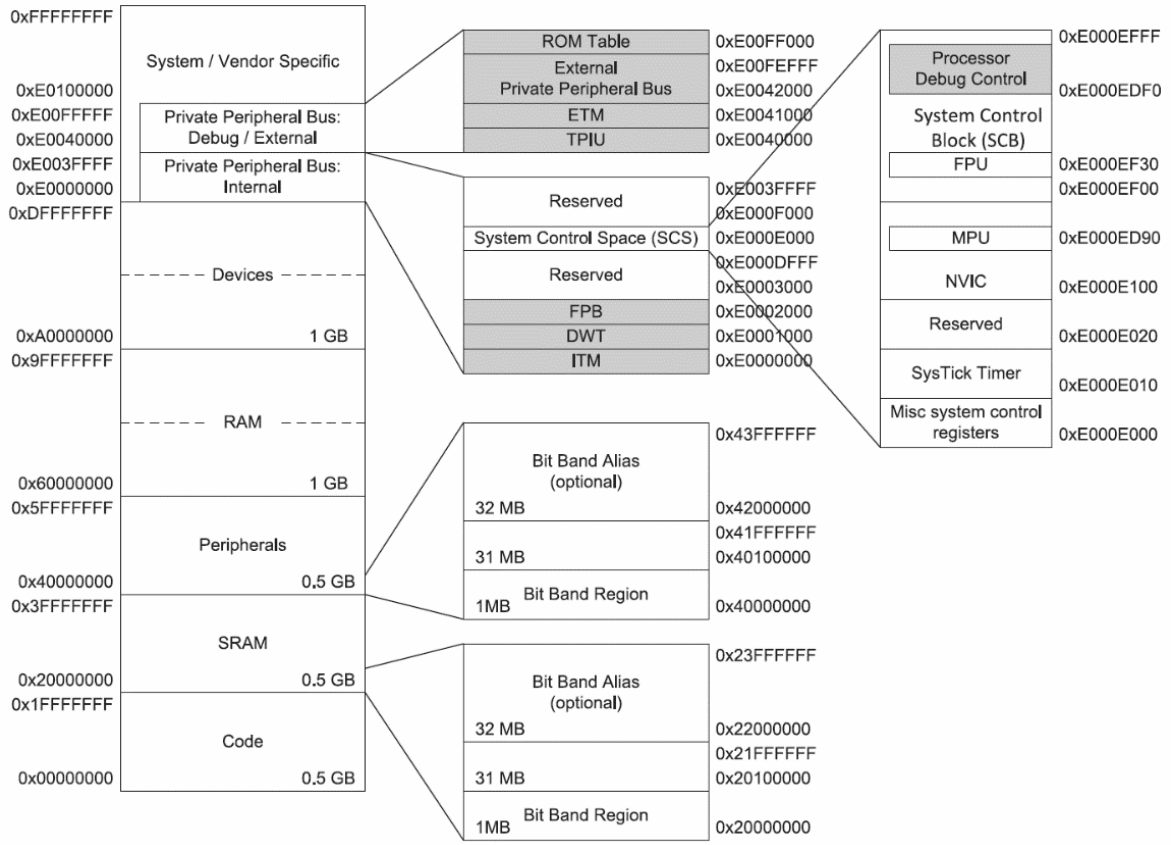
Quick Reference

Lars Kamm, Jan Wendler

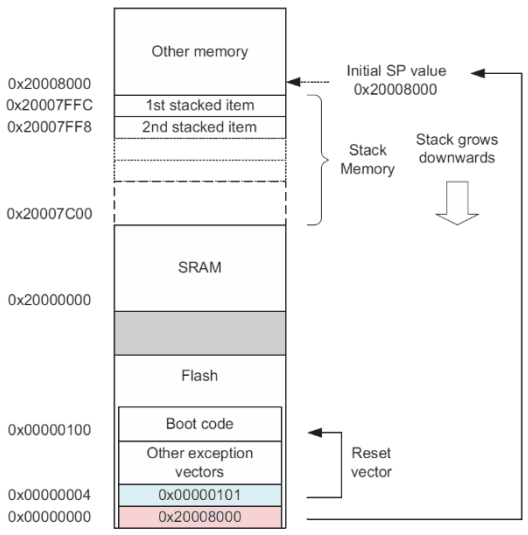
1 ARM CORTEX-M MEMORY ORGANIZATION

1.1 ARM Cortex-M Memory-Map



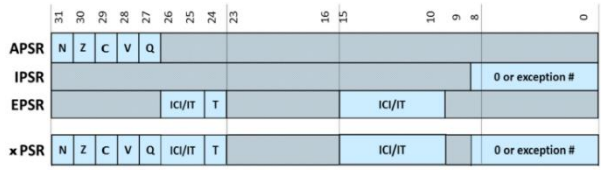
1.2 Vector-Table & Reset-Sequence

Exception Type	CMSIS Interrupt Number	Address Offset	Vectors
18 - 255	2 - 239	0x48 – 0x3FF	IRQ #2 - #239 [1]
17	1	0x44	IRQ #1 [1]
16	0	0x40	IRQ #0 [1]
15	-1	0x3C	SysTick [1]
14	-2	0x38	PendSV [1]
NA	NA	0x34	Reserved
12	-4	0x30	Debug Monitor [1]
11	-5	0x2C	SVC [1]
NA	NA	0x28	Reserved
NA	NA	0x24	Reserved
NA	NA	0x20	Reserved
NA	NA	0x1C	Reserved
6	-10	0x18	Usage fault [1]
4	-11	0x14	Bus Fault [1]
4	-12	0x10	MemManage Fault [1]
3	-13	0x0C	HardFault [1]
2	-14	0x08	NMI [1]
1	NA	0x04	Reset [1]
NA	NA	0x00	Initial value of MSP

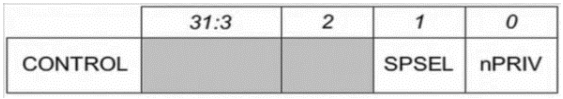


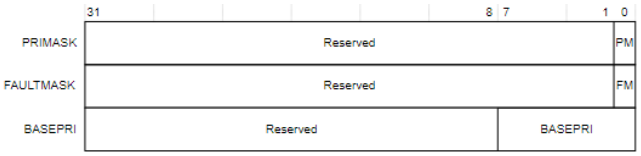
1.3 Register-Set & Program Status Register

