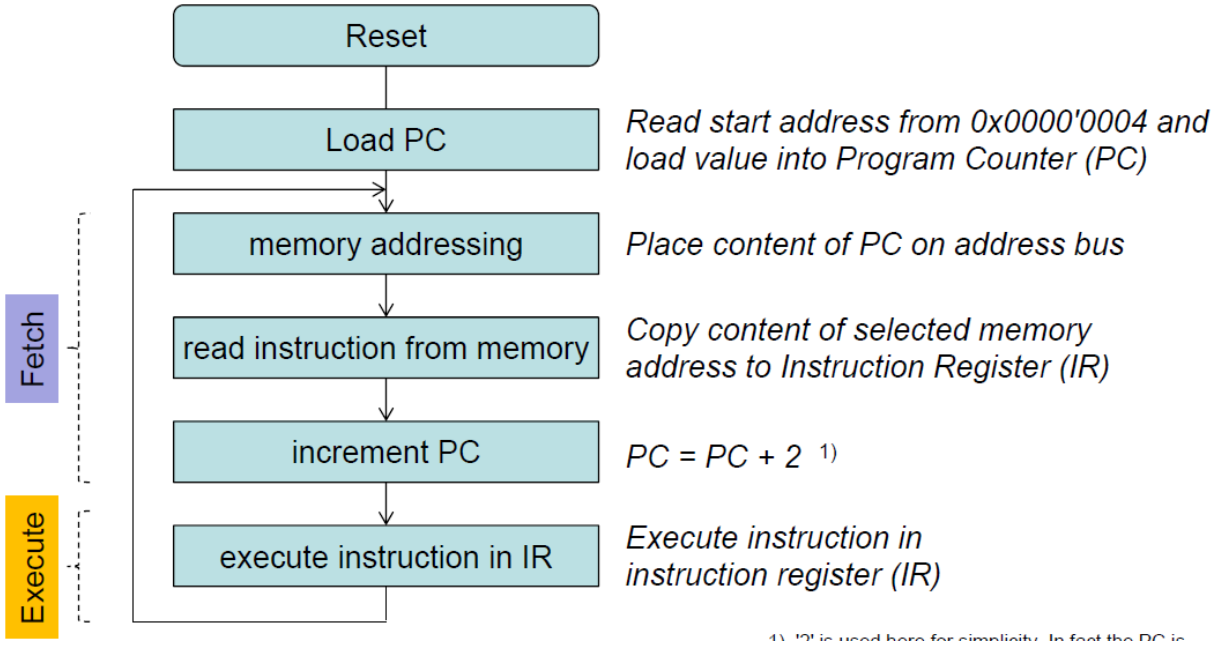


Abbildung 37 Program Execution

8 Data Transfer Instruction

ARM ist eine typische Load/Store Architektur → man muss zuerst die Daten in ein Register laden, führt die Operation durch und zum Schluss wird das Resultat in das Memory gespeichert

