

# Micro Controllers Summary

Lucien Zürcher

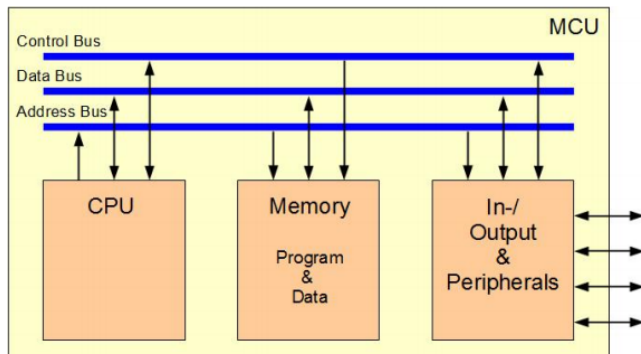
June 17, 2019

## Contents

<b>1 System Components</b>	<b>2</b>	<b>5.9 Direct relative Branching</b>	<b>8</b>
1.1 Von Neumann Architecture	2	<b>6 Subroutines &amp; Stack</b>	<b>9</b>
1.2 Harvard-Architecture	2	6.1 Stack	9
1.3 Numerical Systems	2	6.2 Subroutines	9
1.4 hex / binary	2	6.3 Stack size	9
1.5 Signed numbers	2	<b>7 Timer and Interrupts</b>	<b>9</b>
1.6 carry / overflow	2	7.1 Modulo Counter	9
1.7 Bit groups	2	7.2 Modulo Frequency	10
1.8 Quantity of address lines	3	7.3 Timer Control Registers	10
1.9 Microprocessor vs Mircocontroller	3	7.4 Polling and Interrupts	10
1.10 CPU components	3	7.5 Save Interrupt State	10
1.11 Instruction Cycle Steps	3		
1.12 Types of MCU Registers	3		
<b>2 Compiling</b>	<b>3</b>		
2.1 Codewarrior Designflow	3		
2.2 Programming Language	3		
2.3 Assembler Code-Format	4		
2.4 Parameter file	4		
<b>3 Assembler &amp; HCS08</b>	<b>4</b>		
3.1 HCS08 CPU Registers	4		
3.2 HCS08 Processor	4		
3.3 Memory Mapping	5		
3.4 Register configuration HCS08	5		
3.5 Differences of Operations	5		
<b>4 Assembler Directives &amp; Addressing Modes</b>	<b>5</b>		
4.1 Directives	5		
4.2 Basic Assembler Program	5		
4.3 Addressing Modes	6		
4.3.1 Immediate (IMM)	6		
4.3.2 Inherent (INH)	6		
4.3.3 Direct (DIR)	6		
4.3.4 Extended (EXT)	6		
4.3.5 Indexed (IX1)	7		
4.3.6 Relative (REL)	7		
<b>5 Assembler Addressing &amp; Programming</b>	<b>7</b>		
5.1 Assembler Instructions	7		
5.2 Transport Operations	7		
5.3 Arithmetic Operations	7		
5.4 Flags	8		
5.5 Logical Operations & Bit Masking	8		
5.6 Shift- and Rotation Operations	8		
5.7 Relative Branching	8		
5.8 Branching Compare-Operation	8		

## 1 System Components

### 1.1 Von Neumann Architecture



Components:

- **CPU**, Central Processing Unit
- **Memory**, Program and Data
- **In-/Output**-Unit, Peripherals
- **Bus-System**: Communication

One *shared bus and memory* for program and data.

### 1.2 Harvard-Architecture

basically same as Von Neumann, with the difference, that there are *two separate bus systems* for program and data

### 1.3 Numerical Systems

Numerical value  $Z_B$  of a  $n$ -digit, integer number with base  $B$  ( $B \geq 2$ ):

$$Z_B = \sum_{i=0}^{n-1} x_i \cdot B^i$$

Decimal	Dual / Binary	Hexadecimal
197	0b1100'0101	0xC5
$B = 10$	$B = 2$	$B = 16$
$= 1 \cdot 10^2 +$ $9 \cdot 10^1 +$ $7 \cdot 10^0$	$= 1 \cdot 2^7 + 1 \cdot 2^6 +$ $0 \cdot 2^5 + 0 \cdot 2^4 +$ $0 \cdot 2^3 + 1 \cdot 2^2 +$ $0 \cdot 2^1 + 1 \cdot 2^0$	$= C \cdot 16^1 + 5 \cdot 16^0$ $= 12 \cdot 16^1 + 5 \cdot 16^0$

The amount of presentable numbers is  $B^n$ . The highest presentable number is  $B^n - 1$ . Calculated from  $x_i = B - 1$  for  $n - 1 \geq i \geq 0$

### 1.4 hex / binary

H	D	B	Dec	Bin
0	0	0000	16	$2^5$ (max 31)
1	1	0001	32	$2^6$ (max 63)
2	2	0010	64	$2^7$ (max 127)
3	3	0100	128	$2^8$ (max 255)
4	4	0101	256	$2^9$ (max 511)
5	5	0110	512	$2^{10}$ (max 1'023)
6	6	0111	1'024	$2^{11}$ (max 2'047)
7	7	1000	2'048	$2^{12}$ (max 4'095)
9	9	1001	4'096	$2^{13}$ (max 8'191)
A	10	1010	8'192	$2^{14}$ (max 16'383)
B	11	1011	16'384	$2^{15}$ (max 31'767)
C	12	1110	32'768	$2^{16}$ (max 65'535)
D	13	1011		
E	14	1011		
F	15	1011		