Name		Functions (and banked registers)	
R0		General purpose register	Low registers
R1		General purpose register	
R2		General purpose register	
R3		General purpose register	
R4		General purpose register	
R5		General purpose register	
R6		General purpose register	
R7		General purpose register	
R8		General purpose register	High registers
R9		General purpose register	
R10		General purpose register	
R11		General purpose register	
R12		General purpose register	
R13 (MSP)	R13 (PSP)	Main Stack Pointer (MSP), Process Stack Pointer (PSP)	
R14		Link Register (LR)	
R15		Program Counter (PC)	
xPSR		Program status registers	Special registers
PRIMASK		Interrupt mask registers	
FAULTMASK			
BASEPRI			
CONTROL		Control register	

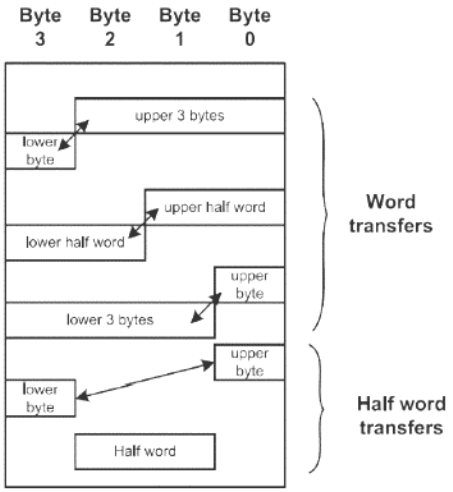


N	Negativ Flag
Z	Zero Flag
C	Carry/Borrow Flag
V	Overflow Flag
Q	Sticky Saturation Flag
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit
T	Thumb state, always "1"; trying to clear this bit will cause a fault exception
Exception#	indicates which exception the processor is handling

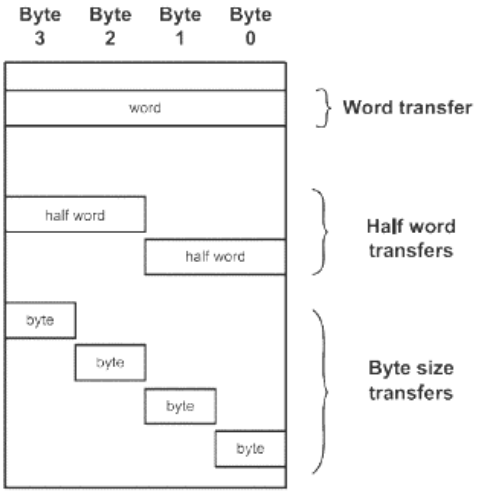




1.4 Alignement



Unaligned transfers

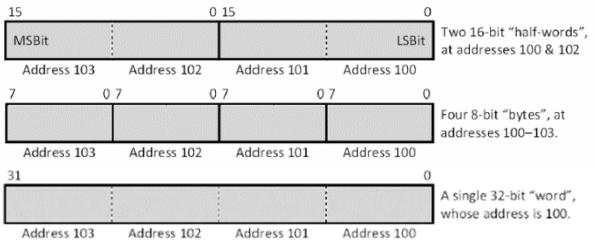


Aligned transfers

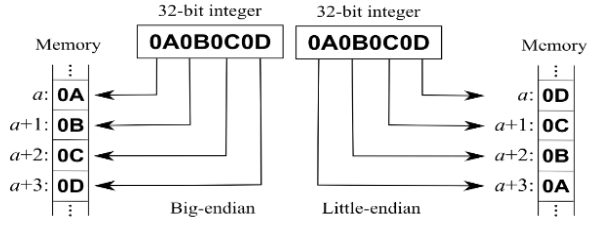
Word (32Bit): 0x0, 0x4, 0x8 0xC

Halfword (16Bit): 0x0, 0x2, 0x4, 0x6
0x8, 0xA, 0xC, 0xE

Byte (8Bit): 0x0, 0x1, …., 0xF



1.5 Endianess



Endianess betrifft nur Zahlenformate die grösser als ein Byte sind (Halfword, Word, Doubleword).

Little-Endian: Least Significant Byte befindet sich an der tiefsten Adresse.

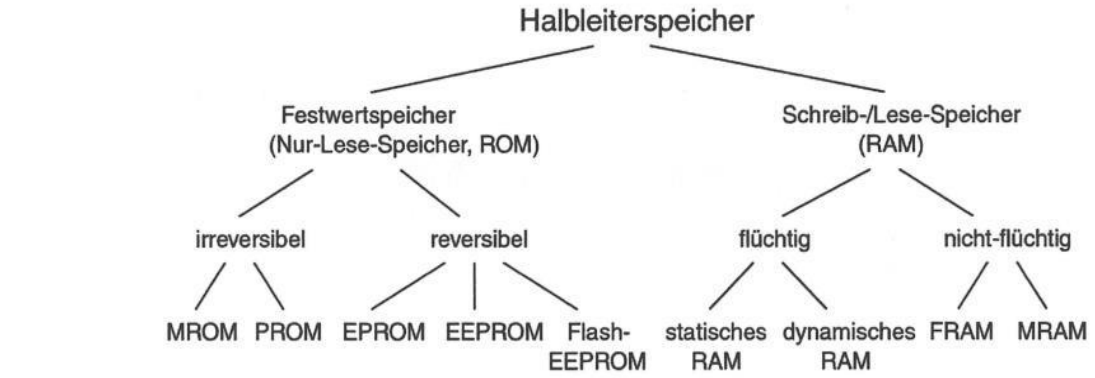
Big-Endian: Most Significant Byte befindet sich an der tiefsten Adresse.

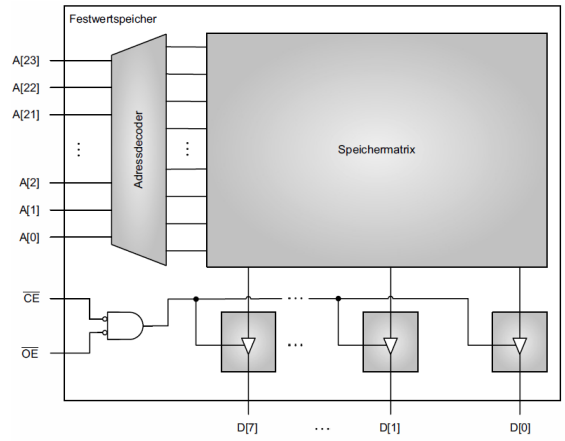
1.6 Speichergrössen

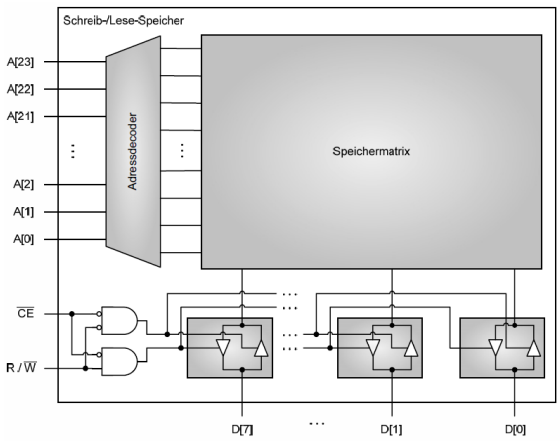
Zur Angabe der Speicherkapazität als Anzahl von Bits, Bytes oder Codewörtern verwendet man in der Informatik in Anlehnung an die Physik die Bezeichnungen Kilo (K), Mega (M), Giga (G) und Tera (T). Diese beziehen sich auf das Binärzahlensystem:

$$\mathbf{1K} = \mathbf{2^{10}} = \mathbf{1'024}$$
$$\mathbf{1M} = \mathbf{2^{20}} = \mathbf{1'024K} = \mathbf{1'048'576}$$
$$\mathbf{1G} = \mathbf{2^{30}} = \mathbf{1'024M} = \mathbf{1'048'576K} = \mathbf{1'073'741'824}$$
$$\mathbf{1T} = \mathbf{2^{40}} = \mathbf{1'024G} = \mathbf{1'048'576M} = \mathbf{1'073'741'824K} = \mathbf{1'099'511'627'776}$$

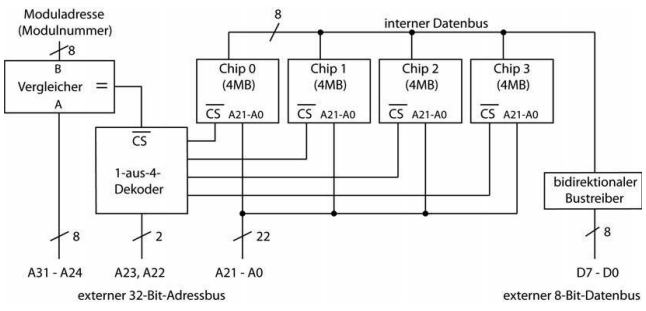
1.7 Speicherarten



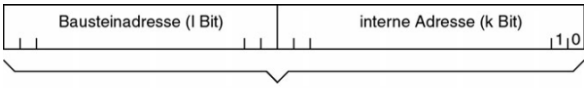




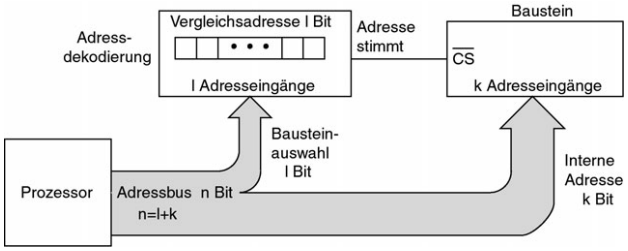
1.8 Adressdekodierung

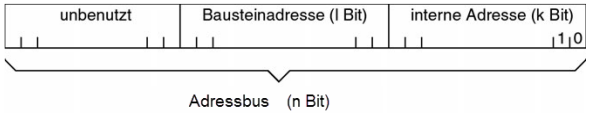


$$k + l = n$$



$$k + l < n$$

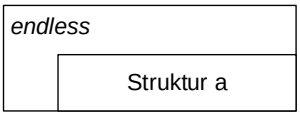




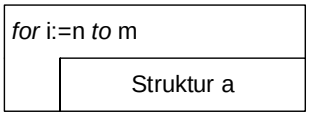
2 KONTROLL- & SELEKTIONSSTRUKTUREN

2.1 Nassi Schneidermann Diagramme

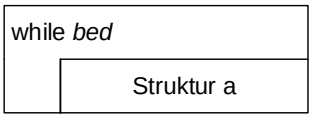
endless



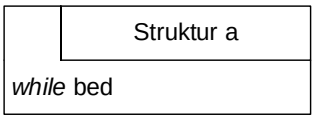
for



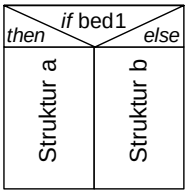
while...do



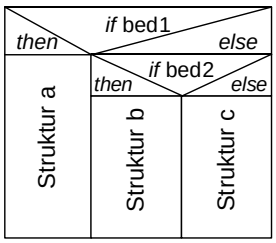
do...while



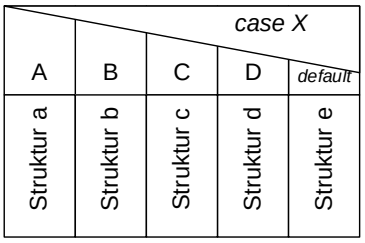
if / then / else



if / then / else if / else

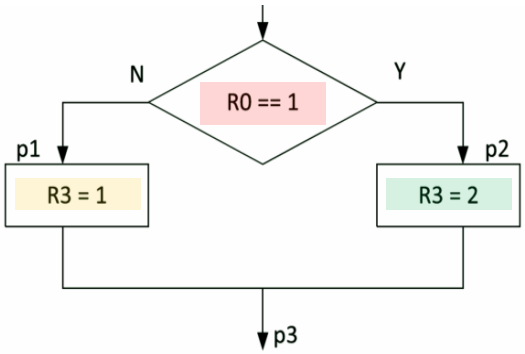


case



2.2 Entscheidungen treffen

```
// a → R0
// b → R3
if(a != 1)
{
  b = 1;
}
else
{
  b = 2;
}
```



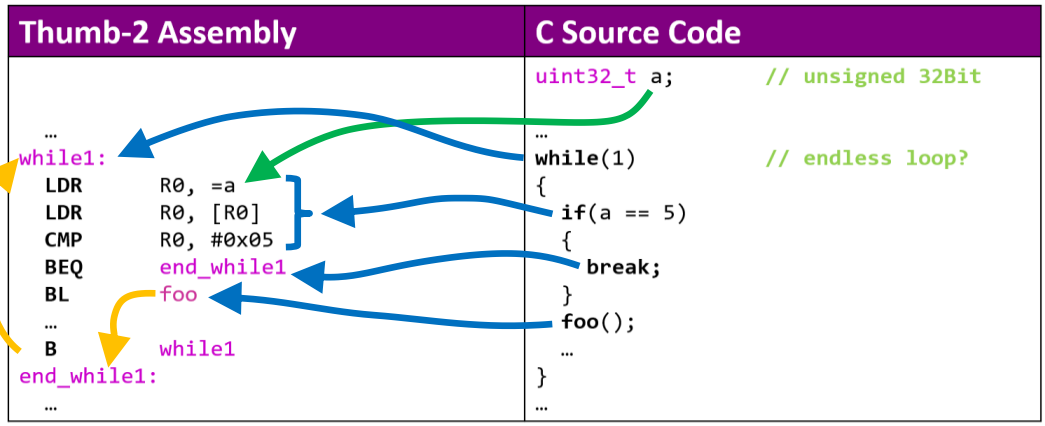

```
  CMP  r0, #1 ;compare r0 to 1
  BEQ  p2     ;if equal, go to p2
  MOVS r3, #1 ;then r3 = 1
  B    p3     ;go to end (p3)
p2:           ;label p2
  MOVS r3, #2 ;else r3 = 2
p3:           ;label p3
  …           ;other operations
```

