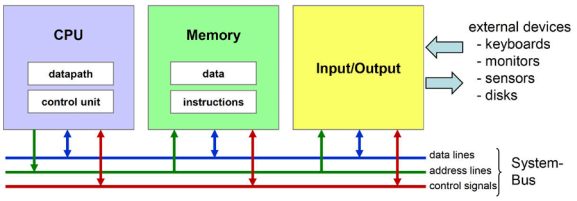
Computer Systems

Computer Engineering:
- Architecture of computer systems
- Combination of hardware and software → Embedded systems

Von Neumann Architecture

Hardware Components:

CPU	**Central Processing Unit or processor**
Memory	**stores instructions and data**
Input / Output	**interface to external devices**
System-Bus	**electrical connection of blocks**



CPU (Intel, ARM, AMD)

Instructions and data are stored in the same memory.
Master auf der Adress-Linie (im System-Bus).
Reads or writes to/from Memory or I/O.

Datapath
- ALU (Arithmetic and Logic Unit)
- Registers: Executes ALU operations and holds intermediate results
  Fast but limited storage, Register is **4 Bytes** (32 Bits)

Control Unit
Reads and executes/interprests opcode instructinos (Finite State Machine)
Operations:
- Data transfer: Registers ⟷ Memory
- Arithemtic and logic operations
- jumps (goto, if-else)

Memory

Each **1 Byte Memory-cell** is addressed with a **4 Byte memory-address.**

2^N addresses, from 0 to 2^N –1. N = Number of address lines.
Speichermedien (main memory):
- Random Access Memory (RAM, read/write)
- Read Only Memory (ROM, read)

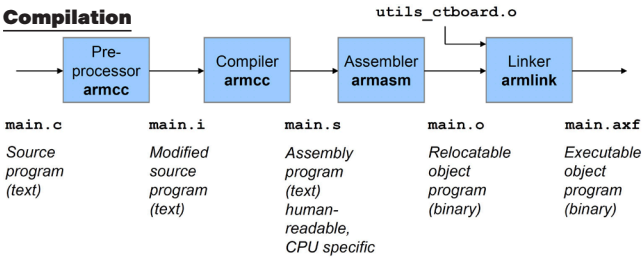
Main memory (Arbeitsspeicher): volatile (flüchtig) und non-valatile, SRAM (static RAM), Zugriff auf einzelne Bytes
Sekundärspeicher: non-valatile: ROM (factory programmed), Flash (in system programmable), Blockweiser Zugriff

System-Bus

Single bus for instructinos and data.

Address Lines: 2^N addresses means N address lines.
Constrol Signal: CPU signals is read or write access + handles bus timing
Data Lines: Transfer of data. 4/8/16/32/64 data lines.

Compilation



Program Execution on Embedded Systems

1. **Host** (Developer-Environment) compiles and links program.
2. Loader on **target** (Device) loads executable from host (through Serial interface) to RAM and copies into non-volatile (persistent) memory (Flash) = Firmware update (eingebettete Software).
3. Loader on target jumps to main() and starts execution. (System operation without host)

Sizes
- **Opcode** is **2 Bytes** (16 Bits, only even instruction addresses)
- **Register** stores **4 Bytes** (32 Bits)
- **Memory address** is **4 Byte** long
- **Memory-cell** stores **1 Byte**

CPU (Cortex-M Architecture)

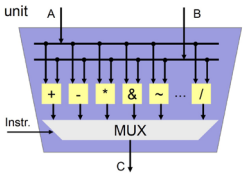
Hardware Plattforms

FPGA (Field-programmable gate array): Logik der Transistoren sind programmierbar. Anwendungs-Bsp.: Firmare update CRC Polynom.

CPU Model
- Instructions Set
- Processing width (8/16/32 bit)
- Register set (Register = Speicherplatz im Processor. Schnell verfügbar)
- Addressing modes: How memory an IO can be accessed

CPU Components: Core Registers, ALU (32-bit), Flags (APSR), Control Unit with IR (Instruction Register), Bus Interface

32-bit ALU (Arithmetic Logic Unit)

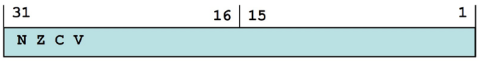


2 Register-Werte werden in der ALU bearbeitet. Resultat geht zurück in Register.

Alle Operationen (+, –, *, &, etc.) werden **parallel** ausgeführt. Die Control Unit wählt dann das benötigte Resultat aus (Instr. – MUX → C).

Flag-Register / APSR (Application Processor Status Register):

Bits set by CPU based on ALU result: **N**egative, **Z**ero, **C**arry, **O**verflow

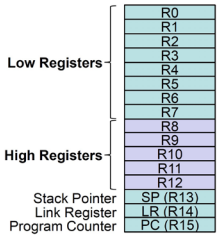


Core Registers

Used for temporary storage of data and addresses.

Low Registers R0 – R7: 3-bit addresses
High Registers R8 – R12: 4-bit addresses

Program Counter (PC): Address of next instruction
Link Register (LR): Saved PC. Where to jump back to after returning from a procedure.
Stack Pointer (SP): Address in the stack

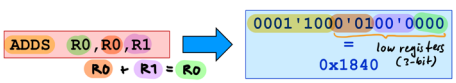


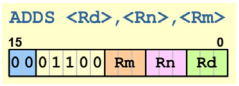
Control Unit
- Controls executions based on instructions in IR.
- Generates control signals for all other CPU components.

Intruction Register (IR): Machine code **(opcode)** of instruction that executed. Intruction Register is 4 Bytes.

Instruciton Set

Assembler translates assembly → binary instructions:





Basic Assembly Program

Memory address	Opcode	Label	Instr.	Operands	Comments
00000000	20A5	demoprg	MOVS	R0,#0xA5	; copy 0xA5 into R0
00000002	2111		MOVS	R1,#0x11	; copy 0x11 into R1
00000004	1840		ADDS	R0,R0,R1	; add contents of R0 and R1
					; store result in R0
00000006	4A00		LDR	R2,=0x2000	; load address into R2
00000008	6010		STR	R0,[R2]	; store content of R0 at
					; the address given by R2
0000000A	00002000				

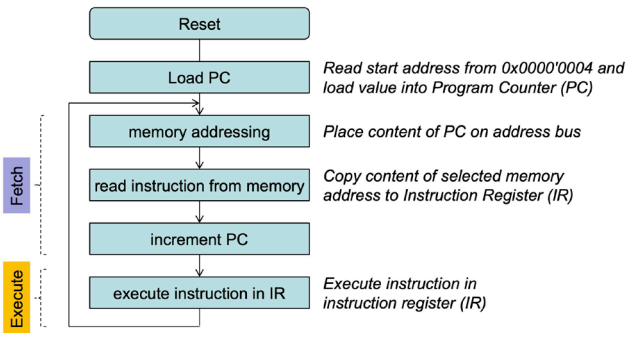
Listfile

Opcode = 16-bit instruction (machine code)
(Linker still will add an offset to those memory addresses.)

