Definitionen, Bemerkungen & Beispiele

CT_Zusammenfassung

1 Lernziele

1.1 Lecture_Control_Structures

At the end of this lesson you will be able
- to explain the basic concepts of structured programming
- to enumerate and explain the basic elements of a structogram
- to comprehend how a C-compiler implements control structures in assembly language
  - if-then-else
  - do-while loops
  - while loops
  - for loops
  - switch statements
- to program basic structograms in assembly language

1.2 Lecture_Subroutines_and_Stack

At the end of this lesson you will be able
- to explain why subroutines are important in program development
- to explain how a processor stack works
- to interpret and explain the code for subroutine calls in assembly
- to implement leaf and non-leaf functions in Cortex-M assembly
- to explain the difference between leaf and non-leaf functions

1.3 Lecture_Parameter_Passing

At the end of this lesson you will be able
- to outline the basic ideas of the ARM Procedure Call Standard
- to enumerate the different registers in the ARM Procedure Call Standard
- to explain how registers are saved (caller-saved and callee-saved registers)
- to outline what stack frames are and how they are implemented
- to interpret assembly programs and relate register values to function parameters and return values
- to interpret and explain assembly programs which use stack frames
- to implement basic stack frames in Cortex-M0 assembly
- to outline what an Application Binary Interface (ABI) is
- to explain why the ABI is important for modular programming

1.4 Lecture_Modular_Coding_Linking

At the end of this lesson you will be able
- to explain the concepts behind modular programming
- to appropriately partition C and assembly programs into modules
- to explain the steps involved from source to the executable program
- to interpret map files of object files and executable programs
- to explain the main tasks of a linker: merging, resolution, relocation
- to explain the rules the linker applies for resolution and relocation
- to explain the difference between static and dynamic linking
- to explain the concept of source level debugging

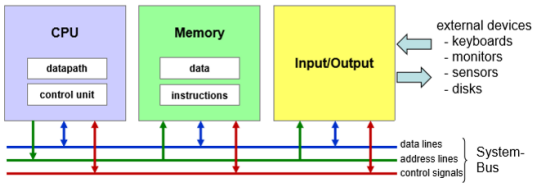
1.5 Lecture_Exceptional_Control_Flow-Interrupts

At the end of this lesson you will be able
- to explain advantages and disadvantages of polling and interrupt-driven I/O
- to distinguish the different types of exceptions on a Cortex-M3/M4
- to explain how the Cortex-M3/M4 recognizes and processes exceptions
- to explain the vector table of the Cortex-M3/M4
- to understand the basic functionality of the Nested Vectored Interrupt Controller (NVIC)
  - to enable and disable interrupts
  - to set and clear interrupts by software
  - to prioritize exceptions
  - to know how programmed priorities influence preemption of service routines
  - to explain how simultaneously pending interrupts are processed
- to implement a simple interrupt service routine in Cortex-M assembly
- to explain potential data consistency issues due to interrupts and to give potential examples

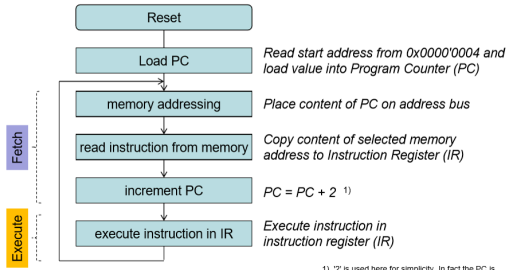
1.6 Lecture_Increasing_system_Performance

At the end of this lesson you will be able
- to explain different types of bus architectures
- to understand the difference between von Neumann and Harvard architecture
- to understand RISC and CISC paradigms
- to describe the idea of pipelining
- to calculate processing performance improvement through pipelining
- to describe the basics of parallel computing

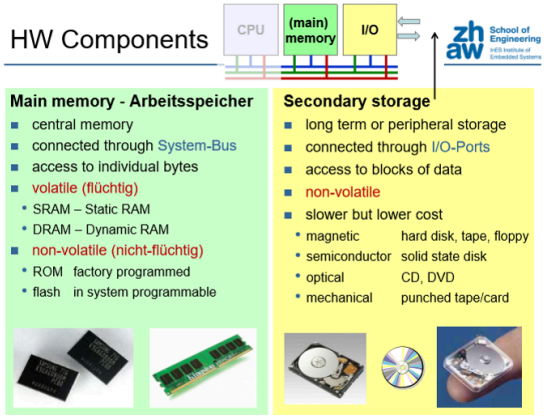
2 Lecture_Computer_Architecture



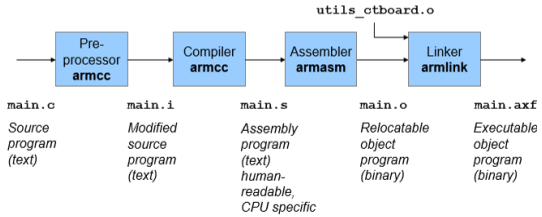
- **CPU - Control Unit**
  - Ist Finite State Machine (FSM)
  - Befehle holen (Fetch), dekodieren (Decode), ausführen (Execute)
    - Fetch holt Instruktion aus Speicher
    - Decode interpretiert Instruktion und erzeugt Steuersignale in der Control Unit
    - Execute führt Instruktion aus (ALU-Operation, Speicherzugriff, ...)
  - Steuerung von ALU, Registern, Speicher, Peripherie



