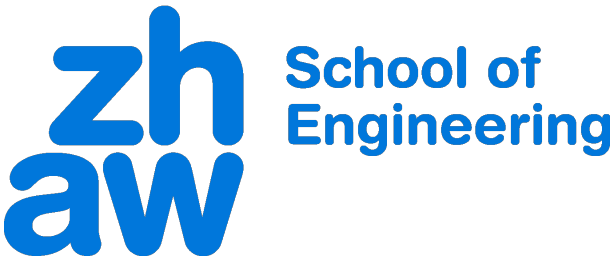
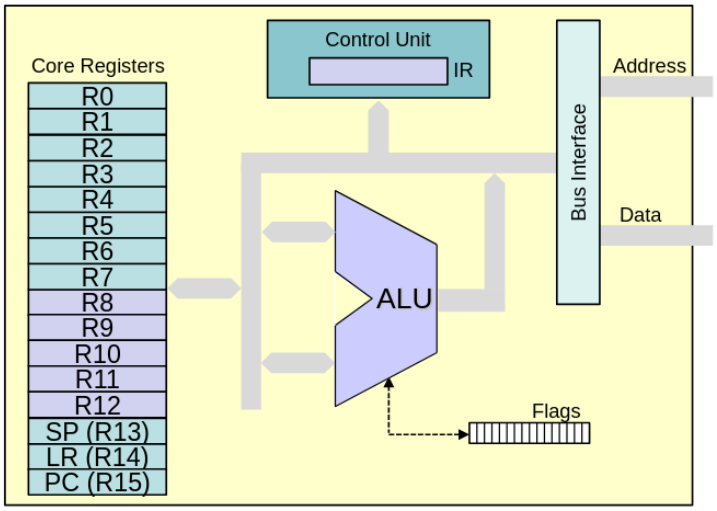
ZHAW Zurich University of Applied Sciences
Winterthur

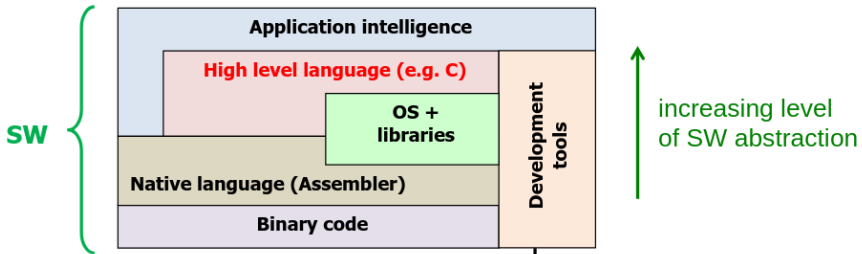


Zusammenfassung CT1
Studienwochen 1-14

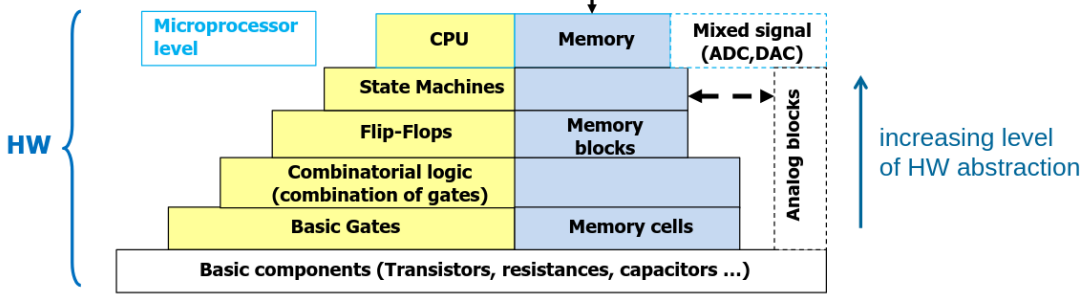
Written by: Severin Sprenger
(w/ inputs from lienhyan)
June 17, 2026
Zf. CT1 SW 1-14

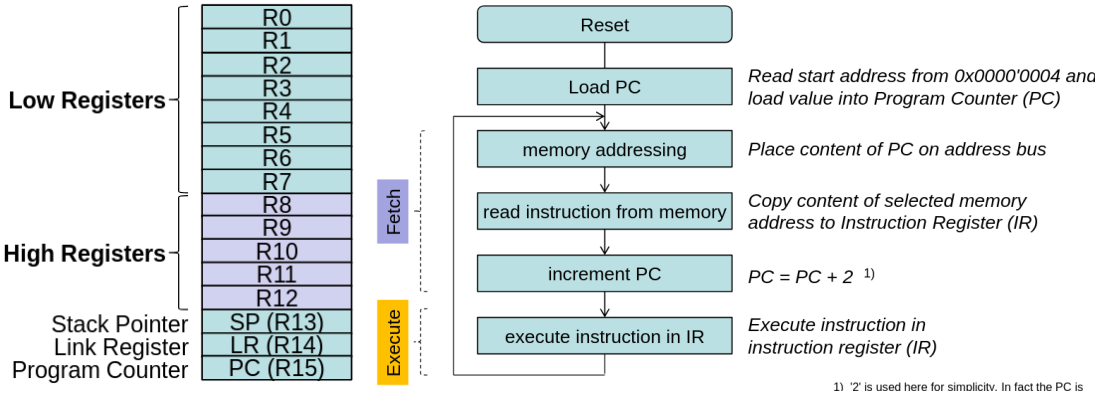
1 CPU



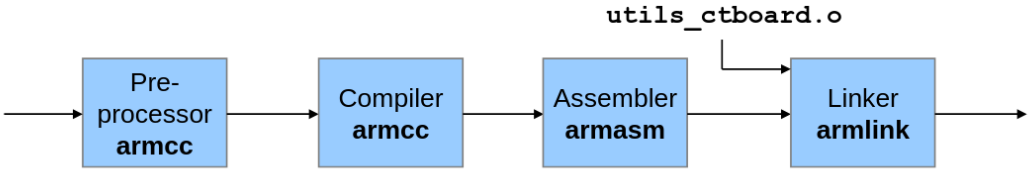


load SW into memory





2 Compiler etc.



main.c	main.i	main.s	main.o	main.axf
Source program (text)	*Modified source program (text)*	*Assembly program (text) human-readable, CPU specific*	*Relocatable object program (binary)*	*Executable object program (binary)*

Type	Frequency	
Data transfer	43%	
Control flow	23%	
Arithmetic	15%	Data processing
Compare	13%	
Logical	5%	
Miscellaneous	1%	

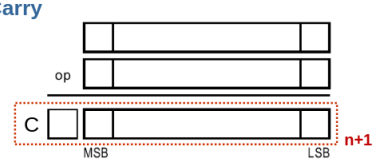
3 Arithmetic Operations

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) [1]
RSBS	Reverse Subtract (negative)	-1 • A
MULS	Multiplication	A • B

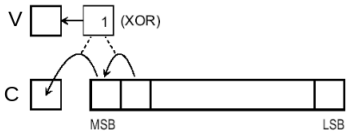
Mnemonic	Instruction	Function	C-Operator
ANDS	Bitwise AND	Rdn & Rm	a & b
BICS	Bit Clear	Rdn & !Rm	a & ~b
EORS	Exclusive OR	Rdn $ Rm	a ^ b
MVNS	Bitwise NOT	!Rm	~a
ORRS	Bitwise OR	Rdn # Rm	a ¦ b

Flag	Meaning	Action	Operands
Negative	MSB = 1	N = 1	signed
Zero	Result = 0	Z = 1	signed , unsigned
Carry	Carry	C = 1	unsigned
Overflow	Overflow	V = 1	signed

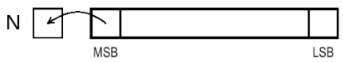
Carry



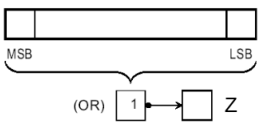
Overflow



Negative



Zero




```
0101 * 0011
     0011
    0000
   0011
  0000
 00001111
```

unsigned

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

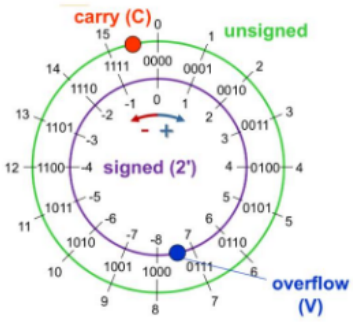
zero extension of multiplier

signed

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

sign extension of multiplier

- **unsigned** Interpretation
  - Program must check **carry flag (C)** after operation
  - **C = 1 for Addition** **C = 0 for Subtraction**
    - Result cannot be represented (not enough digits / no negative numbers)
    - Full turn on number circle must be added or subtracted → odometer effect
  - Overflow flag (V) irrelevant
- **signed** Interpretation
  - Program must check **overflow flag (V)** after operation
  - **V = 1** means
    - Not enough digits available to represent the result
    - Full turn on number circle must be added or subtracted → odometer effect
  - Carry flag (C) irrelevant
- **unsigned**
  - Addition → C = 1 → carry result too large for available bits
  - Subtraction → C = 0 → borrow result less than zero → no negative numbers
- **signed**
  - Addition → potential overflow in case of operands with equal signs
  - Subtraction → potential overflow in case of operands with opposite signs
- **Processors do not distinguish signed and unsigned**
  - Users (resp. compilers) have to know, whether they are working with signed or unsigned numbers
  - Processor always calculates flags for both cases
    - carry (C) unsigned
    - overflow (V) signed
      negative (N)



