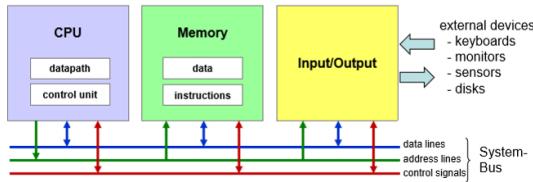


# Definitionen, Bemerkungen & Beispiele

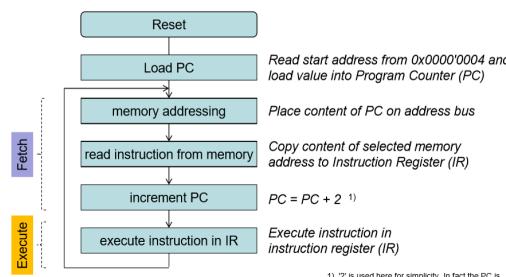
## CT\_Zusammenfassung

### 1 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



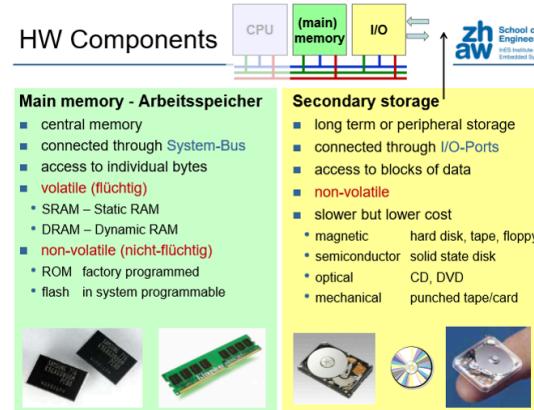
#### • CPU - Datapath

- ALU (Arithmetic Logic Unit): führt Operationen aus
- Register: schnelle Zwischenspeicher (CPU-intern)
- Busse: verbinden Komponenten (Adress-, Daten-, Steuerbus)

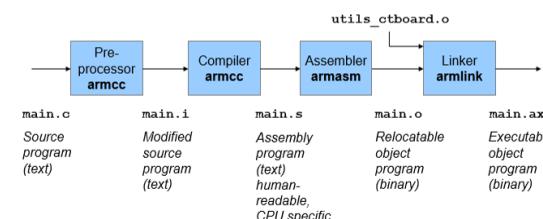
#### • Memory

- 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  $\rightarrow$  N Bit Adressbus  $\rightarrow$   $0 \dots 2^{N-1}$  Adressen
- RAM/ROM/Flash (extern, langsamer)



#### • Toolchain



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

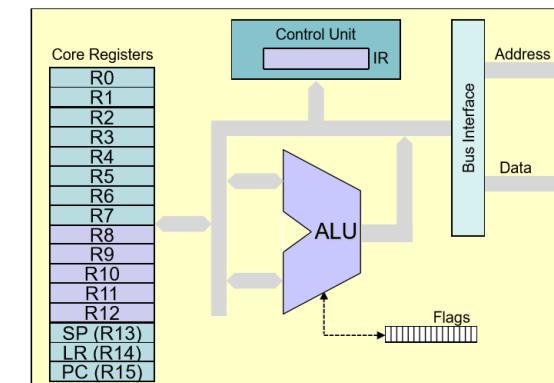
### 2 Lecture\_Cortex-M\_Architecture

#### • Instruction Set Architecture (ISA)

- Schnittstelle HW  $\leftrightarrow$  SW (Befehle, Register, Adressierung, Datentypen, ...)
- CT1 (Cortex-M0): **32-bit RISC**, Load/Store, Thumb-Subset

#### • Architecture

- Adressraum: 32-bit Adressen  $\rightarrow$  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)

```

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

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

```

#### Stolpersteine:

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

## 2.1 constants @ 0x08000200

Bytes (hex)	Bedeutung
05 00 00 00	limit DCD 5 ( <b>little endian</b> )
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) ( <b>little endian</b> )
? ? ? ? ? ? ? ?	unbenutzt/unknown

## 2.2 variables @ 0x20000000

Bytes (hex)	Bedeutung
0A 00 00 00	counter DCD 10 ( <b>little endian</b> )
all 0	buffer SPACE 10 ( <b>unknown</b> )
FF FF 80 FF 00 FF 80 FE	colors DCW FFFF, FF80, FF00, FE80 ( <b>little endian halfwords</b> )
pad pad	Padding (Alignment vor DCD)

C8 00 00 00	last_counter DCD 200 ( <b>little endian</b> )
-------------	--

#### • Endianness

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

#### Little Endian (Beispiel 0xA1B2C3D4 @ addr) addr+0: D4

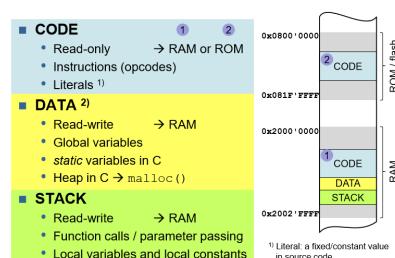
addr+1: C3 addr+2: B2 addr+3: A1 **Big Endian (Beispiel**

#### 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	0xB2	
0x30000000	0xA2	Var3 (32-bit integer)
0x30000001	34 (dezimal)	
0x30000002	0x54	
0x30000003	0xF	
0x30000004	0x92	Var4 (unsigned char)
0x30000005	0x03	Var5 (char)

#### • Memory Map

- Tiefste Adresse oben gezeichnet
- Berechnung Speichergrösse: Endadresse - Startadresse + 1 (in Bytes)

