
Kapitel 2

Architektur von Mikrocontrollern am Beispiel Freescale HCS12

2.1	Grunddaten der Microcontroller-Familie HCS12	2
2.2	Hello Embedded World	7
2.3	Registermodell, Datentypen und Adressierungsarten	15
2.4	Übersicht der wichtigsten Maschinenbefehle	22
2.5	Programmierbeispiele 1	32
2.6	Stack	35
2.7	Programmierbeispiele 2	38

Anhang: Freescale CodeWarrior HCS12 Entwicklungsumgebung

2.1 Grunddaten der Microcontroller-Familie HCS12

2.1 Grunddaten der Microcontroller-Familie HCS12

- **Von Neumann**-Architektur
- Complex Instruction Set (**CISC**)
- Datenwortbreite $n_{\text{DAT}} = \mathbf{16 \text{ bit}}$
- Adresswortbreite $n_{\text{ADR}} = 16 \text{ bit}$
- Kleinste adressierbare Einheit $n_{\text{min}} = 1 \text{ Byte}$
 - Adressraum $N = 2^{n_{\text{ADR}}} \cdot n_{\text{min}} = 2^{16} \text{ Byte} = 64 \text{ KB}$
Erweiterung durch Speicherbank-Umschaltung
- Befehle und Daten können bei beliebigen Adressen beginnen
- Mehrbyte-Werte: Speicherreihenfolge **Big Endian** (Most Significant Byte zuerst)
Bei Adressangaben genügt Angabe der Adresse des ersten Bytes (und der Länge)
Bsp.: 16bit Wert \$4433 ab Offsetadresse \$0103

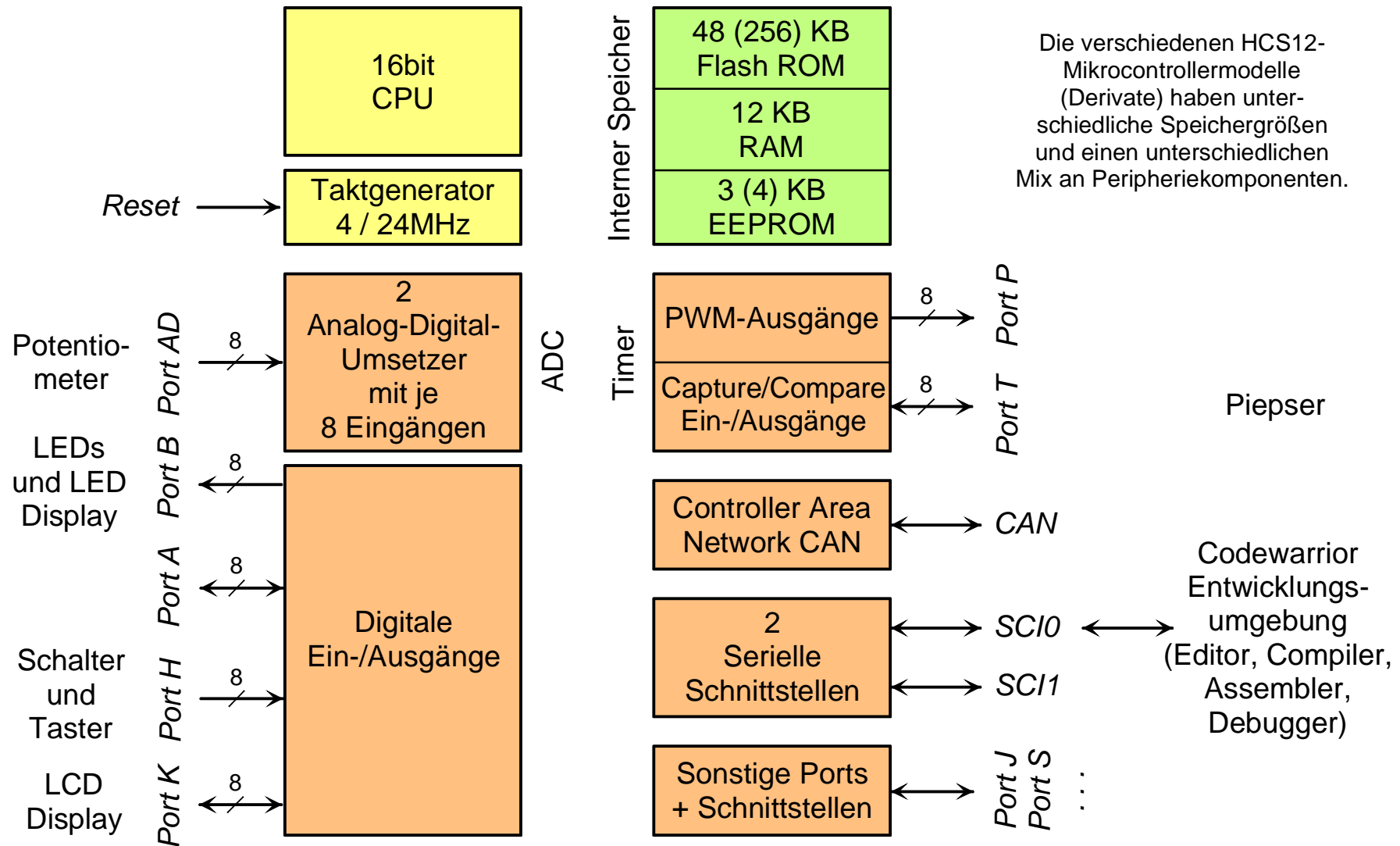
Adresse	Inhalt
0	...
...	...
\$0103	
\$0104	
...	...