8.1 Befehle

MOV / MOVS

Kopiert einen Registerwert in ein anderes Register

MOV → low and high registers

MOVS → only low registers (0 – 7 Register) S = update of flags

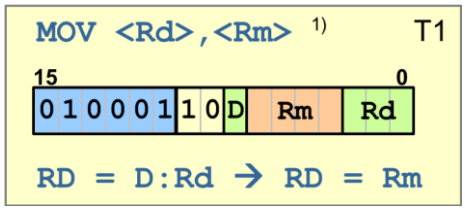


Abbildung 38 Object-Code MOV

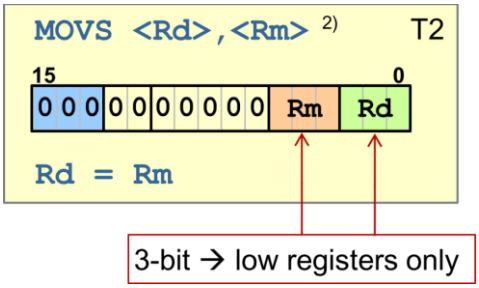


Abbildung 39 Object-Code MOVS

EQU

Analog DEFINE in C

LDR

Load Register

Laden immer ein ganzes Word

LDRB

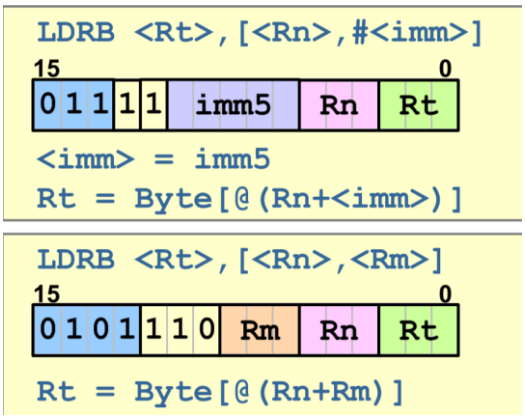


Abbildung 40 Object-Code LDRB

LDRH

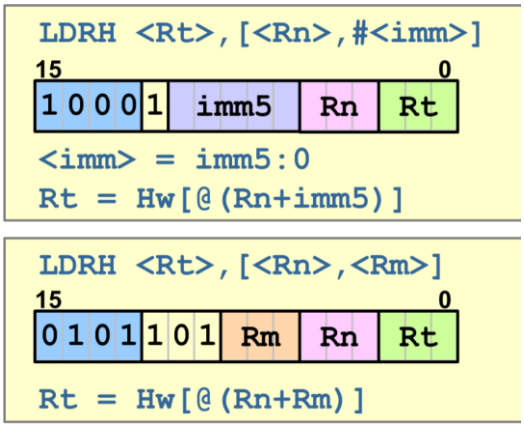


Abbildung 41 Object-Code LDRH

STR

Speichert Daten

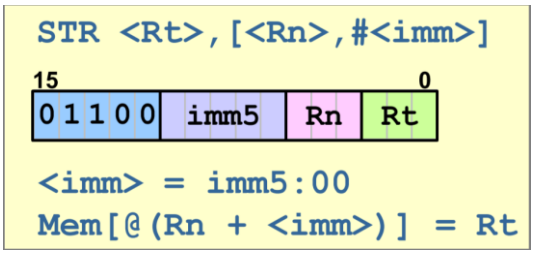


Abbildung 42 Object-Code STR

9 Arithmetische Operationen

9.1 Flags

Für den Prozessor spielt es keine Rolle ob die Zahl unsigned oder signed ist. APSR steht für Application Program Status Register

Dabei gibt es vier grundsätzliche Funktionalitäten:

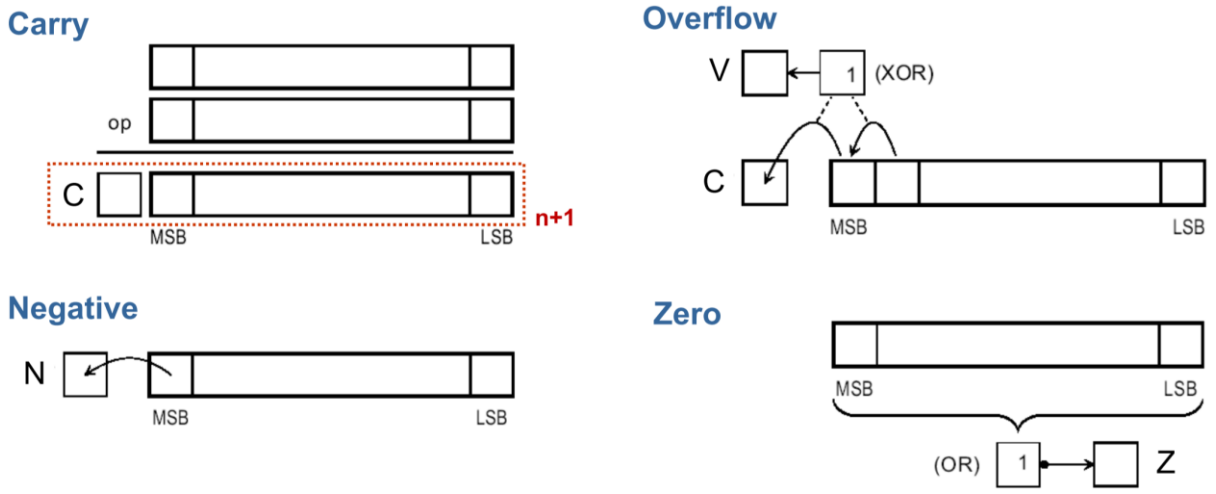
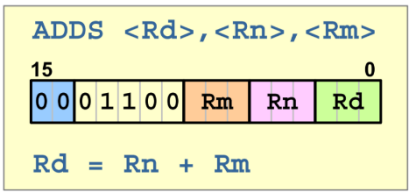


