

# Summary Micro-Controller FS18

Alex Neher

July 3, 2018



# Contents

<b>1</b>	<b>Microcontroller</b>	<b>1</b>
1.1	Components . . . . .	1
1.2	Numerical systems . . . . .	2
1.2.1	Example . . . . .	2
1.2.2	Two's Complement . . . . .	2
1.3	Logic Gates . . . . .	2
1.4	Instruction Set Cycle . . . . .	2
1.5	Assembler . . . . .	4
1.5.1	Directives . . . . .	4
1.5.2	Addressing modes . . . . .	4
1.5.3	Assembler Programming Operations . . . . .	7
1.5.3.1	Data Transport . . . . .	7
1.5.3.2	Arithmetic Operations . . . . .	8
1.5.3.3	Logic and Bitmasking Operations . . . . .	8
1.5.3.4	Shift- and Rotation Operations . . . . .	9
1.5.3.5	Branching . . . . .	10
1.5.3.6	Comparing . . . . .	10
1.5.3.7	Flags . . . . .	11
1.6	Stack . . . . .	12
1.7	Subroutines . . . . .	13
1.8	Timer . . . . .	14
1.9	Interrupts . . . . .	15
<b>2</b>	<b>C</b>	<b>17</b>



# Chapter 1

## Microcontroller

### 1.1 Components

Micro controllers are so called **Single Chip** Computer, meaning everything is on a single PCB, as opposed to e.g. a 'normal' PC.

MC consist of at least four components:

**CPU:** Central Processing Unit

**Memory:** Where programs and data are stored

**IO/Input-Output:** Communication with Peripherals

**Bus-System:** Connects the components

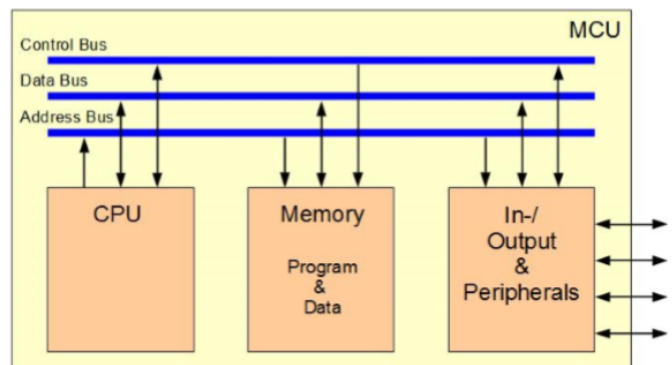


Fig.. 1.1.1: Von-Neumann Architecture

There are two different architectures:

**Von-Neumann:** One shared bus for program and data. Program and data are in the same memory. Often found in low-cost MCs

**Harvard:** Two separate bus systems for program and data. Often found in high-performance MCs

Usually, a read/write-operation goes through four steps:

1. CPU puts the address on the address bus
2. Either the memory or the IO claim the address as their
3. CPU tells the component via the control bus whether the operation is read or write
4. *read:* The memory or IO places the data of the requested address on the data bus  
*write:* The CPU writes the data on the mentioned address via data bus

## 1.2 Numerical systems

In MC, variables and constants are seldom stored as decimal value. Rather they're either stored as a binary or a hexadecimal value.

In general, mathematical terms an  $n$ -digit integer to the base  $B$  can be expressed as:

$$\sum_{i=0}^{n-1} x_i * B^i = X_0 * B^0 + x_1 * B^2 + [...] + x_{n-1} * B^{n-1}$$

Or easier: *Multiply the  $n$ -th digit of the integer with  $B^{n-1}$  starting from the right with  $n = 0$*

### 1.2.1 Example

1100'0101<sub>2</sub> (binary) to decimal

$$= 1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 0 * 2^3 + 0 * 2^4 + 0 * 2^5 + 1 * 2^6 + 1 * 2^7$$

$$= 1 + 0 + 4 + 0 + 0 + 0 + 64 + 128$$

$$= \underline{\underline{197_{10}}}$$

If you have to convert a number between two 'exotic' systems, say base 8 to base 3, it's usually easier to convert it to decimal first and then convert it to the desired system again ( $x_8 \rightarrow x_{10} \rightarrow x_3$ ). An exception to that is binary to hexadecimal and vice versa. One digit in hexadecimal represents four digits in binary, so you can directly convert blocks of four:

1100'0101<sub>2</sub> to hexadecimal:

$$1100 = 0 * 2^0 + 0 * 2^1 + 1 * 2^2 + 1 * 2^3 = 0 + 0 + 4 + 8 = 12_{10} = C_{16}$$

$$0101 = 1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 0 * 2^3 = 1 + 0 + 4 + 0 = 5_{10} = 5_{16}$$

$$\rightarrow 1100'0101_2 = C5_{16}$$

### 1.2.2 Two's Complement

Especially in MC-technology, signed numbers (that can also be negative) are mostly stored as *two's complement*. You basically take the binary number, invert every digit and add one. So -28 would be stored as

$$28_{10} = 16 + 8 + 4 = 2^2 + 2^3 + 2^4 = 0001'1100_2$$

invert

$$0001'1100 \rightarrow 1110'0011$$

add one

$$1110'0011 + 1 = \underline{\underline{1110'0100}}$$

## 1.3 Logic Gates

**Multiplexer:** The multiplexer is a combinational logic circuit, which allows us to select one of many input lines and route it to the single, common output line. The demultiplier does the exact opposite: it takes one input and you can select to which output line it is routed