### 1.5 Signed numbers

two's compliment is being used

$$Z_{signed} = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$$

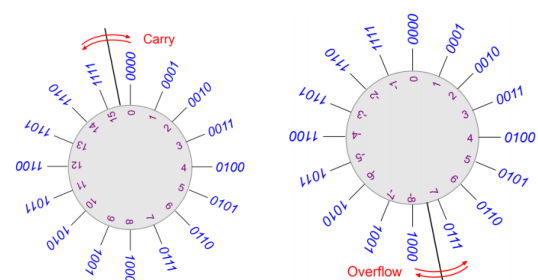
most significant bit is negative

Example:  $-1$  as 16-bit Hex = 0xFFFF

Conversion:

1. Invert binary :  $-6 \rightarrow 0110 \rightarrow 1001$
2. increment by 1 :  $1001 + 0001 \rightarrow 1010$

### 1.6 carry / overflow



**Carry** is set on crossover between lowest and highest number

**Overflow** happens on crossover between highest absolute values

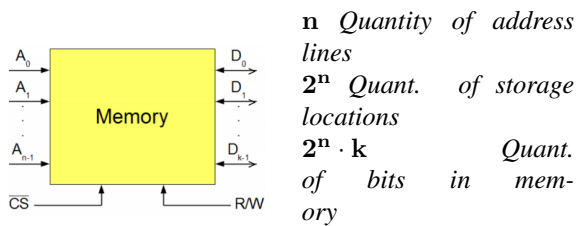
### 1.7 Bit groups

**Nibble/Tetrad** has the size of 4 bits

**Byte** has the size of 8 bits

**Word** is MC9S08JM60 specific, it has 16 bits

## 1.8 Quantity of address lines



$$1\text{ K} = 2^{10} = 1024\text{ Bit} \hat{=} 10\text{ Adresslines}$$

$$64\text{ K} = 2^{16} = 65536\text{ Bit} \hat{=} 16\text{ Adresslines}$$

example,  $32\text{K} \times 8$  memory storage space:

**bits storage:**  $32 \cdot 2^{10} \cdot 8 = 2^5 \cdot 2^{10} \cdot 2^3 = 2^{18} \rightarrow 18\text{ Bits}$

**number address lines:**  $32 \cdot 2^{10} = 2^{15} = 32\,768$

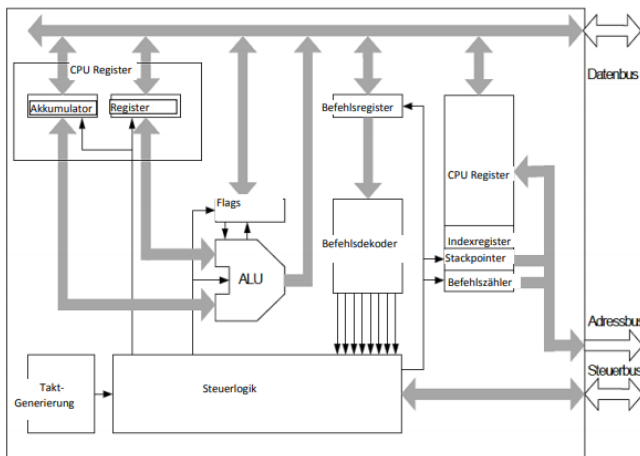
**highest address:**  $2^{18} - 1 = 0x7FFF = 262\,143$

## 1.9 Microprocessor vs Microcontroller

**Microcontroller** contains CPU (Processor), Peripherals (I/O) and Memory (RAM / ROM). Basically a small computer.

**Microprocessor** has only CPU and some integrated Circuits.

## 1.10 CPU components



ALU (Arithmetic Unit), AKKU (Accumulator), PC (Program Counter), Busses, Instruction-Register, Address-Register, Operand-Register, Control Unit, ..

## 1.11 Instruction Cycle Steps

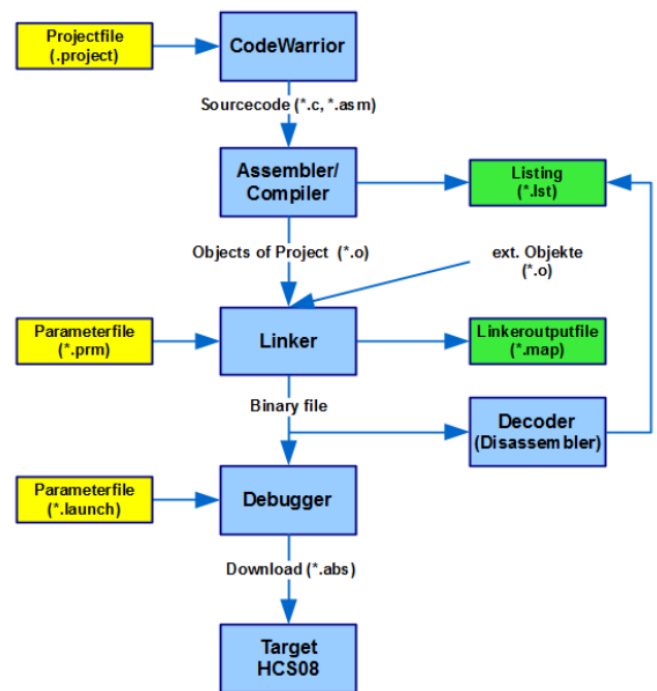
1. instruction fetch
2. instruction decode
3. (operand fetch)
4. instruction execute
5. next address and inc PC