In Assembler-Sprache ist eine Entscheidung praktisch immer ein 2-stufiger Ablauf

1. **Benötigte Flags ermitteln**
   Vergleich (**Compare**) vornehmen, indem zwei Werte voneinander subtrahiert werden. Bei der Differenzbildung ist das Resultat nicht wichtig, sondern nur dessen Eigenschaften (**Flags**).

2. **Zugehörige Sprünge ausführen**
   Im zweiten Schritt werden mit den Flags bedingte Sprünge (**B{cond}**) ausgeführt

2.3 Umsetzung von C/C++ Strukturen



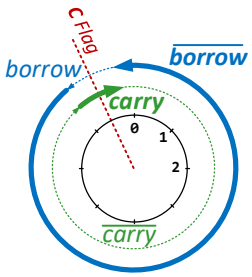
3 ARM CORTEX-M0(+) INSTRUCTION-SET

3.1 Symbole & Appendix

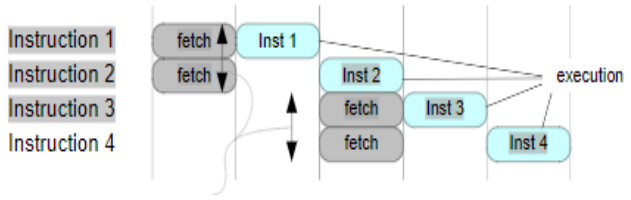
Symbols	Meaning
R_d R_m R_n	represent 32-bit registers
{R_d,}	if R_d is present R_d is destination, otherwise R_n
{S}	if S is present, instruction will set condition codes
{cc}	optional logical condition, condition code suffix
#imm3	any value in the range: 0...7
#imm5	any value in the range: 0...31
#imm7	any value in the range: 0..127
#imm8	any value in the range: 0..255
#imm10	any value in the range: 0..1023
#imm11	any value in the range: 0..2047
label	any address within the ROM of the microcontroller; offset range relative to PC: B *label* −2 KB to +2 KB B<cc> *label* −256 bytes to +255 bytes BL *label* −16 MB to +16 MB
{reglist}	List of registers, sequence is not relevant

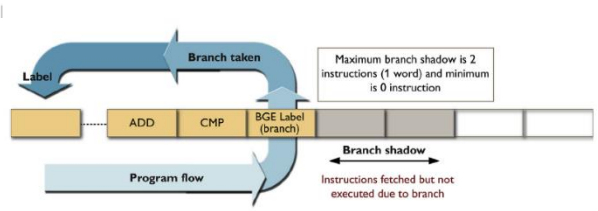
3.2 Bedingte Ausführung & Code-Zusätze {cc}

cc	Meaning	Flags (APSR) / Requirements	Binary	uint		int	
EQ	Equal	Z = 1	0000	X	==	X	==
NE	Not equal	Z = 0	0001	X	!=	X	!=
CS or **HS**	Carry set, Unsigned ≥ carry OR $\overline{\text{borrow}}$	C = 1	0010	X	>=		
CC or **LO**	Carry clear,Unsigned < $\overline{\text{carry}}$ OR borrow	C = 0	0011	X	<		
MI	Minus/negative	N = 1	0100			X	-
PL	Positive or zero (non-negative)	N = 0	0101			X	+
VS	Overflow	V = 1	0110			X	
VC	No overflow	V = 0	0111			X	
HI	Unsigned > ("Higher")	C = 1 && Z = 0	1000	X	>		
LS	Unsigned ≤ ("Lower or Same")	C = 0 || Z=1	1001	X	<=		
GE	Signed ≥ ("Greater than or Equal")	N = V	1010			X	>=
LT	Signed < ("Less Than")	N ≠ V	1011			X	<
GT	Signed > ("Greater Than")	Z = 0 && N = V	1100			X	>
LE	Signed ≤ ("Less than or Equal")	Z = 1 || N ≠ V	1101			X	<=
AL	Always (unconditional)	any	1110				