**FlipFlop:** Idunno

## 1.4 Instruction Set Cycle

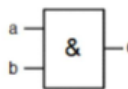
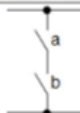

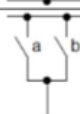

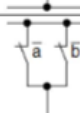

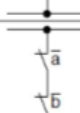
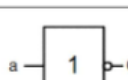
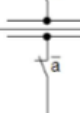
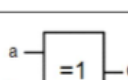
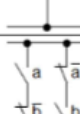

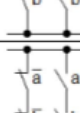
Logic Gates																			
Function	Symbol	Equation	Table	Switch Setup															
AND		$Q = a \cdot b$	<table><tr><th>b</th><th>a</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	b	a	Q	0	0	0	0	1	0	1	0	0	1	1	1	
b	a	Q																	
0	0	0																	
0	1	0																	
1	0	0																	
1	1	1																	
OR		$Q = a + b$	<table><tr><th>b</th><th>a</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	b	a	Q	0	0	0	0	1	1	1	0	1	1	1	1	
b	a	Q																	
0	0	0																	
0	1	1																	
1	0	1																	
1	1	1																	
NAND		$Q = \overline{a \cdot b}$	<table><tr><th>b</th><th>a</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	b	a	Q	0	0	1	0	1	1	1	0	1	1	1	0	
b	a	Q																	
0	0	1																	
0	1	1																	
1	0	1																	
1	1	0																	
NOR		$Q = \overline{a + b}$	<table><tr><th>b</th><th>a</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	b	a	Q	0	0	1	0	1	0	1	0	0	1	1	0	
b	a	Q																	
0	0	1																	
0	1	0																	
1	0	0																	
1	1	0																	
NOT		$Q = \overline{a}$	<table><tr><th>a</th><th>Q</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	a	Q	0	1	1	0										
a	Q																		
0	1																		
1	0																		
XOR		$Q = \overline{a} \cdot b + a \cdot \overline{b}$ $Q = a \oplus b$	<table><tr><th>b</th><th>a</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	b	a	Q	0	0	0	0	1	1	1	0	1	1	1	0	
b	a	Q																	
0	0	0																	
0	1	1																	
1	0	1																	
1	1	0																	
XNOR		$Q = a \cdot b + \overline{a} \cdot \overline{b}$ $Q = \overline{a \oplus b}$	<table><tr><th>b</th><th>a</th><th>Q</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	b	a	Q	0	0	1	0	1	0	1	0	0	1	1	1	
b	a	Q																	
0	0	1																	
0	1	0																	
1	0	0																	
1	1	1																	

Fig.. 1.3.1: Fundamental logic-Gates used in MCs

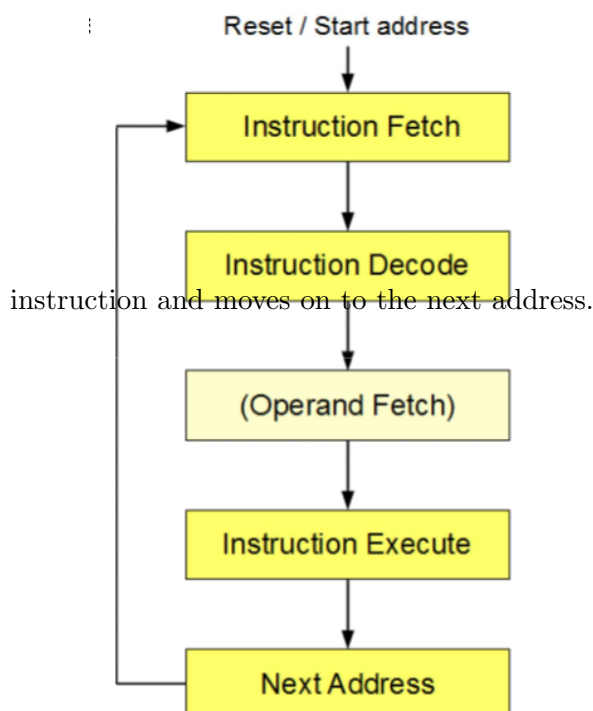


Fig.. 1.4.1: Visualisation of the Instruction Set Cycle

.5The way a CPU executes instructions can be shortened to **FDE**. It stands for **F**etch, **D**ecode **E**xecute. S As can be seen in fig.

1.4.1, the CPU first fetches the instruction from the memory, then it decodes it and decides if it has to fetch a second operand (e.g. for an addition). Afterwards it executes said