▸ Berechnung 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

```

## 3 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 grössze 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)

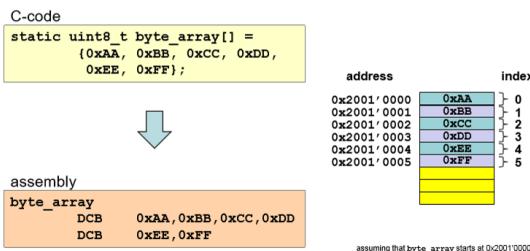
- Relative Adressierung: Adresse wird relativ zu PC berechnet
  - z.B. LDR Rt,[PC,#imm] (PC-relative)
- Indirekte Adressierung: Adresse steht in Register
  - z.B. LDR Rt,[Rn] (Register Indirect)

#### • 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};

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 0xC → R0  
5 Load content of xp → R1  
i.e. address of x is now in R1  
6 Store R0 at address given by R1

## 4 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.

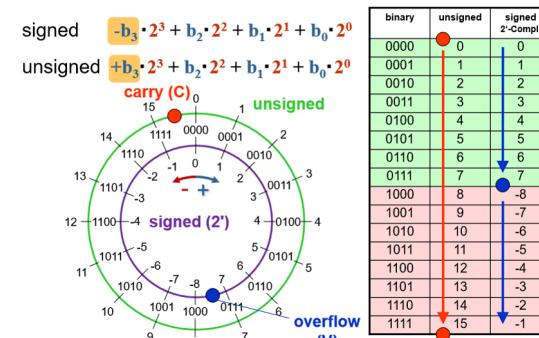
Hinweis: Carry aus der high-Addition wäre ein "65stes Bit" und wird hier nicht weiter gespeichert.

#### • Einerkomplement

- Positive Zahlen: normale Binärdarstellung
- Negative Zahlen: invertieren
- Bereich bei n Bit:  $-(2^{n-1} - 1) \dots 2^n - 1$

#### • Zweierkomplement

- Positive Zahlen: normale Binärdarstellung
- Negative Zahlen: invertieren + 1
- Bereich bei n Bit:  $-2^{(n-1)} \dots 2^{(n-1)} - 1$
- Subtraktion: A - B = A + (B + 1) oder (A + (-B)) + 1



#### • Flags (APSР)

- 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

## 4.1 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**

## 4.2 Addition: op1 + op2

**Q1:** Ist die Summe  $\geq 0x100$ ? → 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 gleiche 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

## 4.3 Subtraktion: op1 - op2

**Q1:** Brauchst du ein Borrow? (**unsigned Vergleich**)

- Ist op1  $\geq$  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

## 4.4 Mini-Merker (sehr schnell)

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

## 5 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  $\rightarrow$  (cast)a > (cast)b  $\rightarrow$  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

#### Integer Casting in C

##### signed ↔ unsigned

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

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

##### Casts in red area

→ Small negative numbers turn into large positive numbers

→ Large positive numbers turn into small negative numbers

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

### 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

- unsigned -> 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

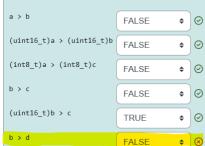
#### • Stoplersteine bei if-Statements

- Vergleiche immer im gleichen Typ (signed/unsigned)
- Sonst falsche Ergebnisse möglich (z.B.  $-1 > 0$  bei unsigned)

##### Vergleichsoperationen nach Casting

Gegeben ist die folgende Variableninitialisierung in C. Geben sie an, wie das Ergebnis des jeweiligen Vergleiches evaluiert wird!