Bei anderen Herstellern, z.B. Intel, Infineon, häufig auch Little Endian:
Niederwertiges Byte zuerst

*1 Motorola/Freescale verwendet für hexadezimale Zahlen die Darstellung \$. . . statt . . . h, d.h. \$4433 = 4433h

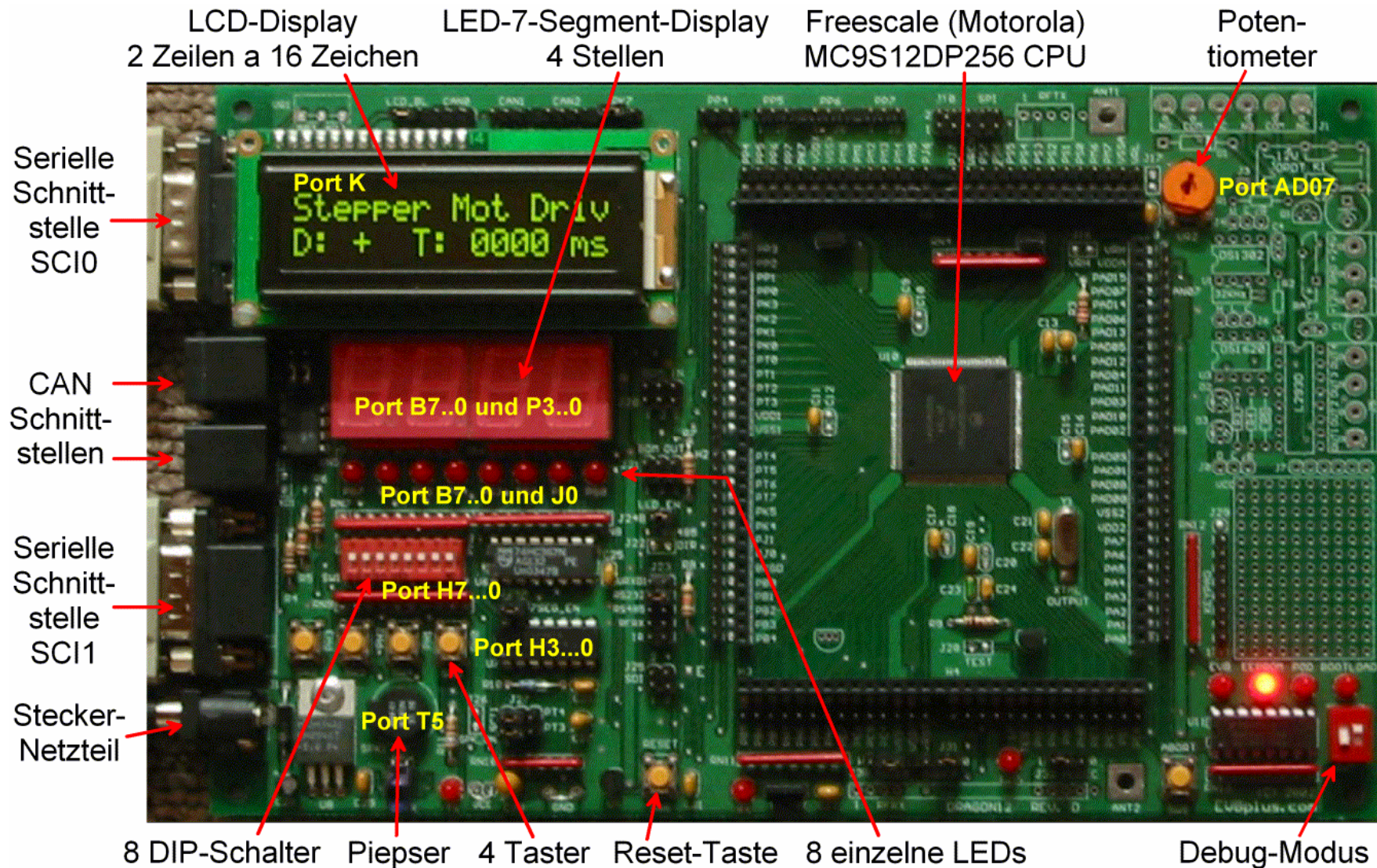
2.1 Grunddaten der Microcontroller-Familie HCS12