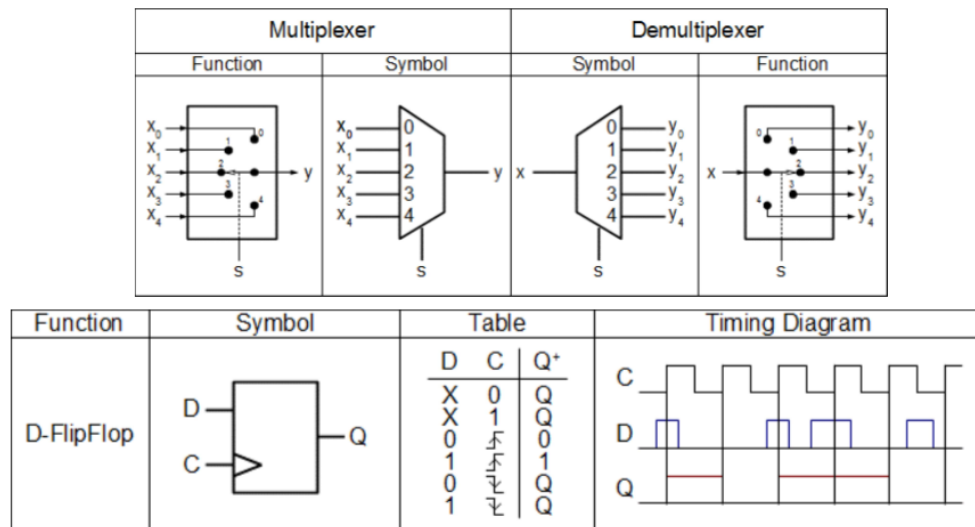


Fig.. 1.3.2: Visualisation of the (de)multiplexer and the flipflop

## 1.5 Assembler

### 1.5.1 Directives

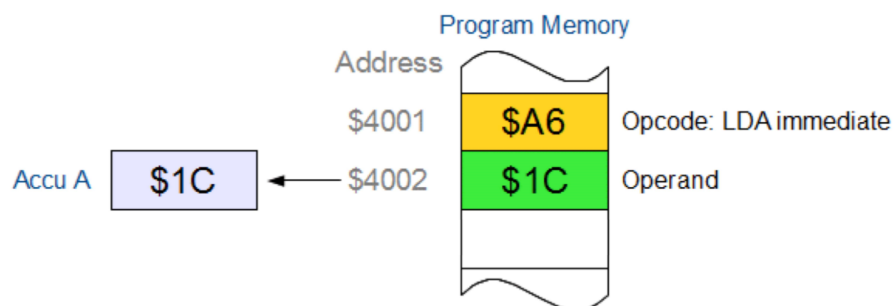
Put simply, Assembler directives are instructions that direct the assembler to do something:

Directive	Description	Example	Explanation
SECTION	Defines the beginning of a re-locatable section	ConstSec: SECTION	Puts the whole ConstSec-Section in the same RAM-section
EQU	Assign an expression to a name. Not redefinable	MaxElement: EQU 20	search-and-replace every <i>MaxElement</i> in the code with 20
DC	Defines one or more constants and theirnames	DC B \$AA	Pointer to address \$AA at every <i>Alarm</i>
DS	Allocates memory to variables	DS W 3	Reserves 3 words of RAM. 1 Word = 2 Bytes

### 1.5.2 Addressing modes

There are six different addressing modes that are supported by our CPU:

**Immediate:** 1 Byte operand in the instruction

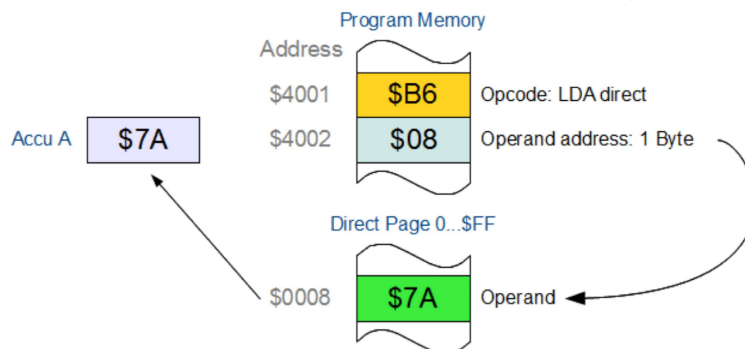




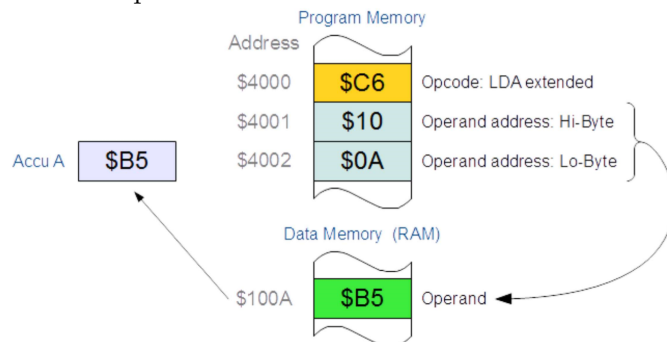
**Inherent:** No operand required



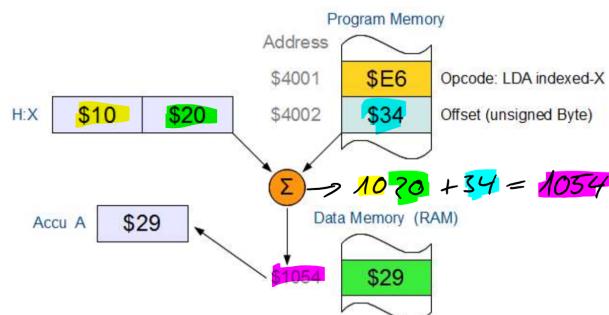
**Direct:** Operands must be stored in the Direct Page (From 0x0000 to 0x00AF)



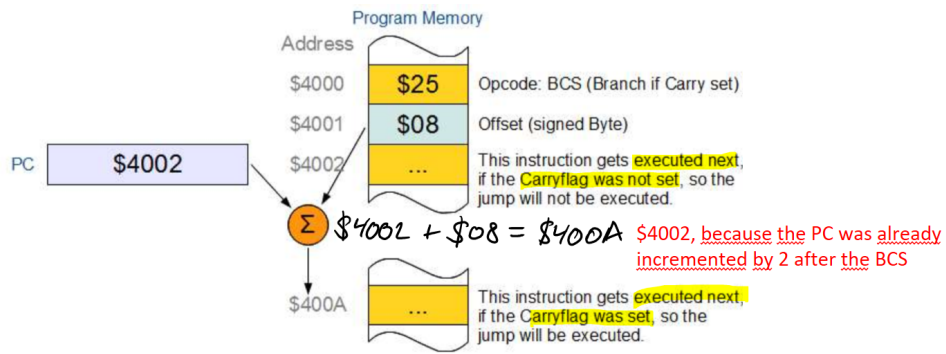
**Extended:** Operand can be stored in the whole 64k Memory