```
int16_t a = 0xFF12;
int16_t b = 0xFFA2;
int16_t c = 0x7F12;
uint16_t d = 0x7FA2;
```



## 6 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)

## 6.1 Multiplikation mit Konstante (Synthese)

### 6.1.1 Schritt 1: Konstante in Binär zerlegen

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

K = 23 = 0b00010111  $\Rightarrow$  Einsen bei  $i = 0, 1, 2, 4 \Rightarrow 23 \times x = (x << 0) + (x << 1) + (x << 2) + (x << 4)$

### 6.1.2 Schritt 2: Addier-Plan erstellen (minimal denken)

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

### 6.1.3 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)

### 6.1.4 Tipp für den ersten Shift LSLS R0, R0, #x

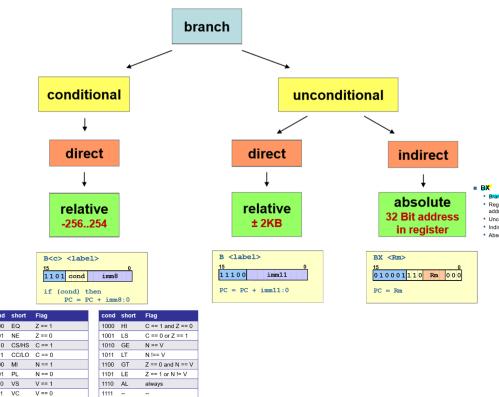
Manchmal beginnt die Vorlage mit einem größeren Startshift #x, um direkt zum ersten gesetzten Bit zu springen:

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

Wenn K ungerade  $\rightarrow$  Bit0=1  $\rightarrow$  i\_min=0  $\Rightarrow$  x=0 (kein Vorschub nötig) Wenn K gerade (z.B. 40=0b00101000)  $\rightarrow$  i\_min=3  $\Rightarrow$  x=3

## 7 Lecture\_Branches

### 7.1 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 realtive (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 Yu: The definite Guide to the ARM Cortex M3, Page 63

#### • 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

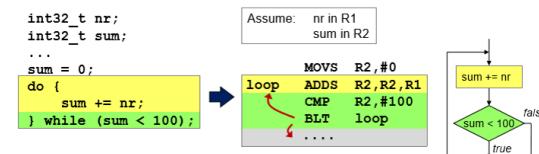
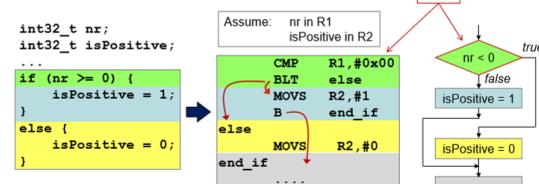
- TST Rn, Rm → führt Rn AND Rm aus (setzt N und Z, Resultat wird verworfen)
- Prüft ob gemeinsame gesetzte Bits in Rn und Rm vorhanden sind (Z=0 wenn ja, Z=1 wenn nein)

## 8 Lecture Control Structures

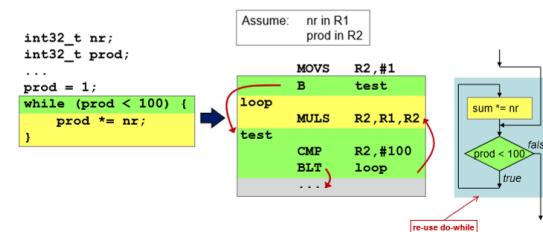
### • Strukturierte Programme

- Sequence (einfach nacheinander)
- Selection (if/else, switch)
- Iteration (while, do-while, for)

- Compiler translates selection into assembly code
  - uses conditional and unconditional jumps

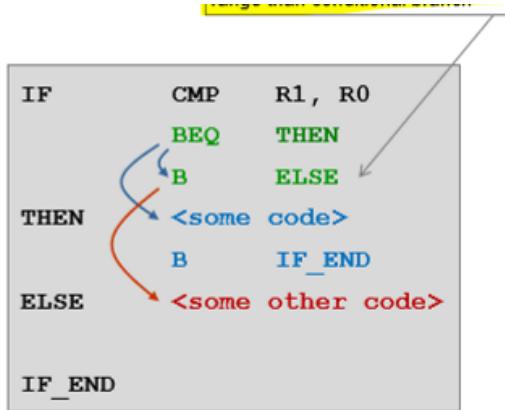


- Compiler translates pre-test loop to assembly code
  - Re-using structure of do-while (pre-test loop)



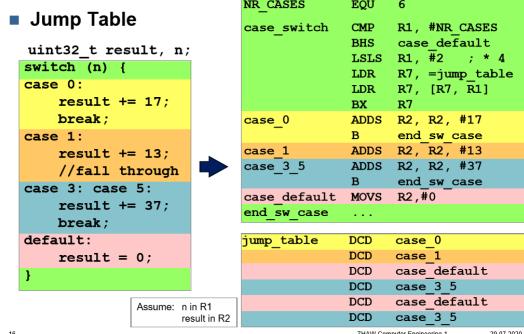
### • Limitations of Conditional Branches

- Nur bedingte Sprünge (keine Schleifen direkt)
- Nur relativ (PC-relative)
- Nur begrenzte Reichweite ( $\pm 256$  Bytes).
  - Lösung siehe Bild (Wenn if block zu gross dann else mit B ansteuern)



Code requires additional branch in case when <some code> is too long

#### • Switch Statement



- jumtable: Ist in Var section (muss bekannt sein)
- Kurzablauf:
  - Prüfen ob Wert (R1) im Bereich -> außerhalb: default
  - Index berechnen (R1 \* 4)
  - Startadresse der jumptable holen (R7)
    - Zieladresse holen (LDR R7,[R7,R1])
    - Springen (BX R7)

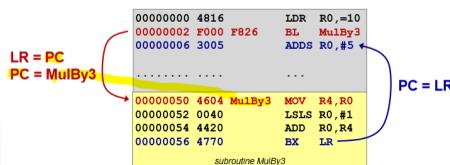
## 9 Lecture\_Subroutines\_and\_Stack

#### • Subroutine Definition

- called by NAME
- Internal design not visible to caller -> information hiding
- can be reused (DRY principle)
- Function -> returns value; Procedure -> no return value
- Call/Return
  - Call: BL func -> LR bekommt Rücksprungadresse
  - Return: BX LR (oder POP {...,PC})

#### ▪ Change of control flow

- Call Save PC to Link Register (LR)
- Return Restore PC from LR



#### • Unterschiede

##### ▸ B label vs BL label

- B: setzt nur den PC auf das Ziel (Programm läuft dort weiter). Kein Rücksprung wird gespeichert