Blockschaltbild des Modells MC9S12DP256 (siehe [3.0 S. 16], [3.2 Fig. 1-1])



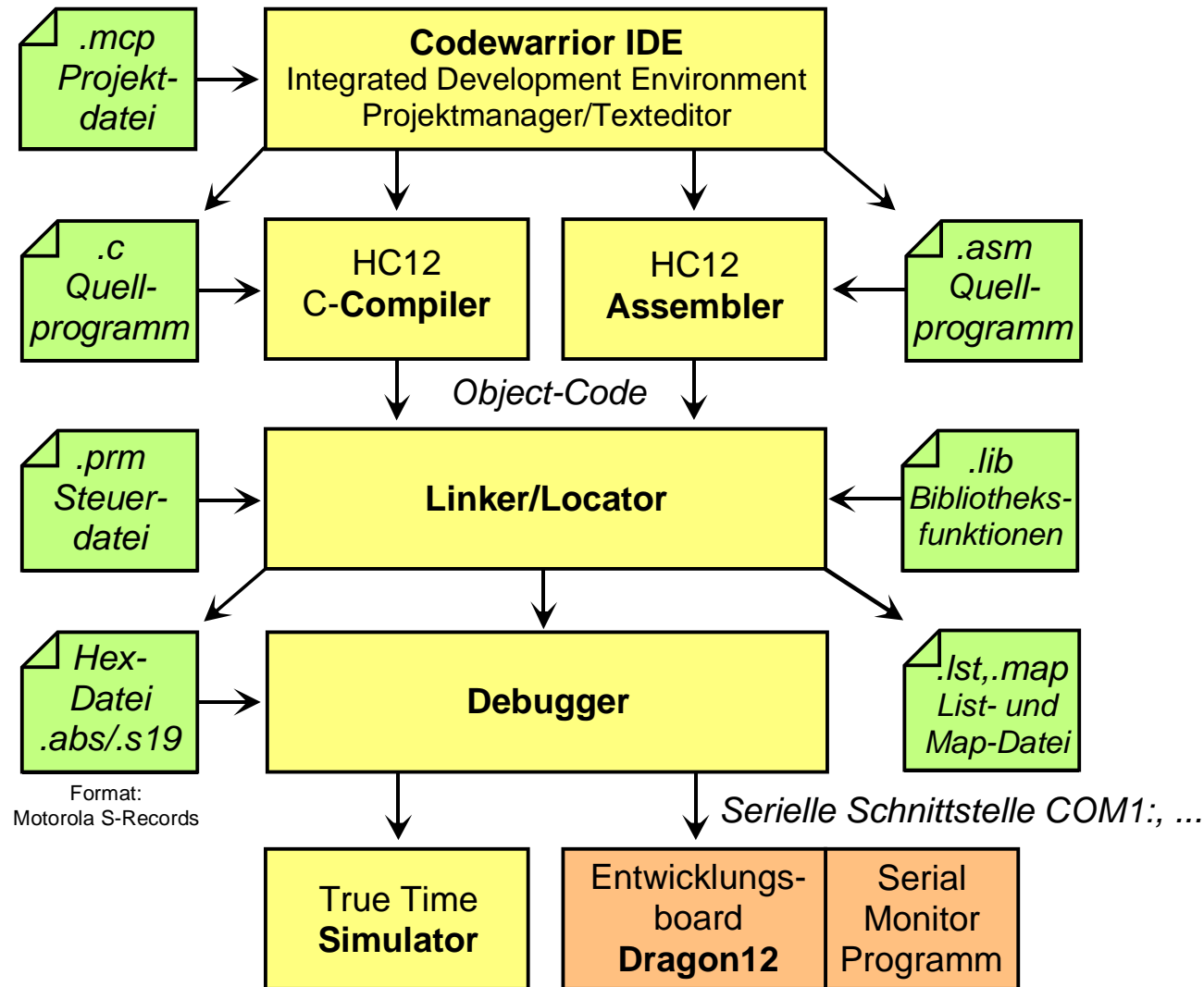
2.1 Grunddaten der Microcontroller-Familie HCS12