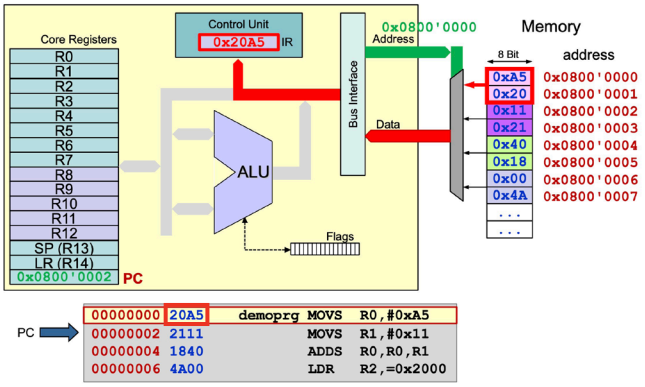
Instructino Types
- Data Transfer (load, move, store)
- Data Processing (Arithemtic and logic operations)
- Control Flow: Branches (goto, if-else) and Function calls (subroutines)

Program Execution

Precondition, Loader has:
- copied executable code into memory
- stored code start address in memory



Example of step: Read Instruction from memory (and Increment PC)

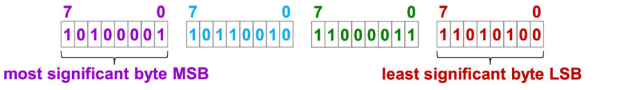


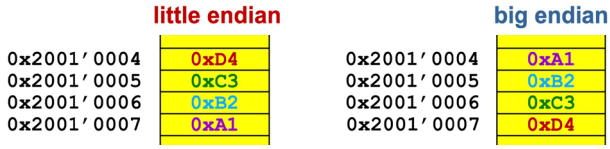
Integer Types (ARM Cotex-M)

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

Little vs Big Endian

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





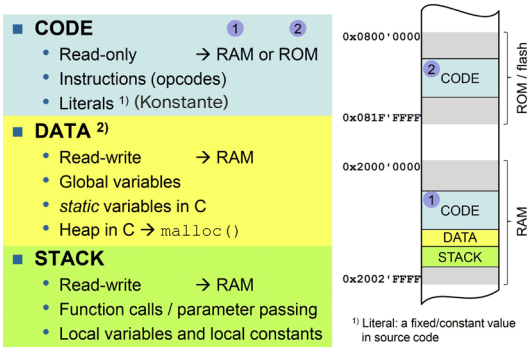
Big endian ist komplizierter für die Speicherbearbeitung. Cortex ist little end.
Little Endian: Niedrige Adressen für niedrige Bits.

Alignment

- **Half-word aligned:** Variables align on **even addresses**
- **Word aligned:** Variables align on addresses that are **divisible by four**

PC (Program Counter) is **halfword aligned.** Each instruction is 2 bytes.
SP (Stack Pointer) is **word aligned.** Each entry is 4 bytes (= Register size).

Memory Map (Graphical Layout of Object File Sections)



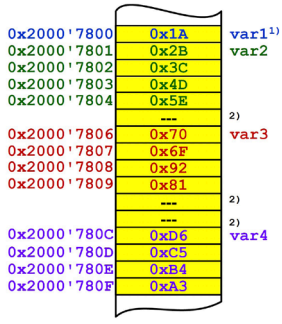
`CODE READONLY` → ROM
`CODE READWRITE` → RAM (volatile), wird grundsätzlich nicht verwendet.

AREA: Ein Assembly Programm ist in Sektionen (areas) aufgeteilt.

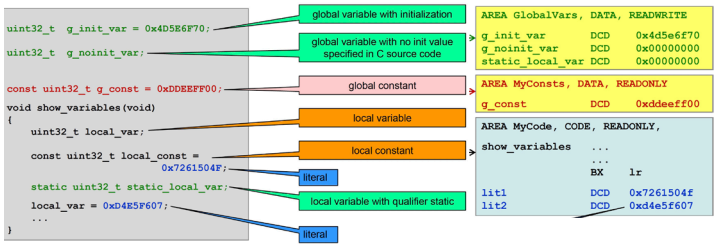
Alignment

Half-word aligned Variables aligned on **even addresses.**

Word aligned Variables aligned on **addresses divisible by four.**




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



Memory Allocation Directives

Initialized data

- **DCB** **1 Byte**
- **DCW** **Half-words,** 2 Bytes, addr. must be **even**
- **DCD** **Words** (word aligned), 4 Bytes, addr. must be **divisible by 4**

Can be located in DATA or CODE area

```
var1  DCB  0x1A
arr1  DCB  0x2B, 0x3C, 0x4D, 0x5E ; Array of 4 × 1 Byte
arr2  DCW  0x6F70, 0x8192         ; Array of 2 × 2 Byte
var2  DCD  0xA3B4C5D6             ; Word of 4 Byte
```

Uninitialized data

```
data1  SPACE  256  ; Reserve 256 bytes
```

Data Transfer

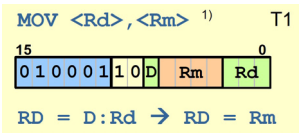
Transfer Types

CPU	→ CPU	**Register to** register
Memory/IO	→ CPU	**Load data / literals** from memory
CPU	→ memory/IO	**Store data** to memory

Register to Register

MOV low and high registers (any to any), does not update flags
MOVS low registers only (R0–R7), and **update flags (APSR)**

```
MOV R5,R8 ; Content of R8 into R5
opcode
0b0100011001000101 or 0x4645
```




```
;MOVS   R1,R8    ; not possible: high reg
;MOVS   R8,R1    ; not possible: high reg
;MOVS   R8,R9    ; not possible: high reg

;MOV R1, #0x12345678  ; zu gross für low registers
LDR R1, =0x12345678   ; macht Literal Pool
```

Loading from Memory

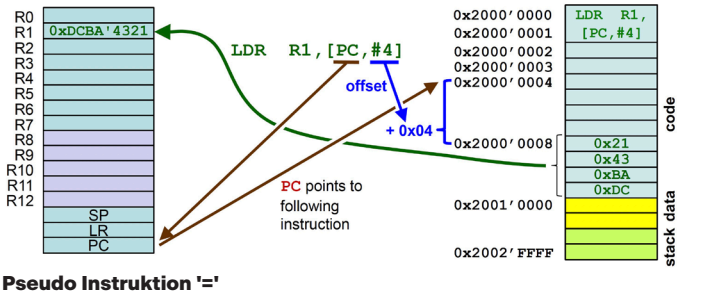
Loading Literals from Source (immediate Data):

```
MOVS R1, #0x1C ; MOVS <Rd>, #<imm8>    low registers only

MY_CONST EQU  0x12          ;MOVS   R8,#MY_CONST8   high reg not possible
         MOVS r1, #MY_CONST ;MOVS   R1,#0x100       immediate out of range
```

Moves the immediate value directly into register a register from source code.

Loading Data from Memory:

```
LDR <low Rt>, [<low Rn>, #<imm5>]   ; pointer arithemtic
; Load content of [base + offset] into Rt. Only low registers.
```



Pseudo Instruktion '='

Kann nur mit LDR verwendet werden. Loads a value into a register **from a memory address.** The assembler will generate a location in memory (a literal pool) to hold the value of the variable.

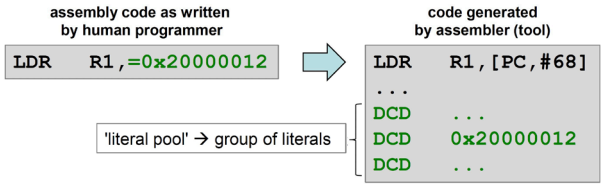
```
LDR R5, mylita      ; Content of mylita

LDR R5, =0x20003000 ; Literal value
; translates into LDR R5, [PC, #offset]

LDR R5, =CONST_A    ; Value of CONST_A
; translates into LDR R5, [PC, #offset]

LDR R5, =mylita     ; Address of mylita
```

Literal Pool

Der Assembler macht noch keine Adress-Zuweisungen, reserviert aber Platz am Ender der Code-Sektion dafür (DCB, DCW, DCD).
Der Assembler berechnet den Offset für uns und lädt die Daten.

Assembler → List-File (.lst) mit opcodes
Einige Instruktionen noch nicht fertig assembliert.
Man kann ablesen, wieviel Platz die AREAs eines Moduls benötigen.

Linker → Map-File (.map) mit Adressen (Value)
Enthält Informationen über den Linking-Prozess.
Welches Symbol sich wo im Speicher befindet.