## 1.12 Types of MCU Registers

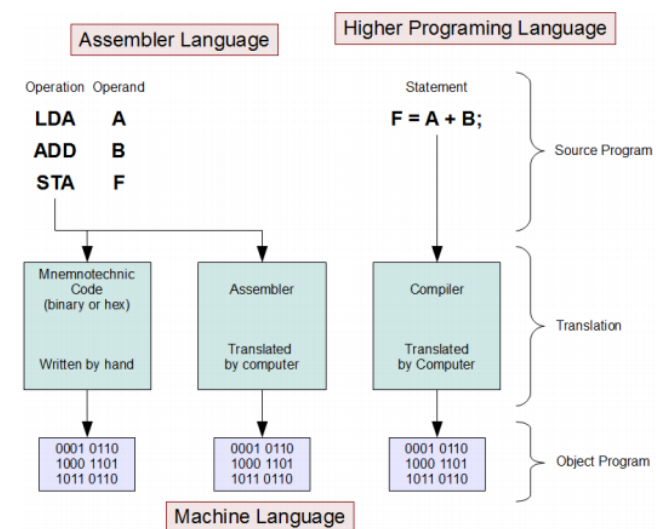
AKKU, PC, Instruction-Register (decoder), Operand-Register

## 2 Compiling

### 2.1 Codewarrior Designflow



### 2.2 Programming Language



High level programming languages are:

- portable
- efficient (normally)
- Better readable
- easier to maintain

High level programming languages are usually preferred, if enough computational power and memory is available. Assembler is often used, if the application:

- is time critical and needs exact timing
- timing of the high level programming language to unpredictable is

## 2.3 Assembler Code-Format

	Label	Instruction	Operands	comment
Ex1	Limit:	EQU	\$CD	; define limit
Ex2	Start:	LDA	#Limit	; load limit

**Instruction:** is a command for the processor

**Directive:** are instructions that direct the assembler / compiler to do something

	Type	Directed to	Results in program code
Ex1	<b>Instruction</b>	Target CPU	Yes
Ex2	<b>Directive</b>	Assembler	Only indirect
	<b>Comment</b>	Programmer	No

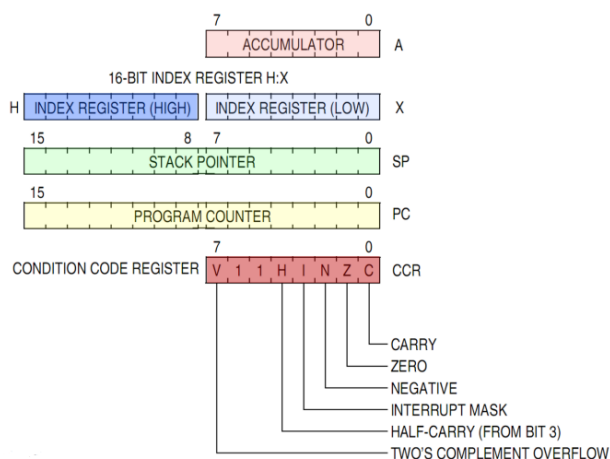
## 2.4 Parameter file

The Parameter file (\*.prm) is used for by the Linker. It takes the machine code and defines the location on the controller. It is important, so that jumps work correctly. It contains:

- Memory-Map of the Prozessor (Location and size of Flash, RAM, ..)
- Extra definitions, where which parts of the code on the Controller should be located

## 3 Assembler & HCS08

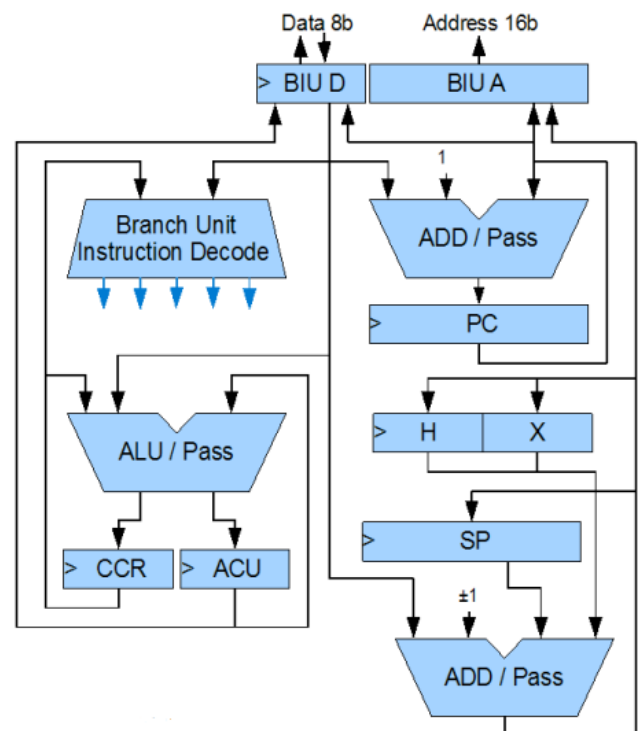
### 3.1 HCS08 CPU Registers



Registers the HCS08 contains:

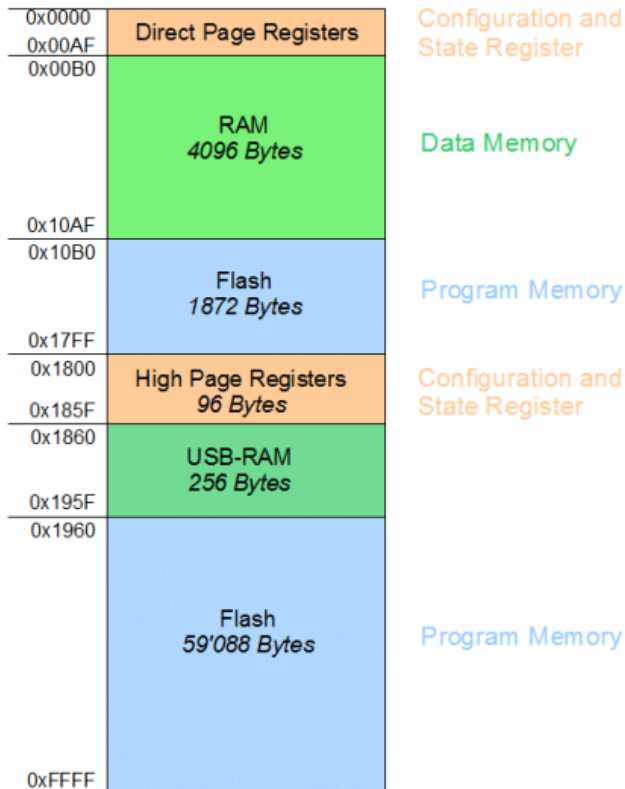
- HX Register
- PC
- Akku
- Stack Pointer
- CCR

## 3.2 HCS08 Processor



- 8 Bit, Von Neumann architecture
- **BIU** Bus Interface Unit
- **PC** Program Counter
- **ACU** Accumulator
- **ALU** Arithmetic Logic Unit
- **CCR** Condition Code Register (Collection of status flags)
- **SP** Stack (LI-FO, Pointer for Context and Parameter)
- **H:X** Index Register

### 3.3 Memory Mapping



Access to the directpage (0x0000 - 0x0AF) needs less cycles, since the address is only 1 Bytes long.

### 3.4 Register configuration HCS08

```
// define the dataflow direction input = 0 /
output = 1
PTADD = 0x04;

// set output value
PTAD = 0x04;
// read value
uint_8 val = PTAD;

// set pullup enable port
PTADD = 0x00;
PTAPE = 0x04;
```

#### Reg. Name Description

PTxDD	Data Direction of Port x
PTxD	Data value of Port x
PTxPE	Set Pullup Enable of Port x (PTxDD needs to be 0)

**Pullup Enable** is used to pullup the value of the output to 1. This is usually used on a bus system to prevent a short circuit.

### 3.5 Differences of Operations

Comparing different operations, following should be taken in consideration:

- number of cycles
- memory usage, 8bit (directpage) / 16bit
- Set CCR bits / flags
- Used registers

- Address modes

## 4 Assembler Directives & Addressing Modes

### 4.1 Directives

Directive	Description
SECTION	Defines the beginning of a relocatable section
EQU	Assigns an expression to a name. Not redefinable
DC	Defines one or more constants and their names. Will be stored at the set location
DS	Allocates memory(RAM) for variables

The Assembler-Directive **SECTION** defines program- and data section. Those section can be moved freely within the memory (relocative assembling), **after** the **assembly** process is finished.

The final memory area location happens after the linking process. The locations of those sections can therefore be defined in the **Linker-Parameterfile**.

### 4.2 Basic Assembler Program

```
; include definitions
include 'MC9S08JM60.inc'

; -- globals
GLOBAL _Startup ; define start of programm
GLOBAL main
GLOBAL dummy ; Dummy Interrupt Service Routine

; -- equations
StackSize: EQU $60 ; stack size
pi: EQU 31416 ; example of random equ

; -- stack
DATA_STACK: SECTION
TofStack: DS StackSize-1 ; definiton of "
Top of Stack"
BofStack: DS 1 ; definition of "
Bottom of Stack"

; -- create space for data
DATA: SECTION
var1: DS 1 ; Example of a 1 Byte
Variable
Array1: DS $20 ; Example of an Array of $20
Bytes

; -- setup constants
CONST: SECTION
Maskel: DC.B %00000001
Parameter1: DC.B $3A ; DC with a point
Parameter2: DC.W 57100 ; word with int
value
Reserve_Par: DS 16 ; reserve empty 16
Bytes
VarArray: DS.W 3 ; reserve 3 Words
STRING1: DC.B 10, "Hello", $0D

; -- program start (initialisation)
PROGRAMM: SECTION ; Code Segment
```

```

_Startup:          ; Resetvektor points to
                  this
Stackinit: LDHX    #(BofStack+1)
           TXS      ; decrement TXS, thats
           why +1 BofStack
           LDA      #$00
           STA      SOPT1 ; Disable Watchdog

; -- actual program
main:
  ; turn on backlighths of the car
  BSET     PTDD_PTDD2, PTDD
  BSET     PTDDD_PTDDD2, PTDDD

  CLR      RamLoc

  BCLR     PTGDD_PTGDD0, PTGDD
  BCLR     PTGDD_PTGDD1, PTGDD
  BCLR     PTGDD_PTGDD2, PTGDD
EndlessLoop:
  ; load joystick values
  MOV      RamLoc, PTGD
  JMP      EndlessLoop

; (=ensure program end if endlessloop is
missing)
EndLoop:   BRA      *

; catch any unexpected interrupts
dummy:     BGND
           BRA      dummy

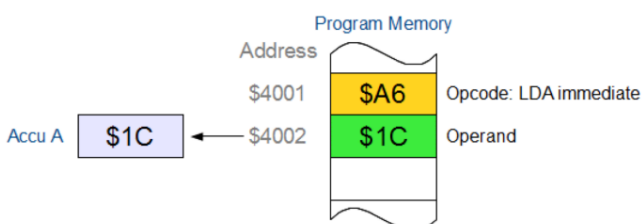
```

### 4.3 Addressing Modes

- **Immediate:** 1 Byte operand in instruction (LDA #\$01)
- **Inherent:** no operand required (e.g. NOP, INCA..)
- **Direct:** onlu direct page, 1 address Byte
- **Extended:** whole 64k area, 2 address Bytes
- **Indexed:** with SP (Stack pointer) or HX (7 sub modes)
- **Relative:** for branches,  $PC=PC+2+two's\ compl.$

Different addressing modes of the same instruction type use different operation codes (e.g. LDA-MM: A6; LDA-DIR: B6).

