Micro Controllers Summary

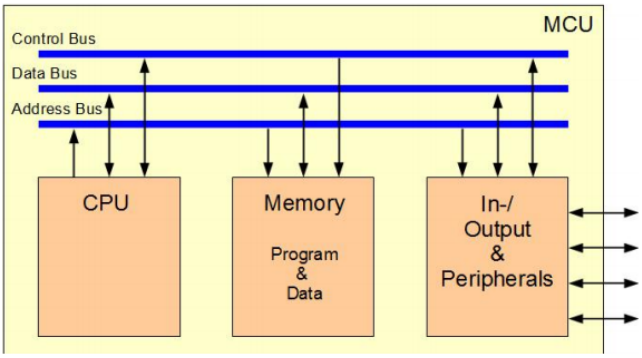
Lucien Zürcher

June 20, 2019

Contents

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 beeing used

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

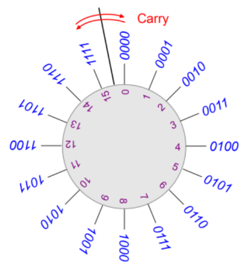
most significant bit is negative

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

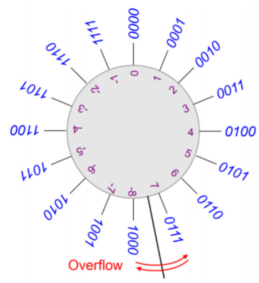
Conversion:

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 absolut values*

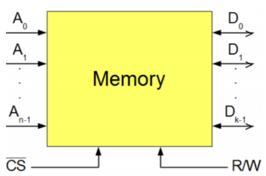
1.7 Bit groups

Nibble/Tetrade *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



n *Quantity of address lines*
$\mathbf{2^n}$ *Quant. of storage locations*
$\mathbf{2^n \cdot k}$ *Quant. of bits in memory*

1 K	=	2^{10}	=	1024 Bit	≙	10 Adresslines
64 K	=	2^{16}	=	65536 Bit	≙	16 Adresslines

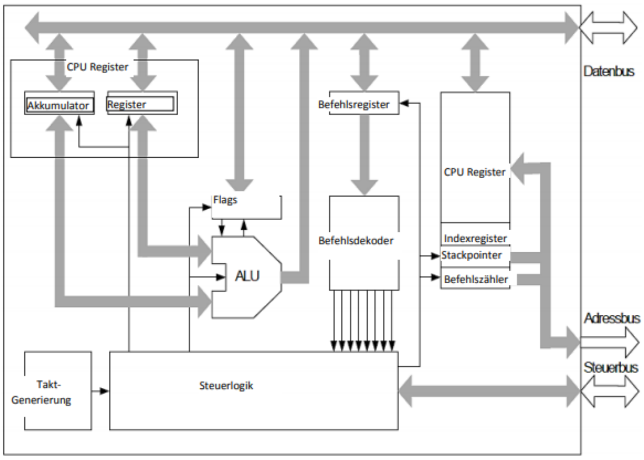
example, $32K \times 8$ *memory storage space:*

bits storage: $32 \cdot 2^10 \cdot 8 = 2^5 \cdot 2^10 \cdot 2^3 = 2^18 \to 18$ *Bits*
number address lines: $32 \cdot 2^10 = 2^15 = 32\,768$
highest address: $2^{18} - 1 = 0x7FFFF = 262'143$

1.9 Microprocessor vs Mircocontroller

Mircocontroller *contains CPU (Processor), Peripherals (I/O) and Memory (RAM / ROM). Basically a small computer.*
Mircoprocessor *has only CPU and som integrated Circuits.*