Abbildung 43 Flags

9.1.1 Adds

- Einzig für niederige Register (low register)
- Resultat mit zwei Operanden
- Flag wird jeweils updated




```
00000002 18D1      ADDS      R1,R2,R3
00000004 1889      ADDS      R1,R1,R2
00000006 1889      ADDS      R1,R2       ; the same (dest = R1)
00000008           ;ADDS     R9,R2       ; not possible (high reg)
00000008           ;ADDS     R1,R10      ; not possible (high reg)
```

Abbildung 44 Beispiele für Adds

Wichtig! Bei Unsigned spielt das Carry-Flag eine Rolle. Bei Signed spielt der Overflow eine Rolle, der Overflow ist nur möglich wenn man zwei Zahlen mit gleichem Vorzeichen addiert. Das Zero-Flag spielt bei beiden eine Rolle. Das negative-Flag spielt nur bei Signed eine Rolle

Bei der Addition bedeutet das Carry-Flag dass das Resultat ungültig ist.

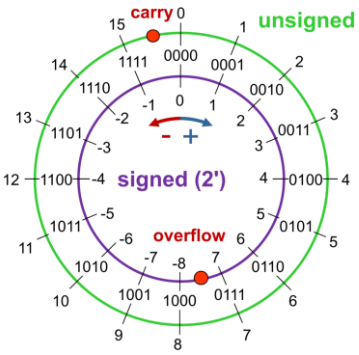
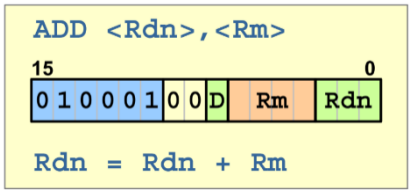


Abbildung 45 Signed / Unsigned mit Carry / Overflow

9.1.2 Add

- Flag wird **nicht** updated
- Hohe und niederige Register (high and low register)
- <Rdn> -> Resultat und Operand




```
00000008 4411     ADD       R1,R1,R2     ; low regs
0000000A 44D1     ADD       R9,R9,R10    ; high regs
0000000C 44D1     ADD       R9,R10       ; the same (dest = R9)
0000000E 4411     ADD       R1,R1,R2
0000000E          ;ADD      R1,R2,R3     ; not possible
```

Abbildung 46 Beispiele für Add

9.1.3 SUBS

Bei der Subtraktion wird das zweite Element negativ dargestellt und danach wird eine normale Addition durchgeführt → Rn + (-Rm), wobei -Rm das Zweierkomplement ist.

Bei der Subtraktion bedeutet das Carry-Flag dass das Resultat gültig ist.

