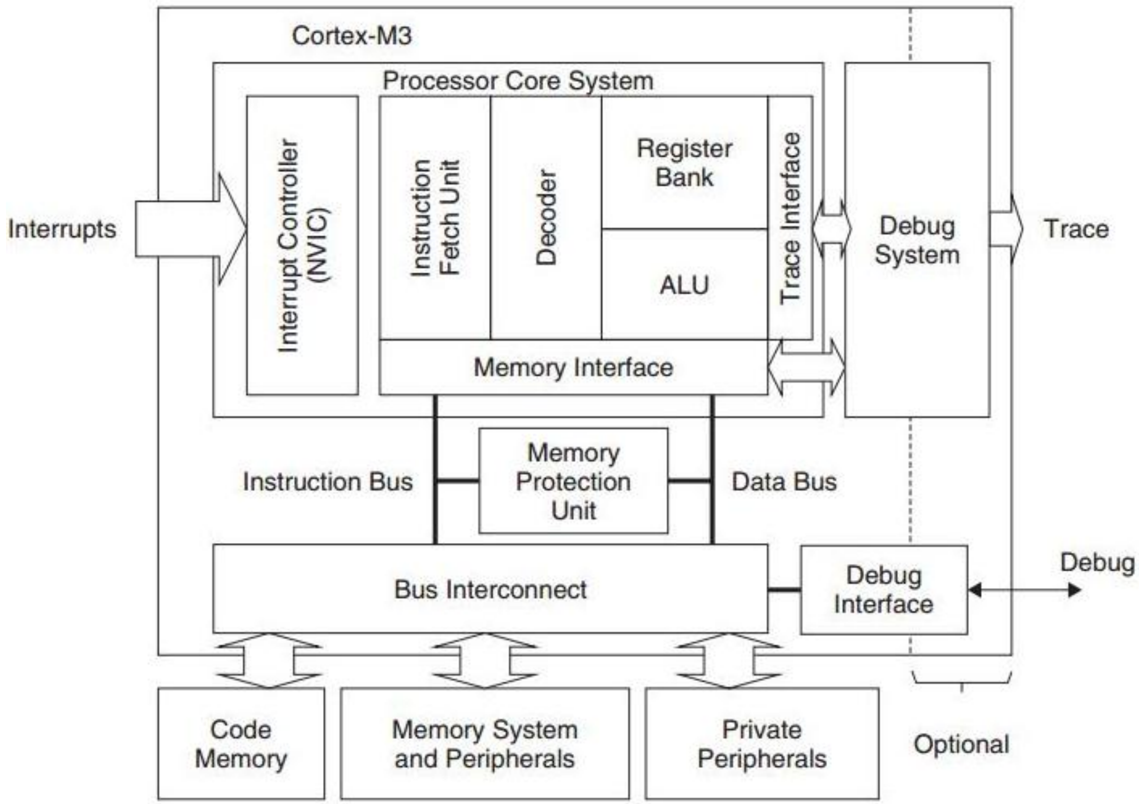
Vorlesungsskript zu „Vertiefung Programmieren“ CPU und Assembler

Dozent: Dipl.-Inf. (FH) Andreas Schmidt

ARM Cortex-M3/-M4



ARM Cortex-M3

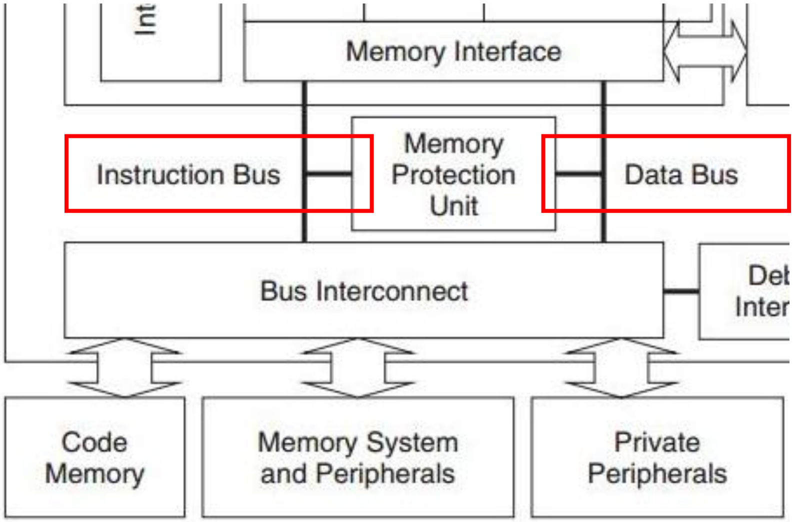
- 32-Bit Architektur
- Thump-2 Instruction Set
- Dreistufige Pipeline