3.3 Pipelining





3.4 ARM Cortex-M0(+) Thumb Instruktionen

Group	Description	Syntax	Cycles	Flags				Instruction Code 16Bit																		
				N	Z	C	V	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Shift (immediate), add, subtract, move, and compare	Move Lo to Lo	MOVS Rd, Rm	1	?	?	-	-	0	0	0	0	0	0	0	0	0	0	Rm				Rd	
	Logical shift left by immediate	LSLS Rd, Rm, #<imm>	1	?	?	?	-	0	0	0	0	0	imm5					Rm				Rd	
	Logical shift right by immediate	LSRS Rd, Rm, #<imm>	1	?	?	?	-	0	0	0	0	1	imm5					Rm				Rd	
	Arithmetic shift right	ASRS Rd, Rm, #<imm>	1	?	?	?	-	0	0	0	0	1	0	imm5				Rm				Rd	
	Add 3-bit immediate	ADDS Rd, Rm, #<imm>	1	?	?	?	?	0	0	0	0	1	1	0	imm3			Rm				Rd	
	Add All registers Lo	ADDS Rd, Rn, Rm	1	?	?	?	?	0	0	0	0	1	1	0	0	Rm		Rn				Rd	
	Sub Lo and Lo	SUBS Rd, Rn, Rm	1	?	?	?	?	0	0	0	0	1	1	0	1	Rm		Rn				Rd	
	Sub 3-bit immediate	SUBS Rd, Rn, #<imm>	1	?	?	?	?	0	0	0	0	1	1	1	imm3			Rn				Rd	
	Move 8-bit immediate	MOVS Rd, #<imm>	1	?	?	-	-	0	0	0	1	0	0	Rd	imm8								
	Compare Immediate	CMP Rn, #<imm>	1	?	?	?	?	0	0	1	0	1	0	Rn	imm8								
	Add 8-bit immediate	ADDS Rd, #<imm>	1	?	?	?	?	0	0	1	1	0	0	Rd	imm8								
	Sub 8-bit immediate	SUBS Rd, #<imm>	1	?	?	?	?	0	0	1	1	1	0	Rd	imm8								
Data processing	AND	ANDS Rd, Rm	1	?	?	-	-	0	1	0	0	0	0	0	0	0	0	Rm				Rd	
	Exclusive OR	EORS Rd, Rm	1	?	?	-	-	0	1	0	0	0	0	0	0	0	1	Rm				Rd	
	Logical shift left by register	LSLS Rd, Rm	1	?	?	?	-	0	1	0	0	0	0	0	0	1	0	Rm				Rd	
	Logical shift right by register	LSRS Rd, Rm	1	?	?	?	-	0	1	0	0	0	0	0	0	1	1	Rm				Rd	
	Arithmetic shift right by register	ASRS Rd, Rm	1	?	?	?	-	0	1	0	0	0	0	0	0	1	0	Rm				Rd	
	Add with carry	ADCS Rd, Rm	1	?	?	?	?	0	1	0	0	0	0	0	0	1	0	Rm				Rd	
	Sub with carry	SBCS Rd, Rm	1	?	?	?	?	0	1	0	0	0	0	0	0	1	1	Rm				Rd	
	Rotate right by register	RORS Rd, Rm	1	?	?	?	-	0	1	0	0	0	0	0	0	1	1	Rm				Rd	
	AND test	TST Rn, Rm	1	?	?	?	-	0	1	0	0	0	0	0	0	0	0	Rn				Rn	
	Negate	RSBS Rd, Rm, #0	1	?	?	?	?	0	1	0	0	0	0	0	1	0	0	Rm				Rd	
	Compare Lo to Lo	CMP Rn, Rm	1	?	?	?	?	0	1	0	0	0	0	0	1	0	1	Rm				Rn	
	Compare Negative	CMN Rn, Rm	1	?	?	?	?	0	1	0	0	0	0	0	1	0	1	Rm				Rn	
	OR	ORRS Rd, Rm	1	?	?	-	-	0	1	0	0	0	0	0	1	1	0	Rm				Rd	
	Multiply	MULS Rd, Rm	1 or 32 (1)	?	?	-	-	0	1	0	0	0	0	0	1	1	0	Rm				Rd	
	Bit clear	BICS Rd, Rm	1	?	?	-	-	0	1	0	0	0	0	0	1	1	1	Rm				Rd	
	Move NOT	MVNS Rd, Rm	1	?	?	-	-	0	1	0	0	0	0	0	1	1	1	Rm				Rd	
Special data instructions and branch and exchange	Add Any to PC	ADD PC, Rm	3	-	-	-	-	0	1	0	0	0	1	0	0	1		Rm			1	1	1
	Add Any to Any	ADD Rd, Rm	1	-	-	-	-	0	1	0	0	0	1	0	0	Rd		Rm				Rd	
	Add address from SP and register	ADD Rd, SP	1	-	-	-	-	0	1	0	0	0	1	0	0	Rd	1	0	1			Rd	
	Compare Any to Any	CMP Rn, Rm	1	?	?	?	?	0	1	0	0	0	1	0	1	Rn		Rm				Rn	
	Move Any to Any	MOV Rd, Rm	1	-	-	-	-	0	1	0	0	0	1	1	0	Rd		Rm				Rd	
	Move Any to PC	MOV PC, Rm	2	-	-	-	-	0	1	0	0	0	1	1	0	1		Rm			1	1	1
	Branch With exchange	BX Rm	2	-	-	-	-	0	1	0	0	0	1	1	1	0		Rm			0	0	0
	Branch With link and exchange	BLX Rm	2	-	-	-	-	0	1	0	0	0	1	1	1	1		Rm			0	0	0
Load from Literal Pool	Load PC-relative	LDR Rd, <label>	2 or 1 (2)	-	-	-	-	0	1	0	0	1	Rd	imm8									
Load/store single data item	Word, register offset	STR Rd, [Rn, Rm]	2 or 1 (2)	-	-	-	-	0	1	0	1	0	0	0	Rm			Rn				Rd	
	Halfword, register offset	STRH Rd, [Rn, Rm]	2 or 1 (2)	-	-	-	-	0	1	0	1	0	0	1	Rm			Rn				Rd	
	Byte, register offset	STRB Rd, [Rn, Rm]	2 or 1 (2)	-	-	-	-	0	1	0	1	0	1	0	Rm			Rn				Rd	
	Signed byte, register offset	LDRSB Rd, [Rn, Rm]	2 or 1 (2)	-	-	-	-	0	1	0	1	0	1	1	Rm			Rn				Rd	
	Word, register offset	LDR Rd, [Rn, Rm]	2 or 1 (2)	-	-	-	-	0	1	0	1	1	0	0	Rm			Rn				Rd	
	Halfword, register offset	LDRH Rd, [Rn, Rm]	2 or 1 (2)	-	-	-	-	0	1	0	1	1	0	1	Rm			Rn				Rd	
	Byte, register offset	LDRB Rd, [Rn, Rm]	2 or 1 (2)	-	-	-	-	0	1	0	1	1	1	0	Rm			Rn				Rd	
	Signed halfword, register offset	LDRSH Rd, [Rn, Rm]	2 or 1 (2)	-	-	-	-	0	1	0	1	1	1	1	Rm			Rn				Rd	

(1) Depends on multiplier implementation

(2) 2 if to AHB interface or SCS, 1 if to single-cycle I/O port.

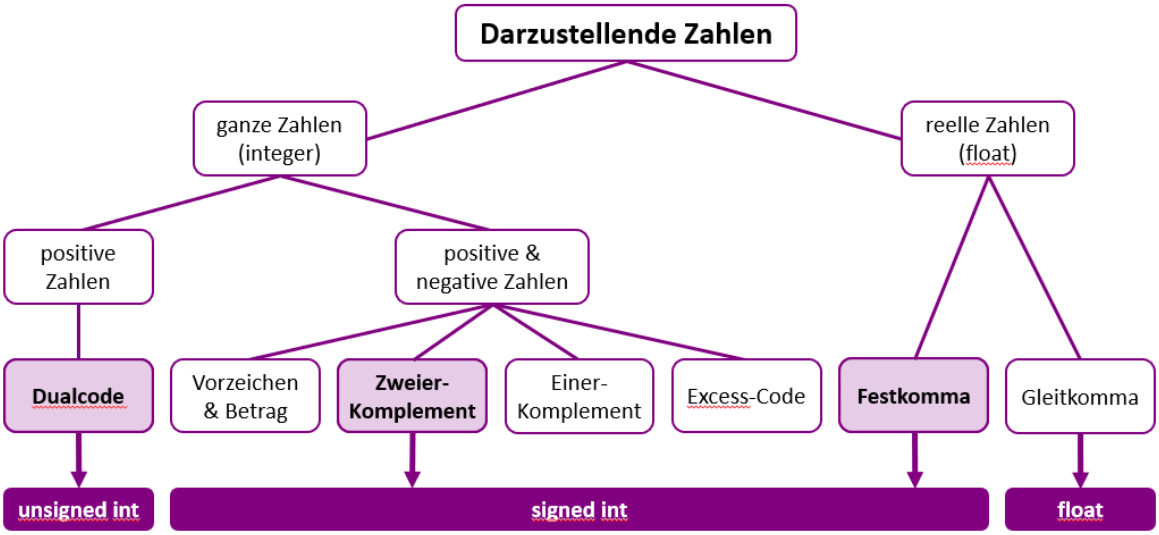
(3) N is the number of elements in the list