- **CPU - Datapath**
  - ALU (Arithmetic Logic Unit): führt Operationen aus
  - Register: schnelle Zwischenspeicher (CPU-intern)
  - Busse: verbinden Komponenten (Adress-, Daten-, Steuerbus)
- **Memory**
  - Byte-adressiert (1 Adresse pro Byte)
  - Speicherzellen: Byte (8 Bit), Halfword (16 Bit), Word (32 Bit)
  - Speicherbereiche: Code, Data, Stack, Heap, Peripherals
  - 2^N Adressen ⟶ N Bit Adressbus ⟶ 0 ... 2^N-1 Adressen
  - RAM/ROM/Flash (extern, langsamer)



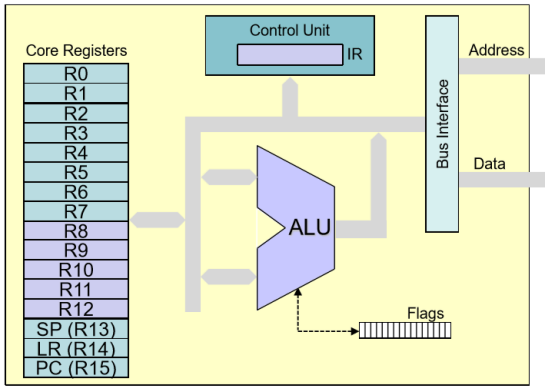
- **Toolchain**



- Host vs. Target (Cross-Toolchain: auf PC bauen, auf Embedded-Target laufen)

3 Lecture_Cortex-M_Architecture

- **Instruction Set Architecture (ISA)**
  - Schnittstelle HW ↔ SW (Befehle, Register, Adressierung, Datentypen, ...)
  - CT1 (Cortex-M0): **32-bit RISC**, Load/Store, Thumb-Subset
- **Architecture**
  - Adressraum: 32-bit Adressen ⟶ Adressraum 0x0000_0000 ... 0xFFFF_FFFF
  - R0–R12: allgemeine Register
  - SP (Zeigt auf zuletzt genutzten Stack-Eintrag), LR (Adresse nach Beenden einer Funktion weiter), PC (Zeigt auf nächsten Befehl)



- **Word alignment**
  - Adressen für Daten müssen auf natürliche Grenzen ausgerichtet sein
  - byte: beliebige Adresse (8-Bit)
  - halfword: gerade Adresse (Vielfaches von 2) (16-Bit)
  - word: durch 4 teilbare Adresse (Vielfaches von 4) (32-Bit)
  - Size of integer ist architectureabhängig, weil Registergröße relevant (CT1: int=32 Bit / 32 Bit Register)

```
                AREA constants, DATA, READONLY
limit           DCD   5
text            DCB   "Hello World"
                DCB   0
text_addr       DCD   text

                AREA variables, DATA, READWRITE
counter         DCD   10
buffer          SPACE 10
colors          DCW   0xFFFF, 0xFF80, 0xFF00, 0xFE80
last_counter    DCD   200
```

Stolpersteine:
- **little endian** (Words/Halfwords werden LSB-first abgelegt)
- **pad** = Padding-Bytes für Alignment

3.1 constants @ `0x08000200`

Bytes (hex)	Bedeutung
`05 00 00 00`	`limit DCD 5` (**little endian**)

`48 65 6C 6C 6F 20 57 6F 72 6C 64 00`	`text DCB "Hello World", 0`
`04 02 00 08`	`text_addr DCD text (= 0x08000204)` **(little endian)**
`? ? ? ?  ? ? ? ?`	`unbenutzt/unknown`

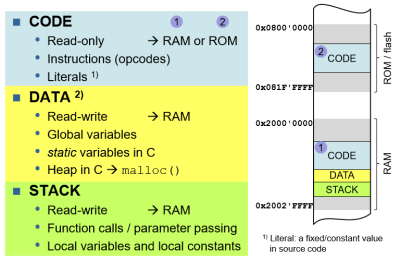
3.2 variables @ `0x20000000`

Bytes (hex)	Bedeutung
`0A 00 00 00`	`counter DCD 10` **(little endian)**
`all 0`	`buffer SPACE 10` **(unknown)**
`FF FF 80 FF 00 FF 80 FE`	`colors DCW FFFF, FF80, FF00, FE80` **(little endian halfwords)**
`pad pad`	`Padding (Alignment vor DCD)`
`C8 00 00 00`	`last_counter DCD 200` **(little endian)**