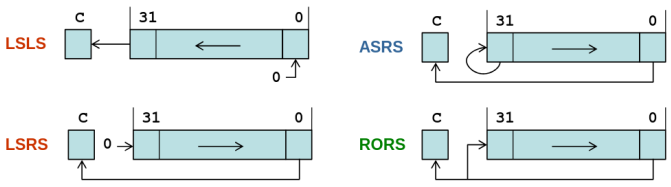
Flag	Meaning	Action	Operands
Negative	MSB = 1	N = 1	signed
Zero	Result = 0	Z = 1	signed , unsigned
Carry	Carry	C = 1	unsigned
Overflow	Overflow	V = 1	signed

- **How are groups of bytes arranged in memory?**
  - little endian → *least significant byte* at lower address
    e.g. Intel x86, Altera Nios, ST ARM (STM32)
  - big endian → *most significant byte* at lower address
    e.g. Freescale (Motorola), PowerPC

3.1 Shifts

- **Overview**

Mnemonic	Instruction	Function
LSLS	Logical Shift Left	$2^n \cdot Rn$ 0 → LSB
LSRS	Logical Shift Right	$2^{-n} \cdot Rn$ 0 → MSB
ASRS	Arithmetic Shift Right	$2^{-n} \cdot \pm A$ MSB → MSB
RORS	Rotate Right	LSB → MSB



3.2 Sign Extensions

- **Sign Extension Cortex-M0 (signed values)**
  - Extend word-length without changing value
  - SXTB Extends an 8-bit value to a 32-bit value
  - SXTH Extends a 16-bit value to a 32-bit value
- **Zero Extension Cortex-M0 (unsigned values)**
  - Extend word-length, fill with zeroes
  - UXTB Extends an 8-bit value to a 32-bit value
  - UXTH Extends a 16-bit value to a 32-bit value

Examples

```
SXTB  R3, R4   ; Extract lowest byte of the value in R4,
               ; sign extend it and write the result to R3
UXTH  R2, R3   ; Extract lower two bytes of the value in R3,
               ; zero extend it and write the result to R2
```

3.3 C Datatype Equivalents

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`

C-Type – signed integers	Size	Term	inttypes.h / stdint.h
`signed char`	8 Bit	Byte	`int8_t`
`short`	16 Bit	Half-word	`int16_t`
`int`	32 Bit	Word	`int32_t`
`long`	32 Bit	Word	`int32_t`
`long long`	64 Bit	Double-word	`int64_t`

4 Stack

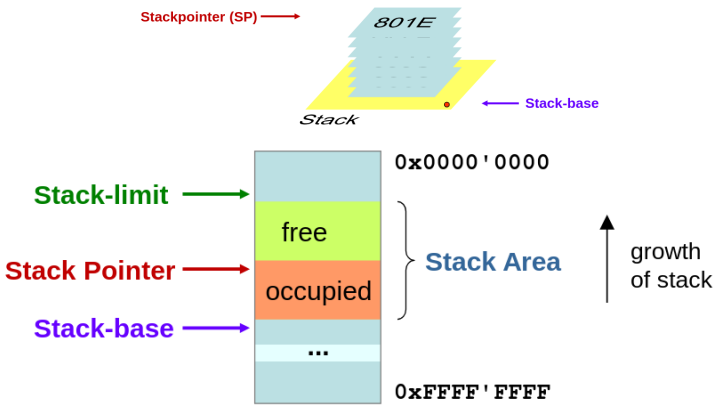
- **PUSH {R2,R3,R6}**

```
00000000 B083      SUB     SP,SP,#12
00000002 9200      STR     R2,[SP]
00000004 9301      STR     R3,[SP,#4]
00000006 9602      STR     R6,[SP,#8]
```

- **POP {R2,R3,R6}**

```
00000008 9A00      LDR     R2,[SP]
0000000A 9B01      LDR     R3,[SP,#4]
0000000C 9B02      LDR     R6,[SP,#8]
0000000E B003      ADD     SP,SP,#12
```

- **Stack as Object**
  - Methods
    - `PUSH()` and `POP()`
  - Data
    - pushed (written) on top of the stack
    - popped (fetched, read) from the top of the stack → LIFO[1)]



5 Switch-Case

- **Jump Table**

```
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
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
                DCD   case_1
                DCD   case_default
                DCD   case_3_5
                DCD   case_default
                DCD   case_3_5
```

6 Subroutines

- **Subroutines**
  - Structured programming
    - → Avoids duplicated code / clear interface
  - Call and return on ARM:
    - `BL <label>` and `BX LR / POP {PC}`
  - Nested subroutines → save LR on stack

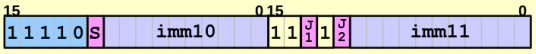
- **Mark start and end of a procedure / function**
  - Used by debugger (tool)
    - Buttons "step over" and "step out"
  - Structure code for reader

```
subrExample     PROC
                PUSH    {...,LR}
                ...
                ...
                POP     {...,PC}
                ENDP
```

- **BL <label>**
  - Store current PC in LR
  - Branch to <label>
    - PC = PC +/- offset
    - offset range -16'777'216 to 16'777'214

Subroutine call through a label
- unconditional
- relative
- direct

```
BL <label>
```

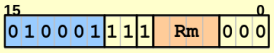



```
I1 = NOT(J1 EOR S); I2 = NOT (J2 EOR S)
<imm> = S:I1:I2:imm10:imm11:0
LR = PC (LSB set to '1')
PC = PC + <imm>
```

- **BLX (register)**
  - Store current PC in LR
  - Address of subroutine in register
  - Branch
    - PC = register
    - Branch address from 0 to 2^{32}

Subroutine call with address in register
- unconditional
- absolute
- indirect

```
BLX <Rm>
```




```
LR = PC - 2 (LSB set to '1')
PC = Rm
```

7 Arrays

```
word_array      DCD     0xFFEEDDCC
                DCD     0xBBAA9988
                DCD     0x77665544
                DCD     0x33221100
```

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

```
                AREA    progData2, DATA, READWRITE
data_array      SPACE   256
```

8 Constants and Literals

```
MY_CONST8   EQU  0x12
            MOVS  R1,#MY_CONST8
```

```
1  CONST_A                    EQU    0x000000AA
2  CONST_B                    EQU    0xBBCCDDEE
3
4  0000003C 4F03              LDR    R7,=0x12
5  0000003E 4F04              LDR    R7,=CONST_A
6  00000040 4F04              LDR    R7,=CONST_B
7  00000042 4D01              LDR    R5,mylita
8  00000044 4D04              LDR    R5,=mylita
9  00000046 E02F              B      wherever
10
11 00000048 FF001122 mylita   DCD    0xFF001122
12
13 0000004C 00000012
14 00000050 000000AA
15 00000054 BBCCDDEE
16 00000058 00000000
```

load **0xFF001122** from address **0x00000048**

load address of mylita from address **0x00000058**

Literal Pool

CONST_A and CONST_B

space where the address of mylita will be stored after linking

9 Conditional Jumps

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

Arithmetic - unsigned: higher and 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

Arithmetic - signed: greater and less

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

10 Modular Coding

Where?

- **Register**
  - Caller and Callee[1] use the same register
- **Global variables**
  - Shared variables in data area (section)
- **Stack**
  - Caller → PUSH parameter on stack
  - Callee → access parameter through `LDR <Rt>,[SP,#<imm>]`

How?

- **pass by value**
  - Handover the value
- **pass by reference**
  - Handover the address to a value

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 a calculation
- Usually, such values are not named in the program source and have a limited lifetime

Variable Register

- A register used to hold the value of a variable, usually one local to a routine, and often named in the source code.
- Cortex-M0 registers R8 – R11 (v5 – v8) are often unused
  - as they are accessible only by few instructions

Argument, Parameter

- Used interchangeably
- Formal parameter of a subroutine

Parameters

- Caller copies arguments to R0 to R3
- Caller copies additional parameters to stack

Returning fundamental data types

• Smaller than word	zero or sign extend to word; return in R0
• Word	return in R0
• Double-word	return in R0 / R1 [1]
• 128-bit	return in R0 – R3 [1]

Returning composite data types *(structs, arrays, ...)*

• Up to 4 bytes	return in R0
• Larger than 4 bytes	stored in data area; address passed as extra argument at function call

```
void caller(void)
{
    uint32_t p = 4;
    uint32_t q = 5;
    uint32_t r = 6;
    uint32_t sum;

    sum = callee(p,q,r);
}
```

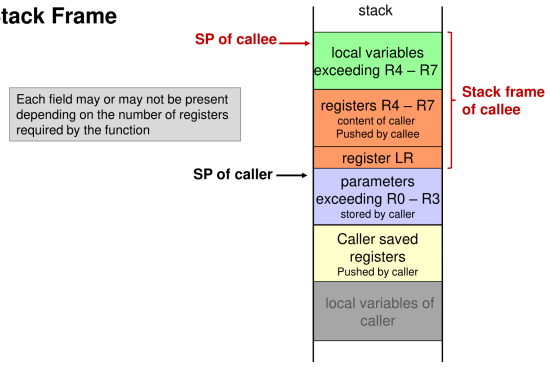
```
uint32_t callee(uint32_t a,
                uint32_t b,
                uint32_t c)
{
    return a + b + c;
}
```