- BL: setzt PC auf Ziel und speichert Rücksprungadresse in LR (Link Register) -> Rücksprung mit BX LR möglich oder POP {...,PC}

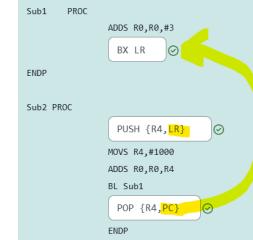
##### ▸ BX Rm vs BLX Rm

- BX Rm: springt zu Adresse in Rm (kein Rücksprung gespeichert). Ändert nur PC
- BLX Rm: springt zu Adresse in Rm und speichert Rücksprungadresse in LR. Ändert PC und LR

#### • Subroutine Sichern

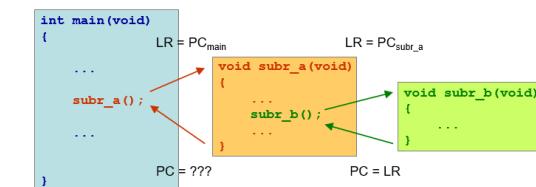
#### Subroutines/Stack

Betrachten Sie die nachfolgend definierten Subroutinen Sub1 und Sub2.  
Ergänzen Sie in den Lücken die nötigen Instruktionen, zur Sicherung der minimal notwendigen Register,



#### • Nested Subroutines

- Subroutine kann andere Subroutinen aufrufen
- LR wird bei jedem BL überschrieben -> vorher sichern (Stack/Register)
- Rücksprungadresse immer im LR



#### • Stack Basics

- Stack wächst zu kleineren Adressen (downwards)
- PUSH {...} / POP {...} sichern/holen Register

#### • Stack Pointer (SP)

- Zeigt auf zuletzt genutzten Stack-Eintrag
- Nach PUSH: SP = SP - 4 × Anzahl Register -> neue Position nach ein
- Nach POP: SP = SP + 4 × Anzahl Register -> neue Position nach aus

#### • 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}

## 10 Lecture\_Parameter\_Passing

### • Application Binary Interface

- Sind Spielregeln für Subroutine Calls
- Definieren wie Parameter übergeben werden
  - Definieren wie Rückgabewerte übergeben werden
  - Definieren welche Register von Caller/Callee gesichert werden müssen

### • How to pass parameters?

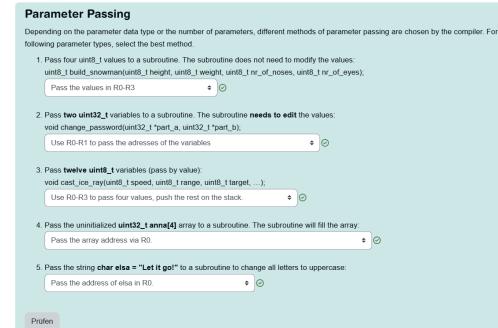
- via Registers (schnell, limitiert)
- via Stack (flexibel, langsamer)
- via globalen Variablen (unsicher, langsam)
- ARM nutzt Register-basiertes Parameter Passing (schnell)
- **Pass by Value**
  - So übergibt ARM Parameter (Kopien in R0–R3)
- **Pass by Reference**
  - So übergibt ARM Adressen (Pointer in R0–R3) und vor allem grosse Datenstrukturen über Stack z.B. Arrays
  - to enumerate and describe the operations of the caller of a subroutine
  - to summarize the structure of a subroutine and describe what happens in the prolog and epilog respectively

### • ARM Procedure Call Standard (AAPCS)

- R0–R3: Argumente / Return-Werte
- R4–R11: callee-saved (müssen von Subroutine gesichert werden)
- R12: scratch (caller-saved)
- SP: Stack Pointer
- LR: Link Register (Rücksprungadresse)
- PC: Program Counter

## Register Usage

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	



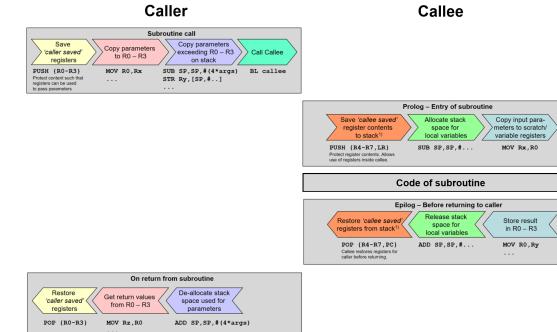
### ‣ Register explained

- Scratch Register:
  - limited lifetime (only within one subroutine)
  - not named, hold temporary values during calculations
- Variable Register:
  - hold values across subroutine calls
  - must be saved/restored by callee if used
  - R8–R12
  - named
- Argument, Parameter:
  - used to pass arguments to subroutines and return values
  - Caller copies R0–R3 and additional parameters on stack

**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

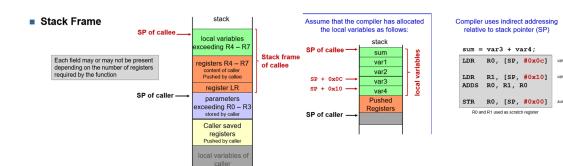


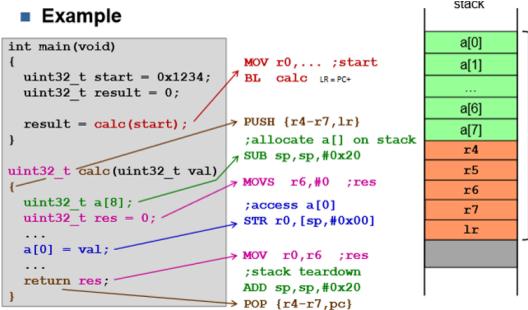
• to explain, interpret and discuss stack frames

• to access elements of a stack frame in assembly

• to understand the build-up and tear-down of stack-frames

### • Stack Frame





### C to Assembly call