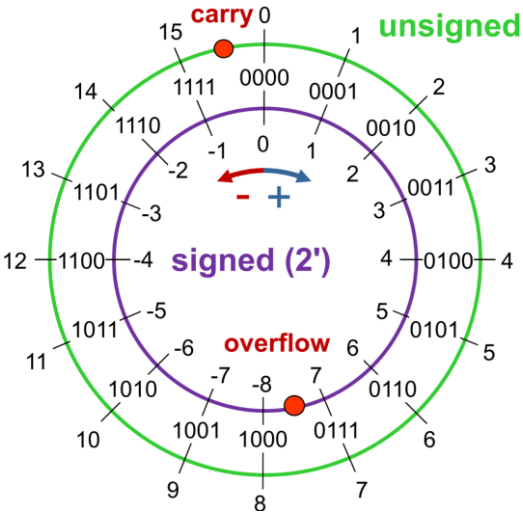


Abbildung 47 Subtraktion

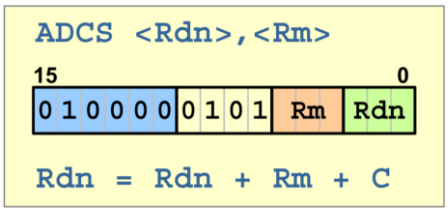
9.1.4 Unsigned Interpretation

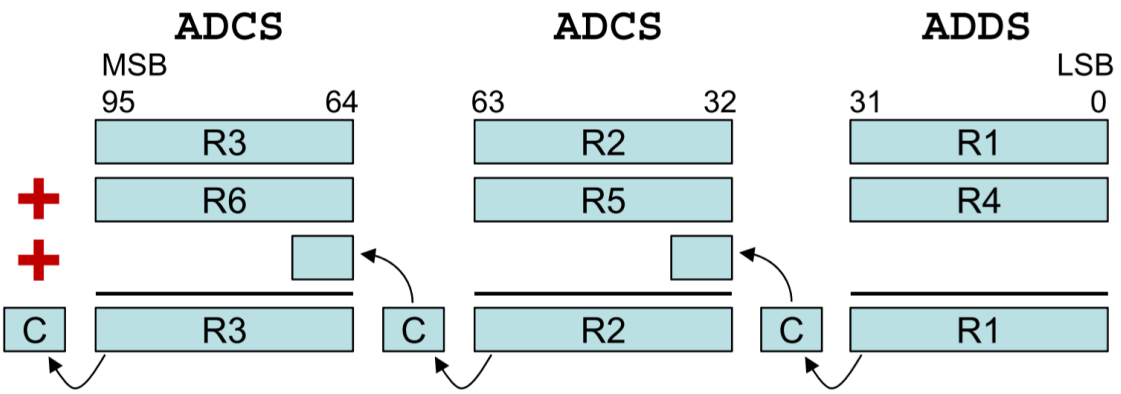
9.1.5 Signed Interpretation

9.1.6 Multi-Word Addition ADCS

Wie addiert man zwei Multi-Word Zahlen

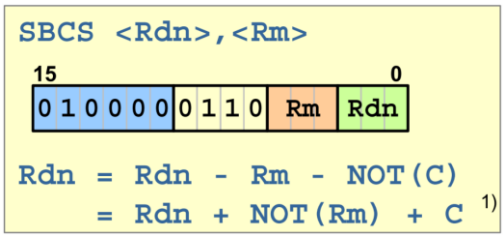
Prüfungsrelevant

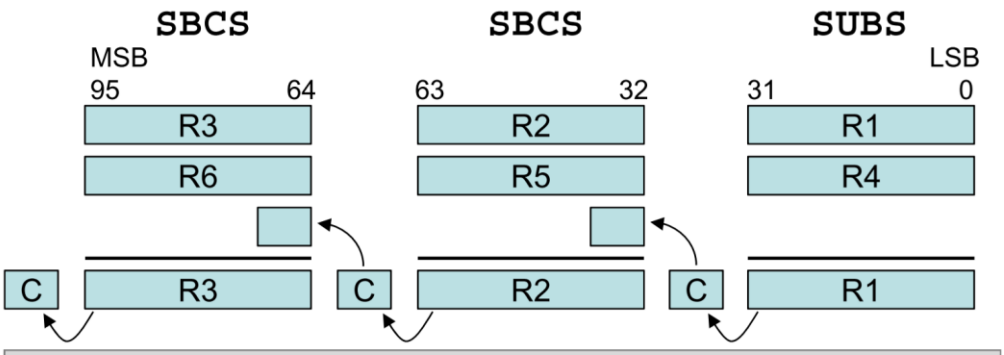





```
; Operand A:          96-bit in R3(MSW),R2,R1(LSW)
; Operand B:          96-bit in R6(MSW),R5,R4(LSW)
; Result = A + B:    96-bit in R3(MSW),R2,R1(LSW)
00000028 1909     ADDS     R1,R1,R4
0000002A 416A     ADCS     R2,R2,R5
0000002C 4173     ADCS     R3,R3,R6
```

9.1.7 Multi-Word Subtraction SBCS






```
; Operand A:     96-bit in R3(MSW),R2,R1(LSW)
; Operand B:     96-bit in R6(MSW),R5,R4(LSW)
; Result A - B:  96-bit in R3(MSW),R2,R1(LSW)
0000002E 1B09    SUBS    R1, R1, R4
00000030 41AA    SBCS    R2, R2, R5
00000032 41B3    SBCS    R3, R3, R6
```

9.1.8 Mutliplication

Man kann die beide Bit-Grössen zusammenzählen, um zu erfahren, wie gross das Resultat wird.
Bspw: 4 Bit * 4 Bit = 8 Bit grosses Resultat

Schriftliches Multiplizieren

```
0101 * 0011
     0011
    0000
   0011
  0000
 00001111
```

unsigned

```
0101 * 1101
   00001101
  00000000
 00001101
00000000
00001000001
```

zero extension of multiplier

signed

```
0101 * 1101
   11111101
  00000000
 11111101
00000000
10011110001
```

sign extension of multiplier