- **Endianness**
  - STM32 (ST) ist **little endian** (LSB liegt an kleinerer Adresse)

Little Endian (Beispiel 0xA1B2C3D4 @ addr) addr+0: D4 addr+1: C3 addr+2: B2 addr+3: A1 **Big Endian (Beispiel 0xA1B2C3D4 @ addr)** addr+0: A1 addr+1: B2 addr+2: C3 addr+3: D4

- **Cortex-M0 Instruction Set**
  - Typen von Instruktionen: Data Transfer, Arithmetic/Logic, Branch, Control, ...



Address	Byte content (decimal, hex, binary)	Variable
0x2FFFFFF7	0x45	
0x2FFFFFF8	0xE2	Var1 (16-bit short)
0x2FFFFFF9	01100010 (binary)	
0x2FFFFFFA	213	
0x2FFFFFFB	25	
0x2FFFFFFC	0x65	Var2 (32-bit unsigned integer)
0x2FFFFFFD	10101101 (binary)	
0x2FFFFFFE	0xA3	
0x2FFFFFFF	0x82	
0x30000000	0xA2	Var3 (32-bit integer)
0x30000001	34 (dezimal)	
0x30000002	0x54	
0x30000003	0xFF	
0x30000004	0x92	Var4 (unsigned char)
0x30000005	0x03	Var5 (char)

- **Memory Map**
  - Tiefste Adresse oben gezeichnet
  - Berechnung Speichergrösse: Endadresse - Startadresse + 1 (in Bytes)
  - Berechnugn Endadresse: Startadresse + Grösse - 1
  - Typische Sektionen:
    - .text: Code (Programm-Instruktionen)
    - .data: initialisierte Daten (Variablen mit Startwert)
    - .bss: uninitialisierte Daten (Variablen ohne Startwert; werden auf 0 gesetzt)
- **Memory Allocation**
  - Stack: wächst zu kleineren Adressen (downwards)
  - Heap: wächst zu grösseren Adressen (upwards)
  - Data Segment: statische/global Variablen
  - Code Segment: Programmcode
  - DCB: Byte, DCW: Half-word, DCD: Word

```
        AREA example1, DATA, READWRITE
var1    DCB  0x1A
var2    DCB  0x2B, 0x3C, 0x4D, 0x5E
var3    DCW  0x6F70, 0x8192
var4    DCD  0xA3B4C5D6
```

4 Lecture_Datatransfer

- **Data Transfer Types**
  - Register-Register (z.B. `MOV Rd,Rm`)
  - Loading Literals (z.B. `LDR Rd,=literal`)
  - Loading Data from Memory (z.B. `LDR Rd,[Rn,#imm]`)
  - Storing Data to Memory (z.B. `STR Rd,[Rn,#imm]`)
  - **Stolpersteine:**
    - Endianness
    - lower Registers (R0–R7 vs R8–R12)
    - LDR with [PC, `#imm`] ist loading literal
    - LDR Rd,label ladet literal
    - LDR Rd,=literal ist pseudo-instruction für loading literal
    - LDR(X), auf X achten (Word/Halfword/Byte)
    - MOV(S) kann nicht mit pseudo-instructionen umgehen (=literal) -> Alternative LDR Rd,=literal verwenden
    - MOV Rd, `#imm` nur für kleine immediates (0–255)
    - **Daran denken:** bei STRH/LDRH und STR/LDR imm-offset vervielfachen (STRH/LDRH: imm×2, STR/LDR: imm×4)
- **Literal-Pool**
  - Bereich im Code-Segment mit Literalen (Konstanten)
  - LDR Rd,=literal ⟶ lädt Wert aus Literal-Pool
  - Literal-Pool wird automatisch vom Assembler verwaltet
  - Berechnung der grösse des Pools: Anzahl Literale × 4 Byte (Word)
    - Was gehört nicht dazu: ALIGN Direktiven, Labels, Kommentare
- **EQU Directive**
  - `NAME EQU value` definiert Konstanten
  - ersetzt NAME durch value im Code (kein Speicher reserviert)
  - zB `BUFFER_SIZE EQU 0x64` -> `LDR R0,=BUFFER_SIZE` wird zu `LDR R0,=0x64`