```

extern void strcpy(char *d, const char *s);
int main(void)
{
    const char *srcstr = "First string ";
    char dststr[] = "Second string";
    strcpy(dststr,srcstr);
    return (0);
}

PRESERVE8
AREA SCopy, CODE, READONLY
EXPORT strcpy

strcpy:
    ; R0 points to destination string
    ; R1 points to source string
    LDRB R2, [R1]          ; Load byte and update address
    ADDS R1, R1, #1
    STRB R2, [R0]           ; Store byte and update address
    ADDS R0, R0, #1
    CMP R2, #0              ; Check for null terminator
    BNE strcpy             ; Keep going if not
    BX LR                  ; Return
END

```

## 11 Lecture Modular Coding Linking

### • Warum modular?

- Komplexität managen, Wiederverwendung, weniger Copy/Paste.
- Es gelten die gleichen Prinzipien wie bei prog1 (high cohesion, low coupling).
- Translationsschritte:
  - Preprocessing (Makros, Includes, bedingte Compilierung)
  - Compilation (C/ASM → Objektdateien .o)
  - Assembly (ASM → Objektdateien .o)
  - Linking (Objektdateien + Libraries → ausführbare Datei .elf/.bin), erst beim Linking werden alle Module zusammengefügt

### • C: Declaration vs. Definition

- Declared before use (z.B. in Header-Datei)
- Defined once (z.B. in Quellcode-Datei)
- Declaration: "Name existiert so" (z.B. uint32\_t f(uint32\_t);)
- Definition: "hier ist der Code / Speicher wird reserviert"

### • Source Code Anatomy

- External linkage: global (über Modul hinweg sichtbar)
- Internal linkage: static (nur innerhalb eines Moduls sichtbar)
- No linkage: lokale Variablen (nur innerhalb einer Funktion sichtbar)

### ■ Example: Internal and external linkage (C)

- All global names have external linkage unless defined static

<code>// square.c</code>	<code>// main.c</code>
<code>... uint32_t square(uint32_t v) {     return v*v; }</code>	<code>... static uint32_t a = 5; static uint32_t b = 7; int main(void) {     uint32_t res;     res = square(a) + b;     ... }</code>

square = external linkage

a = internal linkage  
b = internal linkage  
main = external linkage  
res = no linkage  
square = external linkage<sup>1)</sup>

1) square has external linkage (no static keyword), but no definition in main.c - needs to be resolved by the linker

### • EXPORT and IMPORT

- EXPORT name in Assembly: macht Symbol global sichtbar (external linkage)
- IMPORT name in Assembly: deklariert externes Symbol (muss in anderem Modul definiert sein)

### • Assembly to object file linkage

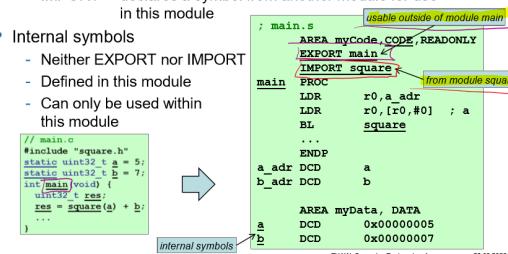
- References: **Imported symbols** translate **global reference symbols** in object file
- global: **Exported symbols** translate to **global symbols**
- local: **Internal symbols** translate to **local symbols**

### • Linkage control

- EXPORT declares a symbol for use by other modules
- IMPORT declares a symbol from another module for use in this module

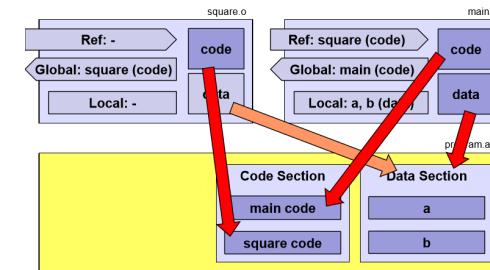
### • Internal symbols

- Neither EXPORT nor IMPORT
- Defined in this module
- Can only be used within this module



### • Linker Tasks

- Merging (Code/Data Sections zusammenfügen)
  - data Section
  - code Section
- Resolve used external symbols
- Relocate addresses



### • Merging Code Section

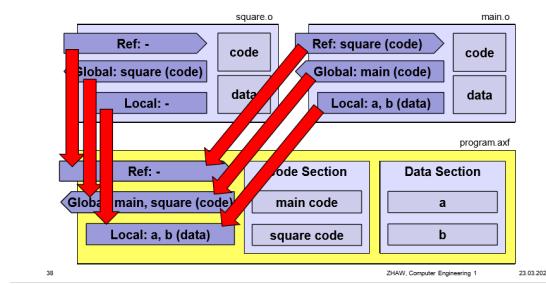
- Merging code sections of main.o and square.o
  - Offset for first code section is **0x00000000** (main.o)
  - Offset for next code section is **0x0000001C** (square.o)

code (main.o)
0x00000000 8510 PUSH {r4..r12} 0x00000002 4604 LDR r0,[pc,#16] 0x00000004 6800 LDR r0,[r1,#0x00] 0x00000006 ffffff BL square 0x0000000A 4903 LDR r1,[pc,#12] 0x0000000C 6800 LDR r1,[r1,#0x00] 0x0000000E 4604 ADD r1,r1,r0 0x00000010 2000 MOV r1,r0 0x00000012 n010 POP {r4..pc} 0x00000014 00000000 DCD 0x00000000 0x00000018 00000000 DCD 0x00000000
code (square.o)
0x0000001C 4604 MOV r1,r0 0x0000001E 6800 LDR r0,[r1,#16] 0x00000020 4348 MOVS r0,r1,r0 0x00000022 4770 BX lr

- Erster Offset ist immer startadresse der Section (0x0000\_0000)

- Alle folgenden Sections werden nacheinander angefügt
- Neue Adresse = vorherige Endadresse + 4 (Im Bild  $0x0000_0018 + 4 = 0x0000_001C$ )