1.10 CPU components



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

1.11 Instruction Cycle Steps

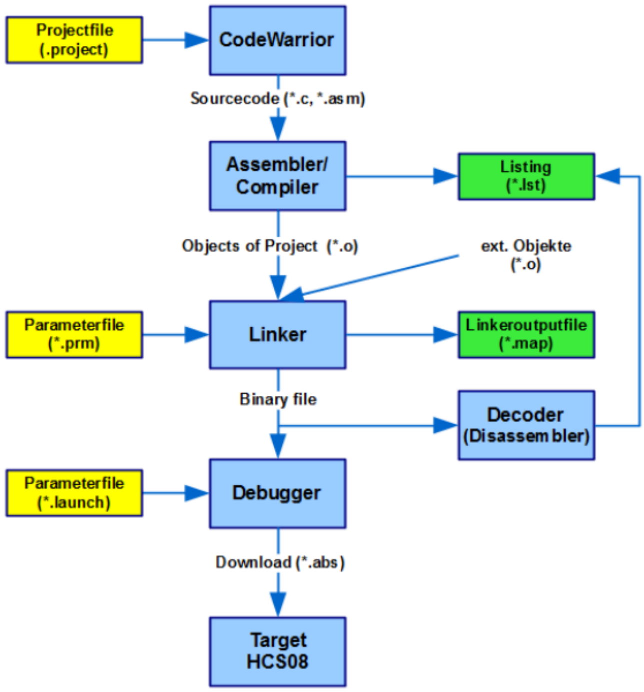
1. instruction fetch
2. instruction decode
3. (operand fetch)
4. instruction execute
5. next address and inc PC

1.12 Types of MCU Registers

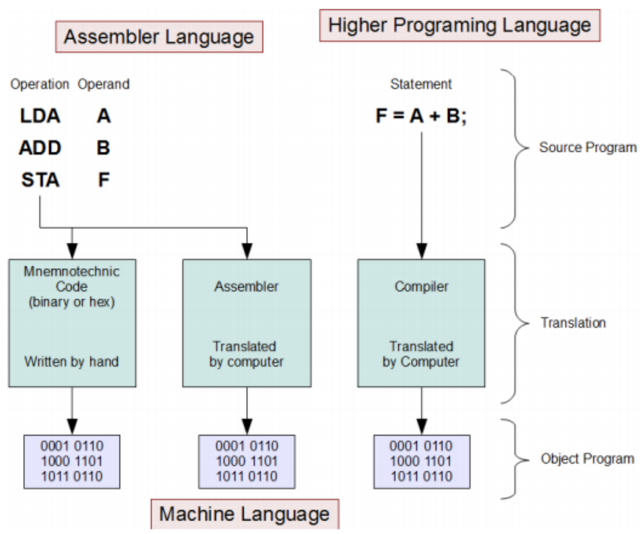
AKKU, PC, Instruction-Register (decoder), Operand-Register

2 Compiling

2.1 Codewarrior Designflow



2.2 Programming Language



High level programming languages are:

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

High level programming languages are usually prefered, 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 unpredictible 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	**Instruction**	Target CPU	Yes
Ex2	**Directive**	Assembler	Only indirect
	Comment	Programmer	No