Interpretation unsigned
5d * 3d = 15d → correct

Interpretation signed
5d * 3d = 15d → correct

Interpretation unsigned
5d * 13d = 65d → correct

Interpretation signed
5d * – 3d = 65d → wrong

Interpretation unsigned
5d * 13d = 241d → wrong

Interpretation signed
5d * – 3d = – 15d → correct

10 Casting in C

Man sollte eine HEX-Zahl erkennen ob die Zahl negativ oder positiv ist. Alles was bei HEX grösser als 8 bei der ersten Zahl ist, ist negativ. Wobei diese Zahl auch gleich die grösse negative Zahl ist.
Dezimalzahlen sind irrelevant

8-bit	hex	unsigned	signed
	0x00	0	0
	...	...	...
	0x7F	127	127
	0x80	128	-128
	...	...	...
	0xFF	255	-1

16-bit	hex	unsigned	signed
	0x0000	0	0
	...	...	...
	0x7FFF	32'767	32'767
	0x8000	32'768	-32'768
	...	...	...
	0xFFFF	65'535	-1

32-bit	hex	unsigned	signed
	0x0000 0000	0	0
	...	...	...
	0x7FFF'FFFF	2'147'483'647	2'147'483'647
	0x8000'0000	2'147'483'648	-2'147'483'648
	...	...	...
	0xFFFF'FFFF	4'294'967'295	-1

Das Casting innerhalb der Zahlen 0 – 7 zwischen unsigned & signed ist kein Problem

binary	unsigned	signed 2' compl.
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

11 Logshift Operations

11.1 Bit Manipulation

➔ Typische Prüfungsaufgabe

Clear bits, e.g. clear bits **5** and **1** in register R1

```
MOVS    R2,#0x22        ;00100010b
BICS    R1,R1,R2
```

Set bits, e.g. set bits **6** und **3** in register R1

```
MOVS    R2,#0x48        ;01001000b
ORRS    R1,R1,R2
```