**Indexed:** Operand is stored in Stack Pointer or H:X Register



**Relative:** Only used with BRANCH-Instructions. Depending on outcome of the Branch



### 1.5.3 Assembler Programming Operations

There are three main assembler instruction types:

- Data Transport
  - Load
  - Store
  - Transfer
  - Move
- Operations
  - Arithmetic
  - Logic
  - Bit-Manipulation/Masking
  - Shift and Rotation
- Branching

#### 1.5.3.1 Data Transport

Data Transport instructions are again divided into four subtypes

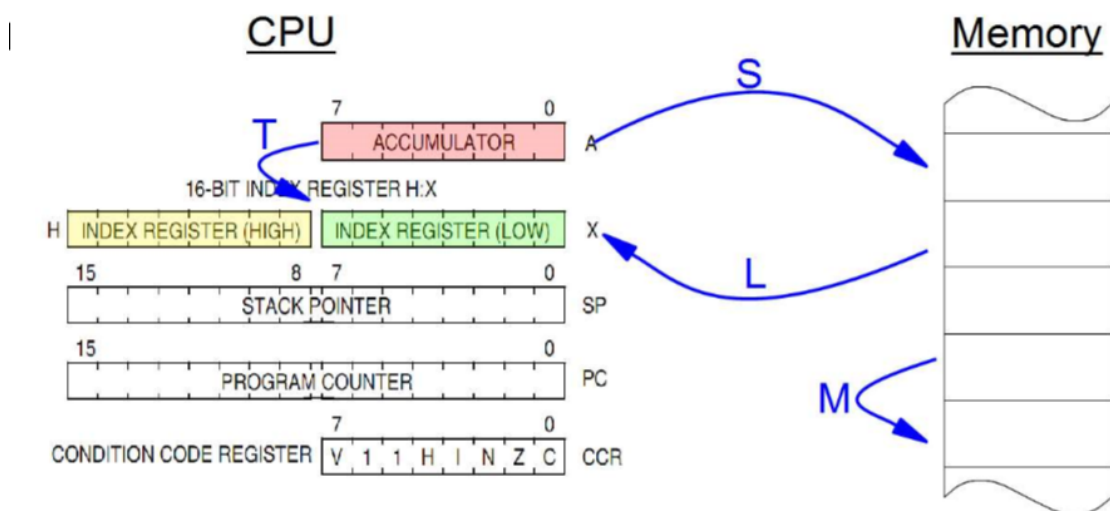


Fig.. 1.5.1: Visualisation of the different data tranfers

**Load:** Data is moved from the memory to the CPU-Register  
Examples: LDA, LDX, LDHX, (PULA, PULX Stack-operations)

**Store:** Data is moved from the CPU-Register to the memory  
Examples: STA, STX, STHX, (PSHA, PSHA Stack-operations)

**Transfer:** Very fast Data Transfer between CPU registers Examples: TAP, TPA, TAX, TSX

**Move:** Data is moved within the memory

### 1.5.3.2 Arithmetic Operations

Arithmetic operations cover the four basic mathematical operations:

**Addition:** ADD, ADC (Addition with Carry Bit → supports numbers > 8bit)

**Subtraction:** SUB, SBC (Addition with Carry Bit → supports numbers > 8bit)

**Multiplication:** MUL Only works with unsigned numbers. Multiplies the content of the accumulator with the content of the X-register and stores the 16bit result in the X:A-registers (LSB in A, MSB in X)

**Division:** DIV Only works with unsigned numbers. Divides whatever is in the H:A-register through whatever is in the X-register. The 8bit result is stored in the accumulator. In case of an overflow or division by 0, the carry bit will be set.

### 1.5.3.3 Logic and Bitmasking Operations

Bitwise Operations change but a single bit of the operand. This can be especially useful e.g. to turn on or off LEDs, which are controlled by a register of 8 bits

ORA#(B6|B3)

Logical OR Operation on Bit 6 and Bit 3 → Sets **only** Bit 6 and Bit 3 in the Accumulator

AND#(B6|B3)

Logical AND Operation on Bit 6 and Bit 3 → Deletes all Bits **except** Bit 6 and Bit 3

BCLR n, Addr

Deletes Bit n on a specific memory address

BSET n, Addr

Sets Bit n on a specific memory address

BIT Addr

AND Operation of the accumulator with the content of the address, *without changing the content of either of them*

CLC

Clears Carry-Flag

SEC

Sets Carry-Flag

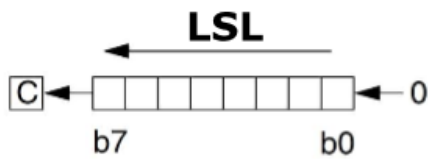
CLI

Delete Interrupt-Mask Bit (→ Interrupt **enable**)

SEI

Sets Interrupt-Mask Bit (→ Interrupt **disable**)

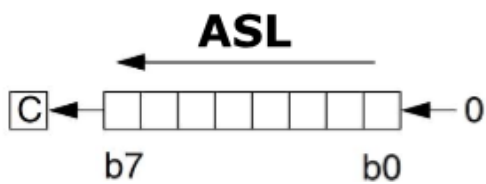
### 1.5.3.4 Shift- and Rotation Operations