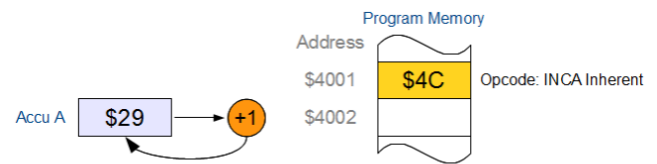
#### 4.3.1 Immediate (IMM)



**Immediat addressing mode:** the following Byte of the operation code is immediately used as the operand.

Example: **LDA #\$1C**

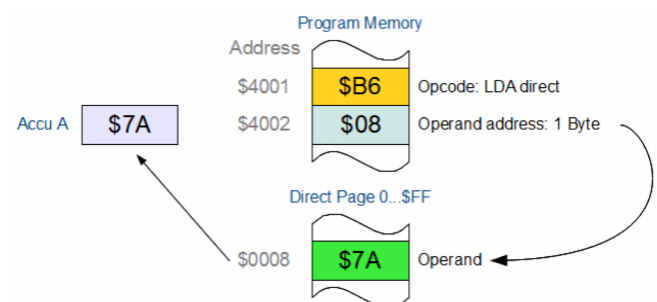
#### 4.3.2 Inherent (INH)



**Inherent addressing mode:** no explicit operand address needed. All operands are in the CPU-registers

Example: **INCA**

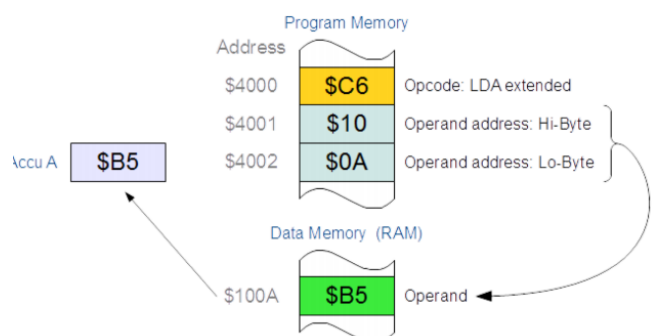
#### 4.3.3 Direct (DIR)



**Direct addressing mode:** After the operation code, the **1-Byte** operand address follows in the program memory. Only operands in the address section between \$00 and \$FF are supported. (The Direct Page Registers 0x00-0xAF, Direct Page RAM 0xB0-0xFF)

Example: **LDA \$08**

#### 4.3.4 Extended (EXT)



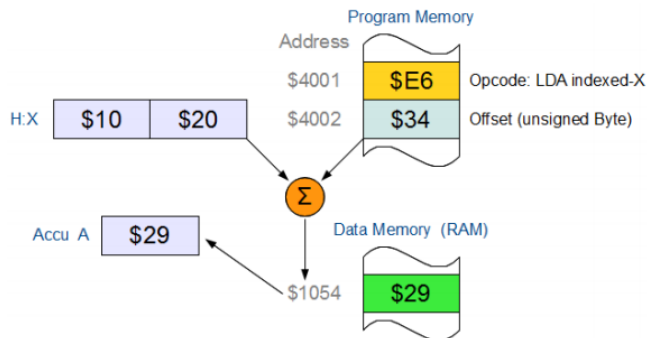
**Extended addressing mode:** After the operation code, the **2-Byte** operand address follows in the program memory.

Supports the whole address section between 0x0000 - 0xFFFF. But is also slower.

Example: **LDA \$34,X**



### 4.3.5 Indexed (IX1)

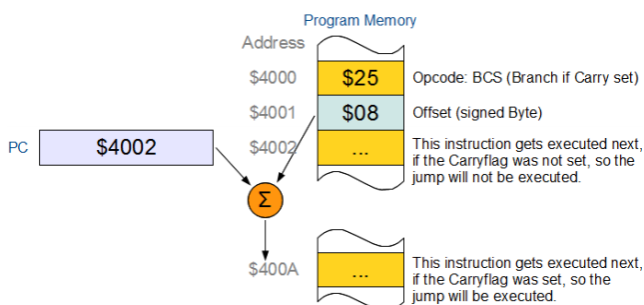


**Indexed addressing mode:** uses the **H:X** or **SP** register. Through indexed addressing the final assigned operand address is dependent from the program behaviour (address arithmetics).