ARM Cortex-M4

- DSP Funktionalität
- opt. FPU

Quelle: RM0440 Reference Manual

ARM Cortex-M3/-M4

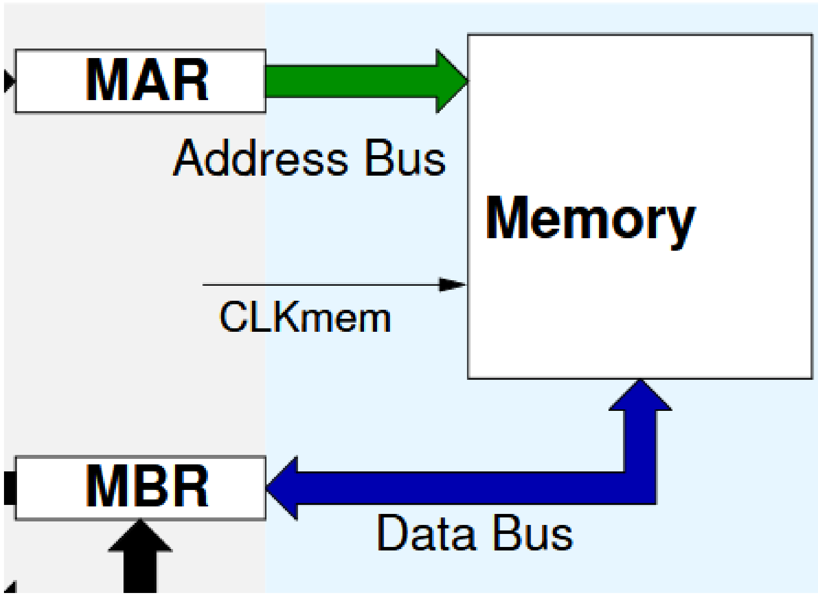


Der Cortex-M3/-M4 besitzt intern eine Harvard-Architektur
➔ Getrennte Bus-Systeme für Befehle und Daten

Das Programmier-Modell des Cortex-M3/-M4 ist allerdings eine Von-Neumann-Architektur
➔ Gesamter Adressraum kann linear adressiert werden

Quelle: RM0440 Reference Manual

CPU Memory



Zugriff auf den Speicher mit Hilfe des Adress- und Daten-Bus

Daten-Bus-Breite bestimmt die Anzahl der gleichzeitig zu übertragenden Bits bei einem Datentransfer