Literal Pool Example

```
CONST_A                 EQU     0x000000AA
; opcode
0000003C 4F03           LDR     R7,=0x12
0000003E 4F04           LDR     R7,=CONST_A ; value of CONST_A
00000042 4D01           LDR     R5,mylita   ; content of mylita
00000044 4D04           LDR     R5,=mylita  ; address of mylita
00000048 FF001122 mylita DCD    0xFF001122
; literal pool
0000004C 00000012
00000050 000000AA
00000058 00000000 ; space for the address of mylita (after linking)
```

C Example

```
static uint32_t g;

void lit_example(void) {
    uint32_t a;
    uint32_t b;
    const uint32_t c = 0x04;
    uint32_t *p;

    a = 0x05;
    b = 0xABCDEF12; 1)
    p = &g;
    ...
}
```

```
                AREA MyCode, CODE, ...
...     ...         ...
000002  2304        MOVS  R3,#4
000004  2005        MOVS  R0,#5
000006  4904        LDR   R1,lit1
000008  4a04        LDR   R2,adr_g 2)
...     ...         ...
000018      lit1    DCD   0xabcdef12
00001C      adr_g   DCD   g 3)

                AREA MyData, DATA, ...
                g       DCD   0x00000000
```

Compiler assigns variables to registers: a → R0, b → R1, c → R3, p → R2

Storing Data into Memory

```
; Equivalent to LDR but other direction
STR <Rt>, [<Rn>, #<imm5>] ; only low registers
```

Summary

```
LDR <into Register> [<from memory>]
STR <from Register> [<into memory>]
```

MOVS <Rd>,<Rm>	Register to register	
MOVS <Rd>,#<imm8>	Loading literals	8-bit literal
LDR <Rt>,[PC,#<imm>]		32-bit literal, PC-relative
LDR <Rt>,[<Rn>,#<imm>] also LDRB and LDRH	Loading data	Register indirect with immediate offset
LDR <Rt>,[<Rn>,<Rm>] also LDRB, LDRH, LDRSB and LDRSH		Register indirect with register offset
STR <Rt>,[<Rn>,#<imm>] also STRB and STRH	Storing data	Register indirect with immediate offset
STR <Rt>,[<Rn>,<Rm>] also STRB and STRH		Register indirect with register offset

Load value into high register (R8-R12)

```
LDR R0,=0x12345678  ; load immediate value
MOV R9,R0
```

I/O (Memory Mapped Items)

Volatile (in C):
- Used for accessing memory mapped items (I/O)
- Compiler darf nichts optimieren. Wenn wir mit I/O arbeiten.

Arrays

C-code

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

assembly

```
halfword_array
        DCW     0x0011,0x2233
        DCW     0x4455,0x6677
        DCW     0x8899,0xAABB
```

Accessing array elements

element address = base address + element size • index

Example

```
Address  Value
                    AREA    my_data, DATA, READWRITE
00000000 11223344 my_array      DCD     0x11223344
00000004 55667788               DCD     0x55667788
00000008 99AABBCC               DCD     0x99AABBCC
0000007C 4906     LDR     R1,=my_array   ; load address of array
00000082 684D     LDR     R5,[R1,#0x04]  ; base R1, immediate offset
```

Arithmetic and Logic Operations

Flags and APSR (Application Program Status Register)

Processor does not know wheter the user is working with unsigned (C) or signed (V). Therefore C and V are always calculated.

Flag	Meaning		Relevant for
Negative	N = 1	if MSB = 1	signed
Zero	Z = 1	if result = 0	signed , unsigned
Carry	C = 1	if carry	unsigned
Overflow	V = 1	if overflow	signed

Mnemonic	Instruction	Function
ADD / ADDS	Addition	A + B
ADCS	Addition with carry	A + B + c
ADR	Address to Register	PC + A
SUB / SUBS	Subtraction	A - B
SBCS	Subtraction with carry (borrow)	A - B - NOT(c)
RSBS	Reverse Subtract (negative)	-1 • A
MULS	Multiplication	A • B

ADDS <Rd>, <Rn>, <Rm>
Update of flags, only low register (low to low)

```
ADDS R1,R2,R3  ; add R2 and R2 and save in R1
ADDS R1,R1,R2
ADDS R1,R2     ; equivalent to above
```

ADD <R0–R12>,<R0–R12>
No update of flags, any to any registers, only 8bit immediate
Same register is used for **first operand** and **result!**

```
ADD R8,R8,R9
ADD R1,R2,R3   ; illlegal
```

Negative Zahlen und 2-er Komplement

Processors do not distinguish signed and unsigned.

Negative Zahlen werden in binär im 2-er Komplement enkodiert. und haben in binär haben eine 1 als MSB.

Umrechnen von binär in dezimal (und vise versa):
Komplement nehmen und +1 rechnen.

Umrechnen in binär: **-5** → 5 → 0101 → 1010 + 1 → **1011**
Umrechnen in dezimal: **1011** → 0100 + 1 → 1010 → 5 → **-5**

Invert a Number (Reverse Subtract)

RSBS <Rd>,<Rn>,#0 Rd = 0 - Rn
Generates 2' complement, Updates flags, Only low register

```
; Example: var = -var
LDR  R0,=var    ; load addr of var
LDR  R1,[R0]    ; get value of var
RSBS R1,R1,#0   ; invert value of var
STR  R1,[R0]    ; store var
```

Substraction / Addition

unsigned numbers → check **carry flag (C)** after operation:
- For **Addition:** **C = 1** (carry / no borrow), result is too large.
  C = 0 result is within range.
- For **Subtraction:** **C = 0** (borrow), result is less than zero (not in range)
  C = 1 result is larger than 0.

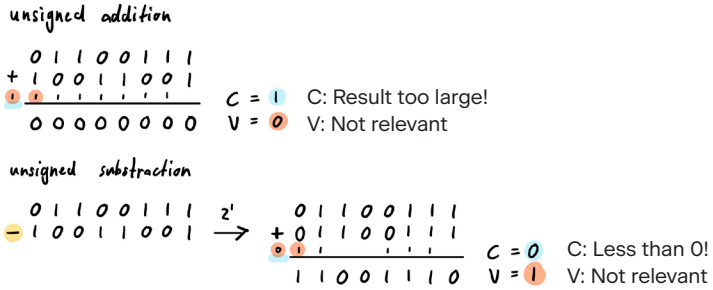
signed numbers → check **overflow flag (V)** after operation:
V = 1: Not enough digits available to represent the result.
Nur möglich wenn **(+a)+b** oder **(–a)–b.**

Unsigned Subtraction

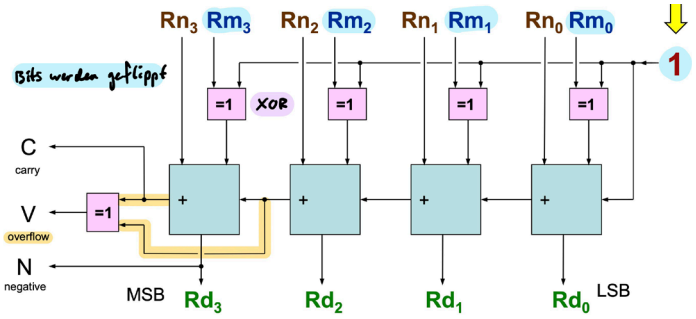
There is no subtraction in Binary → instead: Addition of 2' complement. Bsp:

5 – 2 → 5 + (-2) → 0101 + 1110 → [C=1]0011 = 3 C=1: Result is in range

Beispiele



Substraction in Hardware



1 for Substraction, 0 for Addition

Signed/Unsigned Multiplication

Result requires twice as many binary digits as operands: 4-bit · 4-bit → 8-bit

5 · 3 = 15

unsigned

```
0101 * 1101
  00001101
 00000000
 00001101
00000000
000001000001
```

zero extension of multiplier

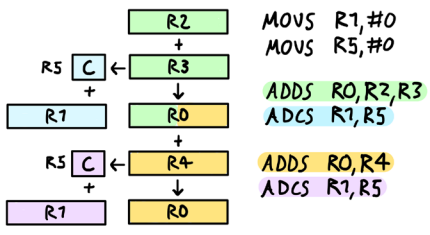
5 · -3 = -15

signed

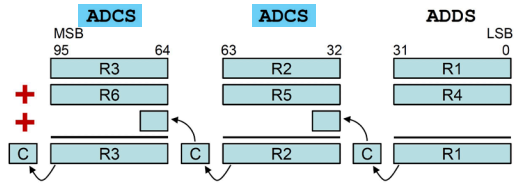
```
0101 * 1101
  11111101
 00000000
 11111101
00000000
100111110001
```

sign extension of multiplier

Example: 32 bis + 32 bit + 32 bit = 64 bit



Multi-Word Arithmetic Example: 96 bit + 96 bit = 96 bit




```
ADDS  R1,R1,R4
ADCS  R2,R2,R5   ; add with carry
ADCS  R3,R3,R6   ; add with carry

MRS   R4,APSR    ; read flags [N,Z,C,V] (upper 4 bits)
LSRS  R4,R4,#24  ; right shift to lower 4 bits
```

Dazwischen darf **keine andere Operation** durchgeführt werden, sonst werden die Flags evtl. ungültig.

Casting

Integer ranges based on word sizes

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

signed ↔ unsigned in C

C casted immer nach unsigned, wenn ein unsigned Operator vorkommt.

Expression	Type	Evaluation
0 == 0U	unsigned	1
-1 < 0	signed	1
-1 < 0U	unsigned	0
2'147'483'647 > -2'147'483'647 - 1	signed	1
2'147'483'647U > -2'147'483'647 – 1	unsigned	0
2'147'483'647 > (int) 2'147'483'648U	signed	1
-1 > -2	signed	1
(unsigned) -1 > -2	unsigned	1

Extension

```
int_8_t  → int16_t  → int32_t  → int64_t   signed
uint_8_t → uint16_t → uint32_t → uint64_t  unsigned
```

Sign Extension, Bsp. 4 Bit → 8 Bit

Unsigned → Zero Extension

1011 → 0000 1011 0011 → 0000 0011

Signed → Sign Extension

1011 → 1111 1011 0011 → 0000 0011

Truncation

```
int64_t  → int32_t  → int16_t  → int_8   signed
uint64_t → uint32_t → uint16_t → uint_8  unsigned
```

Cast **cuts** the left most digits

Unsigned → modulo Operation

```
uint32_t x = 287962;
uint16_t sx = (uint16_t)x;
```

0x000464DA → 287'962
0x64DA → 25'818

Signed → possible change of sign!

```
int32_t x  = 53191;
int16_t sx = (int16_t) x;
```

0x0000CFC7 → 53'191
0xCFC7 → -12'345

Instructions

```
LDR  R0,=0xB5 ; 0b10110101
; extend an 8-bit value to a 32-bit value without changing value
SXTB R1,R0    ; 0xFFFFFFB5
; extend an 8-bit value to a 32-bit value
UXTB R1,R0    ; 0x000000B5
```

Opcode-Aligment

Half-words (2 Bytes): Address must be **even**
Words (4 Bytes): Address must be **divisible by 4**

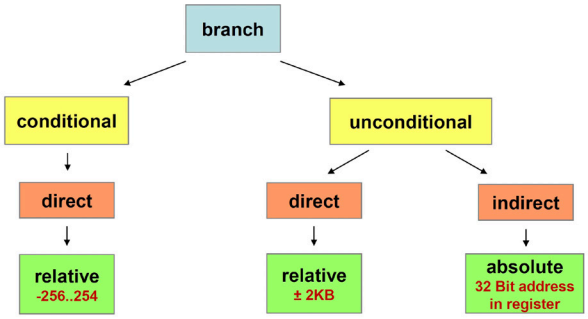
```
opcode                    mnemonic
1000 0 00111 101 011  ←  STRH R3,[R5,#0x0E]  ; half word alignment
       14/2=7                      14

01001 101 00010011    ←  LDR R5,[PC,#0x4C]   ; word alignment
          76/4=19                  76
```

Hex-Werte

```
10 11 12 13 14 15
A  B  C  D  E  F
```

Branch Instructions

Branch = Change of PC (Programm counter)



Direct: Target address is part of instruction. `B label`
Indirect: Target address is in register. `BX R0`

Compare and Test

CMP <Rn>,<Rm> = Rn – Rm → N,Z,C,V
Same as **SUBS** but without storing the result. Only flags are affected.
Used to **compare two values** (lt, gt, eq, etc).

TST <Rn>,<Rm> = Rn & Rm → N,Z
Logical **AND** but without storing the result. Changes N and Z.
Used to **check if a specific bit is set** → `BEQ bit_not_set` or `BNE bit_is_set`

Unconditional Branches

(branch always)

```
B   <label> ; direct/rel,   PC = PC ± offset (imm)
BL  <label> ; direct/rel,   PC = PC ± offset (imm), LR = PC
BLX <Rm>    ; indirect/abs, PC = Rm,                LR = PC – 2
BX  <Rm>    ; indirect/abs, PC = Rm
LDR R0,=label
BX  R0
```

Conditional Branches

Branch only if condition is met.

```
B<c> <label>  ; direct/relativ
```

Unsigned Arithemtic (higher, lower)

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 Arithemtic (greater, less)

Symbol	Condition	Flag	
EQ	Equal	Z == 1	==
NE	Not equal	Z == 0	!=
MI	Minus/negative	N == 1	< 0
PL	Plus/positive or zero	N == 0	>= 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	<=

Flag Dependent

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	< 0
PL	Plus/positive or zero	N == 0	>= 0
VS	Overflow	V == 1	
VC	No overflow	V == 0	

```
BAL   ; branch always
```

Logic Instructions

Bit Manipulations

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

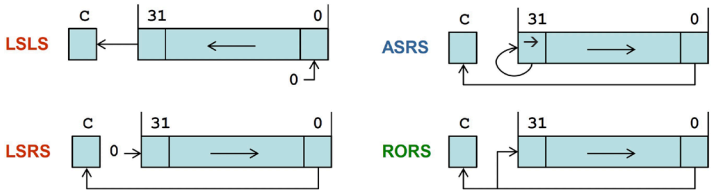
```
MOVS    R2,#0x1C        ;00011100b
EORS    R1,R1,R2
```

Invert all bits

```
MVNS R0,R0
```

Shift / Rotate

LSLS	Logical Shift Left	Für unsigned/signed: $2^n \cdot Rn$ (Multip.)
LSRS	Logical Shift Right	Für unsigned: $2^{-n} \cdot Rn$ (Division)
ASRS	Arithmetic Shift Right	Für signed: $2^{-n} \cdot Rn$ (Division)
RORS	Rotate Right	



Control Structures

If-Then-Else

```
if (nr >= 0) {
    isPositive = 1;
}
else {
    isPositive = 0;
}
```

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

```
    CMP   R1,#0
    BGE   then
    MOVS  R2,#0
    B     end_if
then
    MOVS  R2,#1
end_if
```

Do-While

```
sum = 0;
do {
    sum += nr;
} while (sum < 100);
```

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

While Loops

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

```
    MOVS  R2,#1
    B     test
loop
    MULS  R2,R1,R2
test
    CMP   R2,#100
    BLT   loop
    ...
```