(a) Logical Left-Shift



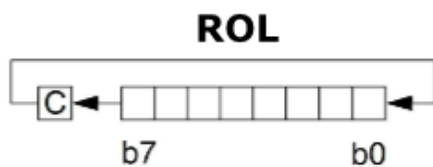
(b) Logical Right-Shift



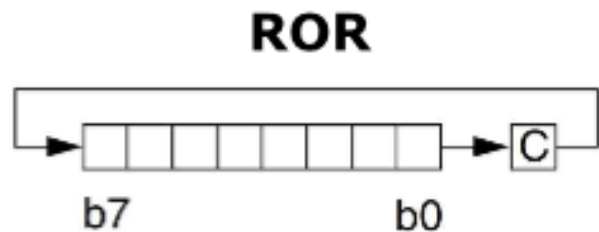
(c) Arithmetic Left Shift



(d) Arithmetic Right-Shift



(e) Rotate Left



(f) Rotate Right

The Left-Shifts are an equivalent to a multiplication by two and Right-Shifts are the equivalent of a division by two:

$0011'0110_2 = 54_{10} \rightarrow LSL \rightarrow 0110'1100_2 = 108_{10}$   
 $0011'0110_2 = 54_{10} \rightarrow LSR \rightarrow 0001'1011_2 = 27_{10}$

### 1.5.3.5 Branching

Oper.	Test	Meaning
BEQ	Z = 1	Branch if equal
BNE	Z = 0	Branch if not equal
BCS	C = 1	Branch if Carry
BCC	C = 0	Branch if no Carry
BMI	N = 1	Branch if Negative
BPL	N = 0	Branch if not Negative
BGT	>	Branch if bigger (signed)
GHI		Branch if bigger (unsigned)
BGE	≥	Branch if bigger/equal (signed)
BHS/BCC		Branch if bigger/equal (unsigned)
BLE	≤	Branch if lesser/equal (signed)
BLS		Branch if lesser/equal (unsigned)
BLT	<	Branch if lesser (signed)
BLO/BLS		Branch if lesser (unsigned)
BEQ	=	Branch if equal
BRA		Branch always (Overriding SP and PC)
BRN		Branch never (Mainly used for debugging, just waits)
BSR		Brnch to subroutine (Saves SP and PC to return to main-routine)

These Branches depend on a single Bit of a memory address in the **direct pages**(0x00 - 0xFF)

Oper.	Meaning
BRCLR n, Addr, Label	Branch to Label if Bit n at Address Addr is not set
BRSET n, Addr, Label	Branch to Label if Bit n at Address Addr is set

### 1.5.3.6 Comparing

Comparing operations compare two values (kinda obvious innit?) and set the flags accordingly, but don't change the compared values:

Oper.	Meaning
CMP opr8	Compares content of accumulator with the 8-bit operand
CPX opr8	Compare content of X-register with 8-bit operand
CPHX opr16	Compare content of HX-register with 16-bit operand

### Example

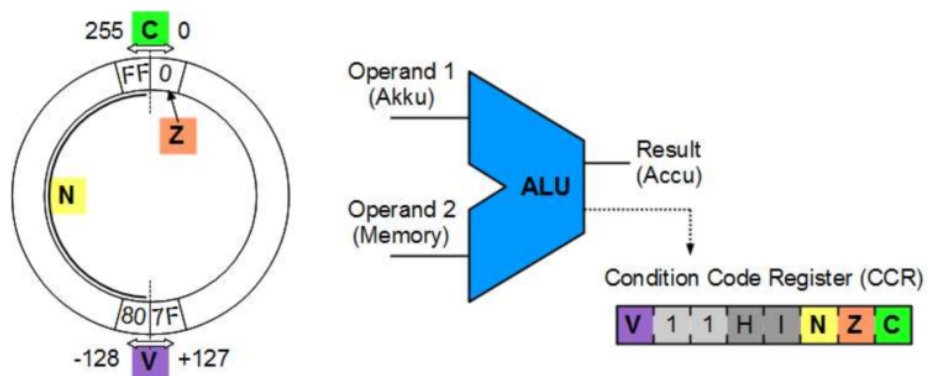
```

1 LDA Op1    //Load Op1 into accumulator
2 CMP Op2    //Compare it to Op2 and sets Zero/Carry/Negative Flag accordingly
3 BMI Label1 //Branch to Label if Negative flag is set (--> Op1 < Op2)

```

Listing 1.1: Test if a value is bigger or smaller and branch accordingly

### 1.5.3.7 Flags



CC	Name	Condition	Relevant for	
Z	Zero	Result = 0	unsigned	signed
N	Negative	Result < 0		signed
C	Carry	0 > Result > 255	unsigned	
V	Overflow	-128 > Result > 127		signed