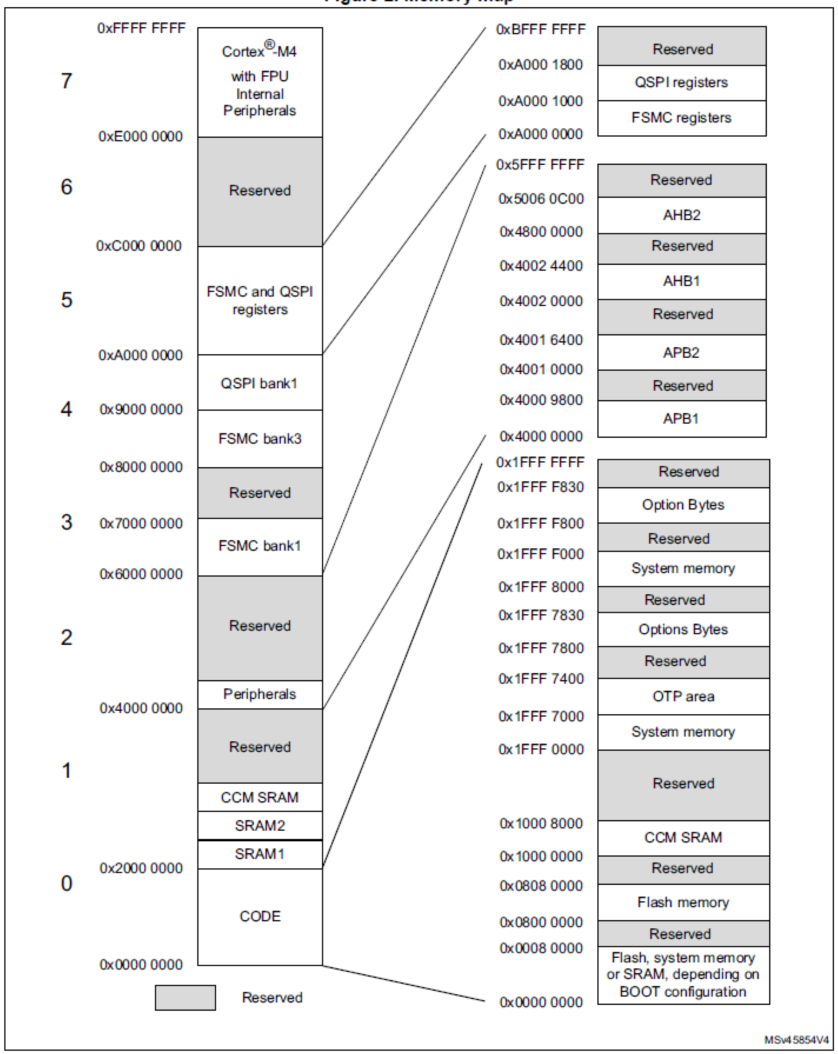
I.d.R.:
Registerbreite = Datenbusbreite

Adress-Bus-Breite bestimmt den adressierbaren Speicherbereich

2^{32} = 4.294.967.296 = 4 GB

CPU Memory

Figure 2. Memory map



Quelle: RM0440 Reference Manual

Speichermodell des Cortex-M3/-M4

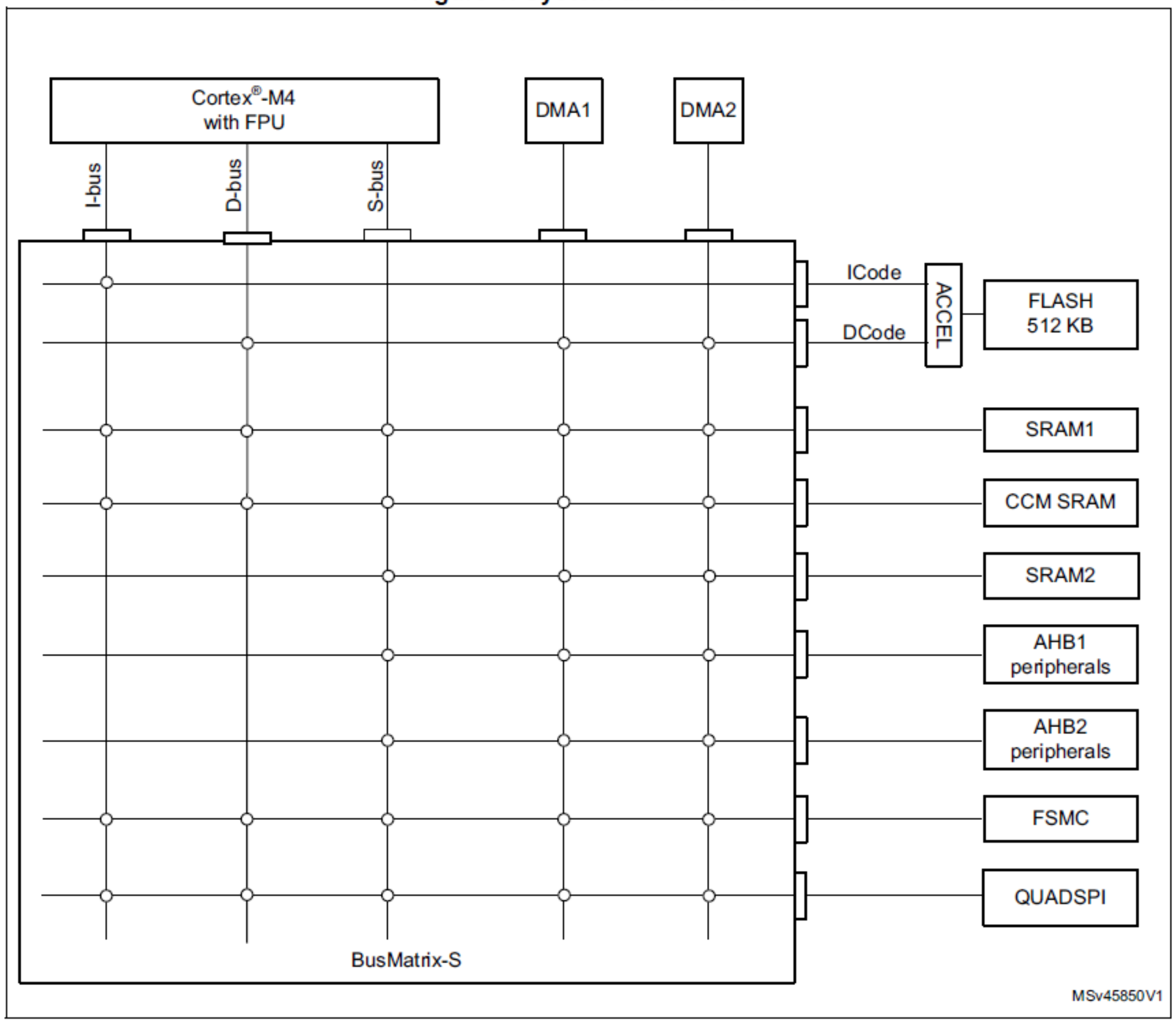
Linearer 32-Bit Adressraum in dem alle Speicher (Flash, SRAM etc.) sowie die Peripherie-Komponenten eingebettet sind.

Wichtig:

Beim Einsatz einer MMU (Memory Management Unit) und virtueller Speicherverwaltung kann der lineare, virtuelle Adressraum sich vom linearen, physikalischen Adressraum unterscheiden.

CPU Memory

Figure 1. System architecture



Quelle: RM0440 Reference Manual

32-Bit Speicher Aufbau

32 Bit „Wortbreite“

0x0014				
0x0010				
0x000C				
0x0008	0D	0C	0B	0A
0x0004	00	00	00	08
0x0000	00	00	00	00
	0	1	2	3

Big Endian: 0x0D0C0B0A
Little Endian: 0x0A0B0C0D

Bei Big-Endian wird das „große Ende“, also der signifikanteste Wert, zuerst abgelegt ➔ also in der niedrigsten Speicheradresse.

Bei Little-Endian wird das „kleinste Ende“, also der am wenigsten signifikante Wert, zuerst gespeichert.

Quelle: intern

32-Bit Speicher Aufbau (andere Darstellung)

32 Bit „Wortbreite“

0x0000	00	00	00	00
0x0004	00	00	00	08
0x0008	0D	0C	0B	0A
0x000C				
0x0010				
0x0014				
	0	1	2	3

Quelle: intern

Assembler – Maschinencode

Eine CPU führt Maschinen-Instruktionen aus, welche aus Sicht der CPU (z.B. ARM Core) aus 16- bzw. 32-Bit Daten bestehen

0xE0823003 ➔ Addition der Werte in Register R2 und R3 und speichert das Ergebnis in R3