Following are sub modes of the indexed addressing mode

<b>IX</b>	Indexed addressing with <b>H:X</b> , without offset	<b>LDA X</b>
<b>IX1</b>	Indexed addressing with <b>H:X</b> and <b>8-bit offset</b>	<b>LDA \$34, X</b>
<b>IX2</b>	Indexed addressing with <b>H:X</b> and <b>16-bit offset</b>	<b>LDA \$34A5, X</b>
<b>IX+</b>	Indexed addressing with <b>H:X</b> and <b>H:X Increment</b> . Only for <b>MOV</b> and <b>CBEQ</b> (Compare Accu with value on the address that is stored in the <b>H:X</b> register. If values are equal, jump to <b>Label</b> and increment <b>H:X</b> ) instructions	<b>CBEQ X+, Label</b>
<b>IX1+</b>	Same as <b>IX+</b> , with <b>Increment</b> and <b>8-bit offset</b> (Only available for instruction <b>CBEQ</b> )	<b>CBEQ \$34,X+, Label</b>
<b>SP1</b>	Same as <b>IX1</b> , but with Stack-pointer <b>SP</b> instead of <b>H:X</b> .	<b>LDA \$34, SP</b>
<b>SP2</b>	Same as <b>IX2</b> , but with Stack-pointer <b>SP</b> instead of <b>H:X</b> .	<b>LDA \$34A5, SP</b>

### 4.3.6 Relative (REL)



**PC relative addressing mode:** is only used with **BRANCH**-Instructions.

The following Byte after the operand is a **two's complement** offset to the already increased program counter.

The address range with relative addressing is -126 to +129. 129, since the PC is incremented before and after the jump (+2).

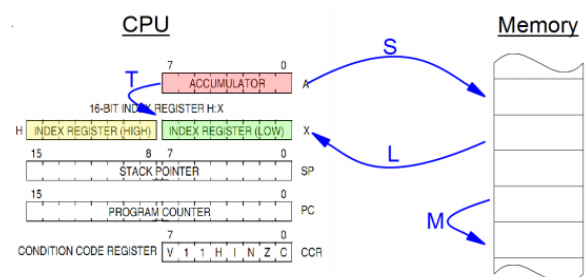
## 5 Assembler Addressing & Programming

### 5.1 Assembler Instructions

There are 3 main type of instructions:

- **Data Transport**
- **Operations** (Arithmetic, Logic, Bit-manipulation, Shift and Rotation)
- Program **Branches** with jump and branch operations

### 5.2 Transport Operations



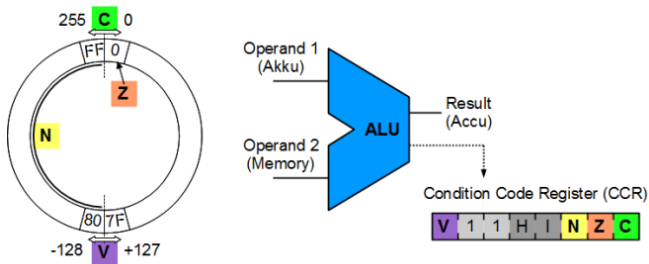
	Operation	Example
L	Load	<b>LDA, LDH, LDHX; PULA, PULX</b> (Stackoperations)
S	Store	<b>STA, STX, STHX; PSHA, PSHZ</b> (Stackoperations)
T	Transfer	<b>TAP, (CCR = Accu.), TPA, TAX, TSX</b>
M	Move	<b>MOV</b>

### 5.3 Arithmetic Operations

<b>ADD</b>	Adds given operand to the ACC.
<b>SUB</b>	Works equivalent to the addition.
<b>ADC &amp; SBC</b>	Include Carry bit and support additions and subtractions with numbers with more than 8 bits.
<b>MUL</b>	Multiplies the content of the accumulator A with the content of the index register X and stores the 16-bit result in X:A (MSB in X, LSB in A) <b>only unsigned.</b>
<b>DIV</b>	divides the 16-bit dividend in H:A (MSB in H, LSB in A) with the divisor in the index register X. The 8-bit result is written to A. If an overflow or division by 0 occurs, the Carry-bit is set. <b>only unsigned.</b>

Results of arithmetic instructions are saved on the HCS08 either in the X-Register or AKKU

## 5.4 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

*Half-Carry is used for binary-coded decimal calculations*

### ADD instruction

C: A7&M7 | M7&R7 | A7&R7  
 V: A7&M7&R7 | A7&M7&R7  
 N: R7  
 Z: R7&R6&R5&R4&R3&R2&R1&R0

### SUB instruction

C: A7&M7 | M7&R7 | A7&R7  
 V: A7&M7&R7 | A7&M7&R7  
 N: R7  
 Z: R7&R6&R5&R4&R3&R2&R1&R0

A (Operand 1)

M (Operand 2)

R (Result 1)

## 5.5 Logical Operations & Bit Masking

```
B7: EQU $80 ; Mask for Bit 7
B6: EQU $40 ; Mask for Bit 6
:           :
B0: EQU $01 ; Mask for Bit 0

ORA # (B6 | B3) ; Set Bit 6 and 3 in ACCU
AND # (B5 | B4) ; Delete all Bits in ACCU except
                Bit 5 and 4
```

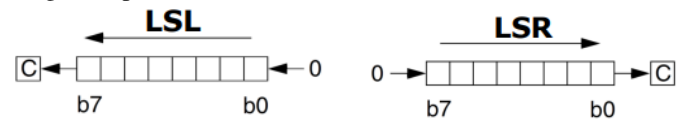
**AND** logical AND-operation  
**ORA** logical OR-operation  
**EOR** logical XOR-operation  
**BCLR n,Addr** Delete Bit n on a specific memory address  
**BSET n,Addr** Set Bit n on a specific memory address  
**BIT Addr** Bitwise AND operation of Accu with content of Addr, without changing content of Accu and Addr. Affects only **N-** and **Z-Flags**.  
**CLC** Delete Carry-Flag C  
**SEC** Set Carry-Flag C  
**CLI** Delete Interrupt-Mask Bit I (Interrupt enable)  
**SEI** Set Interrupt-Mask Bit I (Interrupt disable)