Fig.. 1.5.3: Visualization of all supported flags

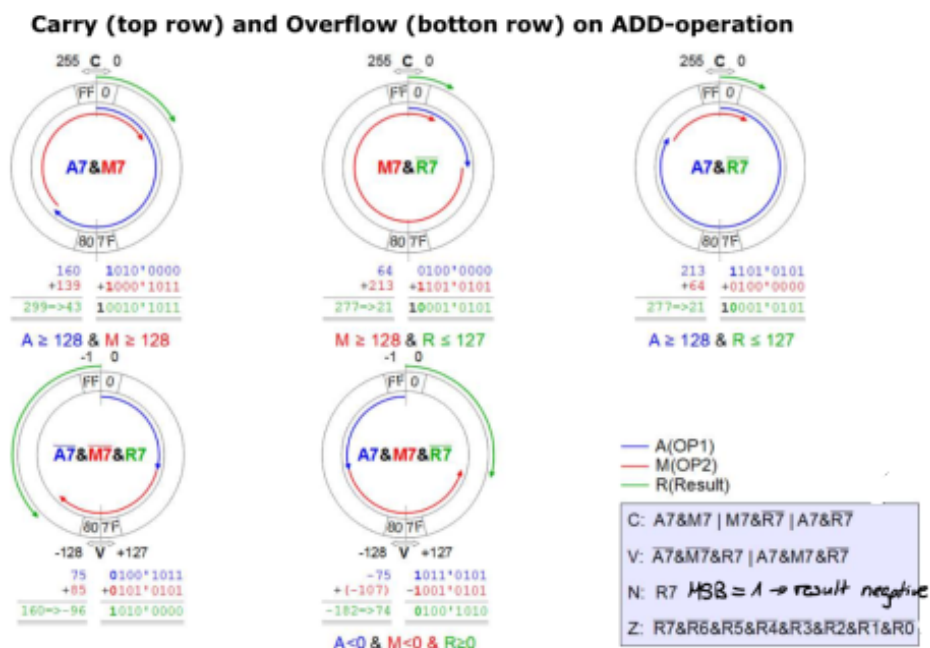


Fig.. 1.5.4: Visualization of carry- and overflow-flags on Additions

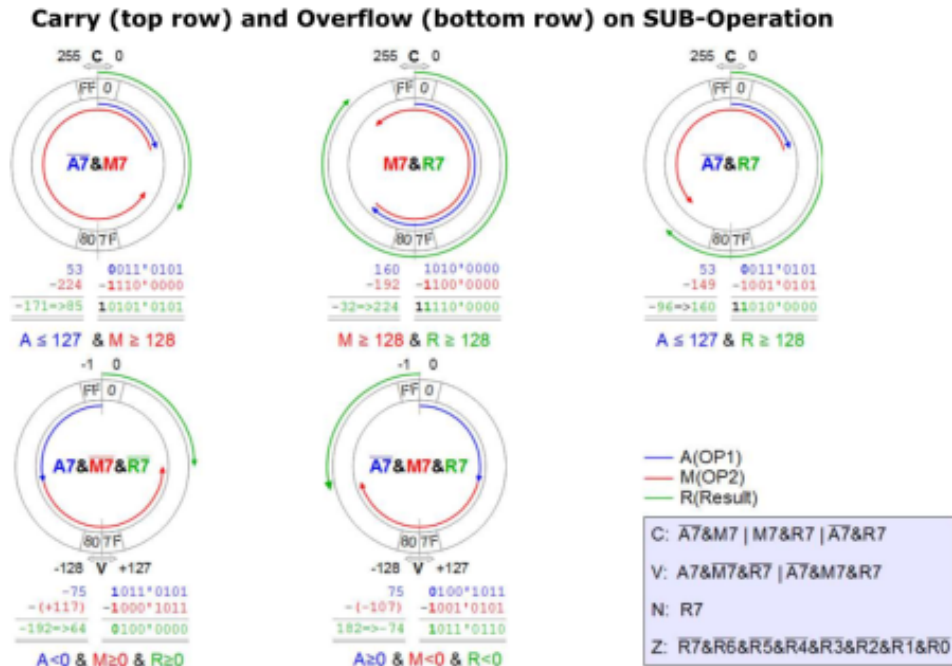


Fig.. 1.5.5: Visualization of carry- and overflow-flags on Subtractions

## 1.6 Stack

The stack is a so-called LIFO-memory (Last In First Out), meaning whatever data is chunked on the stack last, has to be removed, before the data underneath it can be accessed.

The Stackpointer (SP) is a pointer that always points at the next free memory place on the stack. As soon as something is pushed onto the stack (and the address the SP pointed to is therefore no longer free), it gets updated automatically. Same goes after a pull. The addresses are counted from the top of the stack, therefore the higher the stack gets, the lower the addresses.