Um Maschinen-Instruktionen für den Menschen besser verständlich zu machen, wurden sog. Mnemonics eingeführt. Hierbei handelt es sich um lesbare Abkürzungen für Maschinen-Instruktionen

```
add         r11,sp,#0x0
sub         sp,sp,#0xc
ldr         r3,[->globalVarNoInit]
mov         r2,#0x20
str         r2,[r3,#0x0]=>globalVarNoInit
ldr         r3,[->globalVarInit]
ldr         r2,[r3,#0x0]=>globalVarInit
ldr         r3,[->globalVarNoInit]
ldr         r3,[r3,#0x0]=>globalVarNoInit
add         r3,r2,r3
```

Table 21. Cortex-M4 instructions

Mnemonic	Operands	Brief description	Flags	Page
ADC, ADCS	{Rd,} Rn, Op2	Add with carry	N,Z,C,V	3.5.1 on page 83
ADD, ADDS	{Rd,} Rn, Op2	Add	N,Z,C,V	3.5.1 on page 83
ADD, ADDW	{Rd,} Rn, #imm12	Add	N,Z,C,V	3.5.1 on page 83
ADR	Rd, label	Load PC-relative address	—	3.4.1 on page 70

Quelle: intern

Assembler – Maschinencode

Register Transition
Pre-Comment
Label | XRef Header | XRef
+ | Address | Bytes | ... | Mnemonic | Operands | EOL Comment
PCode
Post-Comment
Space

```
00010000 04 b0 2d e5    str     r11,[sp,#local_4]!
00010004 00 b0 8d e2    add     r11,sp,#0x0
00010008 0c d0 4d e2    sub     sp,sp,#0xc
0001000c 40 30 9f e5    ldr     r3,[->globalVarNoInit]              = 00010068
00010010 20 20 a0 e3    mov     r2,#0x20
00010014 00 20 83 e5    str     r2,[r3,#0x0]=>globalVarNoInit       = ??
00010018 38 30 9f e5    ldr     r3,[->globalVarInit]                = 00010060
0001001c 00 20 93 e5    ldr     r2,[r3,#0x0]=>globalVarInit         = 0000ABCDh
00010020 2c 30 9f e5    ldr     r3,[->globalVarNoInit]              = 00010068
00010024 00 30 93 e5    ldr     r3,[r3,#0x0]=>globalVarNoInit       = ??
00010028 03 30 82 e0    add     r3,r2,r3
0001002c 08 30 0b e5    str     r3,[r11,#local_c]
00010030 08 30 1b e5    ldr     r3,[r11,#local_c]
00010034 10 30 83 e2    add     r3,r3,#0x10
00010038 1c 20 9f e5    ldr     r2,[->newResult]                    = 00010064
0001003c 00 30 82 e5    str     r3,[r2,#0x0]=>newResult             = DEADBEEFh
00010040 08 30 1b e5    ldr     r3,[r11,#local_c]
00010044 03 00 a0 e1    cpy     r0,r3
00010048 00 d0 8b e2    add     sp,r11,#0x0
0001004c 04 b0 9d e4    ldr     r11=>local_4,[sp],#0x4
00010050 1e ff 2f e1    bx      lr
```

Quelle: intern

Assembler – Beispiel Maschinencode Decodierung

`00010028 03 30 82 e0       add       r3,r2,r3`

Little Endian → Big Endian Convertierung notwendig

E0 82 30 03 ➔ 0b1110 0000 1000 0010 0011 0000 0000 0011

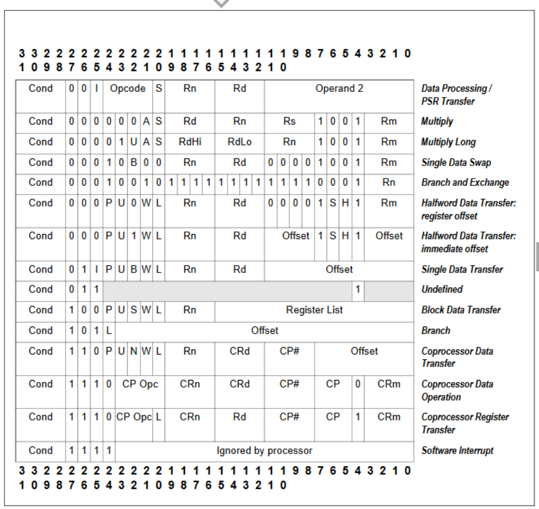
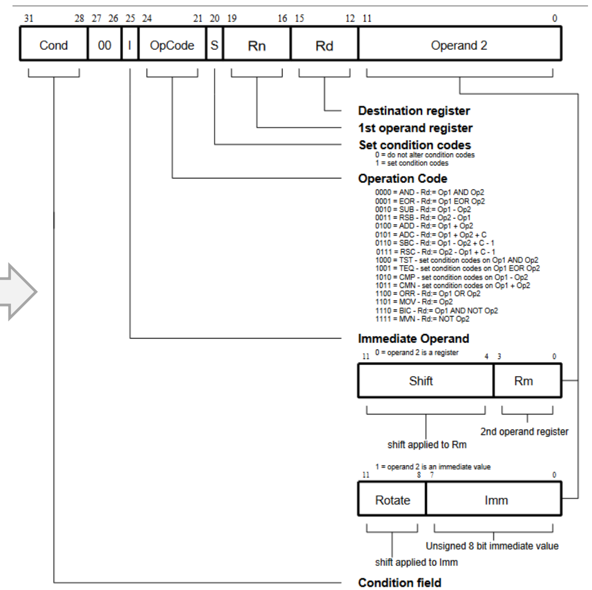