Invert bits, e.g. invert bits **4**, **3** and **2** in register R1

```
MOVS    R2,#0x1C        ;00011100b
EORS    R1,R1,R2
```

Abbildung 48 Bit-Manipulationen

11.2 Shift / Rotate

Mnemonic	Instruction	Function
LSLS	Logical Shift Left	$2^n \bullet Rn$ $0 \rightarrow LSB$
LSRS	Logical Shift Right	$2^{-n} \bullet Rn$ $0 \rightarrow MSB$
ASRS	Arithmetic Shift Right	$2^{-n} \bullet \pm A$ $MSB \rightarrow MSB$
RORS	Rotate Right	$LSB \rightarrow MSB$

Abbildung 49 Shift / Rotate Befehle

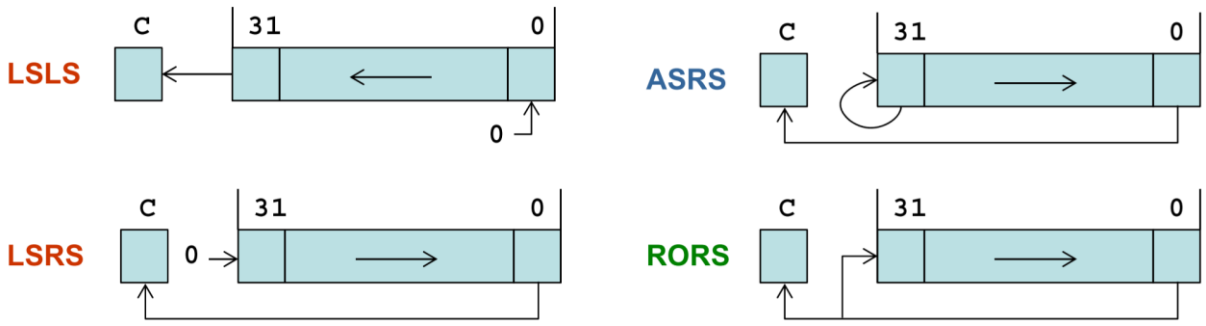


Abbildung 50 Bildliche Erklärung Shift / Rotate

12 Branches / Sprünge

BEQ = «Branch if Equal»

Type = **unconditional** (branch always) oder **conditional** (branch only if condition is met)

Zieladresse = **relative** (target address relative to PC) oder **absolute** (absolute Zieladresse)

Adress hand-over = **direct** (target address of instrucation) oder **indirect** (Zieladresse in Register)