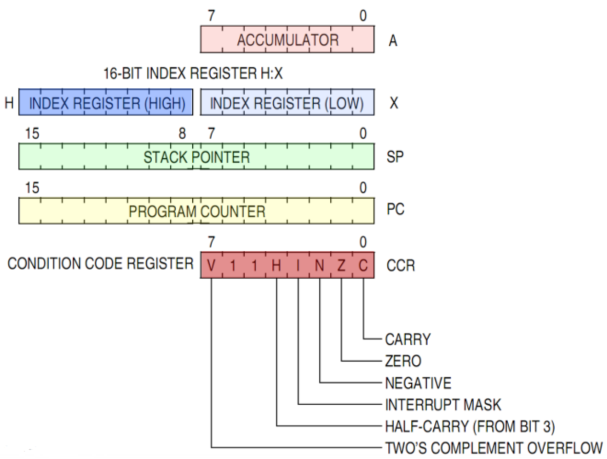
2.4 Parameter file

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

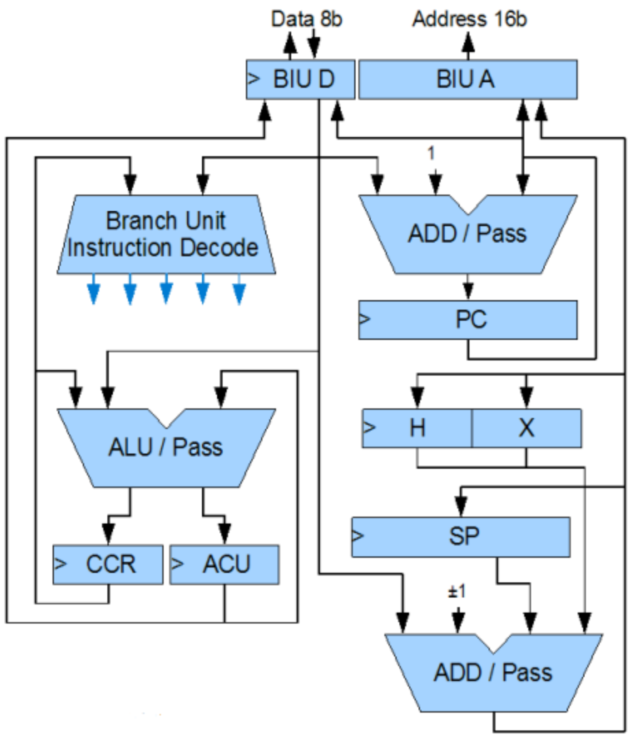
3.1 HCS08 CPU Registers



Registers the HCS08 contains:

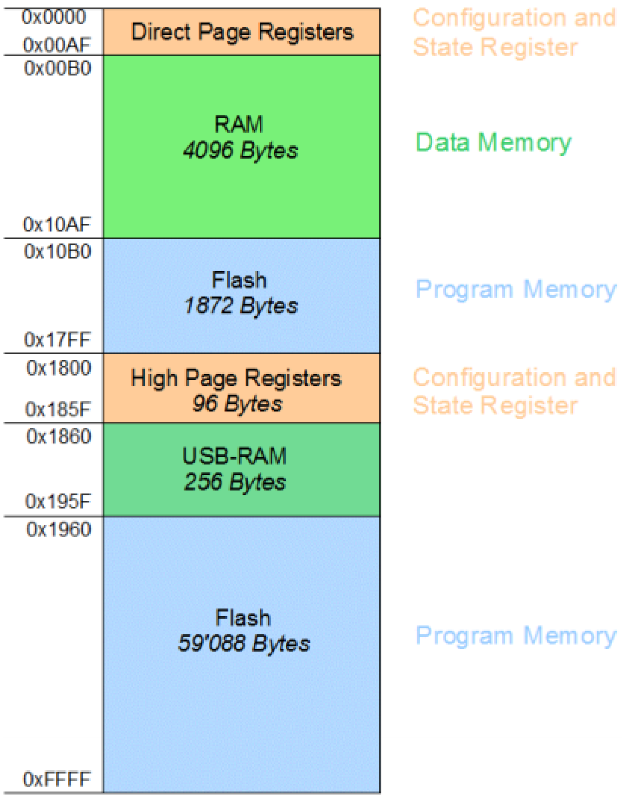
- HX Register
- PC
- Akku
- Stack Pointer
- CCR

3.2 HCS08 Processor



- 8 Bit, Von Neumann archidecture
- **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 therefor 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
Maske1:       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
```

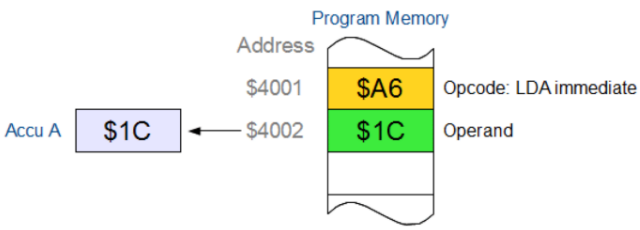
```
_Startup:           ; Resetvektor points to
    this
Stackinit:  LDHX   #(BofStack+1)
            TXS          ; decrement TXS, thats
                 why +1 BofStack
            LDA    #$00
            STA    SOPT1  ; Disable Watchdog

; -- actual program
main:
    ; turn on backligths 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.