Figure 4-1: ARM instruction set formats



4 Bit Condition: 0b1110 ➔ Always
2 Bit Constant 0b00
1 Bit Immediate Op -> 0b0 ➔ Reg.
4 Bit Operand 0b0100 → ADD
1 Bit Condition Cod 0b0 → no altering cond.
4 Bit 1st Operand reg. 0b0010 → R2
4 Bit Dest Reg. 0b0011 → R3
12 Bit Operand 2
  8 Bit Shift operand 0b00000000 → no shift
  4 Bit Reg. 0b0011 → R3

➔Addition R3 = R2 + R3

https://www.scadacore.com/tools/programming-calculators/online-hex-converter/

Quelle: intern

Assembler – Beispiel Maschinencode Decodierung

→ 0b1110 0000 1000 0010 0011 0000 0000 0011

3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Cond	0	0	I	Opcode	S	Rn	Rd	Operand 2																		*Data Processing / PSR Transfer*
Cond	0	0	0	0	0	0	A	S	Rd	Rn	Rs	1	0	0	1	Rm										*Multiply*
Cond	0	0	0	0	1	U	A	S	RdHi	RdLo	Rn	1	0	0	1	Rm										*Multiply Long*
Cond	0	0	0	1	0	B	0	0	Rn	Rd	0	0	0	0	1	0	0	1	Rm							*Single Data Swap*
Cond	0	0	0	1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	Rn	*Branch and Exchange*
Cond	0	0	0	P	U	0	W	L	Rn	Rd	0	0	0	0	1	S	H	1	Rm							*Halfword Data Transfer: register offset*
Cond	0	0	0	P	U	1	W	L	Rn	Rd	Offset	1	S	H	1	Offset										*Halfword Data Transfer: immediate offset*
Cond	0	1	I	P	U	B	W	L	Rn	Rd	Offset															*Single Data Transfer*
Cond	0	1	1		1																					*Undefined*
Cond	1	0	0	P	U	S	W	L	Rn	Register List																*Block Data Transfer*
Cond	1	0	1	L	Offset																					*Branch*
Cond	1	1	0	P	U	N	W	L	Rn	CRd	CP#	Offset														*Coprocessor Data Transfer*
Cond	1	1	1	0	CP Opc	CRn	CRd	CP#	CP	0	CRm															*Coprocessor Data Operation*
Cond	1	1	1	0	CP Opc	L	CRn	Rd	CP#	CP	1	CRm														*Coprocessor Register Transfer*
Cond	1	1	1	1	Ignored by processor																					*Software Interrupt*

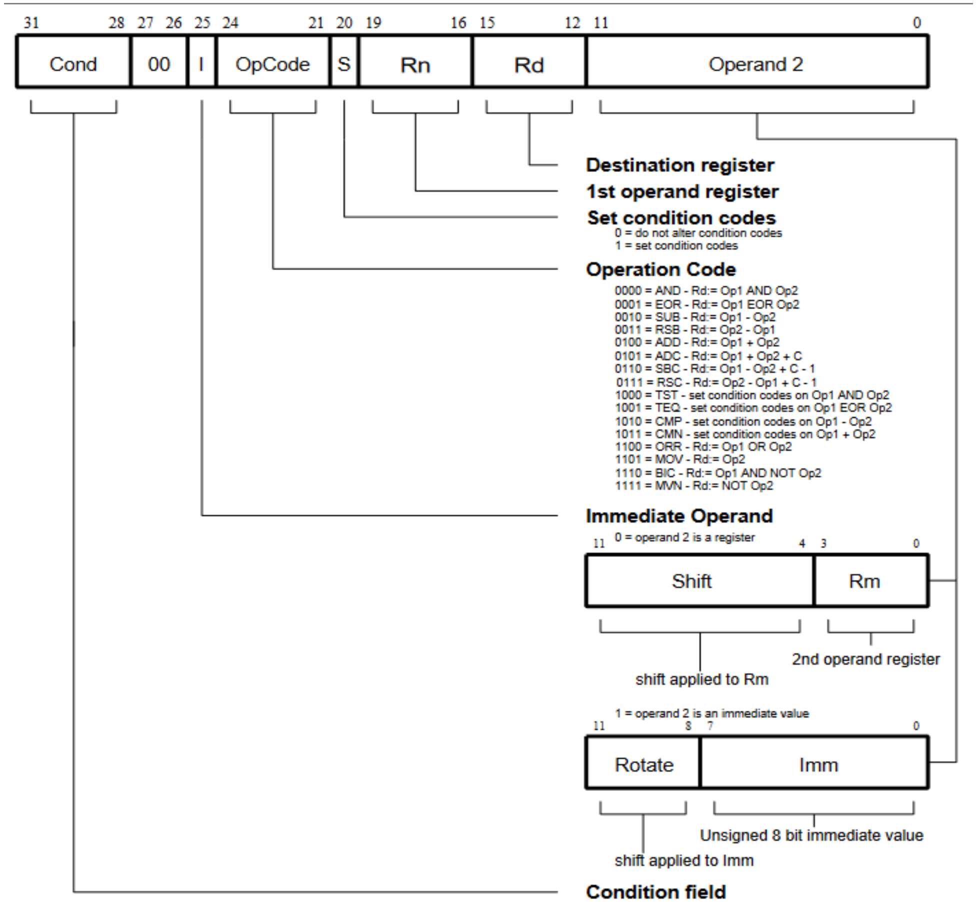
3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 9 8 7 6 5 4 3 2 1 0
1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0