Switch

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

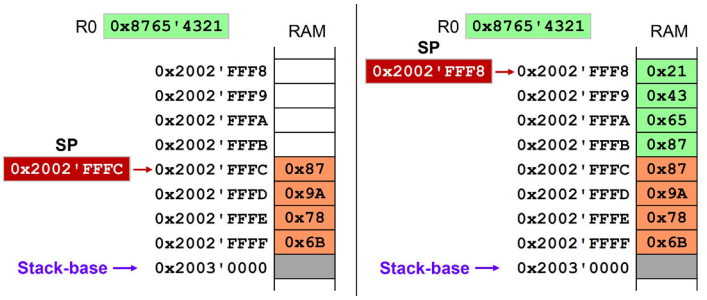
Assume: n in R1
result in R2

```
NR_CASES        EQU   6

case_switch     CMP   R1, #NR_CASES
                BHS   case_default
                LSLS  R1, #2    ; * 4
                LDR   R7, =jump_table
                LDR   R7, [R7, R1]
                BX    R7   ; offset to table
case_0          ADDS  R2, R2, #17
                B     end_sw_case
case_1          ADDS  R2, R2, #13
case_3_5        ADDS  R2, R2, #37
                B     end_sw_case
case_default    MOVS  R2,#0
end_sw_case     ...
```

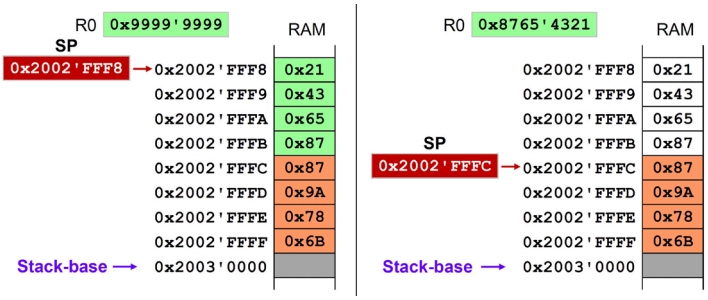
```
jump_table      DCD   case_0          n = 0
                DCD   case_1          n = 1
                DCD   case_default    n = 2
                DCD   case_3_5        n = 3
                DCD   case_default    n = 4
                DCD   case_3_5        n = 5
```

Stack

```
PUSH {R0}   ; Push changes SP (Stack Pointer) and Stack.
```

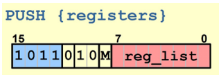



```
POP {R0}    ; Pop changes SP and Registers.
```

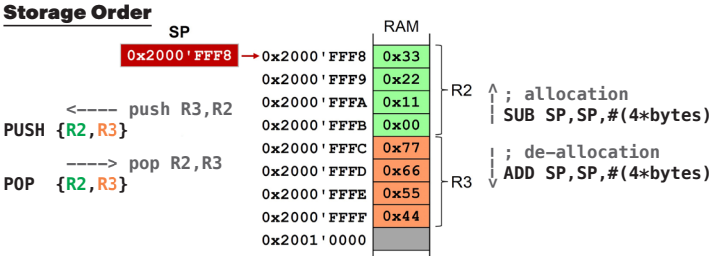


Register List

```
PUSH {R1,R3,R4,R5}    reg_list = 0'0011'1010
POP  {R1,R3,R4,R5}
```



Storage Order



Subroutines and Parameter Passing

Terminology

Routine: Caller
Subroutine: Callee
Procedure: Routine (caller) returning no result.
Function: Routine (caller) returning a result.

Subroutine

R0–R3 and the stack are used as **parameters** and **return values.**
R4–R7 are used by the callee as local variables (must back them up first).
Callee is free to use R0–R3 (without having to back them up).

1. **Caller** uses R0-R3 as arguments and backups them up (if necessary):
   `PUSH {R0-R3}` or `MOV R4,R0`
   Additional arguments are copied onto the stack: `PUSH {R4-R7}`
2. **Callee** pushes R4-R7 (if needed) and LR onto the stack.
3. **Callee** saves the return values to R0-R3 (additionals onto the stack)
4. **Callee** pops R4-R7 (if needed) and PC from the stack, restoring LR.
5. **Caller** gets the return values from R0-R3 (`MOVS R4,R0`).
6. **Caller** restores R0-R3 if necessary: `POP {R0-R3}` or `MOV R0,R4`

Call a subroutine with **`BL <label>`**
Return with **`POP {PC}`** or **`BX LR`** (if R4-R7 were nod used)

```
my_proc PROC   ; PROC/FUNCTION directive for for debugging
        PUSH {R4-R7,LR}
        ; do something with R0-R3 as parameters
        ; save return values to R0-R3
        POP {R4-R7,PC}
        ENDP   ; ENDP/ENDFUNC

; call the subroutine
        PUSH {R0-R3}        ; backup R0-R3 (if necessary)
        BL my_proc          ; call the subroutine
        MOVS R4,R0          ; get return values (if any) in R0-R3
        POP {R0-R3}         ; restore R0-R3 (if necessary)
```

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[1)]
r13	SP	
r14	LR	
r15	PC	

Register contents might be modified by callee

Callee must preserve contents of these registers (Callee saved)

Cortex-M0: Registers r8 – r11 have limited set of instructions. Therefore, they are often not used by compilers.

Scratch Register: Used to hold an intermediate value during calculation.

Pass by Value

```
; Doubles the value in R0 (R0 is argument and return)
double  PROC
        LSLS R0,R0,#1
        BX LR
        ENDP
; caller
        MOV R0,#0x03
        BL double
        ; R0 is now 0x06
```

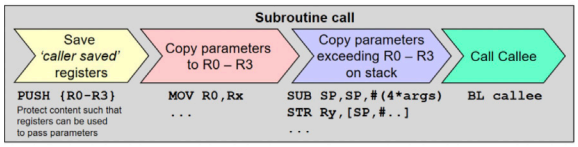
Pass by Reference

Passing an address (by value) is called pass-by-reference = pointer.
Necessary for **any type larger than 4 bytes.**

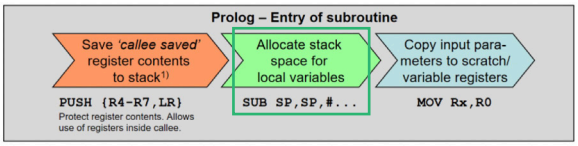
```
my_proc PROC
        PUSH {R4,LR}
        STR  R4,[R0]    ; store something to the variable
        POP  {R4,PC}
        ENDP
; caller
        LDR  R0,=my_var ; pass the address of the variable
        BL   double
```

ARM Procedure Call Standard

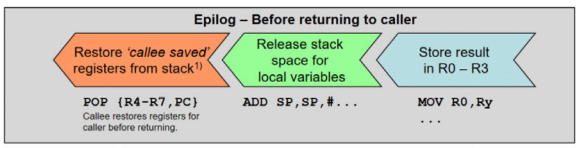
Caller

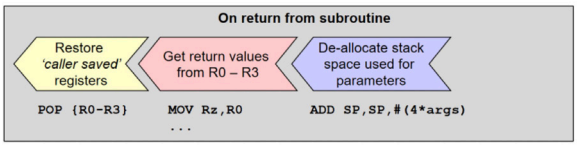


Callee



Code of subroutine





Use the **Stack for local variables** (as opposed to Registers). Example:

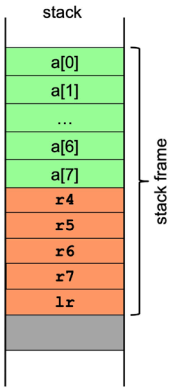
```
my_proc PROC
        PUSH {R4-R7,LR}  ; backup registers and LR
        ; allocate space for 3 local variables
        SUB  SP,SP,#12   ; 3 * 4 bytes
        ; access local variables
        STR  R0,[SP,#0]
        STR  R1,[SP,#4]
        STR  R2,[SP,#8]
        ; deallocate space of local variables after we're done
        ADD  SP,SP,#12
        ; restore registers and PC
        POP  {R4-R7,PC}
        ENDP
```