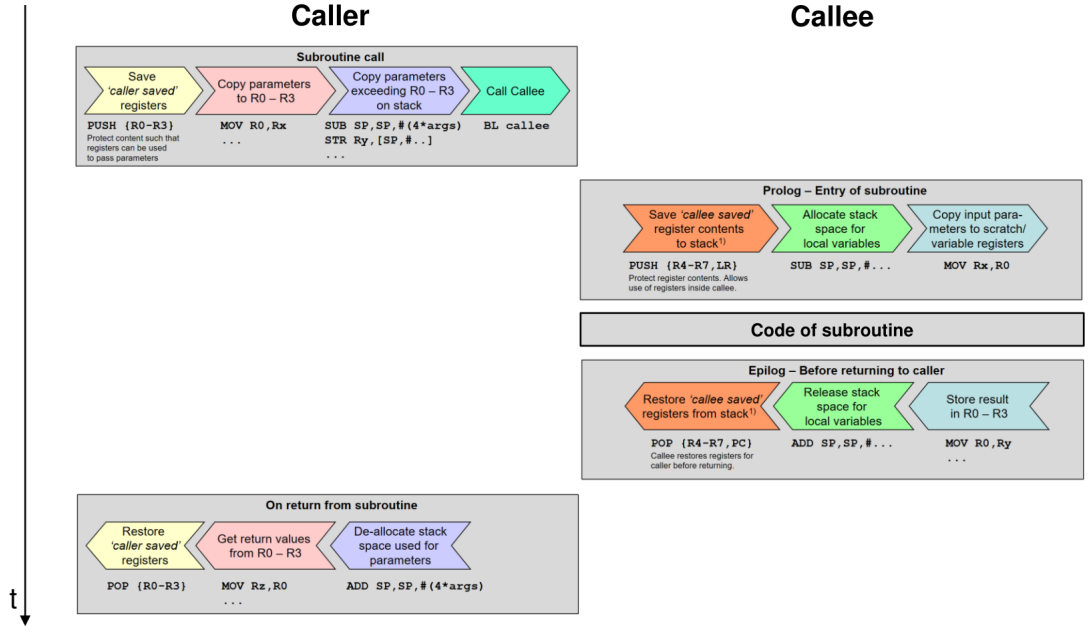
```
MOVS  r4,#4
MOVS  r5,#5
MOVS  r6,#6      local variables R4 – R6

MOV   r2,r6
MOV   r1,r5
MOV   r0,r4      copy parameters to R0 – R2

BL    callee
MOV   r7,r0      copy return value to local variable
```

```
callee PROC
  ADDS  r0,r0,r1
  ADDS  r0,r0,r2   callee uses own copies of parameters
  BX    lr
ENDP
```

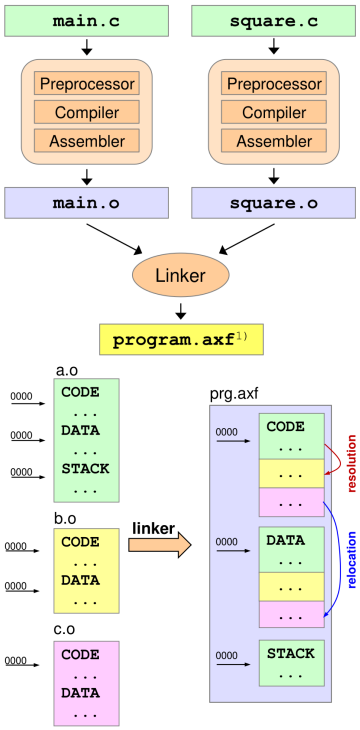
PUSH and POP are omitted in the example

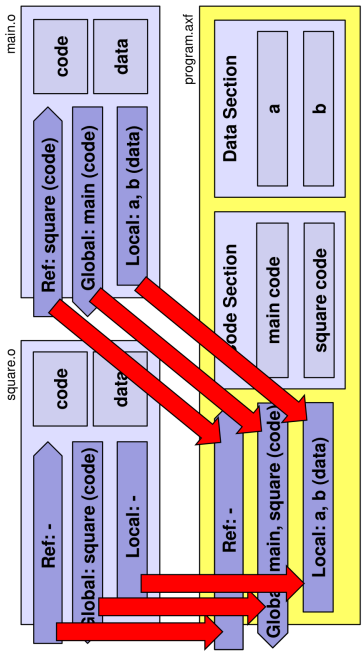
Stack Frame



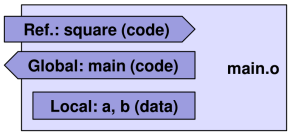


11 Linker



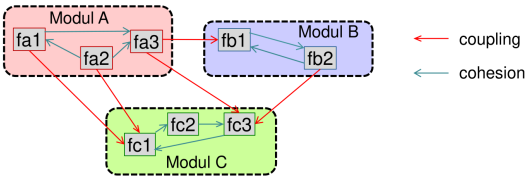



```
IMPORT square
...
EXPORT main
main: ...
...
a:    ...
b:    ...
```



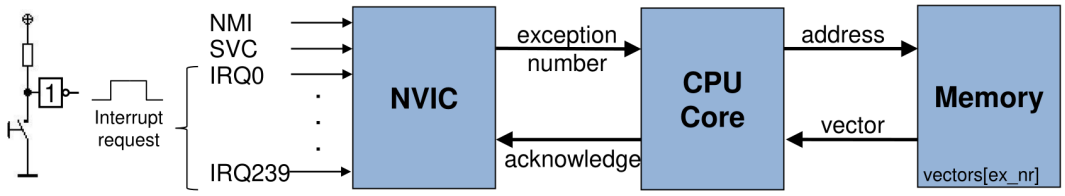
- References
  - Imported symbols from assembly code translate to global reference symbols in the object file
- Global
  - Exported symbols from assembly code translate to global symbols in the object file
- Local
  - Internal symbols from assembly code translate to local symbols in the object file

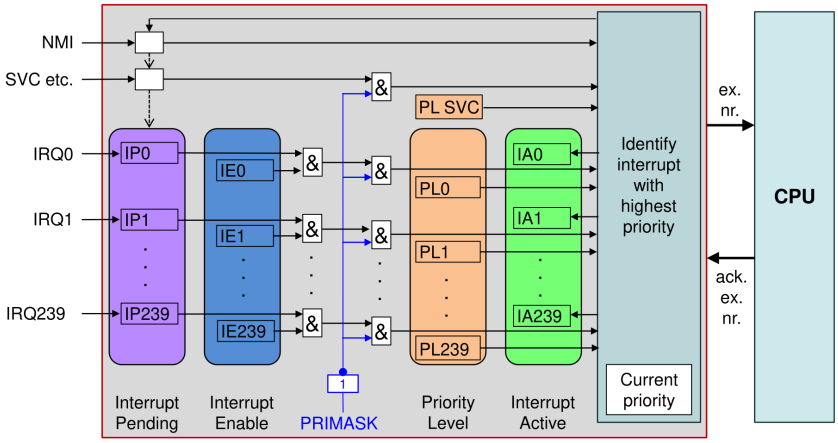
- **High module cohesion**
  - Group together what belongs together
  - Lean external interface
  - Idea: each module fulfills a **single** defined task
- **Low module coupling**
  - Split what does not belong together
  - Little dependencies between modules



2. Das .lst-File wird nach dem Compilieren vom Assembler erzeugt.
3. Im .lst-File können einige Instruktionen noch nicht fertig assembliert sein, weil dem Assembler gewisse Informationen fehlen.
4. Aus dem .lst-File kann man bereits exakt lesen, wieviel Platz die AREAs eines Moduls benötigen werden.

12 Interrupts





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

source: "The definitive Guide to the Cortex-M3"

Non-maskable Interrupt (NMI)

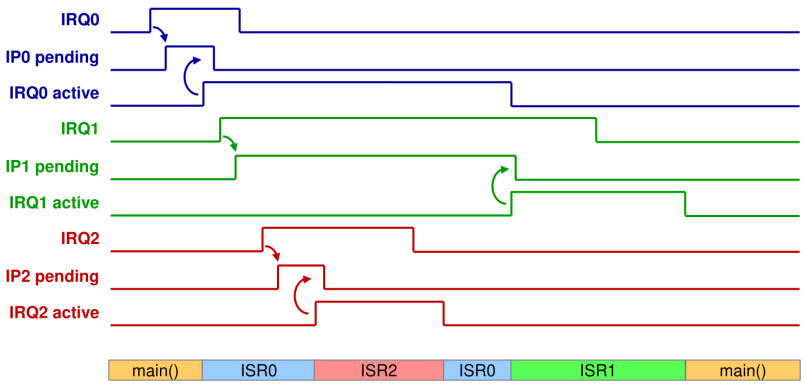
- Power-fail, emergency button, watchdog, ...