Figure 4-1: ARM instruction set formats

Quelle: intern

Assembler – Beispiel Maschinencode Decodierung

➔ 0b1110 00 0 0100 0 0010 0011 0000 0000 0011



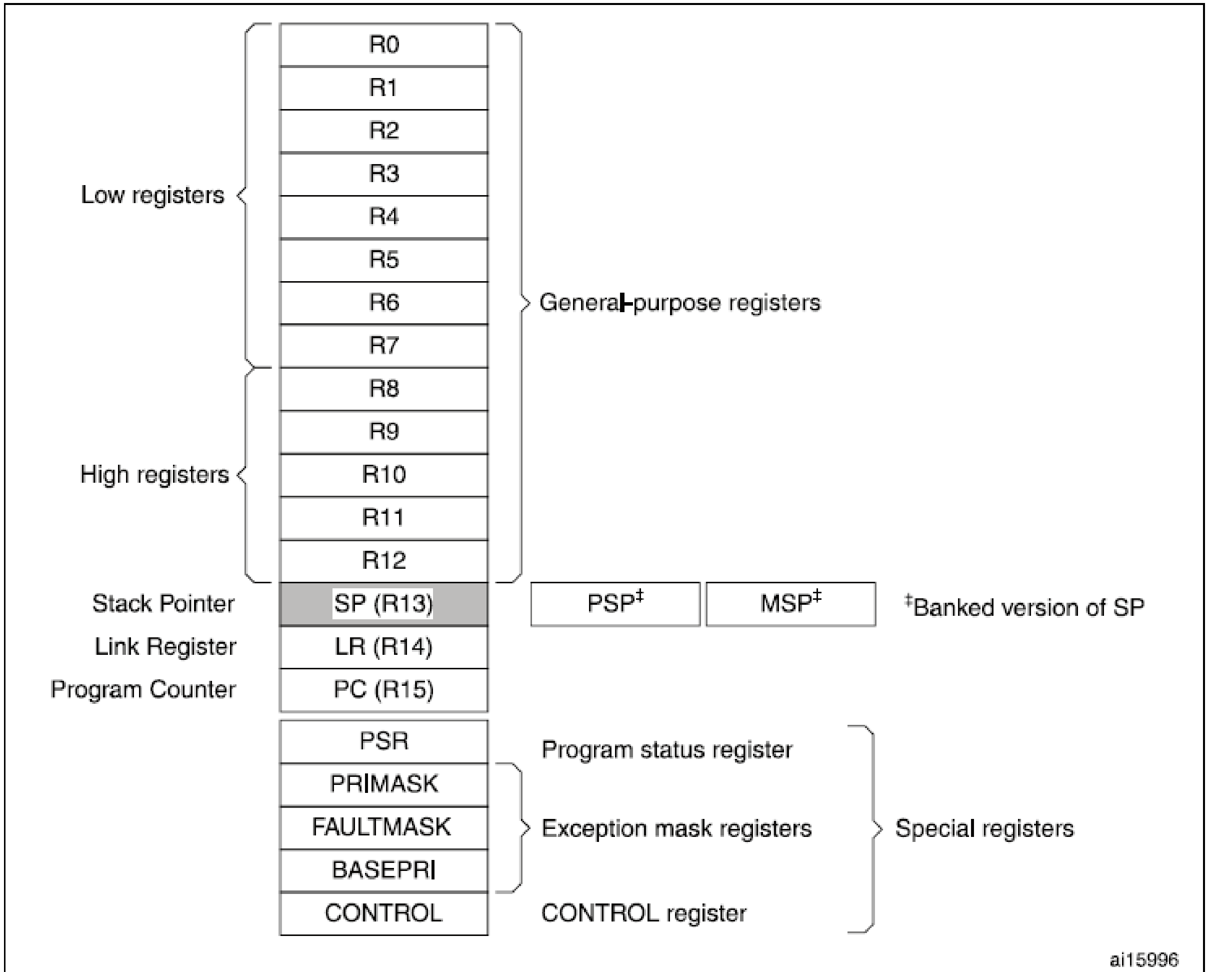
4 Bit Condition: 0b1110 ➔ Always
2 Bit Constant 0b00
1 Bit Immediate Op -> 0b0 ➔ Reg.
4 Bit Operand 0b0100 → ADD
1 Bit Condition Cod 0b0 → no altering cond.
4 Bit 1st Operand reg. 0b0010 → R2
4 Bit Dest Reg. 0b0011 → R3
12 Bit Operand 2
  8 Bit Shift operand 0b00000000 → no shift
  4 Bit Reg. 0b0011 → R3