Stack Frame

Example

```
int main(void)
{
  uint32_t start = 0x1234;
  uint32_t result = 0;

  result = calc(start);
}

uint32_t calc(uint32_t val)
{
  uint32_t a[8];
  uint32_t res = 0;
  ...
  a[0] = val;
  ...
  return res;
}
```

```
MOV r0,... ;start
BL  calc

PUSH {r4-r7,lr}
;allocate a[] on stack
SUB sp,sp,#0x20
                    32d = 8·4
MOVS  r6,#0  ;res

;access a[0]
STR r0,[sp,#0x00]

MOV  r0,r6  ;res
;stack teardown
ADD sp,sp,#0x20
POP {r4-r7,pc}
```



Calling Assembly Subroutine from C

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

```
            PRESERVE8
            AREA    SCopy, CODE, READONLY
            EXPORT strcopy

            ; R0 points to destination string
            ; R1 points to source string
strcopy     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  strcopy      ; Keep going if not
            BX   LR           ; Return
 END
```

Modular Coding

High Cohesion: Each module fulfills a (single) defined task.
Each module can easily be reused.

Low Coupling: Little dependencies between modules.
Modules can be tested individually.

Encapsulation: Information hidign. Internal implementaion are hidden.
An interface may have alternative implementations.

Declaration and Definition

«Declare before used»

Declare

```
uint32_t square(uint32_t v); // square function defined elsewhere
extern uint32_t counter;     // counter variable defined elsewhere
struct S;                    // struct S type defined elsewhere
```

`extern` bedeutet, die Variable (gilt nur für Variablen) ist in einer anderen Unit definiert. Function and type declarations do not need the «extern» keyword.

Definition

```
uint32_t square(uint32_t v) { ... } // square function definition
uint32_t counter;                   // counter variable definition
struct S { ... };                   // struct S type definition
```

Memory is allocated where the definition is.
Each definition is implicitly also a declarations of the given name.

Header Files

Avoids duplication of declarations. Nowhere in the source code we include the source files (.c) directly. The linker will resolve the references to the definitions in the source files.

Linking

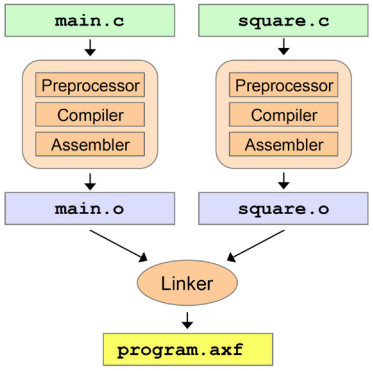
Linker

After compilation and assembling, the linker is responsible for:

1. **Merging code and data section:** Merge all sections of individual object files into (first) one *temporary object file*, then
2. **Symbol resolution:** Search missing addresses of external symbols → merged symbol tables (.symtab)
3. **Symbol relocation:** updating addresses in the code section (.code) (because merging sections invalidates the original addresses).

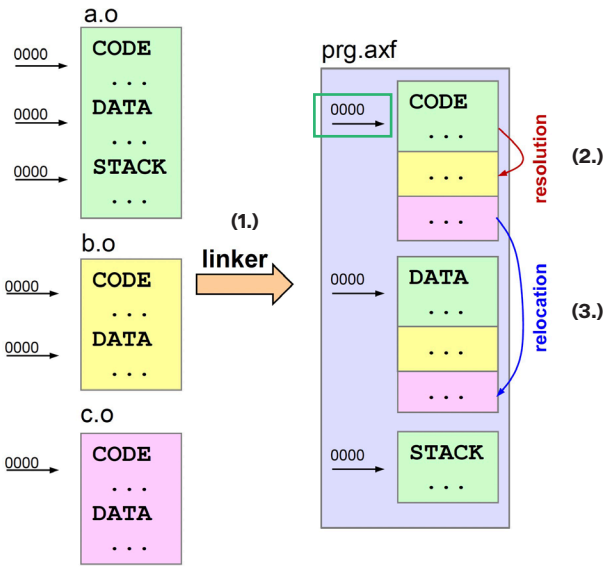
The object files holds placeholders for the memory addresses (functions and variables). The linker replaces these placeholders with the actual memory addresses. In short, Linker **merges object files** and **adds/replaces correct memory addresses.** The linker only cares about object files and is therefore ignorant of the used programming language.

C-declarations/definitions → Assembly symbols → Object file symbols



Each source file gets translated into one object file.

The object files get linked into one executable file.



Module relativ offset

Internal and External Linkage

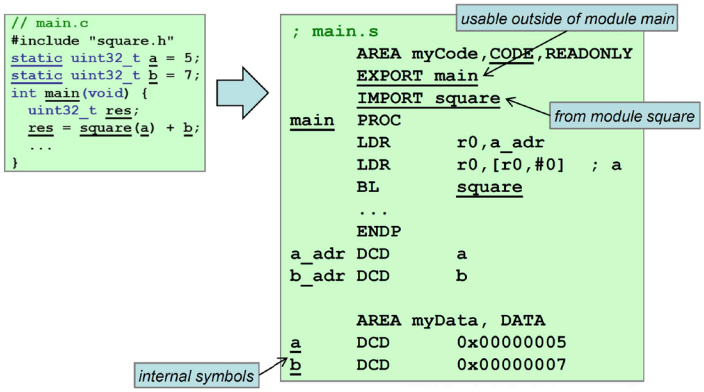
- **External linkage:** All global names, except declared static.
- **Internal linkage:** All (global) variables or functions declared `static`. `static` variables are only visible in the same translation unit.
- **No linkage:** Local variables (e.g. in a function)

```
#include "square.h"
static uint32_t a = 5;
static uint32_t b = 7;
int main(void) {
  uint32_t res;
  res = square(a) + b;
  ...
}
```

```
uint32_t square(uint32_t v) {
  return v*v;
}
```

C-Declarations/Definition → Assembly Symbol

- C-**definitions** with **external linkage** translates into **EXPORT** symbol.
- C-**declarations** with **external linkage** translates into **IMPORT** symbol.
  e.g. **#include** translates into IMPORT symbol.



Assembly Symbol → Object File

Symbol Translations

- **Imported** symbols (**external linkage declaration,** #include)
  → **global reference** symbols (References, ***Gb Ref***)
- **Exported** symbols (**external linkage definitions,** functions and externals)
  → **global** symbols (Global, ***Gb***)
- **Internal** symbols (**internal linkage,** local variables)
  → **local** symbols (Local, ***Lc***)

That means, a **Reference** (imported/global reference) must be declared **Global** (exported/global) in another file, otherwise linkage will fail.

Steps of a Linker

1. Merge

Example of merged **code section:**

code (main.o)

```
0x00000000 B510      PUSH    {r4,lr}
0x00000002 4804      LDR     r0,[pc,#16]
0x00000004 6800      LDR     r0,[r0,#0x00]
0x00000006 f7fffffe  BL      square
0x0000000A 4903      LDR     r1,[pc,#12]
0x0000000C 6809      LDR     r1,[r1,#0x00]
0x0000000E 1844      ADDS    r4,r0,r1
0x00000010 2000      MOVS    r0,#0x00
0x00000012 BD10      POP     {r4,pc}
0x00000014 00000000  DCD     0x00000000
0x00000018 00000000  DCD     0x00000000
```

code (square.o)

```
0x0000001C 4601      MOV     r1,r0
0x0000001E 4608      MOV     r0,r1
0x00000020 4348      MULS    r0,r1,r0
0x00000022 4770      BX      lr
```

Example of merged **data section:**

data (main.o)

```
0x00000000 05000000  DCD   5
0x00000004 07000000  DCD   7
```

Example of merged **symbol table section:**

symbols (main.o)

```
 #  Symbol Name                Value       Bind  Sec  Type  Vis  Size
==============================================================================
 7  a                          0x00000000  Lc     4  Data  De   0x4
 8  b                          0x00000004  Lc     4  Data  De   0x4
11  main                       0x00000001  Gb     1  Code  Hi   0x14
12  square                     0x00000000  Gb   Ref  Code  Hi
```

symbols (square.o)

```
 #  Symbol Name                Value       Bind  Sec  Type  Vis  Size
==============================================================================
 6  square                     0x00000001  Gb     1  Code  Hi   0x8
```

2. Symbols Resolution

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

No relocation done yet. The relative values of the symbols within the modules are not yet relocated to global addresses.