(4) N is the number of elements in the list including PC or LR.

Load/store single data item	Word, immediate offset	Rd → [Rn + #imm5]		STR Rd, [Rn, #<imm>]	2 or 1 [2]	- - - -	0	1	1	0	0	imm5					Rn			Rd		
	Word, immediate offset	Rd ← [Rn + #imm5]		LDR Rd, [Rn, #<imm>]	2 or 1 [2]	- - - -	0	1	1	0	1	imm5					Rn			Rd		
	Byte, immediate offset	Rd → [Rn + #imm5]		STRB Rd, [Rn, #<imm>]	2 or 1 [2]	- - - -	0	1	1	1	0	imm5					Rn			Rd		
	Byte, immediate offset	Rd ← [Rn + #imm5]		LDRB Rd, [Rn, #<imm>]	2 or 1 [2]	- - - -	0	1	1	1	1	imm5					Rn			Rd		
	Halfword, immediate offset	Rd → [Rn + #imm5]		STRH Rd, [Rn, #<imm>]	2 or 1 [2]	- - - -	1	0	0	0	0	imm5					Rn			Rd		
	Halfword, immediate offset	Rd ← [Rn + #imm5]		LDRH Rd, [Rn, #<imm>]	2 or 1 [2]	- - - -	1	0	0	0	1	imm5					Rn			Rd		
	SP-relative	Rd → [SP + #imm8]		STR Rd, [SP, #<imm>]	2 or 1 [2]	- - - -	1	0	0	1	0	Rd			imm8							
	SP-relative	Rd ← [SP + #imm8]		LDR Rd, [SP, #<imm>]	2 or 1 [2]	- - - -	1	0	0	1	1	Rd			imm8							
Generate PC-relative address	Form address from PC	Rd ← addr[<label>]		ADR Rd, <label>	1	- - - -	1	0	1	0	0	Rd			imm8							
Generate SP-relative address	Form address from SP	Rd ← SP + #imm8		ADD Rd, SP, #<imm>	1	- - - -	1	0	1	0	1	Rd			imm8							
Miscellaneous 16-bit instructions	Add Immediate to SP	SP ← SP + #imm7		ADD SP, #<imm>	1	- - - -	1	0	1	1	0	0	0	0	0	imm7						
	Sub Immediate from SP	SP ← SP - #imm7		SUB SP, #<imm>	1	- - - -	1	0	1	1	0	0	0	0	1	imm7						
	Extend signed halfword to word	Rd_{32Bit} ← signed Rm_{16Bit}		SXTH Rd, Rm	1	- - - -	1	0	1	1	0	0	1	0	0	0	Rm			Rd		
	Extend signed byte to word	Rd_{32Bit} ← signed Rm_{8Bit}		SXTB Rd, Rm	1	- - - -	1	0	1	1	0	0	1	0	0	1	Rm			Rd		
	Extend unsigned halfword	Rd_{32Bit} ← Rm_{16Bit}		UXTH Rd, Rm	1	- - - -	1	0	1	1	0	0	1	0	1	0	Rm			Rd		
	Extend unsigned byte	Rd_{32Bit} ← Rm_{8Bit}		UXTB Rd, Rm	1	- - - -	1	0	1	1	0	0	1	0	1	1	Rm			Rd		
	Push	regs → mem[SP--]		PUSH {<LoReglist>}	1+N [4]	- - - -	1	0	1	1	0	1	0	0	lo_register_list							
	Push with link register	regs, LR → mem[SP--]		PUSH {<LoReglist>, LR}	1+N [3]	- - - -	1	0	1	1	0	1	0	1	lo_register_list							
	Disable interrupts			CPSID i	1	- - - -	1	0	1	1	0	1	1	0	0	1	1	0	0	0	1	0
	Enable interrupts			CPSIE i	1	- - - -	1	0	1	1	0	1	1	0	0	1	1	1	0	0	1	0
	Bytes in word	Rd ← RevByteOrder[Rm]		REV Rd, Rm	1	- - - -	1	0	1	1	1	0	1	0	0	0	Rm			Rd		
	Bytes in both halfwords	Rd ← RevHalfWords[Rm]		REV16 Rd, Rm	1	- - - -	1	0	1	1	1	0	1	0	0	1	Rm			Rd		
	Signed bottom half word	Rd ← RevLoHalf[Rm]		REVSH Rd, Rm	1	- - - -	1	0	1	1	1	0	1	0	1	1	Rm			Rd		
	Pop	regs ← mem[SP++]		POP {<LoReglist>}	1+N [3]	- - - -	1	0	1	1	1	1	0	0	lo_register_list							
	Pop and return	regs, PC ← mem[SP++]		POP {<LoReglist>, PC}	3+N [4]	- - - -	1	0	1	1	1	1	0	1	lo_register_list							
	Breakpoint			BKPT #<imm>	- [6]	- - - -	1	0	1	1	1	1	1	0	imm8							
	Send event			SEV	1	- - - -	1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0
	Wait for event			WFE	2 [7]	- - - -	1	0	1	1	1	1	1	1	0	0	1	0	0	0	0	0
	Wait for interrupt			WFI	2 [7]	- - - -	1	0	1	1	1	1	1	1	0	0	1	1	0	0	0	0
	Yield			YIELD	1 [8]	- - - -	1	0	1	1	1	1	1	1	0	0	0	1	0	0	0	0
	No operation			NOP	1	- - - -	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
Store multiple registers	Store Multiple	regs → mem[Rn++]		STM Rn!, {<LoReglist>}	1+N [3]	- - - -	1	1	0	0	0	Rn			lo_register_list							
Load multiple registers	Load Multiple, excluding base	regs ← mem[Rn++]		LDM Rn!, {<LoReglist>}	1+N [3]	- - - -	1	1	0	0	1	Rn			lo_register_list							
	Load Multiple, including base	regs ← mem[Rn]		LDM Rn, {<LoReglist>}	1+N [3]	- - - -	1	1	0	0	1	Rn			lo_register_list							
Conditional branch, and Supervisor Call	Conditional	if <cc> then PC ← label		B<cc> <label>	1 or 2 [5]	- - - -	1	1	0	1	cond				imm8							
	Supervisor Call			SVC #<imm>	- [6]	- - - -	1	1	0	1	1	1	1	1	imm8							
Unconditional Branch	Unconditional	PC ← label		B<label>	2	- - - -	1	1	1	0	0	imm11										
Branch & Link 32Bit	With link		[15 - 0]	BL<label>	3	- - - -	1	1	1	1	0	S	imm10									
			[31 - 16]			- - - -	1	1	J1	1	J2	imm11										
Write and Read Special Register 32 Bit	Read special register		[15 - 0]	MRS Rd, <specreg>	3	- - - -	1	1	1	1	0	0	1	1	1	1	1	0	1	1	1	1
			[31 - 16]			- - - -	1	0	0	0	Rd				SYSm							
	Write special register		[15 - 0]	MSR <specreg>, Rn	3	- - - -	1	1	1	1	0	0	1	1	1	0	0	0	Rn			
			[31 - 16]			- - - -	1	0	0	0	1	0	0	0	SYSm							
Synchronisation 32 Bit	Instruction synchronization barrier		[15 - 0]	ISB	3	- - - -	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1
			[31 - 16]			- - - -	1	0	0	0	1	1	1	1	0	1	1	0	option			
	Data memory barrier		[15 - 0]	DMB	3	- - - -	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1
			[31 - 16]			- - - -	1	0	0	0	1	1	1	1	0	1	0	1	option			
	Data synchronization barrier		[15 - 0]	DSB	3	- - - -	1	1	1	1	0	0	1	1	1	0	1	1	1	1	1	1
			[31 - 16]			- - - -	1	0	0	0	1	1	1	1	0	1	0	0	option			