➔Addition R3 = R2 + R3

Quelle: intern

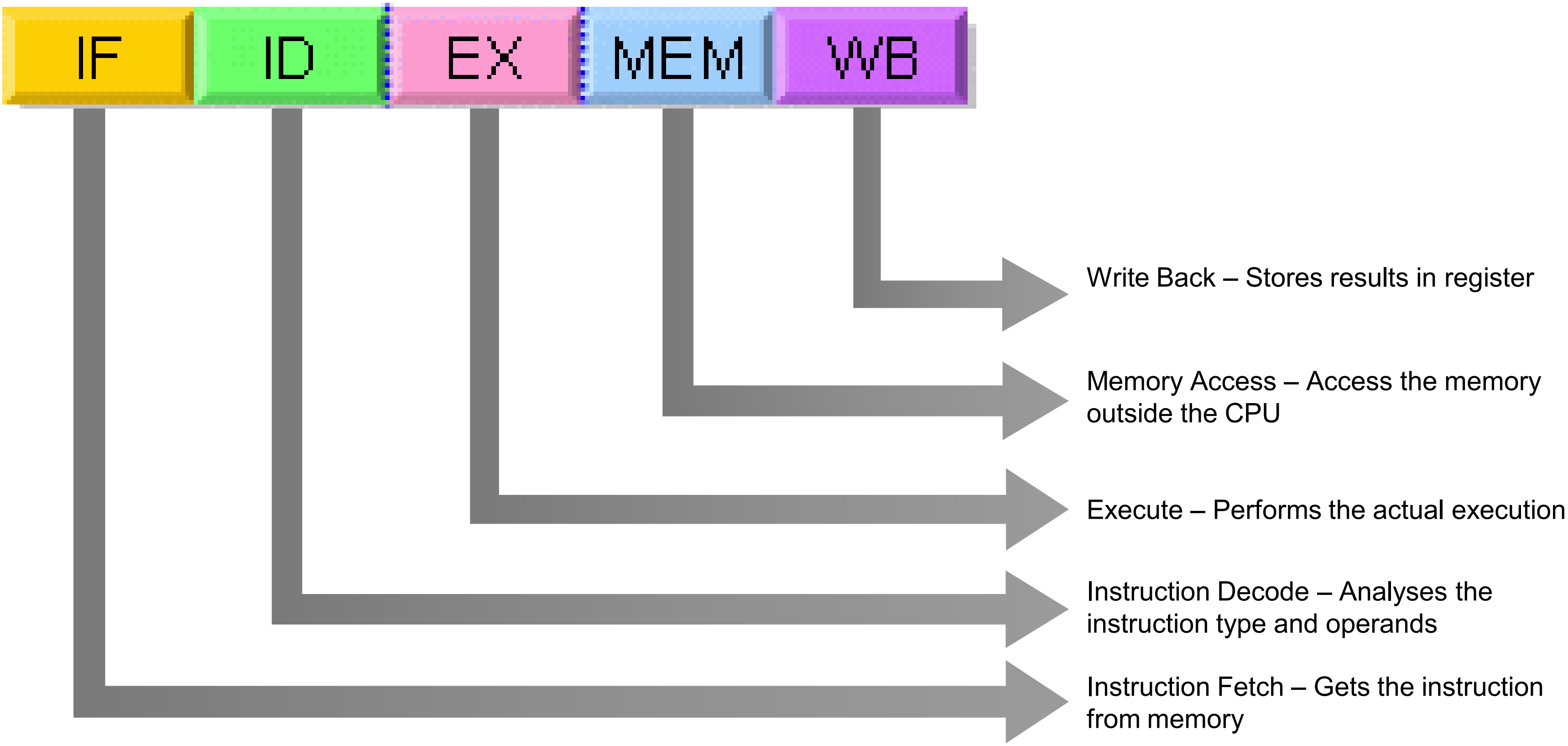
ARM Assembler

Figure 2. Processor core registers



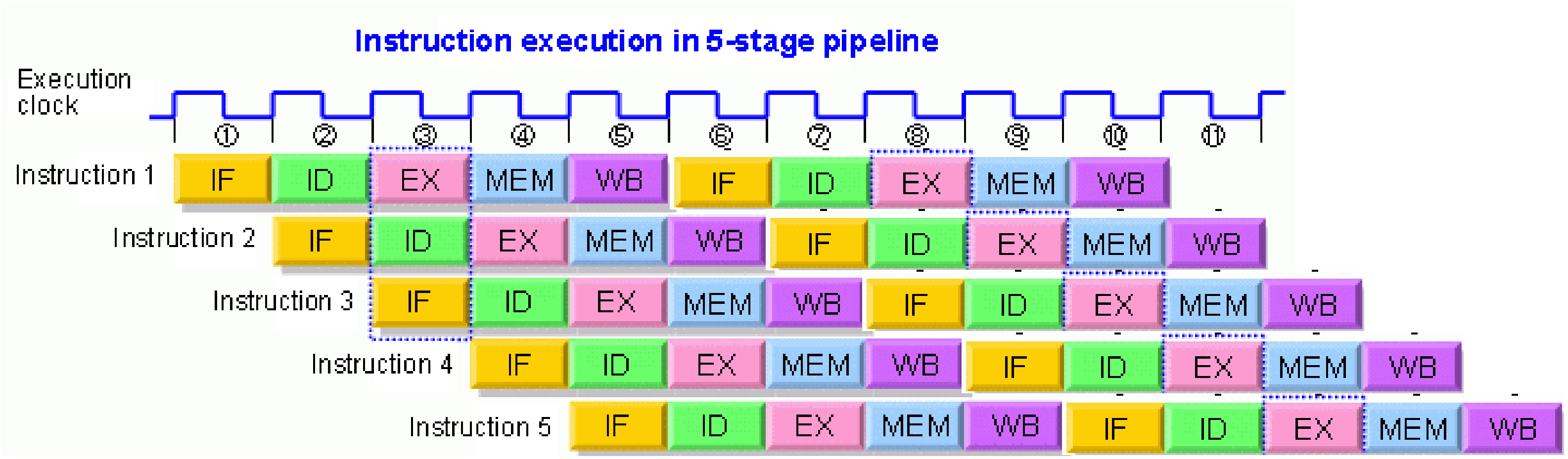
Quelle: PM0214 Programming Manual

CPU Instruction Execution



Quelle: http://www.cs.iit.edu/~cs561/cs350/CPU/5stage.html

CPU Pipeline



1. Instruction Fetch
2. Instruction Decode
3. Execute
4. Memory Access
5. Write Back

Quelle: http://www.cs.iit.edu/~cs561/cs350/CPU/5stage.html

ARM Assembler

ARM is a RISC (Reduced instruction set Computing) processor and therefore has a simplified instruction set (100 instructions or less) and more general purpose registers than CISC.

Unlike Intel, ARM uses instructions that operate only on registers and uses a Load/Store memory model for memory access, which means that only Load/Store instructions can access memory.

This means that incrementing a 32-bit value at a particular memory address on ARM would require three types of instructions (load, increment and store) to first load the value at a particular address into a register, increment it within the register, and store it back to the memory from the register.

Quelle: https://azeria-labs.com/writing-arm-assembly-part-1/

ARM Family vs. ARM Architecture

ARM family	ARM architecture
ARM7	ARM v4
ARM9	ARM v5
ARM11	ARM v6
Cortex-A	ARM v7-A
Cortex-R	ARM v7-R
Cortex-M	ARM v7-M

Quelle: https://azeria-labs.com/writing-arm-assembly-part-1/

Data Types

The data types which can be loaded (or stored), can be signed and unsigned words, halfwords, or bytes.

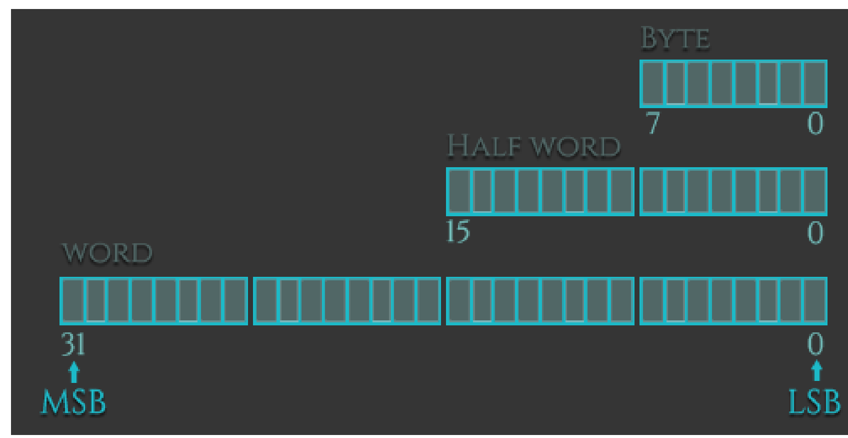
The extensions for these data types are: -h or -sh for halfwords, -b or -sb for bytes, and no extension for words.

The difference between signed and unsigned data types is:

- Signed data types can hold both positive and negative values and are therefore lower in range.
- Unsigned data types can hold large positive values (including ‘Zero’) but cannot hold negative values and are therefore wider in range.