3. Relocation

Relocation Calculation:
address = global base + merge offset + module relative offset

symbol *b*

global base	data section (Architektur abhängig)	0x20000000
merge offset	1st in merged **data section**	+ **0x00000000**
module relative offset	*b* is 2nd after *a*	+ **0x00000004**
address		= **0x20000004**

symbol *square*

global base	user code starts at 0x08000254	0x08000254
merge offset	2nd in merged **code section**	+ **0x0000001C**
module relative offset	(no offset)	+ 0x00000000
address		= **0x08000270**

Relocated symbols

```
 20  a                          0x20000000  Lc     4  Data  De   0x4
 21  b                          0x20000004  Lc     4  Data  De   0x4
186  main                       0x08000255  Gb     1  Code  Hi   0x14
187  square                     0x08000270  Gb     1  Code  Hi   0x8
```

→ **Symbol table** updates merged object files.

Relocated code sections

```
0x08000254 B510      PUSH    {r4,lr}                 ; main
0x08000256 4804      LDR     r0,[pc,#16]
...
0x08000270 4601      MOV     r1,r0                   ; square
0x08000272 4608      MOV     r0,r1
...
```

Adjusted code locations according to relocation table

```
...
0x0800025A F000F809  BL.W    square      ; 0x08000270
...
0x08000268 20000000  DCD     0x20000000
0x0800026C 20000004  DCD     0x20000004
```

Relocated data sections

```
0x20000000 00000005  DCD   5                         ; value of a
0x20000004 00000007  DCD   7                         ; value of b
```

Relocated relocation table entries

```
 0  0x0800025A   10 R_ARM_THM_CALL          12 square
 1  0x08000268    2 R_ARM_ABS32              7 a
 2  0x0800026C    2 R_ARM_ABS32              8 b
```

Aus einem executable können nur Symbole (Variablen und Funktionen) herausgelesen werden. Bei Funktionen fehlt aber die Signatur und bei Variablen den Wert.

Symbol Table - Liste der internen, importierten und exportierten Symbole

Relocation Table - Adressen der Instruktionen und Daten deren Zugriff auf Symbole im Laufe des Linkens angepasst werden müssen

Data Section - Initialisierungswerte von globalen Variablen

Code Section - Maschinen Instruktionen und konstante Daten

Exceptions and Interrupts

Polling

Synchronous and Periodic query of status information in a loop,
e.g. reading status registers SPI, UART, GPIO (pins), etc.

Advantage: Simple to implement, snychronous, deterministic.
Disadvantage: Wastes CPU time, reduced throughput, long reaction time.

Interrupts

Sudden change/interupption of program flow. Interrrupt (e.g. peripheral signal) calls **ISR (Interrupt Service Routine).**

Advantage: No busy wait, better use of CPU time.
Disadvantage: No synchronization (manuall synchronization needed through mutex and semaphore), difficult debugging.

Interrupt Exception Types

Total 255 Exceptions Types (Cortex-M3/M4).

- **Device Interrupt (IRQ0–IRQ239):** Peripherals signal to CPU
- **System Exceptions (Vom System ausgelöst):**
  - **Reset:** Restart of processor (beim Aufstarten der CPU)
  - **NMI (Non-Maskable Interrupt):** Condition that cannot be ignored, e.g. critical hardware error or screenlock on mobile
  - **Faults:** undefined instructions, unaligned accesses, etc.
  - **System Level Calls:** Operating system (OS) calls, Supervisor Call (SVC)

Vector Table

Holds addresses of ISR (Interrupt Service Routine).
Each exception has a different ISR.

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

n in IRQ**n** = Exception# – Exception# of IRQ**0**
e.g. IRQ**1** = 17 – 16 = **1**

Initialization of Vector Table in my programm (Cortex-M3/M4):

```
; 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
…               DCD   ...
…               DCD   ...

                ; Interrupts
16              DCD   IRQ0_Handler       ; ISR for IRQ0
17              DCD   IRQ1_Handler       ; ISR for IRQ1
…               DCD   ...

                AREA  SOURCE_CODE, CODE, READONLY
IRQ0_Handler    PUSH { ... }

                ...         ; interrupt service

                POP { ... }
                BX LR
```

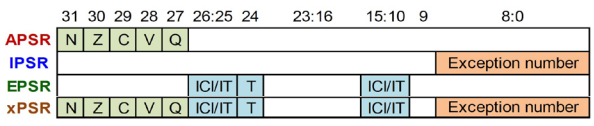
System Exceptions 15 vectors

Interrupts

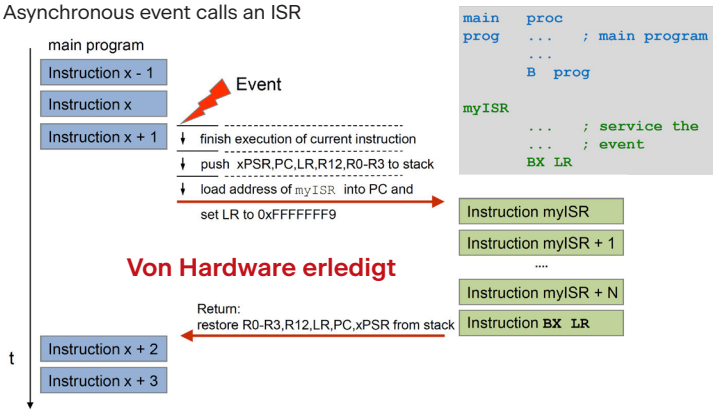
Program Status Register (PSRs)

The hardware will save the current state of the program when an ISR call event occurs. It saves the state into the PSRs.

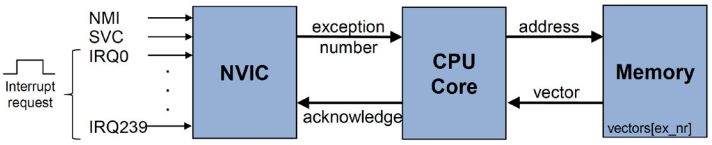
APSR	Application Program Status Register
IPSR	Interrupt Program Status Register
EPSR	Execution Program Status Register
xPSR	Combination of all three PSRs



Asynchronous event calls an ISR



Interrupt-System (Cortex-M3/M4)



NVIC selects and forwards IRQ with highest priority
CPU requests vector from memory
Memory holds vector table (my programm)

Nested Vectored Interrupt Controller (NVIC)

Setzt interrupts auf **Pending, Enable** oder **Active.**
Setzt auch die **Priorität** (Priority Level).

Durch **PRIMASK** (Bitmaske) können interrupts manuell **enabled** oder **disabled** werden.

PRIMASK (Masking of Interrrupts):

PRIMASK = 0 bedeutet alles ist enabled. Setting a bit, disables the interrupt.