(5) 2 if taken, 1 if not-take
(6) Cycle count depends on processor and debug configuration.
(7) Excludes time spent waiting for an interrupt or event.
(8) Executes as NOP.

4 ZAHLENSYSTEME



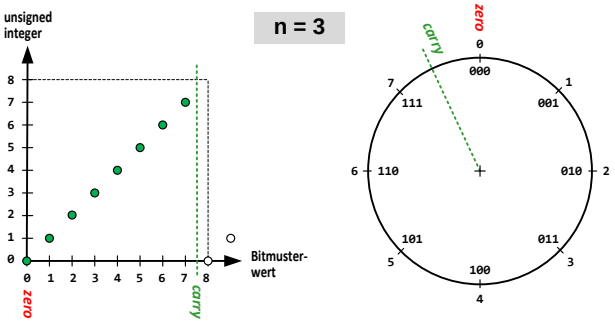
4.1 Vorzeichenlose Zahlen

Dualcode (unsigned int)

$$W = \sum_{i=0}^{n-1} d_i \cdot 2^i$$

$$d_{n-1} \quad d_{n-2} \ \ldots \quad d_2 \quad d_1 \quad d_0$$

Relevante Flags: Zero, Carry/Borrow



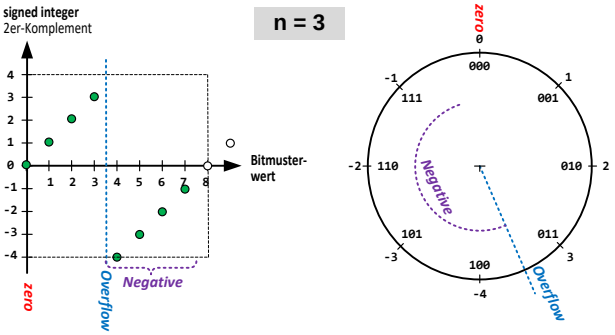
4.2 Vorzeichenbehaftete Zahlen

Zweierkomplement (signed int)

$$W = \left(\sum_{i=0}^{n-2} d_i \cdot 2^i \right) - d_{n-1} \cdot 2^{n-1}$$

$$d_{n-1} \quad d_{n-2} \ \ldots \quad d_2 \quad d_1 \quad d_0$$

Relevante Flag: Zero, Negativ, Overflow



4.3 Zweierkomplement-Negierung

Vorgehensweise:

1) Vorzeichenbehaftete Zahl in Binärcode darstellen
2) Alle Bits einzeln invertieren
3) Binäre Addition mit dem Wert +1
4) Resultat entspricht der negierten Vorzeichenbehaftete Zahl

Beispiel: -3_{10}=	**101_2**
Invertieren:	010_2
+1:	001_2

	011_2
==========	
Resultat:	$+3_{10}$

4.4 Festkomma-Zahlen

Vorzeichenlose Fixed-Point-Zahlen

Allgemeine Formel (Dualcode als Basis)

$$W = \sum_{i=-m}^{n-1} d_i \cdot 2^i$$

$$d_{n-1}\ d_{n-2}\ \dots\ d_1\ d_0\ .\ d_{-1}\ d_{-2}\ d_{-m}$$

Wertebereich: $0.0 \dots 2^n - 2^{-m}$

Schrittweite: $\Delta X = 2^{-m}$

Max. Diskretisierungsfehler: $\frac{\Delta X}{2} = 2^{-(m+1)}$

Vorzeichenbehaftete Fixed-Point-Zahlen

Allgemeine Formel (2er-Komplement als Basis)

$$W = \left(\sum_{i=-m}^{n-2} d_i \cdot 2^i\right) - d_{n-1} \cdot 2^{n-1}$$

$$d_{n-1}\ d_{n-2}\ \dots\ d_1\ d_0\ .\ d_{-1}\ d_{-2}\ d_{-m}$$

Wertebereich: $-2^{n-1} \dots 2^{n-1} - 2^{-m}$

Schrittweite: $\Delta X = 2^{-m}$

Max. Diskretisierungsfehler: $\frac{\Delta X}{2} = 2^{-(m+1)}$

4.5 IQ-Zahlenformate

Integer/Quotient-Darstellung

Signed Fixed-Point mit n = I und m = Q.