- Merging Date Section**
  - Funktioniert gleich wie Code Section
  - Erster Offset ist immer startadresse der Section
  - zb Section1: 0x00 DCD 5, Section2: 0x04 DCD 7 etc...

- Resolve**
  - Ist merge für symbol Tables.



## Tasks of a Linker – Example



### Example: Resolve symbols

- Merging symbol table sections of main.o and square.o
 

#	Symbol	Name	Value	Bind	Seo	Type	Vis	Size	symbols (main.o)
7	a		0x00000000	Lc	4	Data	De	0x4	
8	b		0x00000004	Lc	4	Data	De	0x4	
11	main		0x00000000	Gb	1	Code	Hi	0x14	
12	square		0x00000000	Gb	1	Code	Hi	0x14	
6	square		0x00000001	Gb	1	Code	Hi	0x8	

#	Symbol	Name	Value	Bind	Seo	Type	Vis	Size	resolved symbols
20	a		0x00000000	Lc	4	Data	De	0x4	(main.o)
21	b		0x00000004	Lc	4	Data	De	0x4	(main.o)
22	main		0x00000000	Gb	1	Code	Hi	0x14	(main.o)
187	square		0x00000001	Gb	1	Code	Hi	0x8	(square.o)

The relative values of the symbols within the modules are not yet relocated to global addresses. Therefore, the linker needs to remember for which module/section the relative address is given.

No relocation done yet.

### Relocate

- Ist ausrechnen der relativen finalen Adressen im main.o
- Formel: **new address = base address + offset + module relative address**
  - base address: startadresse der Section im finalen Executable

- offset: wo ist das Modul in der Section (nach merge)
- module relative address: Adresse im Modul (vor Relocate)

### Relocation calculations

- new value = global base + merge offset + module relative offset**
  - E.g. symbol **b**:
  - global base = internal SRAM = **0x20000000**
  - merge offset = 1<sup>st</sup> in merged data section = **0x00000000**
  - module relative offset = b is the 2<sup>nd</sup> variable after a = **0x00000004**
  - new value for symbol b = **0x20000004**
- E.g. symbol **square** if user code (like main) starts at **0x08000254**
  - 0x08000254 + 0x0000001C + 0x00000000 = 0x08000270**

Relocated code sections	
0x08000254	BS10 PUSH {r4..lrc} ; main
0x08000256	LDR r0,[pc,#16]
...	
0x08000270	4B01 MOV r1,r0 ; square
0x08000272	4E08 MOV r0,r1
...	
Relocated data sections	
0x20000000	00000000 DCD 5 ; value of a
0x20000004	00000007 DCD 7 ; value of b
Relocated symbols	
22	b Lc 4 Data De 0x4
186	main Gb 1 Code Hi 0x14
187	square Gb 1 Code Hi 0x8
Relocated relocation table entries	
0	0x0800025A 10 R_ARM_TDRZ_CALL 12 square
1	0x08000268 2 R_ARM_ABS32 7 a
2	0x0800026C 2 R_ARM_ABS32 8 b
Adjusted code locations according to relocation table	
...	
0x0800025A	F000F809 BLW square ; 0x08000270
...	
0x08000268	20000000 DCD 0x20000000
0x0800026C	20000004 DCD 0x20000004

### Static Linking

- Alle Objektdateien und Libraries werden zur Build-Zeit zusammengefügt
- Ergebnis: eine ausführbare Datei (.elf/.bin)
- Vorteile: Einfach, schnell, keine Abhängigkeiten zur Laufzeit
- Nachteile: Größere Datei, keine Updates einzelner Module möglich

### Dynamic Linking

- Module werden zur Laufzeit geladen (shared libraries)
- Vorteile: Kleinere ausführbare Datei, Updates einzelner Module möglich
- Nachteile: Komplexer, Abhängigkeiten zur Laufzeit, Performance-Overhead

### Map File

- Textdatei mit Speicherlayout der ausführbaren Datei

- Enthält Adressen und Größen von Sektionen, Symboltabellen
- Nützlich für Debugging, Optimierung, Speicheranalyse

### Source level Debugging

- needs mapping Machine address to Source code line
- needs mapping Memory location and source code types
- often Provided in in obj file eg .elf format

## 12 Lecture\_Exceptional\_Control\_Flow-Interrupts

### Polling vs Interrupt-driven I/O

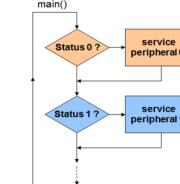
- Polling: Ist einfach periodisch prüfen ob ein Ereignis eingetreten ist
  - einfach, deterministisch, aber Busy-Wait (CPU-Zeit verschwendet)
- Interrupt: Ist ereignisgesteuert (Peripherie signalisiert CPU)
  - effizienter, multitasking-fähig
  - schnelle Reaktion, aber keine Synchronisation zwischen main und ISR + Debugging schwieriger

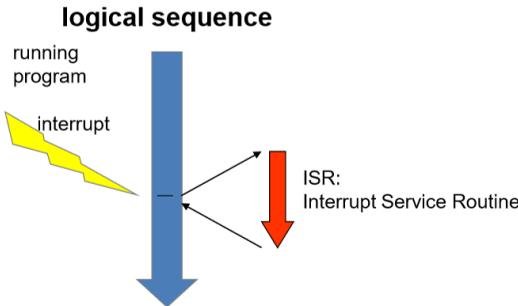
```
while(1)
{
    if ( read_byte(ADDR_BUTTON_A) ) {
        execute_task_A();
    }