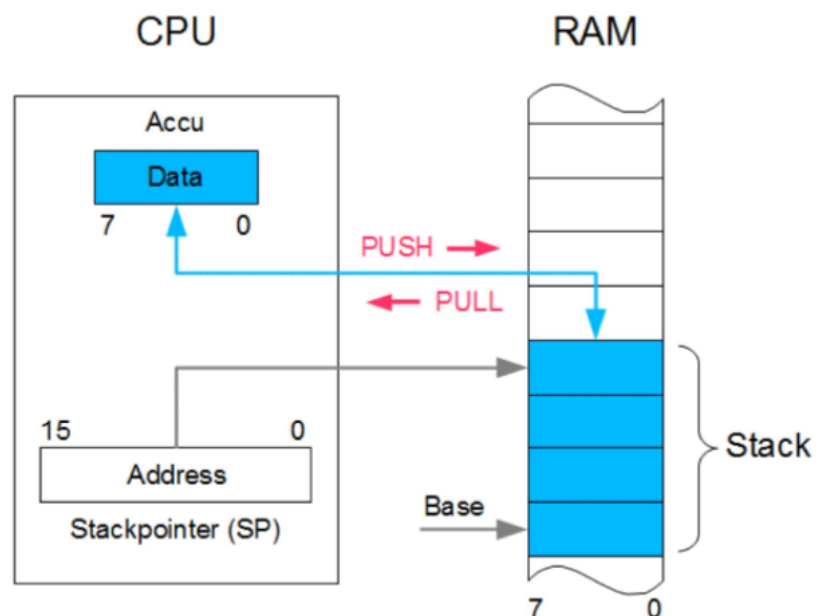


Fig.. 1.6.1: Stack



```

1 Stacksize: EQU $40      //Set the stacksize to hex 40 = 64 dec
2
3 DATA:      SECTION      //Start of DATA Section
4 TofStack:   DS Stacksize-1 //Reserve memory for the stack
5 BofStack:   DS 1
6
7 PROGRAM:    SECTION
8             //Initializing SP(+1, bc SP always points to the first free address)
9             LDHX #(BofStack+1) //LDHX = Load value into H:X-Register
10            TXS           //Transfer SP to H:X Register
11
12            PSHA //Push accumulator to Stack
13            PSHX //Push X-register to Stack
14
15            //Important! Reverse order bc of LIFO
16            PULX //Pull X-Register from Stack
17            PULA //Pull accumulator from Stack

```

Listing 1.2: Initialization of a stack

## 1.7 Subroutines

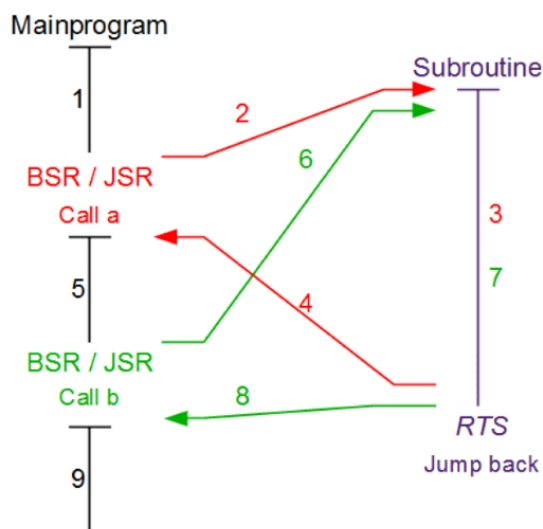


Fig.. 1.7.1: Process of a subroutine

Per definition, a subroutine is *a sequence of computer instructions for performing a specified task that can be used repeatedly*. So in other words, a subroutine is a **method**. A subroutine has its own scope, so e.g. variables defined in a subroutine won't be accessible from the main-program.

The process of a subroutine can be divided into five rough steps:

1. Save Program Counter to the Stack
2. Load the called subroutine
3. Execute the loaded subroutine
4. Get the Program Counter from the Stack
5. Return to Program Counter + 2  
(PC + 1 = BSR/JSR)

Of course, subroutines have both pros and cons, however, there are far more pros than cons:

## Pros

- Repeated command sequences are stored only once in memory  
→ Less memory usage
- Repeated command sequences are programmed and tested only once  
→ Less development effort
- Programs can be built in a modular way  
→ Lower risk of errors
- Programs can be developed by multiple persons at the same time  
→ Higher productivity
- Parts of the program can be compiled independent of each other  
→ Shorter compile time, libraries with standard functions

## Cons

- The call of the subroutine, parameter passing and the return jump need additional time and resources  
→ Slower program execution

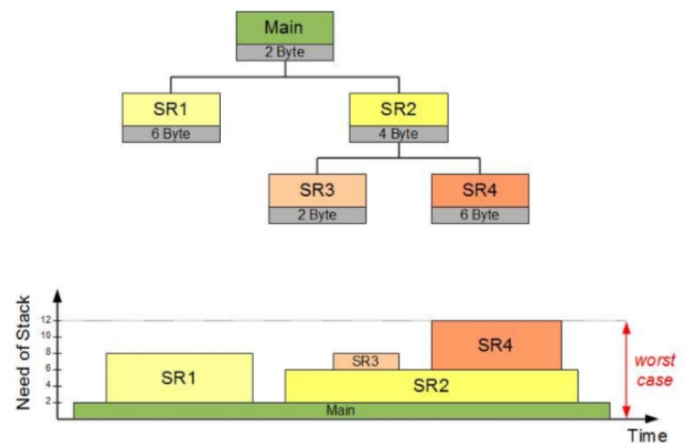
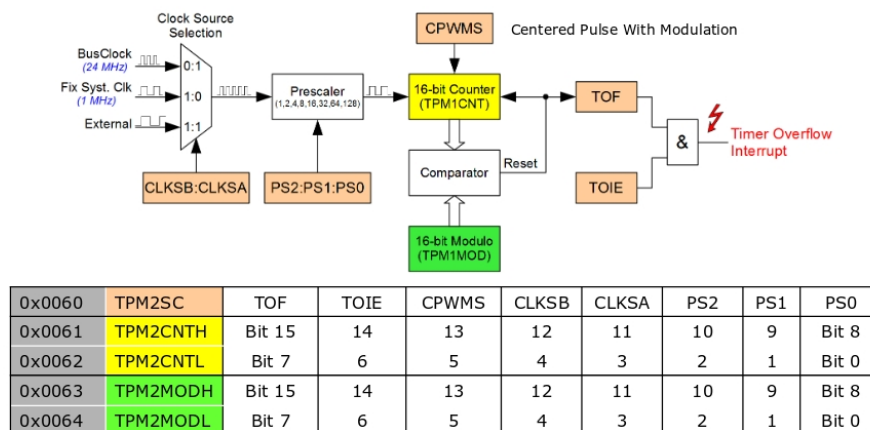


Fig.. 1.7.2: Stack consumption on subroutines

## 1.8 Timer



On given frequency of the chosen clock source, the time  $T_{TOF}$  (between two Timer-Overflow events) is calculated out of the configured Prescaler- and Modulo-values as following:

$$T_{TOF} = (MOD + 1) \cdot PS / f_{CLK}$$

Fig.. 1.8.1: Overview of the timer-system in the MC

In the top-right corner of fig. 1.8.1 a multiplexer can be seen. There, you can select from which clocks the ticks should be counted from:

0:0 Clock is off

1:0 Fix System Clock (Set to 1MHz)

0:1 Bus-Clock (24MHz in our case)

1:1 External Clock

Afterwards, the clock-signal pass through the **prescaler**. If you were to set a timer that counts every tick of a 24MHz clock, you would need to set the timer ridiculously high or it would be done within microseconds.