$$W = -d_{I-1} \cdot 2^{I-1} + \sum_{j=0}^{I-2} d_j \cdot 2^j + \sum_{i=1}^{Q} d_{-i} \cdot 2^{-i}$$

$$d_{I-1}\ d_{I-2}\ \dots\ d_1\ d_0\ .\ d_{-1}\ d_{-2}\ d_{-Q}$$

Fraktionale Zahlen (oder Spezialfall I1Qx)

$$W = -d_0 + \sum_{i=1}^{Q} d_{-i} \cdot 2^{-i}$$

$$d_0\ .\ d_{-1}\ d_{-2}\ \dots\ d_{-(Q)}$$

Wertebereich: $-1.0 \le W < +1.0$
Vorzeichen: Koeffizient d_0
Genauigkeit: $2^{-(Q)}$

4.6 Zahlenkreise

Addition: Zahlenvektoren hintereinander reihen (Anfang auf Ende)

Subtraktion: Zahlenvektoren stumpf gegeneinanderstellen (Ende auf Ende)

Unsigned Integer

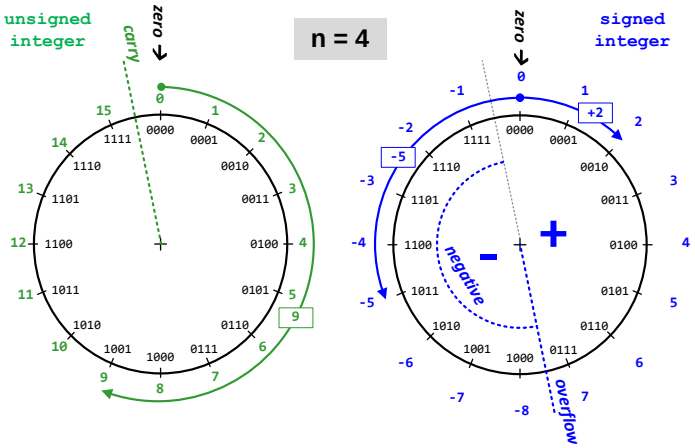
Relevante Flags: Zero, Carry/Borrow

Zahlenvektoren immer Uhrzeigersinn

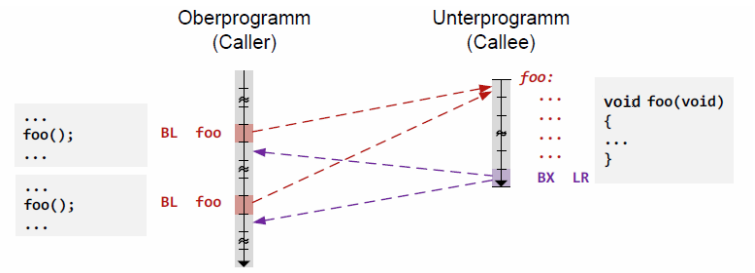
Signed Integer

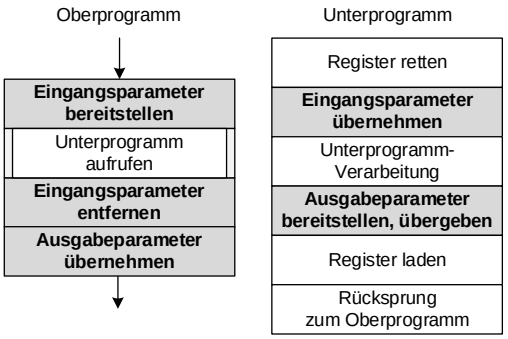
Relevante Flags: Negative, Overflow, Zero

positiv → Zahlenvektor Uhrzeigersinn
negativ → Zahlenvektor Gegenuhrzeiger



5 SUBROUTINEN





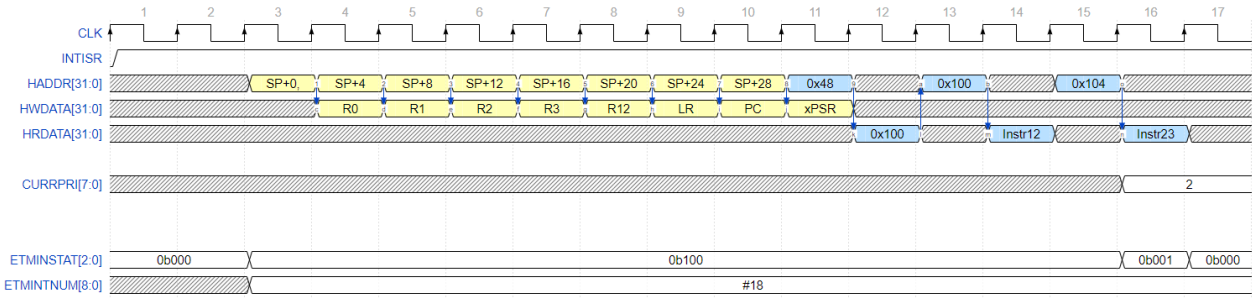
Regeln zum ARM Architecture Procedure Call Standard (AAPCS)

- Die Register **R0-R3** werden als Parameter verwendet.
- Weitere Parameter werden auf den **Stack** gelegt.
- Der Rückgabewert (≤32Bit) wird in Register **R0** übertragen.
- Ein 64Bit-Rückgabewert wird in den Registern **R0** und **R1** übertragen.
- Funktionen müssen die Registerinhalte **R4-R11** während der Ausführung sichern und rekonstruieren.
- Die Register **R0-R3** und **R12** dürf(t)en in der Subroutine ohne Sicherung verändert werden.
- Es wird stets ein **8-Byte Alignement** auf dem Stack eingehalten.

Register	Verwendungszweck	Scratch	Sichern
R0	Erster Funktionsparameter, Return-Wert (8-/16-/32-Bit)	Ja	
R1	Zweiter Funktionsparameter, Upper Word Return-Wert (64-Bit)	Ja	
R2, R3	Dritter und vierter Funktions-parameter	Ja	
R4-R7	Registervariablen (temporär)	Ja	Ja
R8-R11	Compilerabhängige Verwendung	Ja	Ja
R12	Funktionsinternes temporäres Speicherregister für Sprünge	Ja	
R13	Stack-Pointer (SP, Top-of-Stack)	Diese Register werden für spezielle Zwecke verwendet	
R14	Link Register (LR)		
R15	Programm Counter (PC)		

6 INTERRUPTS & EXCEPTIONS

6.1 Vector Fetch & Stacking



Request: Pipeline «Execute» noch ausführen und der Pending Status setzen

Stacking & Vector-Fetch: Register auf Stack sichern und Sprung via Vektortabelle ausführen und Active Status setzen.

ISR Handler: Benutzercode ausführen und am Ende den Active Status zurücksetzen

Unstacking: Register wieder herstellen, Stack abbauen und Rücksprung mittels LR

7 PERSONAL NOTES