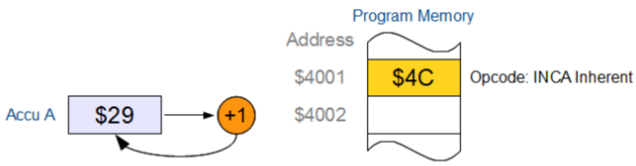
Different addressing modes of the same instruction type use differnet operation codes (e.g. LDA-MM: A6; LDA-DIR: B6).

4.3.1 Immediate (IMM)



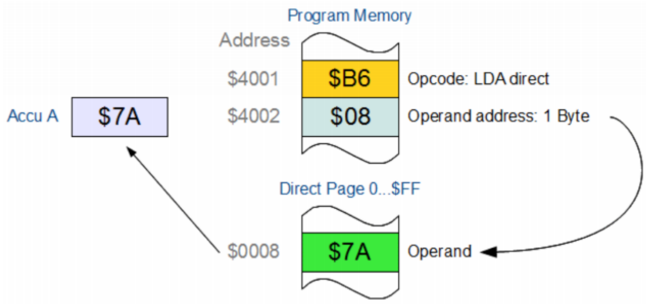
Immediat adressing *mode: the following Byte of the operation code is immediately used as the operand.*
Example: ***LDA #$1C***

4.3.2 Inherent (INH)



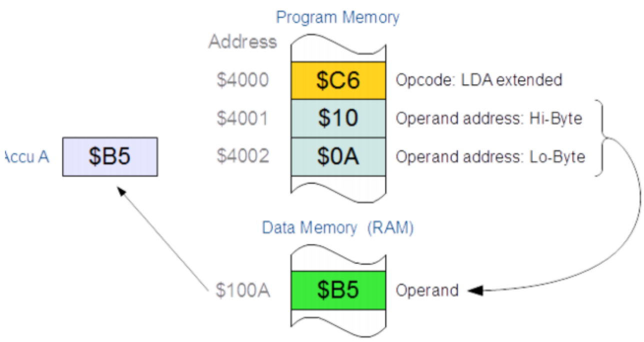
Inherent addressing *mode: no explicit operand address needed. All operands are in the CPU-registers*
Example: ***INCA***

4.3.3 Direct (DIR)



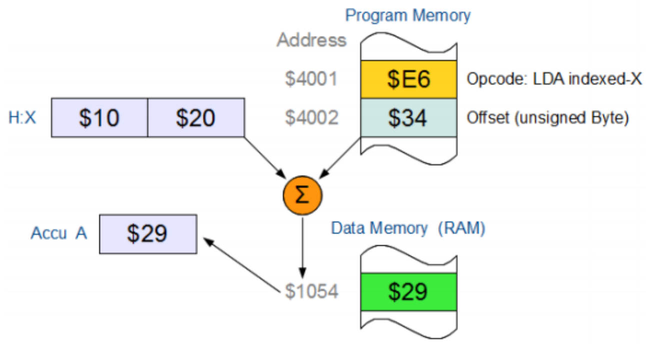
Direct addressing *mode: After the operation code, the* ***1-Byte*** *operand address follows in the program memory. Only operands in the address section between $00 and $FF are supported. (The Direct Page Registers 0x00-0xAF, Direct Page RAM 0xB0-0xFF)*
Example: ***LDA $08***

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***

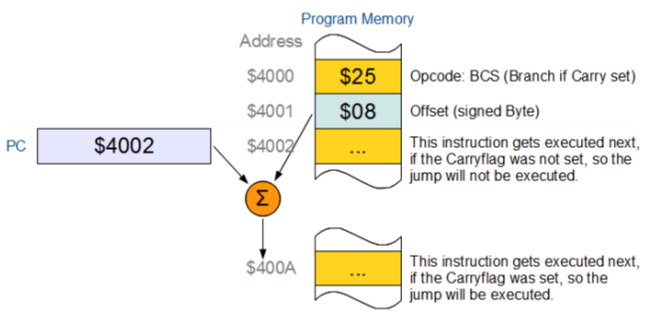
4.3.5 Indexed (IX1)



Indexed addressing *mode: uses the* ***HX*** *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

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

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 relaive addressing is -126 to +129. 129, since the PC is incremented before and after the jump (+2).