    if ( read_byte(ADDR_BUTTON_B) ) {
        execute_task_B();
    }

    if ( read_byte(ADDR_BUTTON_C) ) {
        execute_task_C();
    }

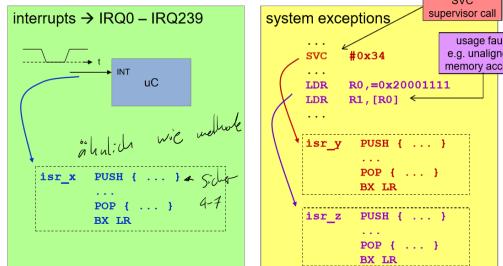
    ...
}
```





- to explain how the Cortex-M3/M4 recognizes and processes exceptions
- to explain the vector table of the Cortex-M3/M4
- to distinguish the different types of exceptions on a Cortex-M3/M4
- **Exceptions (Cortex-M3/M4 Sicht)**
  - System Exceptions (Reset, NMI, Faults, SVC, ...)
  - Interrupts IRQ0...IRQ239 (Peripherals, auch software-triggerbar)
  - Sind eigentlich das gleiche, aber unterschiedliche Prioritäten und Vektoren

#### ■ Examples for Exceptions



## System Exceptions

Exception Number	Exception Type	Priority	Description
1	Reset	-3 (Highest)	Reset
2	NMI	-2	Nonmaskable interrupt (external NMI input)
3	Hard Fault	-1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage Fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus Fault	Programmable	Bus error, occurs when AHB interface receives an error response from a bus slave (also called prefetch abort if it is an instruction fetch or data abort if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error or trying to access coprocessor (the Cortex-M3 does not support a coprocessor)
7-10	Reserved	NA	
11	SVCcall	Programmable	System Service call
12	Debug Monitor	Programmable	Debug monitor (breakpoints, watchpoints, or external debug requests)
13	Reserved	NA	
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

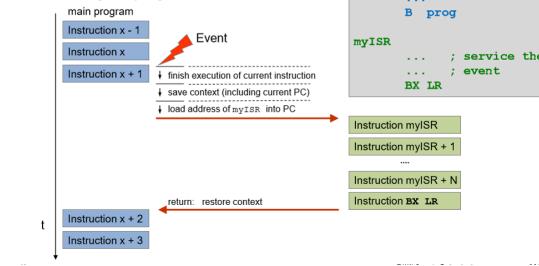
source: "The definitive Guide to the Cortex-M"

### ■ Interrupts: change of programm Flow

Buch vergleicht: finish -> Satz fertig lesen, save -> Buchzeichen, load address -> springen zum Interrupt Handler

#### ■ Interrupts: Event calls ISR

- Change of program flow



### ■ Context Save/Restore bei ISR Entry/Return

- Hardware stackt automatisch: xPSR, PC, LR, R12, R0–R3, EX stored
- Am schluss von myISR BX LR (LR = EXC\_RETURN Wert) → Hardware entstackt automatisch
- nicht automatisch gesichert: R4–R11

Weiche Schritte werden beim Eintritt in eine ISR von der CPU ausgeführt, um die sich der Entwickler der ISR nicht selbst kümmern muss?

Wählen Sie eine oder mehrere Antworten:

Sichern (Push) von xPSR, PC, LR, R12, R0 – R3 auf dem Stack

Beenden der Anweisung, die aktuell ausgeführt wird

Laden des Interruptvektors in PC und von EXC\_RETURN in LR

- to understand the basic functionality of the Nested Vectored Interrupt Controller (NVIC)

### • NVIC Grundfunktionen

- 240 Interrupts (IRQ0–IRQ239) -> trigger von Peripherie (high level signal)
- NVIC ist Hardware-Modul im Cortex-M, leitet Interrupts an CPU weiter (physikalisch)
- CPU rechnet vector addresse aus, basierend auf IRQ-Nummer
- Vektor-Tabelle: Liste von Adressen für jeden Exception/Interrupt (im Flash bei 0x0000\_0000)
- Alle Register werden automatisch gespeichert -> Entwickler muss nur ISR schreiben

### • Vector Table

#### ■ Which ISR Shall the Processor Call?

- Each exception has a different ISR

Memory Addr.	31	0	Exception Nr.
0x0000'0000	Top of Stack	0	0
0x0000'0004	Reset	1	1
0x0000'0008	NMI	2	2
0x0000'000C	Hard Fault	3	3
...	...	...	...
0x0000'002C	SVC	11	11
...	...	...	...
0x0000'0038	PendSV	14	14
0x0000'003C	SysTick	15	15
0x0000'0040	IRQ0	16	16
0x0000'0044	IRQ1	17	17
...	...	...	...
0x0000'03FC	IRQ239	255	255

System Exceptions 1 – 15

Interrupts 0 – 239

IRQn → Exception Nr. (n + 16)

Example: IRQ3 → Exception Nr. 19

ZHAW Computer Engineering 1 9.9.2024

### Vector Table (Cortex-M3/M4)



#### ■ Initialization

```

; Vector Table Mapped to Address 0 at Reset
AREA RESET, DATA, READONLY
0 _Vectors    DCD _initial_sp      ; Top of Stack
1           DCD Reset_Handler    ; Reset Handler
2           DCD NMI_Handler     ; NMI Handler
3           DCD HardFault_Handler ; Hard Fault Handler
4           DCD ...
5           DCD ...
; Interrupts
16          DCD IRQ0_Handler    ; ISR for IRQ0
17          DCD IRQ1_Handler    ; ISR for IRQ1
...
; AREA SOURCE_CODE, CODE, READONLY
IRQ0_Handler PUSH { ... }
...
; interrupt service
POP { ... }
BX LR

```

System Exceptions 15 vectors

Interrupts

## • Interrupts Control

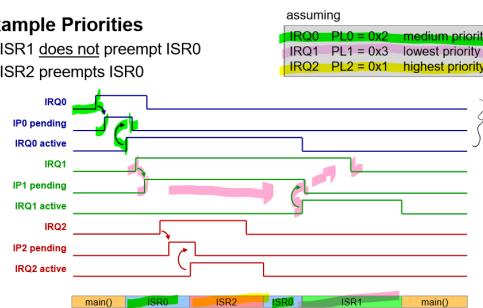
- Inactive: Interrupt nicht aktiv (Standard)
- Pending: Interrupt ist angefordert (warten auf Service) ( $IRQn = 1$ ) (Interrupt disabled)
  - Pending Register: Setzen/Löschen von Pending Bits
- Active: Interrupt wird gerade serviert (Interrupt enabled)
  - Active Status Register: Lesen ob Interrupt aktiv ist (bit setzt wenn aktiv, cleared wenn fertig)
- Active and Pending: Interrupt wird gerade serviert, aber erneut angefordert

## • Prioritize exceptions

- Jede Exception/Interrupt hat Priorität (0 = höchste, 255 = niedrigste)
- Priorität wird in NVIC konfiguriert (4 Bits genutzt  $\rightarrow$  16 Stufen)
  - Prio setzen von  $IRQn$ : via LDR  $R0,=PL\_REG\_IRQn$ , dann prio in R1 schreiben und speichern
  - **Achtung** je tiefer die Zahl, desto höher die Priorität (0 = höchste Prio)
- **Preemption:** Höher priorisierte Interrupts können tiefer priorisierte unterbrechen
  - z.B. IRQ1 (Prio 2) kann IRQ2 (Prio 5) unterbrechen
- **Tail-Chaining:** Wenn ISR endet und ein anderer Interrupt pending ist, wird dieser sofort gestartet (ohne Rückkehr zu main)
- **Simultaneously pending**
  - Wenn mehrere Interrupts gleichzeitig pending sind, wird der mit der höchsten Priorität zuerst bedient
  - NVIC wählt automatisch den nächsten Interrupt basierend auf Priorität und Pending-Status aus

### ■ Example Priorities

- ISR1 **does not** preempt ISR0
- ISR2 preempts ISR0



**Wichtig:** IRQ set by HW - cleared by SW

### • Enable/Disable Interrupts

- Alle Ein/Ausschalten in asm: (CPSID i / CPSIE i)
- All Ein/Ausschalten in C: `_disable_irq()`, `_enable_irq()`