The prescaler can be set to 1, 2, 4, 8, 16, 32, 64 or 128 and filters the ticks. If you were to set the prescaler to e.g. 16, then it would only let every 16-th tick pass.

Every tick that passes the prescaler increases the 16-bit counter by one. This counter compares its new value with the value of the comparator, which itself gets its value from the 16-bit Modulo, which is set by the user. As soon as the values of the counter and the comparator match, it triggers a reset that resets the counter to 0 and triggers a TOF, a Time Overflow Flag. If the TOIE, the Time Overflow Interrup Enabler is enabled, the TOF triggers an interrupt, that can e.g. start a subroutine.

The time (in seconds) to trigger the TOF can be calculated with the formula

$$T_{TOF} = (Modulo + 1) * PrescalerValue * ClockFrequency$$

## 1.9 Interrupts

Interrupts are a MCs way of handling exceptions. An Interrupt can react automatically to events and interrupt the main-program to execute a subroutine. However, the state of the main-program has to be preserved, so that after the execution of the subroutine, the main-program can continue at the point where it was before the interrupt.

Another, less efficient method of exception handling would be **polling**, where a service keeps asking the main-program if a certain thing has already happened yet. If it did, the subroutine is executed, if it hasn't, the service will keep asking. That way, the time of the interrupt can be easily foreseen, but it takes a lot more resources to make a request every  $x$  milliseconds compared to just setting an interrupt.

As mentioned before, the state of the main-program needs to be saved before jumping into a subroutine. In contrast to a manual subroutine call with BSR/JSR, the CPU-state will be saved to the stack **automatically** (fig. 1.9.1). However, the H-register must be saved manually, if it should be preserved.

The CPU also needs a so-called **interrupt-vector**, which sits at a predefined address in the memory. Once an interrupt is triggered, the CPU jumps to the corresponding vector, which points at the start-address of the interrupt-subroutine, so the CPU knows where the subroutine starts.

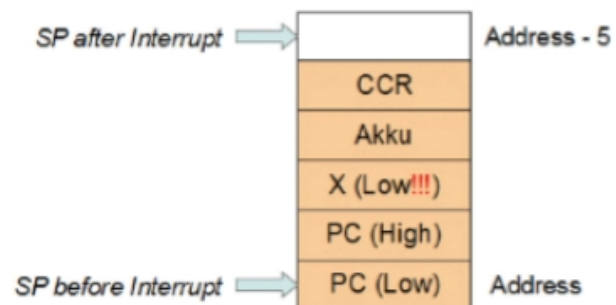


Fig.. 1.9.1: Automatic save of the CPU-state

By default, the MC doesn't support nested interrupts. So if the program is already in an interrupt-subroutine, it will not go into a second, 'deeper' subroutine, but will queue the second subroutine. Different interrupts have different priorities. The interrupt with the *lowest interrupt-vector number* has the *highest priority* and will therefore be executed first thing after the current subroutine is done.

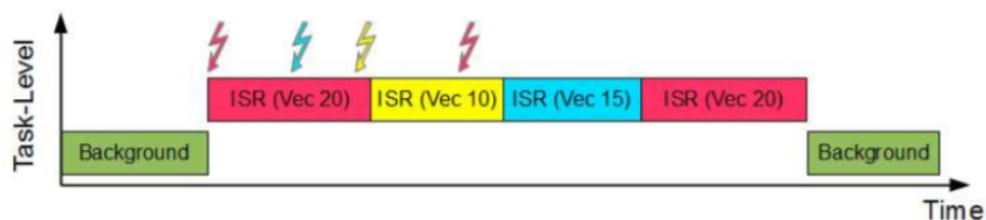


Fig.. 1.9.2: Queued interrupt subroutines with different priorities

## Programming Interrupts

While programming interrupts, some things have to be kept in mind:

- Interrupts rely on a **Stack** to preserve the main-program state
- Interrupt vectors have to be defined with the start-address of the ISR
- Once an interrupt is triggered/fired, its interrupt-flag has to be deleted, because else the interrupt would fire right again, causing an infinite interrupt-loop
- Save the H-Register before jumping to an ISR (if needed)
- Delete the interrupt flag in the ISR after execution, because else the interrupt will only fire exactly once
- Use CLI to release the interrupts globally

```
1  //.prm file for Interrupt Vectors
2  VECTOR ADDRESS 0xFFC4 ISR_RTI
3  VECTOR ADDRESS 0xFFC6 errISR_IIC
4  VECTOR ADDRESS 0xFFC8 errISR_ACOMP
5  VECTOR ADDRESS 0xFFCA errISR_ADC
6
7  //definition of the ISR
8  interrupt void ISR_RTI(void){
9      RTCSC_RTIF = 1;    //Clear interrupt flag
10 }
11
12 void main(void){
13     EnableInterrupts;
14     for(;;){
15         //Do stuff here
16     }
17 }
```

Listing 1.3: Programming of an interrupt

## Chapter 2

# C