Example Priorities

- ISR1 does not preempt ISR0
- ISR2 preempts ISR0

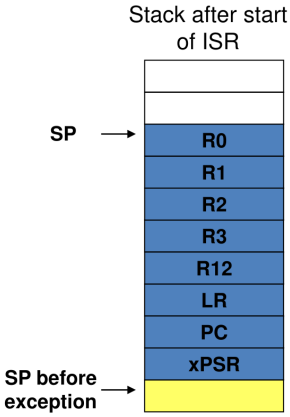
assuming

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



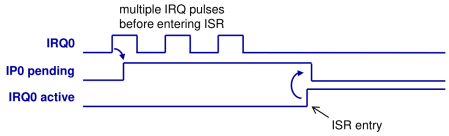
Preemption

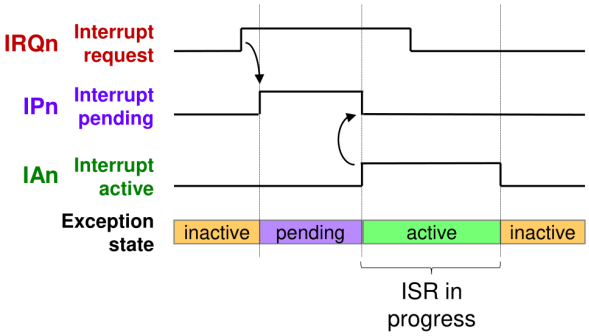
- Service routine A temporarily interrupts service routine B
- Assigned priority level for each exception
  - Levels define whether A can preempt B
- Fixed priorities
  - Reset (-3), NMI (-2), hard fault (-1)
- All other priorities are programmable



Multiple IRQ request pulses before entering ISR

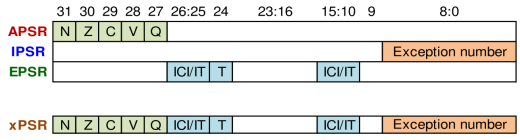
- Treated as single interrupt
- Events are lost



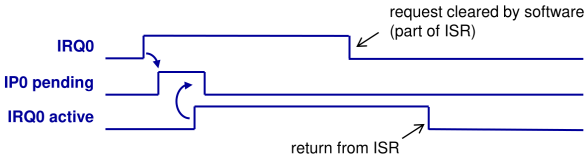


Program Status Registers (PSRs)

- APSR Application Program Status Register
- IPSR Interrupt Program Status Register
- EPSR Execution Program Status Register



- xPSR Combination of all three PSRs




```
; 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   ...                 } System Exceptions 15 vectors
...             DCD   ...

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

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

                ...         ; interrupt service

                POP { ... }
                BX LR
```

		Assembly	C
- Disable	set PRIMASK	`CPSID[1)] i`	`__disable_irq();`
- Enable	clear PRIMASK	`CPSIE[1)] i`	`__enable_irq();`

Enable IRQ3

```
SETENA0 EQU   0xE000E100
        ...

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

Disable IRQ3

```
CLRENA0 EQU   0xE000E180
        ...

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

Trigger IRQ3 by Software

```
SETPEND0  EQU 0xE000E200
          ...

          LDR  R0,=SETPEND0
          MOVS R1,#0x08
          STR  R1,[R0]
```

Delete Pending IRQ3

```
CLRPEND0 EQU 0xE000E280
         ...

         LDR  R0,=CLRPEND0
         MOVS      R1,#0x08
         STR  R1,[R0]
```

ARM and ST use different names for the same register

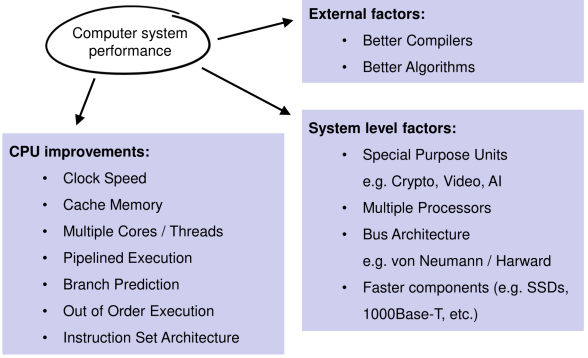
Test if IRQ3 is active

```
ACTIVE0 EQU 0xE000E300
        ...

        LDR  R0, =ACTIVE0
        MOVS R1, #0x08
        LDR  R2, [R0]
        TST  R1, R2
        BEQ  ...
```

Set priority of IRQ3 to 5

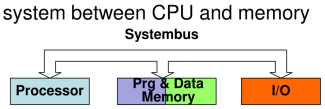
```
PL_IRQ3  EQU   0xE000E403
         ...

         LDR   R0, =PL_REG_IRQ3
         MOVS  R1, #0x50
         STRB  R1, [R0]
```

13 Optimizations



von Neumann Architecture

- Same memory holds program and data
- Single bus system between CPU and memory



Harvard Architecture

- "Mark I" at Harvard University (Howard Aiken, 1939-44)
- Separate memories for program and data
- Two sets of address/data buses between CPU and memory

RISC

- Load / Store Architecture
- Data processing instructions only available on registers

CISC

- One of the operands of an instruction may directly be a memory location

Example: **Balance = Balance + Credit**

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

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

RISC advantages

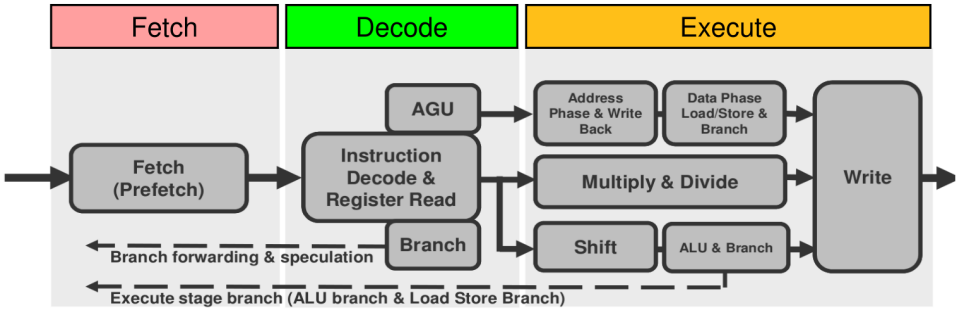
- Simple and fast instruction decoding
- More registers/cache on silicon area (less memory access)
- Several data paths possible (superscalar computers)
- Higher clock frequencies
- Effective compiler optimization with limited, generic instructions
- Easy pipelining, shorter pipelines (instruction size / duration)

CISC advantages

- Less program memory needed with complex instructions
- Short programs may work faster with less memory accesses

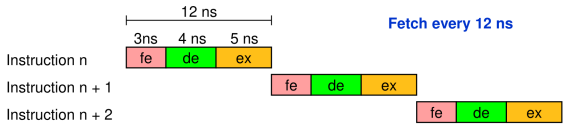
CISC and RISC more and more indistinguishable

- Modern x86 CPUs convert instructions (CISC to RISC)

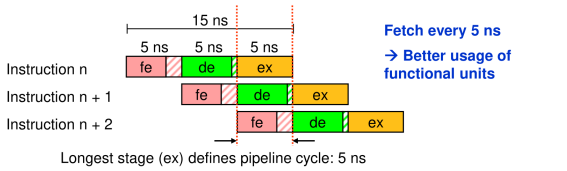


Source: ARM

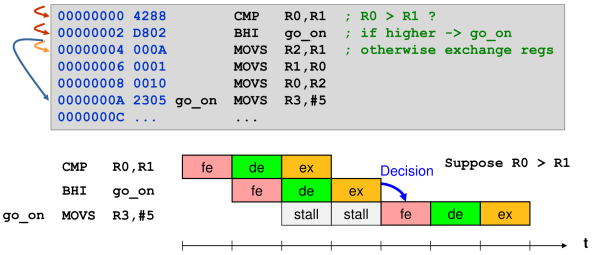
- **Sequential execution**



- **Pipelined execution**



- **Control Hazards**
  - Branch / jump decisions occur in stage 3 (ex)
  - Worst case scenario – conditional branch taken:



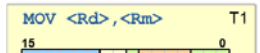
Ideas to further improve pipelining:

- Branch prediction:
  - Store last decision made for each conditional branch
    → probability is high that the same decision is taken again
- Instruction prefetch:
  - Fetch several instructions in advance
    → better use of the system bus
    → possibility of 'Out of Order Execution'
- Out of Order Execution:
  - If one instruction stalls (e.g. a LDR from slow memory), it might be possible to already execute the next instruction

ARM v6-M Instruction Set

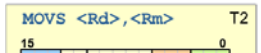
MOV

MOV <Rd>,<Rm> T1



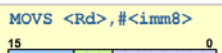
RD = D:Rd → RD = Rm

MOVS <Rd>,<Rm> T2



Rd = Rm

MOVS <Rd>,#<imm8>



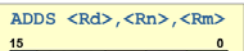
Rd = <imm8>

ADD

ADD <Rdn>,<Rm>

Rdn = Rdn + Rm

ADDS <Rd>,<Rn>,<Rm>

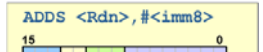


Rd = Rn + Rm

ADDS <Rd>,<Rn>,#<imm3>

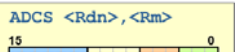
Rd = Rn + <imm3>

ADDS <Rdn>,#<imm8>



Rdn = Rdn + <imm8>

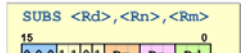
ADCS <Rdn>,<Rm>



Rdn = Rdn + Rm + C

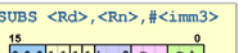
Subtract

SUBS <Rd>,<Rn>,<Rm>



Rd = Rn - Rm
= Rn + NOT(Rm) + 1

SUBS <Rd>,<Rn>,#<imm3>

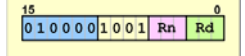


Rd = Rn - <imm3>
= Rn + NOT<imm3> + 1

SUBS <Rdn>,#<imm8>

Rdn = Rdn - <imm8>
= Rdn + NOT<imm8> + 1

RSBS <Rd>,<Rn>,#0



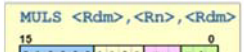
Rd = 0 - Rn

SBCS <Rdn>,<Rm>

Rdn = Rdn - Rm - NOT(C)
= Rdn + NOT(Rm) + C [1]