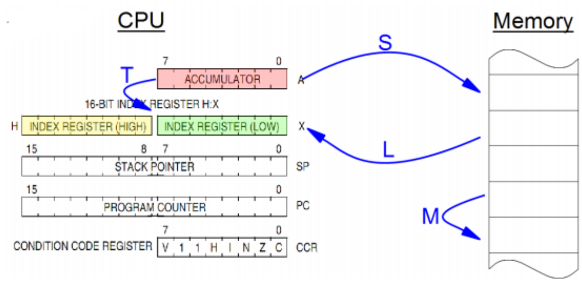
5 Assembler Addressing & Programming

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



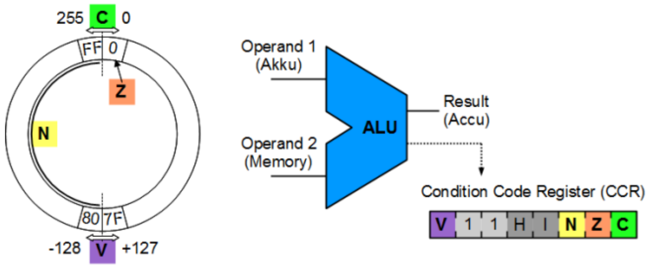
	Operation	Example
L	Load	*LDA, LDX, LDHX; PULA, PULX (Stackoperations)*
S	Store	*STA, STX, STHX; PSHA, PSHZ (Stackoperations)*
T	Transfer	*TAP, (CCR = Accu.), TPA, TAX, TSX*
M	Move	*MOV*

5.3 Arithmetic Operations

ADD	*Adds given operand to the ACC.*
SUB	*Works equivalent to the addition.*
ADC & SBC	*Include Carry bit and support additions and subtractions with numbers with more then 8 bits.*
MUL	*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)* ***only unsigned****.*
DIV	*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.* ***only unsigned****.*

Results of arithmetic instructions are saved on the HCS08 eather 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: $A7\&M7 \mid M7\&\overline{R7} \mid A7\&\overline{R7}$

V: $\overline{A7}\&\overline{M7}\&R7 \mid A7\&M7\&\overline{R7}$

N: $R7$

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$

SUB instruction

C: $\overline{A7}\&M7 \mid M7\&R7 \mid \overline{A7}\&R7$

V: $A7\&\overline{M7}\&\overline{R7} \mid \overline{A7}\&M7\&R7$

N: $R7$

Z: $\overline{R7}\&\overline{R6}\&\overline{R5}\&\overline{R4}\&\overline{R3}\&\overline{R2}\&\overline{R1}\&\overline{R0}$

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*

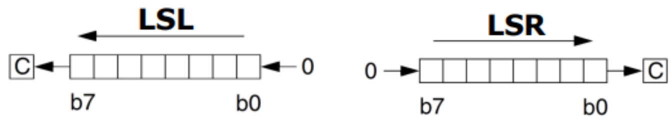
CLI *Delete **Interrupt**-Mask Bit I (Interrupt **enable**)*