- **Adressierungsarten (typisch)**
  - Relative Adressierung: Adresse wird relativ zu PC berechnet
    - z.B. `LDR Rt,[PC,#imm]` (PC-relative)
  - Indirekte Adressierung: Adresse steht in Register
    - z.B. `LDR Rt,[Rn]` (Register Indirect)
- **Pseudo instructions**
  - `LDR Rt,=literal` (Literal Pool)
  - `MOV Rd,=literal` (Literal Pool)
  - `ADR Rd,label` (PC-relative Adresse)
  - `NOP` (keine Operation)
- **Arrays**
  - Abfolge von Elementen gleichen Typs im Speicher
  - Elemente liegen hintereinander (contiguous)
  - Adresse des i-ten Elements: base + i · element_size
  - element_size: byte=1, halfword=2, word=4
  - C-Array-Zugriff: `array[index]` ⟶ `*(array + index)`
  - Compiler übersetzt Array-Zugriffe so in Assembly:

C-code

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

assembly

```
byte_array
        DCB     0xAA,0xBB,0xCC,0xDD
        DCB     0xEE,0xFF
```

address		index
0x2001'0000	0xAA	0
0x2001'0001	0xBB	1
0x2001'0002	0xCC	2
0x2001'0003	0xDD	3
0x2001'0004	0xEE	4
0x2001'0005	0xFF	5

assuming that `byte_array` starts at 0x2001'0000

C-Code

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


void access_byte_array(void)
{
    ...
    byte_array[3] = 0x12;
    ...
}
```

Assembly

```
AREA MyData, DATA, READWRITE
byte_array  DCB 0xaa,0xbb
            DCB 0xcc,0xdd
            DCB 0xee,0xff

AREA MyCode, CODE, READONLY
access_byte_array
                ...
            1 MOVS    r0,#0x12
            2 LDR     r1,adr_b
            3 STRB    r0,[r1,#3]
                ...

adr_b       DCD     byte_array
```

1. Load value to be stored into R0
2. Load base address from label below [1)]
3. Store R0 to base address plus offset

- Achtung: bei mehr als halfword zugriff STR/LDR verwenden!
- **Pointer & Address Operator**
  - `&var` ⟶ Adresse von var
  - `*ptr` ⟶ Wert an Adresse ptr

Pointer and Address Operator

C-Code

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

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

Assembly

```
AREA MyData, DATA, READWRITE
x           DCD     0x00000000
xp          DCD     0x00000000

AREA MyCode, CODE, READONLY
pointer_example
                ...
            1 LDR     r0,adr_x
            2 LDR     r1,adr_xp
            3 STR     r0,[r1,#0]
            4 MOVS    r0,#0xc
            5 LDR     r1,[r1,#0]
            6 STR     r0,[r1,#0]
                ...

adr_x       DCD     x
adr_xp      DCD     xp
```

1. Load address of x → R0
2. Load address of xp → R1
3. Store R0 *(i.e. address of x)* in xp variable *(indirect memory access through R1)*
4. Load immediate value 0x0C → R0
5. Load content of xp → R1 *i.e. address of x is now in R1*
6. Store R0 at address given by R1

5 Lecture_Arithmetic_Operations

- **HW - Addition/Subtraction**
  - Addition: Bitweise Addition mit Carry (Volladdierer-Kette)
  - Subtraction: A - B = A + (B + 1) (Zweierkomplement)
- **SW - Multiword Operation**
  - Zerlege grosse Zahlen in mehrere Wörter (z.B. 64-Bit in 2×32-Bit)
  - Addiere/Subtrahiere Wort für Wort, übertrage Carry/Borrow

Registerbelegung für 64-bit Addition:

- A in R1:R0 (R1 = high32, R0 = low32)
- B in R3:R2 (R3 = high32, R2 = low32)

64-bit Addition: A = A + B

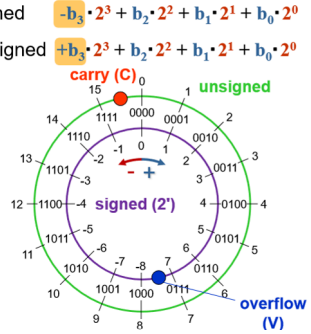
```
ADDS  R0, R0, R2     ; low32  (setzt C)
ADCS  R1, R1, R3     ; high32 + Carry

R0 = low32(Resultat)

R1 = high32(Resultat)