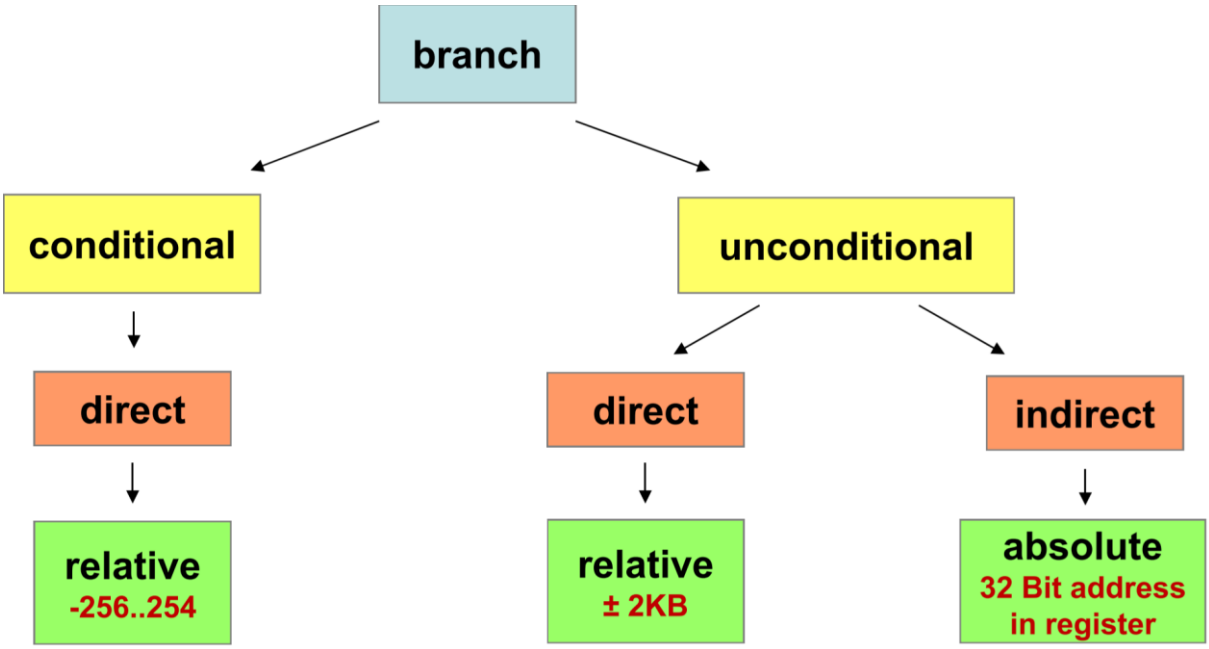
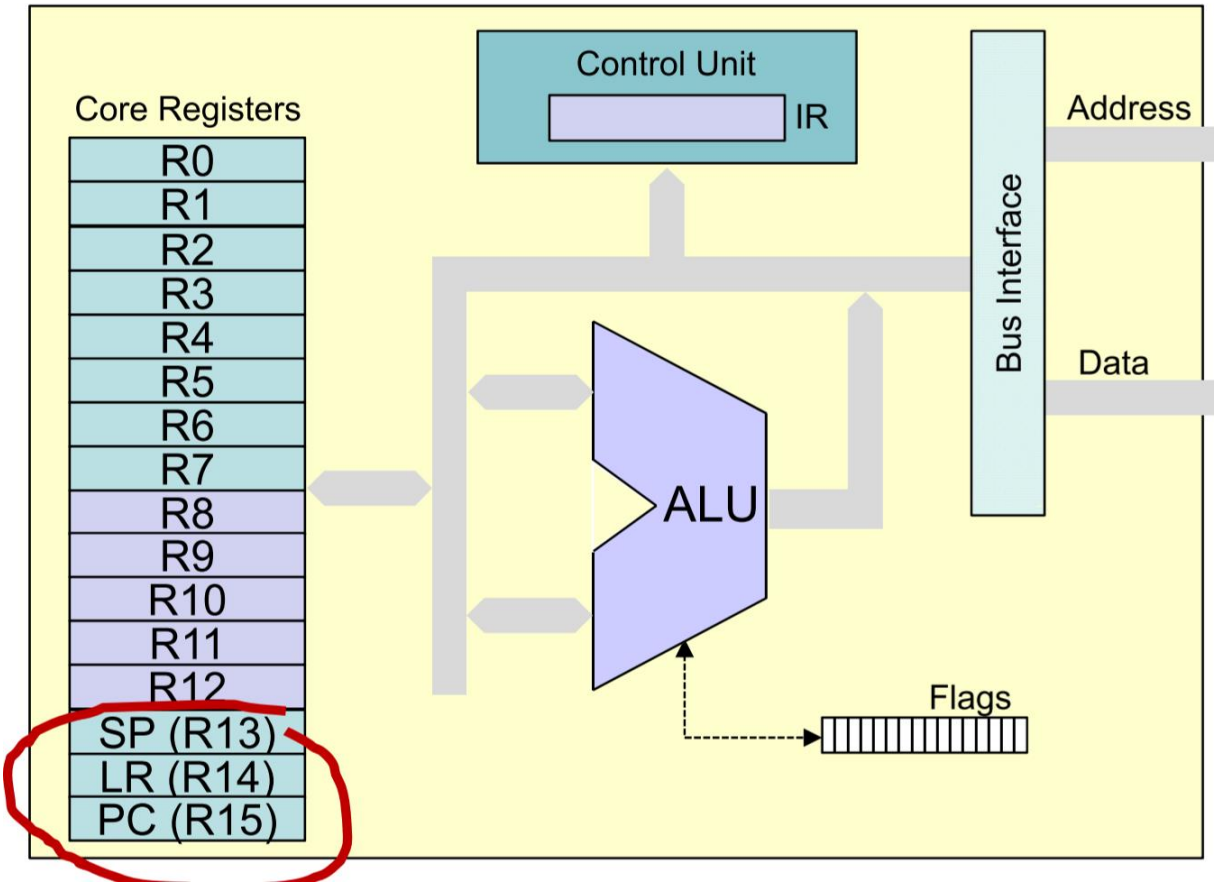