SEI *Set Interrupt-Mask Bit I (Interrupt disable)*

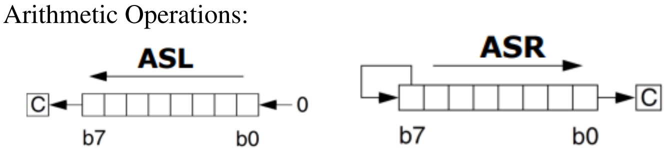
5.6 Shift- and Rotation Operations

in direction MSB (left) **in direction LSB (rigth)**

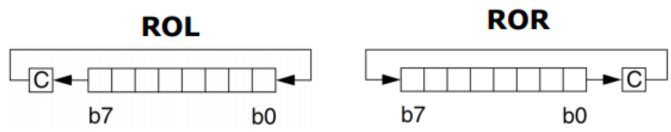
Logical Operations:



Arithmetic Operations:



Multiplication by 2, ASL=LSL *Division by 2*



5.7 Relative Branching

Unconditional Branch

Oper.	Meaning
BRA	Branch **always**
BRN	Branch never
BSR	Branch to **subroutine**

Testing a Single Flag

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

Arithmetic Comparison of Accu and Memory Location

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

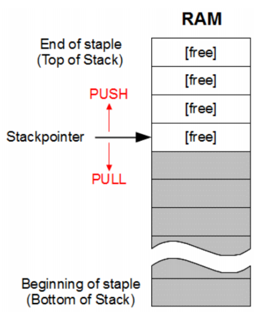
5.9 Direct relative Branching

*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



*The stack is a special memory section (in RAM) that works after the Last-In-First-Out (**LIFO**) principle.*
*It is addressed over the Stackpointerregister **SP** of the CPU.*
***PUSH** put and increment SP*
***PULL** get and decrement SP*
Stack grows from high addresses to lower