Gesamt: A = R1:R0

Warum zuerst low?
Nur so wird das Carry aus der low-Addition korrekt in
die high-Addition übernommen.
```

Hinweis: Carry aus der high-Addition wäre ein “65stes Bit” und wird hier nicht weiter gespeichert.

- **Einerkomplement**
  - Positive Zahlen: normale Binärdarstellung
  - Negative Zahlen: invertieren
  - Bereich bei n Bit: -(2^n-1 −1) ... 2^n-1 −1
- **Zweierkomplement**
  - Positive Zahlen: normale Binärdarstellung
  - Negative Zahlen: invertieren + 1
  - Bereich bei n Bit: −2^(n-1) ... 2^(n-1)-1
  - Subtraktion: A - B = A + (B + 1) oder (A + (-B)) + 1

signed $-b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$

unsigned $+b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0$



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

- **Flags (APSR)**
  - N: MSB des Resultats = 1
  - Z: Resultat = 0
  - C: Carry/NoBorrow (**unsigned**)
  - V: Overflow (**signed**)
  - CPU unterscheidet **nicht** signed/unsigned ⟶ berechnet C und V immer

5.1 Vorbereitungs-Fragen (immer gleich)

1) **Welche Sicht ist gefragt?**

- **unsigned** ⟶ wichtig ist **C** (carry/borrow)
- **signed (2's complement)** ⟶ wichtig ist **V** (overflow)

2) **Welche Grenze ist kritisch?**

- unsigned 8-bit: **0..255**
- signed 8-bit: **−128..+127** (Hex: 0x80..0x7F)

3) **Resultat schnell?**

- Rechne in Hex **byteweise** und merke: **über 0xFF = Carry**, **unter 0x00 = Borrow**

5.2 Addition: `op1 + op2`

Q1: Ist die Summe ≥ 0x100? ⟶ Ja: `C=1` (Carry raus) ⟶ Nein: `C=0` **Achtung Carry (8-bit):** Wenn Summe über `0xFF` geht, bleibt nur die **letzte Byte** übrig (die letzten 2 Hex-Stellen). Merke: dann ist `C=1` und es ist wie **"Summe – 0x100".**

Resultat (hex) schnell:

- Addiere in Hex. Wenn du "über FF" kommst: **schreib nur die letzten 2 Hex-Stellen** als Resultat, und merke dir `C=1`. (Beispiel: 0x82+0x12 = 0x94, kein Carry)

Q2: Overflow (signed) passiert nur wenn beide gleiches Vorzeichen haben:

- Sind **beide** op1 und op2 im selben Vorzeichenbereich?
  - Positiv: 0x00..0x7F
  - Negativ: 0x80..0xFF

⟶ Nein: `V=0`

⟶ Ja: Schau das Resultat an: hat es plötzlich das **andere** Vorzeichen?

- Ja ⟶ `V=1`
- Nein ⟶ `V=0`

Q3: N und Z (vom Resultat)

- `Z=1` wenn Resultat = 0x00, sonst 0
- `N=1` wenn Resultat im Bereich 0x80..0xFF liegt (MSB=1), sonst 0

5.3 Subtraktion: `op1 - op2`

Q1: Brauchst du ein Borrow? (unsigned Vergleich)

- Ist `op1 >= op2` (unsigned)?
  - Ja ⟶ **kein Borrow** ⟶ `C=1`
  - Nein ⟶ Borrow nötig ⟶ `C=0`

Resultat (hex) schnell:

- Subtrahiere in Hex. Wenn du "unter 00" gehst, **leihe 0x100** (also +256) und merke dir: Borrow war nötig (`C=0`). (Praktisch: du bekommst automatisch wieder eine 2-Hex-Stellen Zahl.)

Q2: Overflow (signed) bei SUB passiert nur wenn Vorzeichen verschieden sind:

- Haben op1 und op2 **unterschiedliches** Vorzeichen?

⟶ Nein: `V=0`

⟶ Ja: Schau das Resultat an: hat es das **andere** Vorzeichen als op1?

- Ja ⟶ `V=1`
- Nein ⟶ `V=0`

Q3: N und Z (vom Resultat)

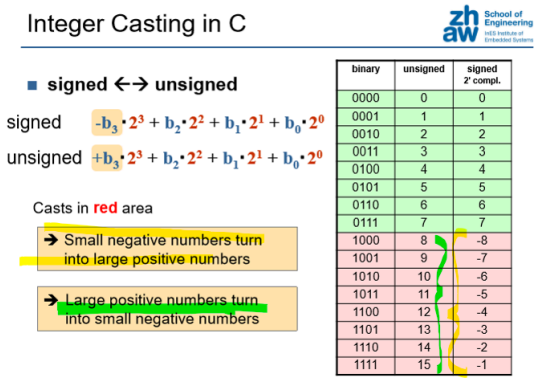
- `Z=1` wenn Resultat = 0x00, sonst 0
- `N=1` wenn Resultat 0x80..0xFF (MSB=1), sonst 0

5.4 Mini-Merker (sehr schnell)

- **C bei ADD:** "über FF hinaus?"
- **C bei SUB:** "musste ich borgen?" ⟶ ja ⟹ C=0, nein ⟹ C=1
- **V bei ADD:** "gleiches Vorzeichen rein, anderes raus?"