Abbildung 51 Branchübersicht

13 Subtroutinen und Stack

Relevant für die Subroutinen sind folgende Register



PC = Programm Code → Steht nächste Instruktion

Subroutine / Prozeduren / Funktionen / Methoden

- Sequenz von Instruktionen um einen (Sub)Aufgabe auszuführen
- Aufgerufen beim Namen
- Interface und Funktionalität bekannt
- Internes Design und Implementation sind versteckt → Informationen hiding
- Können von verschiedenen Orten aufgerufen werden
- Eine Prozedur gibt keine result values, Funktionen hingegen schon

Stack

Stacke-Base ist die grösste Addresse, Stack-Limit ist die kleinste Addresse. Wird etwas push(), dann wird die Addresse um 4 dekrementiert (immer ein Word), beim pop() wird 4 inkrementiert. Die Stack-Adresse ist immer durch 4 teilbar

Der Stack-Pointer steht beim Starten des Programms bei der Addresse 0x0000'0000

14 Parameter passing

Layout of data	• Size, alignment, layout of fundamental data types
Register Usage	• What are the registers used for
Memory Sections and Stack	• Code, read-only data, read-write data, stack, heap
Stack	• Full-descending, word-aligned, …
Subroutine Calls	• Mechanism using LR and PC
Result Return	• Returning arguments through r0 (and r1 – r3)
Parameter Passing	• Passing arguments in r0-r3 and on stack

15 Glossar

Fachbegriff	Beschreibung
CPU	Central Processing Unit oder Prozessor
ALU	Arithmetic and Logic Unit
FSM	Finite State Machine
RAM	Random Access Memory (read / write)
ROM	Read Only Memory (read)
SRAM	Static RAM
DRAM	Dynamic RAM
I/O	Input/Output
SSD	Solid State Disk
FA	Full-Adder → Voll-Addierer
Periode T	T0 + T [s]
Frequenz f	1 / T [Hz]
Duty cycle	T1 / T [-]

16 Verständnishilfen

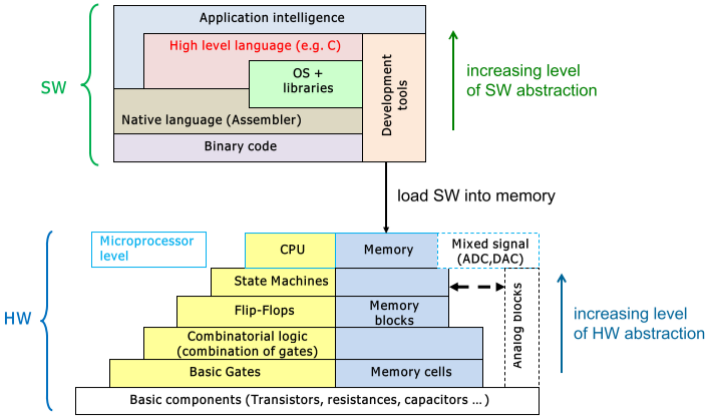


Abbildung 52 Interaktion zwischen Hardware und Software