Multiply

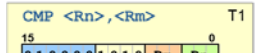
MULS <Rdm>,<Rn>,<Rdm>



Rdm = Rn * Rdm

Compare

CMP <Rn>,<Rm> T1



Rn - Rm → N,Z,C,V

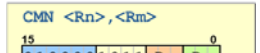
CMP <Rn>,<Rm> T2

Rn - Rm → N,Z,C,V

CMP <Rn>,#<imm8>

Rn - <imm8> → N,Z,C,V

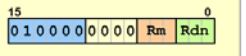
CMN <Rn>,<Rm>



Rn + Rm → N,Z,C,V

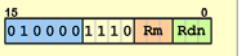
Logical

ANDS <Rdn>,<Rdn>,<Rm>



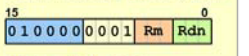
Rdn = Rdn & Rm

BICS <Rdn>,<Rdn>,<Rm>



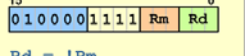
Rdn = Rdn & !Rm

EORS <Rdn>,<Rdn>,<Rm>



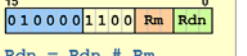
Rdn = Rdn $ Rm

MVNS <Rd>,<Rm>



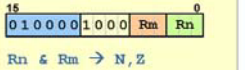
Rd = !Rm

ORRS <Rdn>,<Rdn>,<Rm>



Rdn = Rdn # Rm

TST <Rn>,<Rm>



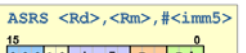
Rn & Rm → N,Z

Shift/Rotate

ASRS <Rdn>,<Rdn>,<Rm>

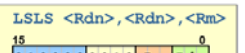
Rdn = shift Rdn right
by Rm<7:0> bits,
fill with MSB [2]

ASRS <Rd>,<Rm>,#<imm5>



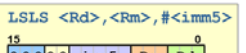
Rd = shift Rm right
by <imm5> bits
fill with MSB

LSLS <Rdn>,<Rdn>,<Rm>



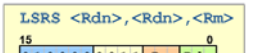
Rdn = shift Rdn left
by Rm<7:0> bits,
fill with zeros [2]

LSLS <Rd>,<Rm>,#<imm5>



Rd = shift Rm left
by <imm5> bits
fill with zeros

LSRS <Rdn>,<Rdn>,<Rm>

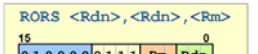


Rdn = shift Rdn right
by Rm<7:0> bits,
fill with zeros [2]

LSRS <Rd>,<Rm>,#<imm5>

Rd = shift Rm right
by <imm5> bits
fill with zeros

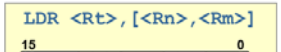
RORS <Rdn>,<Rdn>,<Rm>



Rdn = cyclic rotate right

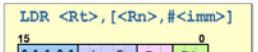
Load

LDR <Rt>,[<Rn>,<Rm>]



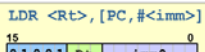
Rt = Mem[@(Rn+Rm)]

LDR <Rt>,[<Rn>,#<imm>]



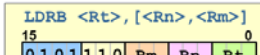
<imm> = imm5:00
Rt = Mem[@(Rn + <imm>)]

LDR <Rt>,[PC,#<imm>]



<imm> = imm8:00
Rt = Mem[@(PC + <imm>)]

LDRB <Rt>,[<Rn>,<Rm>]

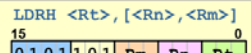


Rt = Byte[@(Rn+Rm)]

LDRB <Rt>,[<Rn>,#<imm>]

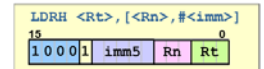
<imm> = imm5
Rt = Byte[@(Rn+<imm>)]

LDRH <Rt>,[<Rn>,<Rm>]



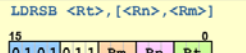
Rt = Hw[@(Rn+Rm)]

LDRH <Rt>,[<Rn>,#<imm>]



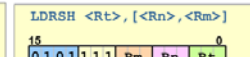
<imm> = imm5:0
Rt = Hw[@(Rn+imm5)]

LDRSB <Rt>,[<Rn>,<Rm>]



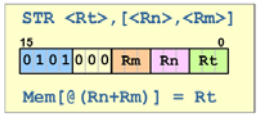
Rt = sign_extend(Byte[@(Rn+Rm)])

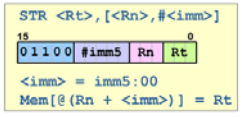
LDRSH <Rt>,[<Rn>,<Rm>]

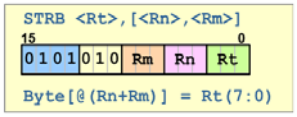


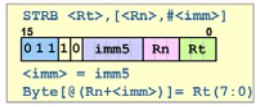
Rt = sign_extend(HWord[@(Rn+Rm)])

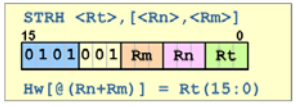
Store

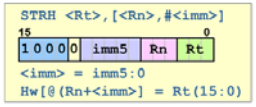






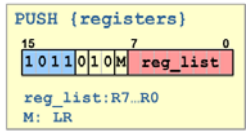


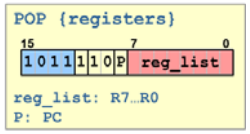




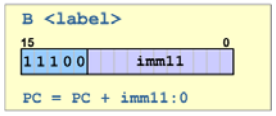
Load/Store Multiple

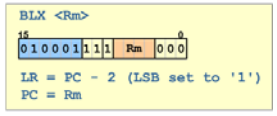
Push/Pop

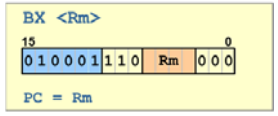


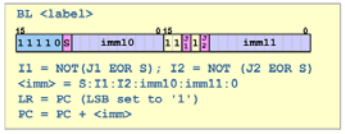


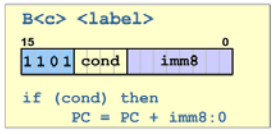
Branch





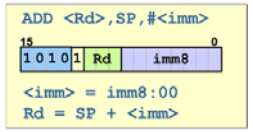


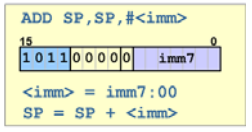


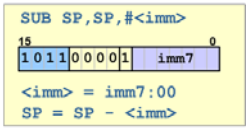


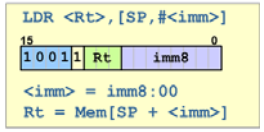
cond	short	Flag	cond	short	Flag	cond	short	Flag
0000	EQ	Z == 1	0110	VS	V == 1	1100	GT	Z == 0 and N == V
0001	NE	Z == 0	0111	VC	V == 0	1101	LE	Z == 1 or N != V
0010	CS/HS	C == 1	1000	HI	C == 1 and Z == 0	1110	AL	always
0011	CC/LO	C == 0	1001	LS	C == 0 or Z == 1	1111	--	--
0100	MI	N == 1	1010	GE	N == V			
0101	PL	N == 0	1011	LT	N != V			

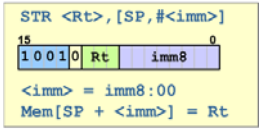
Stack Operations



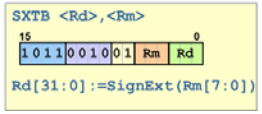


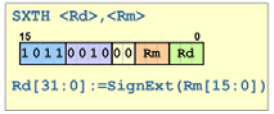


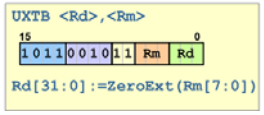


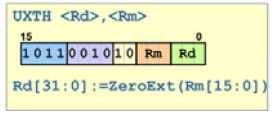


Extend









Pseudo Instructions

```
LDR <Rt>, <label>    =>   LDR <Rt>, [PC, #<imm>]
LDR <Rt>, =<value>   =>   LDR <Rt>, [PC, #<imm>]
                          …
                          Literalpool
                          DCD  value
```

Weitere Befehle

```
REV     REV16   REVSH   SVC     CPSID   CPSIE   SETEND  BKPT    NOP     SEV
WFE     WFI     YIELD
```

Thumb® 16-bit Instruction Set
Quick Reference Card

This card lists all Thumb instructions available on Thumb-capable processors earlier than ARM®v6T2. In addition, it lists all Thumb-2 16-bit instructions.
The instructions shown on this card are all 16-bit in Thumb-2, except where noted otherwise.
All registers are Lo (R0-R7) except where specified. Hi registers are R8-R15.

Key to Tables				
§	See Table **ARM architecture versions**.		`<loreglist+LR>`	A comma-separated list of Lo registers. plus the LR, enclosed in braces, { and }.
`<loreglist>`	A comma-separated list of Lo registers, enclosed in braces, { and }.		`<loreglist+PC>`	A comma-separated list of Lo registers. plus the PC, enclosed in braces, { and }.