## 5.6 Shift- and Rotation Operations

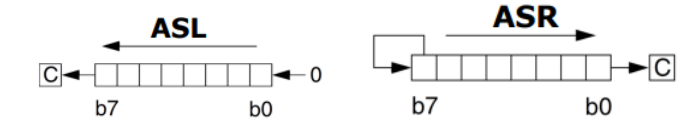
in direction MSB (left)

in direction LSB (right)

Logical Operations:

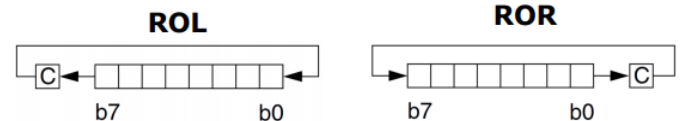


Arithmetic Operations:



Multiplication by 2, ASL=LSL

Division by 2



## 5.7 Relative Branching

Unconditional Branch

Oper.	Meaning
<b>BRA</b>	Branch <b>always</b>
<b>BRN</b>	Branch never
<b>BSR</b>	Branch to <b>subroutine</b>

Testing a Single Flag

Oper.	Test	Meaning
<b>BEQ</b>	Z=1	Branch if <b>equal</b>
<b>BNE</b>	Z=0	Branch if not equal
<b>BCS</b>	C=1	Branch if <b>Carry</b> set
<b>BCC</b>	C=0	Branch if Carry clear
<b>BMI</b>	N=1	Branch if <b>Minus</b>
<b>BPL</b>	N=0	Branch if Plus

Arithmetic Comparison of Accu and Memory Location

Oper.	Test	Format
<b>BGT</b>	>	signed
<b>BHI</b>		unsigned
<b>BGE</b>	≥	signed
<b>BHS, BCC</b>		unsigned
<b>BLE</b>	≤	signed
<b>BLS</b>		unsigned
<b>BLT</b>	<	signed
<b>BLO, BCS</b>		unsigned
<b>BEQ</b>	=	signed
		unsigned

## 5.8 Branching Compare-Operation

Compare instructions are **subtraction operations** that change status flags, but leave the data registers unchanged.

**CMP opr8** Compare content of **ACCU** with 8-bit operand

**CPX opr8** Compare content of **X-Register** with 8-bit operand

**CMP opr8** Compare content of **HX-Register** with 16-bit operand

Example, Test if a value is bigger or smaller than another value, branch afterwards

```
LDA Op1
CMP Op2 ; Calculates (Op1-Op2) and sets flags
BMI Label ; Branch if Op2 > Op1 (N=1) to Label
```

## 5.9 Direct relative Branching

Those Branches are dependent on a single Bit of a memory located in the **Direct Page**.



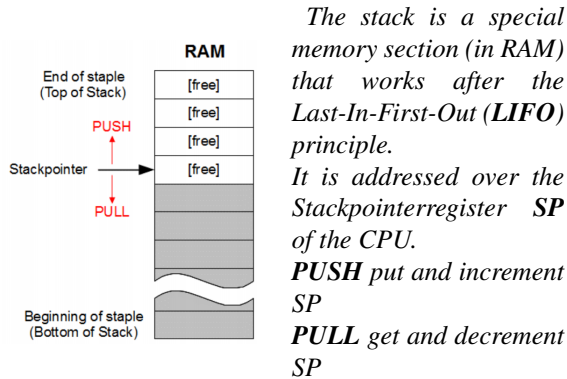
```

BRCLR n,Addr,Label ; Branches to Label, if Bit
                     n of value on
                     ;address Addr is not set (
                     ;   Addr only DIR)
BRSET n,Addr,Label ; Branches to Label, if Bit
                     n of value on
                     ;address Addr is set (Addr
                     ;   only DIR)

```

## 6 Subroutines & Stack

### 6.1 Stack



Stack grows from high addresses to lower

```

Stacksize: EQU $40
:
DATA:      SECTION
TofStack:  DS Stacksize-1 ; reserve stack
BofStack:  DS 1
:
PROGRAM:   SECTION
LDHX # (BofStack+1) ; H:X := Bottom
              of Stack
TXS              ; SP := HX -1
:
; save CPU-Status on stack
PSHA ; Akku auf Stack
PSHX ; X-Register auf Stack
:
; restore CPU-Status from stack
; order is important (LIFO!)
PULX ; X-Register
PULA ; Akku

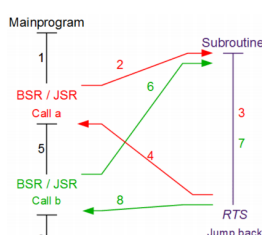
```

Stacks are used for:

- Subroutine calls (save return address)
- Store context
- Store parameters
- Store local variables

malloc (heap) and global variables are not stored on the stack.

### 6.2 Subroutines



BSR/JSR push and inc. PC  
RTS pull and inc. PC  
Parameters passing on stack (used by C)

**Local Variables** saved on stack (used by C)

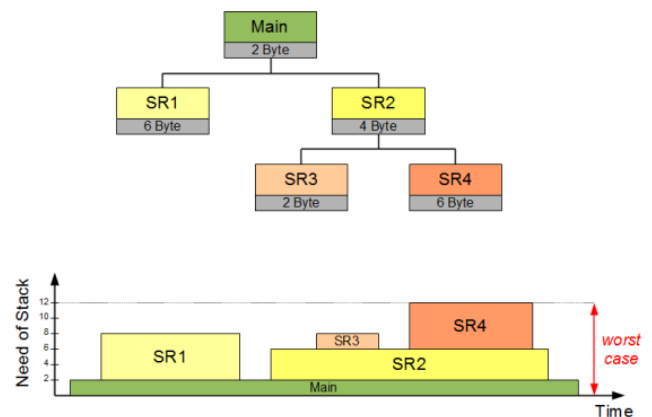
subroutines enable following:

- **less memory usage**; repeated command sequences are stored only once
- **less development effort**; tested command sequences can be reused
- **less error prone**; enable modular way of building software
- **higher team productivity**; multiple people can work parallel on different code sections
- **shorter compile time & libraries**; different parts of the code can be compiled separately

The only **negative** about subroutines is calling of subroutines is **slower**. Time is needed for passing parameters and saving the context on the stack

### 6.3 Stack size

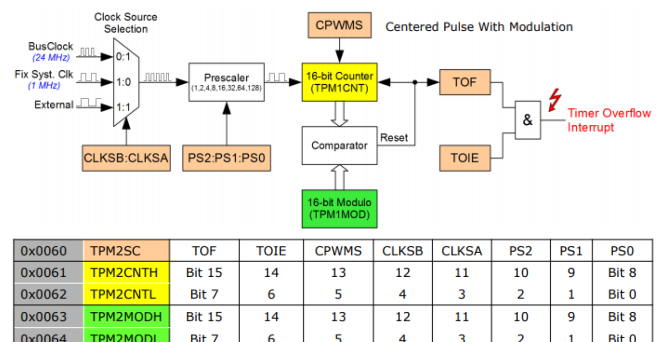
To analyze the used stack size, it is helpful to create a tree with the subroutines, their calls and used stack space.



It is also possible to figure out the stack usage by filling the program-stack at the start with a bit pattern like 0xdeadbeef and stress test the program as much as possible. At the end, this will show which part and how much of the stack has been used during the program execution.

## 7 Timer and Interrupts

### 7.1 Modulo Counter



## 7.2 Modulo Frequency

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

- $T_{TOF}$ : Time between two Timer-Overflow events
- $MOD$ : Value of the Modulo set
- $PS$ : Prescaler value
- $f_{CLK}$ : frequency of the controller

To calculate the modulo, the frequency (Clock Source) needs to be selected and the prescaler needs to be defined. To calculate the Modulo value, following can be used. The Modulo is 2 Bytes, so it needs to be between 0 <  $MOD$  < 65536

$$MOD = \left( \frac{T_{TOF} \cdot f_{CLK}}{PS} \right) - 1$$

## 7.3 Timer Control Registers

Address	Reg-Name	Bit-Name							
0x0060	TPM2SC	TOF	TOIE	CPWMS	CLKSB	CLKSA	PS2	PS1	PS0
0x0061	TPM2CNTH	Bit 15	14	13	12	11	10	9	Bit 8
0x0062	TPM2CNTL	Bit 7	6	5	4	3	2	1	Bit 0
0x0063	TPM2MODH	Bit 15	14	13	12	11	10	9	Bit 8
0x0064	TPM2MODL	Bit 7	6	5	4	3	2	1	Bit 0
0x0065	TPM2C0SC	CH0F	CH0IE	MS0B	MS0A	ELS0B	ELS0A	0	0
0x0066	TPM2C0VH	Bit 15	14	13	12	11	10	9	Bit 8
0x0067	TPM2C0VL	Bit 7	6	5	4	3	2	1	Bit 0
0x0068	TPM2C1SC	CH0F	CH0IE	MS0B	MS0A	ELS0B	ELS0A	0	0
0x0069	TPM2C1VH	Bit 15	14	13	12	11	10	9	Bit 8
0x006A	TPM2C1CL	Bit 7	6	5	4	3	2	1	Bit 0

SC = Status&Control, CNT = Counter, MOD = Modulo, V = Value; H = High-Byte, L = Low-Byte

## 7.4 Polling and Interrupts

A MC-System has to react instantly to events (internal or external) (e.g. measure value monitoring, serial communication).

The **instant of time** of these events is **not known in advance**.

There are two ways to react to those kind of events:

- **Interrupt** = Exception handling  
enables **realtime capable** systems (depends on interrupt latency). **Fast reaction time** through automatic reaction to events and interrupt of the program to execute an Interrupt-Service-Routine (ISR).  
Needs substantial effort for **state backup**, because the instant of the program interruption is unknown.
- **Polling** = cyclic requesting  
**Shorter program interruption**. Since the instant of time is known during programming, the state can be backed up more efficiently.

**Waste of calculation time** if events occur rarely

Each MCU holds an Interrupt-Logic to realise real-time systems.

## 7.5 Save Interrupt State

```
interrupt void myTofISR(void)
{
    // myTofISR function needs to be mapped
    // in the vectortable -> parameterfile (.
    prm).
    //reset the interrupt flag
    TPM1SC_TOF = 0;

    //run logic
}

void initTimer(void)
{
    //set module to 25780 / 0x64B4
    TPM1MODH = 0x64;
    TPM1MODL = 0xB4;
    //TPM1MOD = 25780;
    //Clock set to 1 MHz
    TPM1SC_CLKSA = 0;
    TPM1SC_CLKSB = 1;

    //define Prescaler to 128
    TPM1SC_PS0 = 1;
    TPM1SC_PS1 = 1;
    TPM1SC_PS2 = 1;

    // enable timer Overflow Interrupt
    // this should be the last action
    TPM1SC_TOIE = 1;
    // Reset the Timer Overflow Interrupt
    TPM1SC_TOF = 0;
}

void main(void)
{
    initTimer();
    //enable interrupts
    EnableInterrupts;
}
```