- **V bei SUB:** "verschiedene Vorzeichen rein, Resultat kippt gegenüber op1?"

6 Lecture_Casting

- **Casting erklärt**
  - `(type) value` ⟶ Wert wird in anderen Typ umgewandelt
  - implizit (automatisch) oder explizit (durch `(type)`)
  - **Actung!** wenn a = 5 und b = 10 -> (cast)a > (cast)b -> Cast ist nur für Vergleich relevant, a und b bleiben unverändert.
    - Dauerhafter Cast ist nur durch Zuweisung: `b = (uint8_t)a;`
- **Integer casting in C**



Casting Regeln:

- signed to unsigned: Wert bleibt gleich (Bitmuster bleibt gleich)
- unsigned to signed:
  - Wert bleibt gleich : wenn im positiven Bereich;
  - Wert Negativ (1 vorne): **Berechne Wert - 2^n**
- Zu Beachten sind nächsten Punkte:
  - **Extensions** (kleiner ⟶ grösser) (kleiner = weniger Bits)
  - **Truncation** (grösser ⟶ kleiner)

- **Extensions**
  - usigned -> zero-extend (mit Nullen auffüllen)
  - signed -> sign-extend (Vorzeichenbit auffüllen) (Negativzahlen bleiben negativ)
- **Truncation**
  - "Links abschneiden"
  - signed: Vorzeichen kann kippen
  - unsigned: entspricht modulo (Wrap-around)

Cortex-M0 Extend-Instruktionen

- `SXTB` 8⟶32 signed (sign extend)
- `SXTH` 16⟶32 signed
- `UXTB` 8⟶32 unsigned (zero extend)
- `UXTH` 16⟶32 unsigned

- **Stoplersteine bei if-Statements**
  - Vergleiche immer im gleichen Typ (signed/unsigned)

- Sonst falsche Ergebnisse möglich (z.B. −1 > 0 bei unsigned)

7 Lecture_Logic_and_Shift-Rotate_Instructions

Mnemonic	Instruction	Function	C-Operator
ANDS	Bitwise AND	Rdn & Rm	a & b
BICS	Bit Clear	Rdn & !Rm	a & ~b
EORS	Exclusive OR	Rdn $ Rm	a ^ b
MVNS	Bitwise NOT	!Rm	~a
ORRS	Bitwise OR	Rdn # Rm	a ¦ b

flags N = result<31> [1)]
Z = 1 if result = 0
Z = 0 otherwise
C and V unchanged

- ASRS arithmetic right (MSB/Vorzeichen bleibt)
- RORS rotate right (zyklisch)
- **Flag determine**
  - immer bei Instruktionen mit S
    - N: wenn MSB des Resultats = 1
    - Z: wenn Resultat = 0
    - V: immer unchanged (bleibt wie vorher)
  - (ANDS, ORRS, EORS, BICS, MVNS)
    - C: (bleibt wie vorher)
  - (LSLS, LSRS, ASRS, RORS)
    - C:Last bit shifted out (gilt rechts und links)(RORS rechts rausgefallen)

7.1 Multiplikation mit Konstante (Synthese)

7.1.1 Schritt 1: Konstante in Binär zerlegen

- Schreibe K als Binärzahl.
- Markiere alle Bits i, die 1 sind.

`K = 23 = 0b00010111` ⟹ Einsen bei `i = 0,1,2,4` ⟹ `23·x = (x<<0) + (x<<1) + (x<<2) + (x<<4)`

7.1.2 Schritt 2: Addier-Plan erstellen (minimal denken)

- Start: `acc = 0`
- Für jedes gesetzte Bit `i`: addiere `(x<<i)` zum Accumulator.
- **Merke:** Du musst nicht jedes `i` einzeln bauen — du kannst schrittweise shiften. (immer um #1)

7.1.3 Schritt 3: In "Shift + optional Add" übersetzen (wie im Übungs-Template)

Viele Aufgaben geben so ein Muster vor:

- R0 enthält x (der variable Multiplikand)
- R7 ist Akkumulator (Start 0)
- Danach kommt eine Sequenz aus:
  - `LSLS R0, R0, #1` (x wird jeweils verdoppelt)
  - danach **optional** `ADDS R7, R7, R0` (wenn das entsprechende Bit in K = 1 ist)

7.1.4 Tipp für den ersten Shift `LSLS R0, R0, #x`

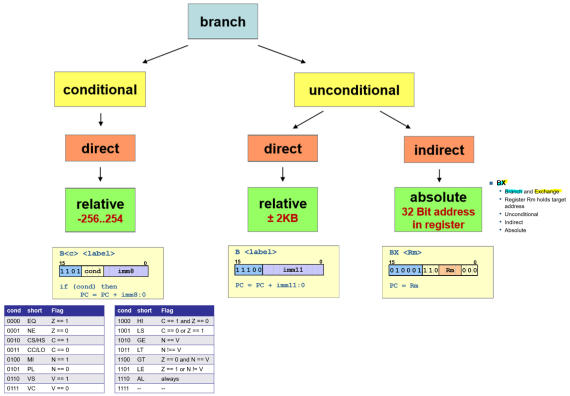
Manchmal beginnt die Vorlage mit einem grösseren Startshift #x, um direkt zum ersten gesetzten Bit zu springen:

- Finde die **kleinste** gesetzte Bitposition `i_min` von K.
- Setze `x = i_min`.
- Danach geht es meist mit #1 Schritten weiter.