Operation		**§**	**Assembler**	**Updates**	**Action**	**Notes**
Move	Immediate		`MOVS Rd, #<imm>`	N Z	Rd := imm	imm range 0-255.
	Lo to Lo		`MOVS Rd, Rm`	N Z	Rd := Rm	Synonym of `LSLS Rd, Rm, #0`
	Hi to Lo, Lo to Hi, Hi to Hi		`MOV Rd, Rm`		Rd := Rm	Not Lo to Lo.
	Any to Any	6	`MOV Rd, Rm`		Rd := Rm	Any register to any register.
Add	Immediate 3		`ADDS Rd, Rn, #<imm>`	N Z C V	Rd := Rn + imm	imm range 0-7.
	All registers Lo		`ADDS Rd, Rn, Rm`	N Z C V	Rd := Rn + Rm	
	Hi to Lo, Lo to Hi, Hi to Hi		`ADD Rd, Rd, Rm`		Rd := Rd + Rm	Not Lo to Lo.
	Any to Any	T2	`ADD Rd, Rd, Rm`		Rd := Rd + Rm	Any register to any register.
	Immediate 8		`ADDS Rd, Rd, #<imm>`	N Z C V	Rd := Rd + imm	imm range 0-255.
	With carry		`ADCS Rd, Rd, Rm`	N Z C V	Rd := Rd + Rm + C-bit	
	Value to SP		`ADD SP, SP, #<imm>`		SP := SP + imm	imm range 0-508 (word-aligned).
	Form address from SP		`ADD Rd, SP, #<imm>`		Rd := SP + imm	imm range 0-1020 (word-aligned).
	Form address from PC		`ADR Rd, <label>`		Rd := label	label range PC to PC+1020 (word-aligned).
Subtract	Lo and Lo		`SUBS Rd, Rn, Rm`	N Z C V	Rd := Rn – Rm	
	Immediate 3		`SUBS Rd, Rn, #<imm>`	N Z C V	Rd := Rn – imm	imm range 0-7.
	Immediate 8		`SUBS Rd, Rd, #<imm>`	N Z C V	Rd := Rd – imm	imm range 0-255.
	With carry		`SBCS Rd, Rd, Rm`	N Z C V	Rd := Rd – Rm – NOT C-bit	
	Value from SP		`SUB SP, SP, #<imm>`		SP := SP – imm	imm range 0-508 (word-aligned).
	Negate		`RSBS Rd, Rn, #0`	N Z C V	Rd := – Rn	Synonym: `NEGS Rd, Rn`
Multiply	Multiply		`MULS Rd, Rm, Rd`	N Z * *	Rd := Rm * Rd	* C and V flags unpredictable in §4T, unchanged in §5T and above
Compare			`CMP Rn, Rm`	N Z C V	update APSR flags on Rn – Rm	Can be Lo to Lo, Lo to Hi, Hi to Lo, or Hi to Hi.
	Negative		`CMN Rn, Rm`	N Z C V	update APSR flags on Rn + Rm	
	Immediate		`CMP Rn, #<imm>`	N Z C V	update APSR flags on Rn – imm	imm range 0-255.
Logical	AND		`ANDS Rd, Rd, Rm`	N Z	Rd := Rd AND Rm	
	Exclusive OR		`EORS Rd, Rd, Rm`	N Z	Rd := Rd EOR Rm	
	OR		`ORRS Rd, Rd, Rm`	N Z	Rd := Rd OR Rm	
	Bit clear		`BICS Rd, Rd, Rm`	N Z	Rd := Rd AND NOT Rm	
	Move NOT		`MVNS Rd, Rm`	N Z	Rd := NOT Rm	
	Test bits		`TST Rn, Rm`	N Z	update APSR flags on Rn AND Rm	
Shift/rotate	Logical shift left		`LSLS Rd, Rm, #<shift>`	N Z C*	Rd := Rm << shift	Allowed shifts 0-31. * C flag unaffected if shift is 0.
			`LSLS Rd, Rd, Rs`	N Z C*	Rd := Rd << Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Logical shift right		`LSRS Rd, Rm, #<shift>`	N Z C	Rd := Rm >> shift	Allowed shifts 1-32.
			`LSRS Rd, Rd, Rs`	N Z C*	Rd := Rd >> Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Arithmetic shift right		`ASRS Rd, Rm, #<shift>`	N Z C	Rd := Rm ASR shift	Allowed shifts 1-32.
			`ASRS Rd, Rd, Rs`	N Z C*	Rd := Rd ASR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.
	Rotate right		`RORS Rd, Rd, Rs`	N Z C*	Rd := Rd ROR Rs[7:0]	* C flag unaffected if Rs[7:0] is 0.

Thumb 16-bit Instruction Set
Quick Reference Card

Operation		§	Assembler	Action	Notes
Load	with immediate offset, word		`LDR Rd, [Rn, #<imm>]`	Rd := [Rn + imm]	imm range 0-124, multiple of 4.
	halfword		`LDRH Rd, [Rn, #<imm>]`	Rd := ZeroExtend([Rn + imm][15:0])	Clears bits 31:16. imm range 0-62, even.
	byte		`LDRB Rd, [Rn, #<imm>]`	Rd := ZeroExtend([Rn + imm][7:0])	Clears bits 31:8. imm range 0-31.
	with register offset, word		`LDR Rd, [Rn, Rm]`	Rd := [Rn + Rm]	
	halfword		`LDRH Rd, [Rn, Rm]`	Rd := ZeroExtend([Rn + Rm][15:0])	Clears bits 31:16
	signed halfword		`LDRSH Rd, [Rn, Rm]`	Rd := SignExtend([Rn + Rm][15:0])	Sets bits 31:16 to bit 15
	byte		`LDRB Rd, [Rn, Rm]`	Rd := ZeroExtend([Rn + Rm][7:0])	Clears bits 31:8
	signed byte		`LDRSB Rd, [Rn, Rm]`	Rd := SignExtend([Rn + Rm][7:0])	Sets bits 31:8 to bit 7
	PC-relative		`LDR Rd, <label>`	Rd := [label]	label range PC to PC+1020 (word-aligned).
	SP-relative		`LDR Rd, [SP, #<imm>]`	Rd := [SP + imm]	imm range 0-1020, multiple of 4.
	Multiple, not including base		`LDM Rn!, <loreglist>`	Loads list of registers (not including Rn)	Always updates base register, Increment After.
	Multiple, including base		`LDM Rn, <loreglist>`	Loads list of registers (including Rn)	Never updates base register, Increment After.
Store	with immediate offset, word		`STR Rd, [Rn, #<imm>]`	[Rn + imm] := Rd	imm range 0-124, multiple of 4.
	halfword		`STRH Rd, [Rn, #<imm>]`	[Rn + imm][15:0] := Rd[15:0]	Ignores Rd[31:16]. imm range 0-62, even.
	byte		`STRB Rd, [Rn, #<imm>]`	[Rn + imm][7:0] := Rd[7:0]	Ignores Rd[31:8]. imm range 0-31.
	with register offset, word		`STR Rd, [Rn, Rm]`	[Rn + Rm] := Rd	
	halfword		`STRH Rd, [Rn, Rm]`	[Rn + Rm][15:0] := Rd[15:0]	Ignores Rd[31:16]
	byte		`STRB Rd, [Rn, Rm]`	[Rn + Rm][7:0] := Rd[7:0]	Ignores Rd[31:8]
	SP-relative, word		`STR Rd, [SP, #<imm>]`	[SP + imm] := Rd	imm range 0-1020, multiple of 4.
	Multiple		`STM Rn!, <loreglist>`	Stores list of registers	Always updates base register, Increment After.
Push	Push		`PUSH <loreglist>`	Push registers onto full descending stack	
	Push with link		`PUSH <loreglist+LR>`	Push LR and registers onto full descending stack	
Pop	Pop		`POP <loreglist>`	Pop registers from full descending stack	
	Pop and return	4T	`POP <loreglist+PC>`	Pop registers, branch to address loaded to PC	
	Pop and return with exchange	5T	`POP <loreglist+PC>`	Pop, branch, and change to ARM state if address[0] = 0	
If-Then	If-Then	T2	`IT{pattern} {cond}`	Makes up to four following instructions conditional, according to pattern. pattern is a string of up to three letters. Each letter can be T (Then) or E (Else).	The first instruction after IT has condition cond. The following instructions have condition cond if the corresponding letter is T, or the inverse of cond if the corresponding letter is E. See Table **Condition Field**.
Branch	Conditional branch		`B{cond} <label>`	If {cond} then PC := label	label must be within – 252 to + 258 bytes of current instruction. See Table **Condition Field**.
	Compare, branch if (non) zero	T2	`CB{N}Z Rn,<label>`	If Rn {== | !=} 0 then PC := label	label must be within +4 to +130 bytes of current instruction.
	Unconditional branch		`B <label>`	PC := label	label must be within ±2KB of current instruction.
	Long branch with link		`BL <label>`	LR := address of next instruction, PC := label	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch and exchange		`BX Rm`	PC := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.
	Branch with link and exchange	5T	`BLX <label>`	LR := address of next instruction, PC := label Change to ARM	This is a 32-bit instruction. label must be within ±4MB of current instruction (T2: ±16MB).
	Branch with link and exchange	5T	`BLX Rm`	LR := address of next instruction, PC := Rm AND 0xFFFFFFFE	Change to ARM state if Rm[0] = 0.
Extend	Signed, halfword to word	6	`SXTH Rd, Rm`	Rd[31:0] := SignExtend(Rm[15:0])	
	Signed, byte to word	6	`SXTB Rd, Rm`	Rd[31:0] := SignExtend(Rm[7:0])	
	Unsigned, halfword to word	6	`UXTH Rd, Rm`	Rd[31:0] := ZeroExtend(Rm[15:0])	
	Unsigned, byte to word	6	`UXTB Rd, Rm`	Rd[31:0] := ZeroExtend(Rm[7:0])	
Reverse	Bytes in word	6	`REV Rd, Rm`	Rd[31:24] := Rm[7:0], Rd[23:16] := Rm[15:8], Rd[15:8] := Rm[23:16], Rd[7:0] := Rm[31:24]	
	Bytes in both halfwords	6	`REV16 Rd, Rm`	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:24] := Rm[23:16], Rd[23:16] := Rm[31:24]	
	Bytes in low halfword, sign extend	6	`REVSH Rd, Rm`	Rd[15:8] := Rm[7:0], Rd[7:0] := Rm[15:8], Rd[31:16] := Rm[7] * &FFFF	