```
LDR R1, =0x10000040 ;enable Interup mask
LDR R0, =REG_SETENA0
STR R1, [R0]
```

Man kann die Bits auch ein- und ausschalten, weil aktive Bits einen Effekt auf das Register haben

### Enable IRQ3

```
SETENA0 EQU 0xE000E100
...
LDR R0, =SETENA0
MOVS R1, #0x08
STR R1, [R0]
```

### Disable IRQ3

```
CLRENA0 EQU 0xE000E180
...
LDR R0, =CLRENA0
MOVS R1, #0x08
STR R1, [R0]
```

**Achtung** 0x08 ist bit 3 (IRQ3) setzen (0b0000\_1000), am besten 1 und dann shiften  $\rightarrow$  LSLS R1, R1, #IRQn. Achtung shift vlt zu gross, aufteilen.



## • Data Consistency Issues

- Interrupts können Variablen ändern, während main darauf zugreift
- Lösung:
  - Disable Interrupts während kritischer Abschnitte (`_disable_irq()`, `_enable_irq()`)

### ■ Example

```
typedef struct {
    uint8_t minutes;
    uint8_t seconds;
} time_t;

static time_t time = { 0, 0 };

int main(void)
{
    while (1) {
        _disable_irq();
        write_byte(ADDR_LED_7_0, time.seconds);
        write_byte(ADDR_LED_15_8, time.minutes);
        _enable_irq();
    }
}
```

time = { 16, 59 }  
1) Output 59  $\rightarrow$  LED\_7\_0  
2) Interrupt  $\rightarrow$  time = { 16, 0 }  
3) Output 16  $\rightarrow$  LED\_15\_8  
 $\rightarrow$  display 16 59 !!!

## • Code Example ISR

```
AREA ISR_Example, CODE, READONLY
EXPORT MyISR
```

```
MyISR
; ISR Code hier
BX LR
END
```

### • CMSIS

- Cortex Microcontroller Software Interface Standard
- Standardisierte Header-Dateien und Funktionen für Cortex-M
- Erleichtert Entwicklung und Portabilität
- Enthält NVIC-Funktionen, System-Initialisierung, etc.
- Methoden: `NVIC_EnableIRQ(IRQn_Type IRQn)`, `NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)`

13

## Lecture\_Increasing\_System\_Performance

- Optimierung ist ein Trade-off

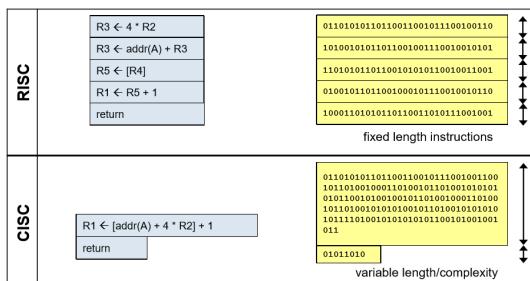
- Geschwindigkeit vs. Komplexität vs. Energieverbrauch vs. Kosten
- Meistens: mehr Geschwindigkeit → mehr Komplexität/Kosten/Energie

#### • Bus-/Speicherarchitektur

- von Neumann: Code+Data über eine Schnittstelle → Bottleneck
- Harvard: getrennte Interfaces → mehr Durchsatz
- Meiste System sind Harvard (Cortex-M)

#### • ISA Paradigmen

- RISC(Reduced Instruction Set Computer): wenige, einfache Instruktionen; Load/Store; gut pipeline-bar
  - Vorteile: einfach und schnell, Mehr Register auf CPU (weniger Speicherzugriffe), mehrere data pfade möglich, höhere Taktfrequenz, besser für Compiler, einfaches Pipelining
  - Nachteile: mehr Instruktionen pro Programm, grösserer Code (mehr Speicher)
- CISC(Complex Instruction Set Computer): viele, komplexe Instruktionen; direkt auf Speicher operierend; schwer pipeline-bar
  - Vorteile: kompakter Code (weniger Speicher), weniger Instruktionen pro Programm, komplexe Operationen in einer Instruktion
  - Nachteile: komplexe Hardware, schwerer zu pipeline-en, längere Taktzyklen, schwieriger für Compiler



#### • Pipelining (FE/DE/EX)

- Es gibt mehrere Stages (z.B. Fetch, Decode, Execute) -> wenn eine Stage von fetch zu decode geht, kann die fetch Stage schon die nächste Instruktion holen
- Erhöht Instruktionsdurchsatz (IPS), nicht Latenz (Zeit pro Instruktion)
- Ideal: 1 Instruktion pro Takt (nach Füllen der Pipeline)
- Pipeline-Takt wird von **langsamster Stage bestimmt**
- Hazards:
  - Data hazard (z.B. LDR braucht Bus → stall)
  - Control hazard (Branch-Entscheid spät → bubbles)

#### Instruction Throughput

- ohne Pipeline: IPS = 1 / (Instruktions-Delay)
- mit Pipeline: IPS  $\approx 1 / (\text{max Stage-Delay})$
- Instruction Delac für eine Instruktion = Summe aller Stage-Delays (FE + DE + EX)
- was bedeutet max Stage-Delay?
  - die langsamste Stage bestimmt den Takt der gesamten Pipeline

#### Special Operations Pipeline pause

- Data Hazard: wenn eine Instruktion auf das Resultat einer vorherigen angewiesen ist (z.B. LDR gefolgt von ADD auf dasselbe Register)
  - Lösung: Pipeline **stallen** (nops einfügen(s)) oder Forwarding (Daten direkt weiterleiten)
  - stallen heisst, dass die Pipeline für einige Zyklen angehalten wird, um Datenabhängigkeiten zu lösen.

#### Optimierungen / Parallelität

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

#### 14 Stolpersteine

- HIER SCREENS VON Aufgaben die ich nicht geschaft habe

#### 15 Häufige Code Snippets

- Hier einige nützliche Code-Snippets