```
ldr = Load Word
ldrh = Load unsigned Half Word
ldrsh = Load signed Half Word
ldrb = Load unsigned Byte
ldrsb = Load signed Bytes

str = Store Word
strh = Store unsigned Half Word
strsh = Store signed Half Word
strb = Store unsigned Byte
strsb = Store signed Byte
```



Quelle: https://azeria-labs.com/arm-data-types-and-registers-part-2/

ARM Instructions

ARM instructions are usually followed by one or two operands and generally use the following template:

```
MNEMONIC{S}{condition} {Rd}, Operand1, Operand2
```

Field	Description
MNEMONIC	Short name (mnemonic) of the instruction
{S}	An optional suffix. If S is specified, the condition flags are updated on the result of the operation
{condition}	Condition that is needed to be met in order for the instruction to be executed
{Rd}	Register (destination) for storing the result of the instruction
Operand1	First operand. Either a register or an immediate value
Operand2	Second (flexible) operand. Can be an immediate value (number) or a register with an optional shift

Operand2 is called a flexible operand, because we can use it in various forms – as immediate value (with limited set of values), register or register with a shift. For example, we can use these expressions as the Operand2:

Quelle: https://azeria-labs.com/arm-instruction-set-part-3/

ARM Assembler

ARM Instructions

Instruction	Description	Instruction	Description
MOV	Move data	EOR	Bitwise XOR
MVN	Move and negate	LDR	Load
ADD	Addition	STR	Store
SUB	Subtraction	LDM	Load Multiple
MUL	Multiplication	STM	Store Multiple
LSL	Logical Shift Left	PUSH	Push on Stack
LSR	Logical Shift Right	POP	Pop off Stack
ASR	Arithmetic Shift Right	B	Branch
ROR	Rotate Right	BL	Branch with Link
CMP	Compare	BX	Branch and eXchange
AND	Bitwise AND	BLX	Branch with Link and eXchange
ORR	Bitwise OR	SWI/SVC	System Call

Quelle: https://azeria-labs.com/arm-instruction-set-part-3/