Thumb 16-bit Instruction Set
Quick Reference Card

Operation		§	Assembler	Action	Notes
Processor state change	Supervisor Call		`SVC <immed_8>`	Supervisor Call processor exception	8-bit immediate value encoded in instruction. Formerly SWI.
	Change processor state	6	`CPSID <iflags>`	Disable specified interrupts	
		6	`CPSIE <iflags>`	Enable specified interrupts	
	Set endianness	6	`SETEND <endianness>`	Sets endianness for loads and saves.	`<endianness>` can be `BE` (Big Endian) or `LE` (Little Endian).
	Breakpoint	5T	`BKPT <immed_8>`	Prefetch abort *or* enter debug state	8-bit immediate value encoded in instruction.
No Op	No operation		`NOP`	None, might not even consume any time.	Real NOP available in ARM v6K and above.
Hint	Set event	T2	`SEV`	Signal event in multiprocessor system.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Wait for event	T2	`WFE`	Wait for event, IRQ, FIQ, Imprecise abort, or Debug entry request.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Wait for interrupt	T2	`WFI`	Wait for IRQ, FIQ, Imprecise abort, or Debug entry request.	Executes as NOP in Thumb-2. Functionally available in ARM v7.
	Yield	T2	`YIELD`	Yield control to alternative thread.	Executes as NOP in Thumb-2. Functionally available in ARM v7.

Condition Field	
Mnemonic	**Description**
`EQ`	Equal
`NE`	Not equal
`CS / HS`	Carry Set / Unsigned higher or same
`CC / LO`	Carry Clear / Unsigned lower
`MI`	Negative
`PL`	Positive or zero
`VS`	Overflow
`VC`	No overflow
`HI`	Unsigned higher
`LS`	Unsigned lower or same
`GE`	Signed greater than or equal
`LT`	Signed less than
`GT`	Signed greater than
`LE`	Signed less than or equal
`AL`	Always. Do not use in `B{cond}`

In Thumb code for processors earlier than ARMv6T2, cond must not appear anywhere except in Conditional Branch (`B{cond}`) instructions.

In Thumb-2 code, cond can appear in any of these instructions (except `CBZ`, `CBNZ`, `CPSID`, `CPSIE`, `IT`, and `SETEND`).

The condition is encoded in a preceding `IT` instruction (except in the case of `B{cond}` instructions).

If `IT` instructions are explicitly provided in the Assembly language source file, the conditions in the instructions must match the corresponding `IT` instructions.

ARM architecture versions	
4T	All Thumb versions of ARM v4 and above.
5T	All Thumb versions of ARM v5 and above.
6	All Thumb versions of ARM v6 and above.
T2	All Thumb-2 versions of ARM v6 and above.

Document Number

ARM QRC 0006E

Change Log

Issue	Date	Change
A	Nov 2004	First Release
B	May 2005	RVCT 2.2 SP1
C	March 2006	RVCT 3.0
D	March 2007	RVCT 3.1
E	Sept 2008	RVCT 4.0

ARM® Thumb® Cortex-M0/M1 Instruction Set ordered by machine code

This card lists all Thumb instructions ordered by machine code to ease manually disassemble Thumb code.
See the respective ***Thumb® 16-bit Instruction Set Quick Reference Card*** for details on the individual instructions.