Wenn K ungerade ⟶ Bit0=1 ⟶ `i_min=0` ⟹ `x=0` (kein Vorschub nötig) Wenn K gerade (z.B. 40=0b00101000) ⟶ `i_min=3` ⟹ `x=3`

8 Lecture_Branches

8.1 Lecture_Branches



- **Unconditional Branches**
  - Springt immer zu Zieladresse
  - direct: Zieladresse wird in label angegeben
    - **Achtung:** wenn der angegebene imm-Offset (nach <<1) > 0x400 (negativ) ist, muss man noch einen **sign-extend** machen. Wert - 0x1000. Negative Zahl dann mit PC + 4 + (neg) imm-Offset rechnen.
  - indirect: Zieladresse steht in Register
    - **BX** Rm kann also an beliebige viele Adressen springen
    - Da absolute bedeutet, dass man genau an diese Adresse springen wird, kann man **beliebig** im Speicher springen.
- **Conditional Branches**
  - Sind immer realative (PC-relative)
- **Flag dependent**
  - abhängig von einem Flag (N,Z,C,V)

Symbol	Condition	Flag
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

- **Arithmetic**
  - abhängig von einem oder mehreren Flags (N,Z,C,V)
  - unsigned:

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
HS (=CS)	Unsigned higher or same	C == 1
LO (=CC)	Unsigned lower	C == 0
HI	Unsigned higher	C == 1 and Z == 0
LS	Unsigned lower or same	C == 0 or Z == 1

  - signed:
    - greater and less

Symbol	Condition	Flag
EQ	Equal	Z == 1
NE	Not equal	Z == 0
MI	Minus/negative	N == 1
PL	Plus/positive or zero	N == 0
VS	Overflow	V == 1
VC	No overflow	V == 0
GE	Signed greater than or equal	N == V
LT	Signed less than	N != V
GT	Signed greater than	Z == 0 and N == V
LE	Signed less than or equal	Z == 1 or N != V

source: Joseph Yiu: The definite Guide to the ARM Cortex M3, Page 63

- to distinguish, apply and explain the instructions CMP,CMN and TEST
- to program bit manipulation operations (set, clear, toggle, test a bit) in assembly language
- **Vergleichsinstruktionen (Compare and Test)**
  - `CMP Rn, Rm` ⟶ führt `Rn - Rm` aus (setzt Flags, Resultat wird verworfen)
    - Ist gleichbedeutend mit Rn >= Rm ? -> res positiv: true, res negativ: false
  - `CMN Rn, Rm` ⟶ führt `Rn + Rm` aus (setzt Flags, Resultat wird verworfen)
    - Ist gleichbedeutend mit Rn + Rm >= 0 ? -> res positiv: true, res negativ: false
  - Ändern keine Register, nur Flags
    - `TST Rn, Rm` ⟶ führt `Rn AND Rm` aus (setzt N und Z, Resultat wird verworfen)
      - Prüft ob gemeinsame gesetzte Bits in Rn und Rm vorhanden sind (Z=0 wenn ja, Z=1 wenn nein)

9 Lecture_Control_Structures

- **Strukturierte Programme**
  - **Sequence** (einfach nacheinander)
  - **Selection** (`if/else`, `switch`)
  - **Iteration** (`while`, `do-while`, `for`)
- **Assembly-Muster**
  - if/else: `CMP` ⟶ `B{cond}` zu else/endif
  - while: test am Anfang (loop_head)
  - do-while: test am Ende
  - switch: Kette von Vergleichen oder Sprungtabelle (je nach Compiler/Range)

while-Loop Skeleton (Assembler-Denke) 1) loop_head: CMP ... 2) B{cond_false} loop_end 3) body ... 4) B loop_head 5) loop_end:

10 Lecture_Subroutines_and_Stack

- **Call/Return**
  - Call: `BL func` ⟶ LR bekommt Rücksprungadresse
  - Return: `BX LR` (oder `POP {...,PC}`)
- **Stack Basics**
  - Stack wächst typischerweise zu kleineren Adressen (downwards)
  - `PUSH {..}` / `POP {..}` sichern/holen Register
- **Register-Sichern**
  - Caller-saved vs. Callee-saved (wichtig für saubere Subroutines)