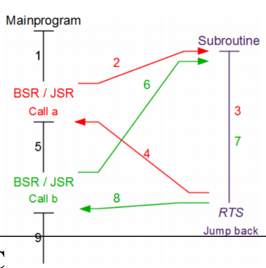
```
Stacksize: EQU $40
             :
DATA:      SECTION
TofStack:  DS Stacksize-1 ; reserve stack
BofStack:  DS 1
             :
PROGRAM:   SECTION
           LDHX #(BofStack+1) ; H:X := Bottom
               of Stack
           TXS                ; SP := HX -1
             :
           ; save CPU-Status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

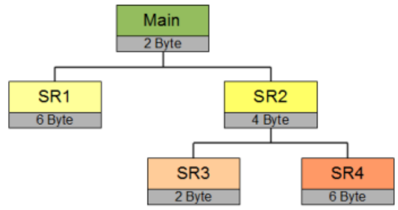
subroutines enable following:

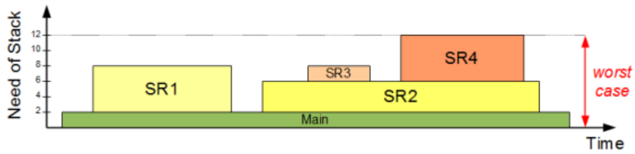
- **less memory usage**; repeated command sequences are stored only once
- **less development effort**; tested command sequences can be reused
- **less error prone**; enable modular way of building software
- **higher team productivity**; multiple people can work parallel on different code sections
-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

*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.

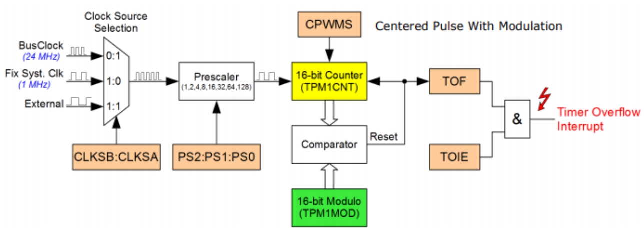




It is also possible to figure out the stack usage by filling the program-stack at the start with an bit pattern like 0xdeadbeef and stress test the program as much as possible. At the end, this will show which part and how much of the stack has been used during the program execution.

7 Timer and Interrupts

7.1 Modulo Counter



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

7.2 Modulo Frequency

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

- $\mathbf{T_{TOF}}$: Time between two Timer-Overflow events
- $\mathbf{MOD}$: Value of the Modulo set
- $\mathbf{PS}$: Presacler value
- $\mathbf{f_{Clk}}$: 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***

$$\mathbf{MOD} = (\frac{\mathbf{T_{TOF}} \cdot \mathbf{f_{Clk}}}{\mathbf{PS}}) - \mathbf{1}$$

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

CLKSB:CLKSA	TPM Clock Source to Prescaler Input
00	No clock selected (TPM counter disable)
01	Bus rate clock
10	Fixed system clock
11	External source

Table 16-4. Prescale Factor Selection

PS2:PS1:PS0	TPM Clock Source Divided-by
000	1
001	2
010	4
011	8
100	16
101	32
110	64
111	128

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:

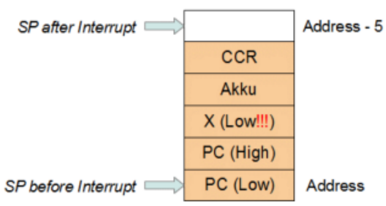
- **Interrupt** = Exception handling
  *enables **realtime capable (+)** systems (depends on interrupt **latency**). **Fast reaction time** through automatic reaction to events and interrupt of the program to execute an Interrupt-Service-Routine (ISR). Needs substancial effort for **state backup (-)**, 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.*
  esier to understand / debug (+)
  ***Waste of caclulation time (-)** if events occure rarely*

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

7.5 Interrupt execution

1. Interrupt called
2. Save current state onto stack
3. Call function
4. By Programming - clear interrupt flag
5. go back to code
6. load saved state from stack
7. keep running where stop before interrupt

7.6 Save Interrupt State



On entrance to an ISR the CPU-State is backed up automatically to the Stack.

*Note: The **H-Register** must be saved „manually“ on the HCS08 (only with Assembler)*

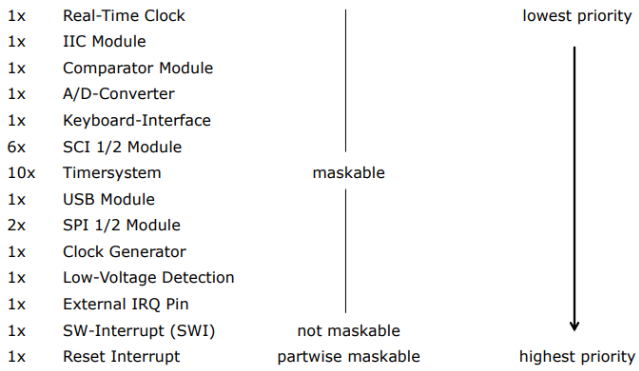
7.7 Difference ISR and Subroutines

ISR = Interrupt Service Routine / Interrupt Subroutine

	ISR	Unterprogramm
Call	spontaneous	BSR/JSR
State backup	automatic	Program (manual)
Return jump	RTI	RTS

7.8 Interrupt Sources Priority

In the MC9S08JM60 there are 29 Interrupt Sources, that are sorted by priority in the Interruptvector-Table

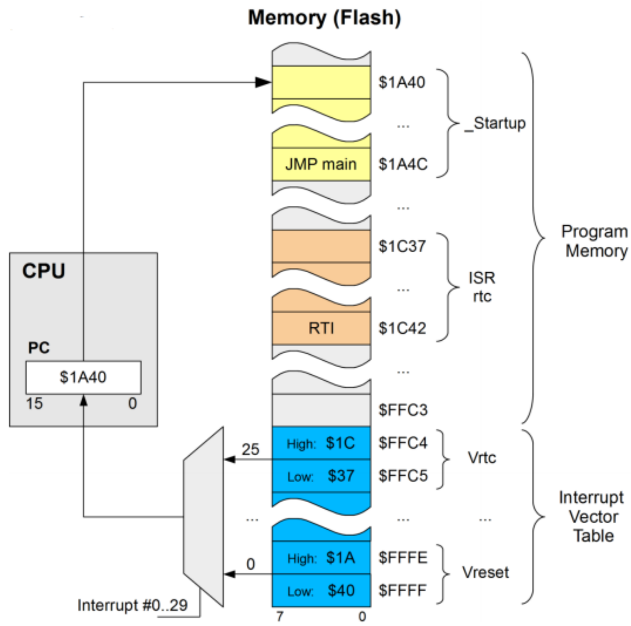


*By default the HCS08 does **not support nested Interrupts**, because the I-Flag gets set on an entrance into an ISR.*
If there are more Interrupt demands, the ISR with the highest priority (lowest vector number) is called first

7.9 Interrupt Counter

Setting the Interrupt Counter will set it always to 0. Reading one of the Counter 8 Bit, the other one will be saved to a shadow register until read from.

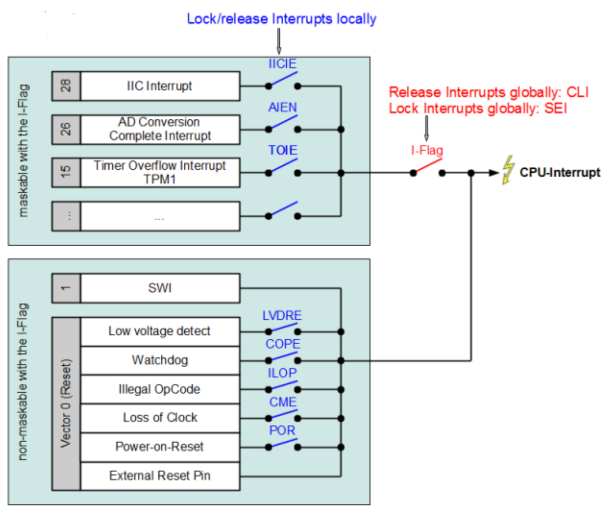
7.10 Interrupt Vectortable