Version 1.3, 2019-08-20, Andreas Gieriet

Zürcher Hochschule für Angewandte Wissenschaften
zhaw School of Engineering
InES Institute of Embedded Systems

```
0000 – 0x0xxx Instructions
0000 0000 00mm mddd    MOVS  Rddd, Rmmm          ; Rddd  = Rmmm              --> alias for LSLS Rddd,Rmmm,#0
0000 0iii iimm mddd    LSLS  Rddd, Rmmm, #0biiiii; Rddd  = Rmmm LSL #0b0iiiii
0000 1iii iimm mddd    LSRS  Rddd, Rmmm, #0biiiii; Rddd  = Rmmm LSR #0b0iiiii

0001 – 0x1xxx Instructions
0001 0iii iimm mddd    ASRS  Rddd, Rmmm, #0biiiii; Rddd  = Rmmm ASR #0b0iiiii
0001 100m mmnn nddd    ADDS  Rddd, Rnnn, Rmmm    ; Rddd  = Rnnn  +  Rmmm
0001 101m mmnn nddd    SUBS  Rddd, Rnnn, Rmmm    ; Rddd  = Rnnn  -  Rmmm
0001 110i iinn nddd    ADDS  Rddd, Rnnn, #0biii  ; Rddd  = Rnnn  +  #0b0iii
0001 111i iinn nddd    SUBS  Rddd, Rnnn, #0biii  ; Rddd  = Rnnn  -  #0b0iii

0010 – 0x2xxx Instructions
0010 0ddd iiii iiii    MOVS  Rddd, #0biiiiiiii   ; Rddd  =          #0b0iiiiiiii
0010 1nnn iiii iiii    CMP   Rnnn, #0biiiiiiii   ; flags = Rnnn  -  #0b0iiiiiiii

0011 – 0x3xxx Instructions
0011 0ddd iiii iiii    ADDS  Rddd, #0biiiiiiii   ; Rddd  = Rddd  +  #0b0iiiiiiii
0011 1ddd iiii iiii    SUBS  Rddd, #0biiiiiiii   ; Rddd  = Rddd  -  #0b0iiiiiiii

0100 – 0x4xxx Instructions
0100 0000 00mm mddd    ANDS  Rddd,  Rmmm         ; Rddd  = Rddd  &  Rmmm
0100 0000 01mm mddd    EORS  Rddd,  Rmmm         ; Rddd  = Rddd  ^  Rmmm
0100 0000 10mm mddd    LSLS  Rddd,  Rmmm         ; Rddd  = Rddd LSL Rmmm
0100 0000 11mm mddd    LSRS  Rddd,  Rmmm         ; Rddd  = Rddd LSR Rmmm
0100 0001 00mm mddd    ASRS  Rddd,  Rmmm         ; Rddd  = Rddd ASR Rmmm
0100 0001 01mm mddd    ADCS  Rddd,  Rmmm         ; Rddd  = Rddd  +  Rmmm +  carry
0100 0001 10mm mddd    SBCS  Rddd,  Rmmm         ; Rddd  = Rddd  -  Rmmm - ~carry
0100 0001 11mm mddd    RORS  Rddd,  Rmmm         ; Rddd  = Rddd ROR Rmmm
0100 0010 00mm mddd    TST   Rddd,  Rmmm         ; flags : Rddd  &  Rmmm
0100 0010 01mm mddd    RSBS  Rddd,  Rmmm, #0     ; Rddd  = 0     -  Rmmm   --> alias for NEGS Rddd, Rmmm
0100 0010 10mm mnnn    CMP   Rnnn,  Rmmm         ; flags : Rnnn  -  Rmmm
0100 0010 11mm mnnn    CMN   Rnnn,  Rmmm         ; flags : Rnnn  +  Rmmm
0100 0011 00mm mddd    ORRS  Rddd,  Rmmm         ; Rddd  = Rddd  |  Rmmm
0100 0011 01mm mddd    MULS  Rddd,  Rmmm, Rddd   ; Rddd  = Rddd  *  Rmmm
0100 0011 10mm mddd    BICS  Rddd,  Rmmm         ; Rddd  = Rddd  & ~Rmmm   --> bit clear
0100 0011 11mm mddd    MVNS  Rddd,  Rmmm         ; Rddd  =        ~Rmmm
0100 0100 dmmm mddd    ADD   Rdddd, Rmmmm        ; Rdddd = Rdddd + Rmmmm
0100 0101 nmmm mnnn    CMP   Rnnnn, Rmmmm        ; flags : Rnnnn - Rmmmm
0100 0110 dmmm mddd    MOV   Rdddd, Rmmmm        ; Rdddd =         Rmmmm
0100 0111 0mmm m...    BX    Rmmmm               ;           PC=Rmmmm  (mmmm==0b1111: unpredictable)
0100 0111 1mmm m...    BLX   Rmmmm               ; LR = IPC+2,PC=Rmmmm  (mmmm==0b1111: unpredictable)
0100 1ttt iiii iiii    LDR   Rttt, [PC, #off]    ; Rttt = [((IPC+4)&~0b011)+0b0iiiiiiii00] --> +1020 max
                       LDR   Rttt, label         ; --> the assembler calculates the above from the label
                       LDR   Rttt, =lab          ; --> pseudo instruction: the assembler stores the lab/lit
                       LDR   Rttt, =lit          ;     in litpool, access PC relative with LDR Rttt,litpool

0101 – 0x5xxx Instructions
0101 000m mnnn nttt    STR   Rttt, [Rnnn, Rmmm]  ; [Rnnn + Rmmm] = Rttt
0101 001m mnnn nttt    STRH  Rttt, [Rnnn, Rmmm]  ; [Rnnn + Rmmm] = Rttt                --> low half
0101 010m mnnn nttt    STRB  Rttt, [Rnnn, Rmmm]  ; [Rnnn + Rmmm] = Rttt                --> low byte
0101 011m mnnn nttt    LDRSB Rttt, [Rnnn, Rmmm]  ; Rttt<sss1> = [Rnnn + Rmmm]<1>       --> low byte
0101 100m mnnn nttt    LDR   Rttt, [Rnnn, Rmmm]  ; Rttt = [Rnnn + Rmmm]
0101 101m mnnn nttt    LDRH  Rttt, [Rnnn, Rmmm]  ; Rttt<0021> = [Rnnn + Rmmm]<21>      --> low half
0101 110m mnnn nttt    LDRB  Rttt, [Rnnn, Rmmm]  ; Rttt<0001> = [Rnnn + Rmmm]<1>       --> low byte
0101 111m mnnn nttt    LDRSH Rttt, [Rnnn, Rmmm]  ; Rttt<ss21> = [Rnnn + Rmmm]<21>      --> low half

0110 – 0x6xxx Instructions
0110 0iii iinn nttt    STR   Rttt, [Rnnn, #off]  ; [Rnnn + 0b0iiiii00] = Rttt          --> +124 max
0110 1iii iinn nttt    LDR   Rttt, [Rnnn, #off]  ; Rttt = [Rnnn + 0x0iiiii00]          --> +124 max

0111 – 0x7xxx Instructions
0111 0iii iinn nttt    STRB  Rttt, [Rnnn, #off]  ; [Rnnn + 0b0iiiii] = Rttt            --> +31 max, low byte
0111 1iii iinn nttt    LDRB  Rttt, [Rnnn, #off]  ; Rttt<0001> = [Rnnn + 0x0iiiii]<1>   --> +31 max, low byte

1000 – 0x8xxx Instructions
1000 0iii iinn nttt    STRH  Rttt, [Rnnn, #off]  ; [Rnnn + 0x0iiiii0] = Rttt           --> +62 max, low half
1000 1iii iinn nttt    LDRH  Rttt, [Rnnn, #off]  ; Rttt<0021> = [Rnnn + 0x0iiiii0]<21> --> +62 max, low half

1001 – 0x9xxx Instructions
1001 0ttt iiii iiii    STR   Rttt, [SP, #off]    ; [SP + 0b0iiiiiiii00] = Rttt          --> +1020 max
1001 1ttt iiii iiii    LDR   Rttt, [SP, #off]    ; Rttt = [SP + 0b0iiiiiiii00]          --> +1020 max

1010 – 0xAxxx Instructions
1010 0ddd iiii iiii    ADR   Rddd, label         ; Rddd = ((IPC+4)&~0b011)+0b0iiiiiiii00   --> +1020 max
1010 1ddd iiii iiii    ADD   Rddd, SP, #off      ; Rddd = SP + 0b0iiiiiiii00              --> +1020 max

1011 – 0xBxxx Instructions
1011 0000 0iii iiii    ADD   SP, SP, #off        ; SP = SP + 0b0iiiiiii00                 --> +508 max
1011 0000 1iii iiii    SUB   SP, SP, #off        ; SP = SP - 0b0iiiiiii00                 --> +508 max
1011 00i1 iiii innn    CBZ   Rnnn, label         ; if Rnnn==zero, PC = IPC+4 + 0x0iiiiii0  --> +126 max
1011 0010 00mm mddd    SXTH Rddd, Rmmm           ; Rddd<ss21> = Rmmm<4321> --> low half
1011 0010 01mm mddd    SXTB Rddd, Rmmm           ; Rddd<sss1> = Rmmm<4321> --> low byte
1011 0010 10mm mddd    UXTH Rddd, Rmmm           ; Rddd<0021> = Rmmm<4321> --> low half
1011 0010 11mm mddd    UXTB Rddd, Rmmm           ; Rddd<0001> = Rmmm<4321> --> low byte
1011 0100 rrrr rrrr    PUSH {reg0-7}             ; rrrrrrrr = Lo reg-mask --> pushes regs to SP (decrements SP)
1011 0101 rrrr rrrr    PUSH {LR,reg0-7]          ; rrrrrrrr = Lo reg-mask --> pushes regs to SP (decrements SP)
1011 0110 0100 xxxx    -                         ; unpredictable
1011 0110 0101 0...    SETEND LE                 ; sets little-endian mode in CPSR
1011 0110 0101 1...    SETEND BE                 ; sets big-endian mode in CPSR
1011 0110 0110 0aif    CPSIE aif                 ; Enable Processor State  --> a=imprecise-abort, i=IRQ, f=FIQ
1011 0110 0111 0aif    CPSID aif                 ; Disable Processor State --> a=imprecise-abort, i=IRQ, f=FIQ
1011 0110 011x 1xxx    -                         ; unpredictable
1011 10i1 iiii innn    CBNZ Rnnn, label          ; if Rnnn!=zero, PC = IPC+4 + 0x0iiiii0   --> + 126 max
1011 1010 00mm mddd    REV   Rddd, Rmmm          ; Rddd<4321> = Rmmm<1234> --> reverse all
1011 1010 01mm mddd    REV16 Rddd, Rmmm          ; Rddd<4321> = Rmmm<3412> --> reverse low half, rev. upper half
1011 1010 10xx xxxx    -                         ; undefined
1011 1010 11mm mddd    REVSH Rddd, Rmmm          ; Rddd<4321> = Rmmm<ss12> --> reverse low half, sign extended
1011 1100 rrrr rrrr    POP  {reg0-7}             ; rrrrrrrr = Lo reg-mask  --> pops regs from SP (increments SP)
1011 1101 rrrr rrrr    POP  {PC,reg0-7]          ; rrrrrrrr = Lo reg-mask  --> pops regs from SP (increments SP)
1011 1110 iiii iiii    BKPT #0biiiiiiii          ; breakpoint, arg ignored by HW
1011 1111 0000 0000    NOP                       ; do nothing
1011 1111 0001 0000    YIELD                     ; do nothing, NOP-Hint: signal to HW to suspend/resume threads
1011 1111 0010 0000    WFE                       ; do nothing, NOP-Hint, wait for event
1011 1111 0011 0000    WFI                       ; do nothing, NOP-Hint: wait for interrupt
1011 1111 0100 0000    SEV                       ; do nothing, NOP-Hint: signal event to multi-processor system
1011 1111 cccc mmmm    ITsel cond                ; if-then: sel=mmmm: T=then/E=else, cond=cccc: as for Bcc<11:8>

1100 – 0xCxxx Instructions
1100 0nnn rrrr rrrr    STMIA Rnnn! {reg0-7}      ; rrrrrrrr = Lo reg-mask, inc Rnnn
1100 1nnn rrrr rrrr    LDMIA Rnnn! {reg0-7}      ; rrrrrrrr = Lo reg-mask, inc Rnnn if Rnnn not in mask
                       LDMIA Rnnn  {reg0-7}      ; rrrrrrrr = Lo reg-mask, load Rnnn if Rnnn in mask

1101 – 0xDxxx Instructions
1101 0000 iiii iiii    BEQ label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0001 iiii iiii    BNE label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0010 iiii iiii    BHS/BCS label             ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0011 iiii iiii    BLO/BCC label             ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0100 iiii iiii    BPL label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0101 iiii iiii    BMI label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0110 iiii iiii    BVS label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 0111 iiii iiii    BVC label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1000 iiii iiii    BHI label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1001 iiii iiii    BLS label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1010 iiii iiii    BGE label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1011 iiii iiii    BLT label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1100 iiii iiii    BGT label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1101 iiii iiii    BLE label                 ; if true, PC = IPC+4 + 0biiiiiiii0 --> -256/+254 max
1101 1110 xxxx xxxx    -                         ; undefined --> can be used for instruction emulation
1101 1111 iiii iiii    SVC #0biiiiiiii           ; supervisor call (formerly called SWI), arg ignored by HW

1110 – 0xExxx Instructions
1110 0iii iiii iiii    B label                   ; PC = IPC+4 + 0biiiiiiiiiii0       --> -2048/+2046 max
1110 1xxx xxxx xxxx    -                         ; 32 bit instructions

1111 – 0xFxxx Instructions
1111 0xii iiii iiii 11y1 ziii iiii iiii BL  label  ; LR=IPC+4,PC=IPC+4+0bXYZii...ii0,X,Y,Z=f(x,y,z), +/-16M
1111 0011 1110 1111 1000 dddd ssss ssss MRS Rdddd,S; Rdddd = special register S (encoded as 0bssssssss)
1111 0011 1000 mmmm 1000 1000 ssss ssss MSR S,Rmmmm; special register S (encoded as 0bssssssss) = Rmmmm
1111 0011 1011 1111 1000 1111 0100 1111 DSB       ; data synchronization barrier
1111 0011 1011 1111 1000 1111 0101 1111 DMB       ; data memory barrier
1111 0011 1011 1111 1000 1111 0110 1111 ISB       ; instruction synchronization barrier
1111 xxxx xxxx xxxx xxxx xxxx xxxx xxxx -          ; other 32 bit instructions
```

1) IPC is the PC of the current instruction (IPC+4 is given by the pipeline, IPC+2/+4 is the return address in the LR)
2) a dot means don't care, but must be set to 0.
3) <4321>: word, <21>: low half word, <1>: low byte, <0001>: zero extend byte, <sss1>: sign extend byte, etc.
4) Undefined instructions can be used to emulate instructions (they trigger the undefined exception).
5) Unpredictable instructions do any unpredictable actions and are therefore illegal instructions.
6) Unallocated codes are undefined unless they are explicitly marked as unpredictable.
7) CBZ, CBNZ, IT are the only 16 bit instructions which are not part of Cortex-M0/M1 Thumb code.
8) BL, DMB, DSB, ISB, MRS, MSR are the only 32 bit instructions as part of the Cortex-M0/M1 instruction set.