Typisches ISR/Subroutine Muster
- Prolog: `PUSH {R4-R7,LR}`
- Epilog: `POP {R4-R7,PC}`

11 Lecture_Parameter_Passing

- **Grundidee Calling Convention (AAPCS-Style)**
  - Argumente: zuerst in R0–R3
  - Return: meist in R0 (ggf. R1 für "zweites Wort")
  - Weitere Argumente / grosse Daten: über Stack
- **Wer muss was retten?**
  - Caller-saved: R0–R3, R12 (und oft LR wenn weiter-call)
  - Callee-saved: typ. R4–R11

Wenn du eine Funktion schreibst, die andere Funktionen aufruft: 1) sichere callee-saved Register, die du verwendest 2) sichere LR, falls du selbst wieder `BL` machst 3) arbeite 4) restore ⟶ return

12 Lecture_Modular_Coding_Linking

- **Warum modular?** Komplexität managen, Wiederverwendung, weniger Copy/Paste
- **C: Declaration vs. Definition**
  - Declaration: "Name existiert so" (z.B. `uint32_t f(uint32_t);`)
  - Definition: "hier ist der Code / Speicher wird reserviert"
- **Linker Tasks**
  - Merging (Code/Data Sections zusammenfügen)
  - Symbol Resolution (Referenzen auflösen)
  - Relocation (Adressen nach dem Zusammenfügen anpassen)
- **Toolchain / Libraries**
  - Native vs. Cross Toolchain
  - Static Libraries (link-time "reinkopiert") vs Dynamic/Shared (load-time)

One-Definition-Rule (Merke)
- Deklarieren: mehrfach ok
- Definieren: in einem Scope nur einmal

13 Lecture_Exceptional_Control_Flow-Interrupts

- **Polling vs Interrupt-driven I/O**
  - Polling: einfach, deterministisch, aber Busy-Wait (CPU-Zeit verschwendet)
  - Interrupt: schnelle Reaktion, aber Synchronisation/ Debugging schwieriger
- **Exceptions (Cortex-M3/M4 Sicht)**
  - System Exceptions (Reset, NMI, Faults, SVC, ...)
  - Interrupts IRQ0...IRQ239 (Peripherals, auch software-triggerbar)
- **Vector Table**
  - Liegt bei Reset an Adresse 0x0000_0000 (Mapping)
  - enthält Start-SP und Handler-Adressen
- **Context Save/Restore bei ISR Entry/Return**
  - Hardware stackt automatisch: xPSR, PC, LR, R12, R0–R3
  - EXC_RETURN wird in LR gesetzt; Return via `BX LR`

Kontext bei ISR Entry (automatisch) push: xPSR, PC, LR, R12, R0–R3 EXC_RETURN typisch: 0xFFFF_FFF9

- **NVIC Grundfunktionen**
  - Enable/Disable (global z.B. PRIMASK / CPSID i, CPSIE i + individuell)
  - Pending/Active Bits
  - Prioritäten ⟶ Preemption

Data Consistency Problem:
- Main liest Struktur, ISR schreibt gleichzeitig ⟶ "gemischte" Anzeige
- Lösung: kritische Sektion mit `__disable_irq(); ... __enable_irq();`

14 Lecture_Increasing_System_Performance

- **Bus-/Speicherarchitektur**
  - von Neumann: Code+Data über eine Schnittstelle ⟶ Bottleneck
  - Harvard: getrennte Interfaces ⟶ mehr Durchsatz
- **ISA Paradigmen**
  - CISC: komplexe Instruktionen
  - RISC: wenige, einfache Instruktionen; Load/Store; gut pipeline-bar
- **Pipelining (FE/DE/EX)**
  - Durchsatz nach "Füllen" ≈ 1 Instruktion pro Pipeline-Takt
  - Pipeline-Takt wird von langsamster Stage bestimmt
  - Hazards:
    - Data hazard (z.B. LDR braucht Bus ⟶ stall)
    - Control hazard (Branch-Entscheid spät ⟶ bubbles)

Instruction Throughput (Idee)
- ohne Pipeline: IPS = 1 / (Instruktions-Delay)
- mit Pipeline: IPS ≈ 1 / (max Stage-Delay)

- **Optimierungen / Parallelität**
  - Branch prediction, Prefetch, Out-of-order (bei grossen CPUs; kann Security-Risiken bringen)
  - Parallel Computing: SIMD, Multithreading, Multicore, Multiprocessor