```
// Extract out of .prm File
VECTOR ADDRESS 0xFFC4 ISR_RTI // RTC
VECTOR ADDRESS 0xFFC6 errISR_IIC // IIC
VECTOR ADDRESS 0xFFC8 errISR_ACMP // ACMP
VECTOR ADDRESS 0xFFCA errISR_ADC // ADC
     Conversion
...
VECTOR ADDRESS 0xFFDA motorBoosterISR // TPM2
     overflow
VECTOR ADDRESS 0xFFDC errISR_TPM2CH1 // TPM2
     channel 1
VECTOR ADDRESS 0xFFDE errISR_TPM2CH0 // TPM2
     channel 0
```

```
VECTOR ADDRESS 0xFFE0 errISR_TPM2O // TPM1
     overflow
VECTOR ADDRESS 0xFFE2 errISR_TPM1CH5 // TPM1
     channel 5
VECTOR ADDRESS 0xFFE4 errISR_TPM1CH4 // TPM1
     channel 4
VECTOR ADDRESS 0xFFE6 errISR_TPM1CH3 // TPM1
     channel 3
VECTOR ADDRESS 0xFFE8 errISR_TPM1CH2 // TPM1
     channel 2
VECTOR ADDRESS 0xFFEA errISR_TPM1CH1 // TPM1
     channel 1
VECTOR ADDRESS 0xFFEC ifrFrontISR // TPM1
     channel 0
```

7.11 Interrupt-Release Logic



7.12 Programming of Interrupts

Following is important for programming interrupts:

- *Define **Interruptvectors**; at the place of the Interruptvector has to be the start address of the ISR (in CW definition in .prm file)*
- *Define and initialise **Stack***
- ***Delete** the Interrupt-**Flags before** you release them, so that the Interrupt does not get fired right away.*
- *Programming of ISR; CPU-State gets backed up automatically (H-Register only through C-Compiler)*
- ***Delete** the Interrupt-**Flag** in the ISR*
- *End the ISR with **RTI** (is done automatically on usage of C-Compiler)*
- Release Interrupts globally (**CLI**) in the main program (typically after initialisation part)

```
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;

    // reset counter
    TPM1CNT = 0

    // 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;
}
```