Assembly: `CPSID <bitmask>` und `CPSIE <bitmask>`
CPSIE/D = Change Processor State Interrupt Enable / Disable

C: `__disable_irq();` und `__enable_irq();`

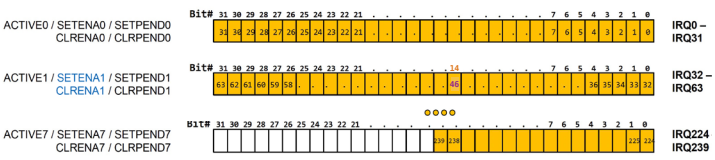
NMI (Non-Maskable Interrupts) können nicht disabled werden.

Enable and Disable IRQ

Example: IRQ**46**

Select **Interrupt Enable Register** and **Bit** to set.

⌊ **46** ÷ 32 ⌋ = **1** → SETENA**1**
46 mod 32 = **14** → **Bit 14**



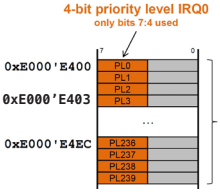
Kontrolle: **1** · 32 + **14** = **46**

```
SETENA1     EQU  0xE000E107
CLRENA1     EQU  0xE000E187

            ; enable IRQ46
            LDR  R0,=SETENA1
            MOVS R1,#0x0E ; set bit 14
            STR  R1,[R0]

            ; disable IRQ46
            LDR  R0,=CLRENA1
            MOVS R1,#0x0E ; unset bit 14
            STR  R1,[R0]
```

Interrupt Enable Registers

Enable / disable: SETENAn / CLRENAn
Trigger / clear: SETPENDn / CLRPEND
Test: ACTIVEn

Set Priority to IRQ

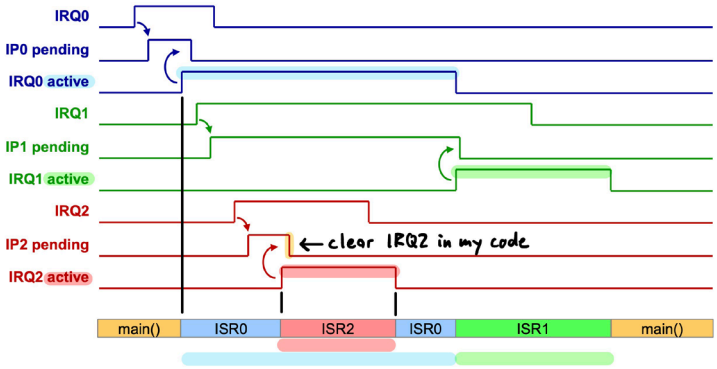



```
; IRQ3 -> PL3
PL_IRQ3     EQU  0xE000E403
            LDR  R0,=PL_REG_IRQ3
            MOVS R1,#0x50 ; set upper 4 bits
            STRB R1,[R0]
```

Nested Exceptions and Priorities

assuming

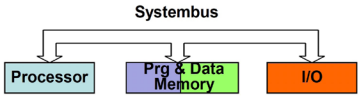
IRQ0	PL0 = 0x2	medium priority
IRQ1	PL1 = 0x3	lowest priority
IRQ2	PL2 = 0x1	highest priority



Bus Architectures

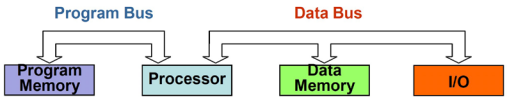
von Neumann Architecture

Single bus system between CPU and memory.
Data + Program (Instructions) share Memory.



Harward Architecture

Two sets of buses. Program and Data have their own Memory.



RISC (ARM) and CISC (Intel) Paradigms

RISC (Reduced Instruction Set Computer)

Fixed length instructions. Load/Store architecture (data processing is between registers).

- Fast decoding, simple addressing
- Less hardware, allows higher clock rates
- More chip space for registers (up to 256)

```
LDR   R0,=Credit
LDR   R1,[R0]
LDR   R0,=Balance
LDR   R3,[R0]
ADDS  R2,R1,R3
STR   R2,[R0]
```

CISC (Complex Instruction Set Architecture)

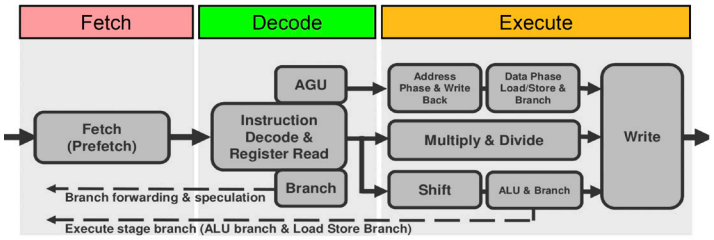
Variable length instructions.
One of the operands of an instruction may directly be a memory location

- Less program memory needed
- Short programs may run faster with less memory accesses

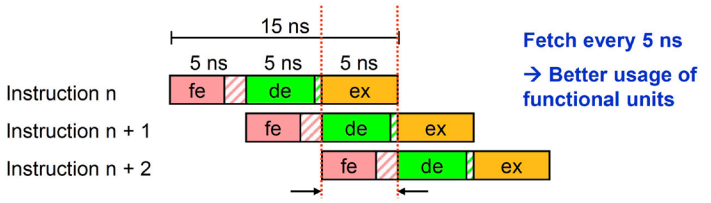
```
MOV  AX, [Credit]
ADD  [Balance], AX
```

Pipelining

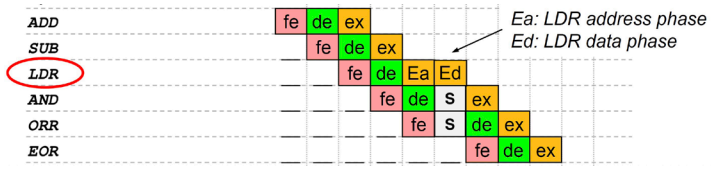
Advantages: Massive performance gain. **Disadvantage:** A blocking stage blocks whole pipeline. Multiple stages may need to have access to the memory at the same time → Memory-Bus Collision.



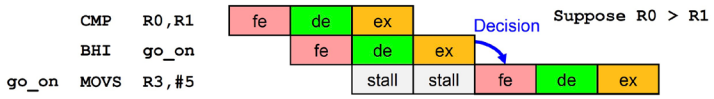
Idea: Fetch the next instruction, while the current one decodes.
Longest stage defines pipeline cycle.



LDR takes 2 instructions → next instruciton must wait (S = stall).



Branch and jump instructions occur in stage 3 (Execute).
Branches stall next decode and execution stages,
because it is unclear which instruction is next (no prefetch).



Instructions per second (IPS)

Without pipelining: *Instruction delay* = *fe* + *de* + *ex*

$$\textit{Instructions per second} = \frac{1}{\textit{Instruction delay}} \quad \text{[IPS]}$$

With pipelining: *Max stage delay* = max (*fe*, *de*, *ex*)

$$\textit{Instructions per second} = \frac{1}{\textit{Max stage delay}} \quad \text{[IPS]}$$

Varia

Edge detection

```
// edge = ~btn_{N-1} & btn_N
uint8_t btn_prev = 0;
while (1) {
  uint8_t btn_value = read_byte(ADDR_BUTTONS);
  uint8_t edge = ~btn_prev & btn_value;
  btn_prev = btn_value;
  // ...
}
```

Vorsätze für Masseinheiten (metric prefix)

peta	P	10^{15}	1 000 000 000 000 000
tera	T	10^{12}	1 000 000 000 000
giga	G	10^{9}	1 000 000 000
mega	M	10^{6}	1 000 000
kilo	k	10^{3}	1 000
hecto	h	10^{2}	100
deca	da	10^{1}	10
deci	d	10^{-1}	0.1
centi	c	10^{-2}	0.01
milli	m	10^{-3}	0.001
micro	µ	10^{-6}	0.000 001
nano	n	10^{-9}	0.000 000 001
pico	p	10^{-12}	0.000 000 000 001
femto	f	10^{-15}	0.000 000 000 000 001