Dragon12-Entwicklungsboard (siehe [3.11] und [2.1 Anhang B])



2.1 Grunddaten der Microcontroller-Familie HCS12

Entwicklungsumgebung Metroworks Codewarrior (siehe [3.12 – 3.17])



2.1 Grunddaten der Microcontroller-Familie HCS12

Memory Map des Modells MC9S12DP256 (siehe [3.0 S. 120], [3.2 Fig. 1-2])

(Der Chip hat mehrere Betriebsmodi, im Labor wird der Normal Single Chip Mode ohne externen Speicher verwendet)

\$0000	Register zur Steuerung der On-Chip-Peripherie 1KB	Alle Peripheriebausteine werden Memory- Mapped adressiert
\$0400	EEPROM 3KB	Das EEPROM ist eigentlich 4KB groß, 1KB wird durch die Peripherieregister verdeckt.
\$1000	RAM 12KB	Stack für Monitor-Programm am Ende des RAM-Bereichs ca. 36B
\$4000	Flash-ROM 16KB	
\$8000	Flash-ROM 16KB	In diesem Bereich können alternativ weitere jeweils 16KB große Flash-ROM-Bereiche ein- geblendet werden (Page Window, Auswahl durch PPAGE-Register) → Speichererweite- rung auf > 64KB
\$C000	Flash-ROM 16KB	\$F780 ... \$FE00: Monitor-Programm für Debugger
\$FFFF		\$FF00 ... \$FFFF: Interrupt-Vektortabelle 256B

2.2 Hello Embedded World

2.2 Hello Embedded World

Seit Kernighan und Ritchie in ihrem Buch „The C Programming Language“ damit begonnen haben, ist das erste Programm, mit dem eine Programmiersprache eingeführt wird, traditionell ein „**Hello World**“-Programm, das einen kurzen Meldungstext auf den Bildschirm ausgibt. Da eingebettete Systeme in der Regel weder Tastatur noch Bildschirm haben, muss man die Idee etwas abwandeln und lässt dort einen „Ausgang wackeln“ (**Toggle Port**), an dem nach Möglichkeit eine Leuchtdiode angeschlossen ist, so dass sich eine **blinkende LED** ergibt.

Zur Lösung der Aufgabe mit dem Dragon12-Entwicklungsboard sind einige Schritte nötig:

Schritt 1: Informationen über die Hardware auf dem Entwicklungsboard

Wo auf dem Entwicklungsboard sind LEDs angeschlossen und wie sind diese anzusteuern?

Ausgangspunkt [3.11]:

ON-BOARD HARDWARE

Getting_started
_Dragon12.pdf S.11

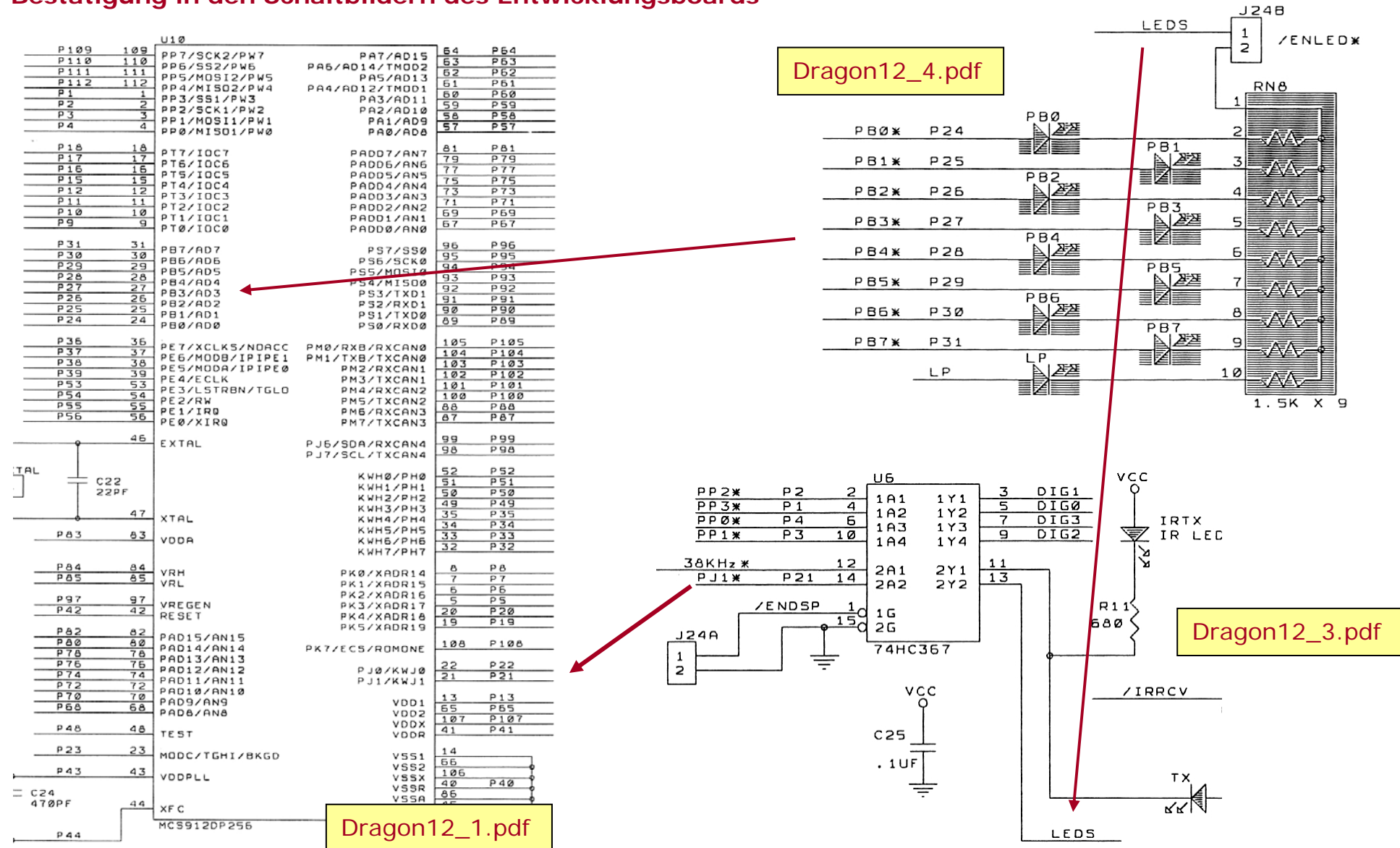
Each port B line is monitored by a LED. It works OK in single chip mode. In order to turn on port B LEDs, the PJ1 (pin 21 of MC9S12DP256) must be programmed as output and set for logic zero. If the board is used in expanded mode, the port B becomes the address/data bus, AD0-AD7, and the LEDs will add too much load on the bus. In order to make it work in expanded mode, J24A and J24B must be removed to disable the 7-segment LED display and the PB0-PB7 LEDs.

Port A is used as the 4X4 keypad interface in single chip mode, but in expanded mode, port A becomes the address/data bus AD8-AD15 and it cannot be connected with a keypad.

Port H is connected to an 8-position DIPswitch. The DIPswitch is connected to GND via the RN9 (eight 4.7K resistors), so it's not dead short to GND. When port H is programmed as an output port, the DIPswitch setting is ignored.

2.2 Hello Embedded World

Bestätigung in den Schaltbildern des Entwicklungsboards



2.2 Hello Embedded World

Schritt 2: Informationen über die Hardware des Mikrocontrollers

Wo liegen die Ports im Adressraum der CPU und wie sind sie zu programmieren?

HCS12 Doku 000-MC9S12DP256.pdf [3.0]: Register Map S.66ff und S.129ff Input/Output Registers

PORTB — Port B Register

Address Offset: \$0001

	Bit 7	6	5	4	3	2	1	Bit 0
Single Chip	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	Unaffected by reset							
Expanded & Periph:	ADDR7/ DATA7	ADDR6/ DATA6	ADDR5/ DATA5	ADDR4/ DATA4	ADDR3/ DATA3	ADDR2/ DATA2	ADDR1/ DATA1	ADDR0/ DATA0
Expanded narrow	ADDR7	ADDR6	ADDR5	ADDR4	ADDR3	ADDR2	ADDR1	ADDR0

Port B bits 7 through 0 are associated with address lines A7 through A0 respectively and data lines D7 through D0 respectively. When this port is not used for external addresses, such as in single-chip mode,

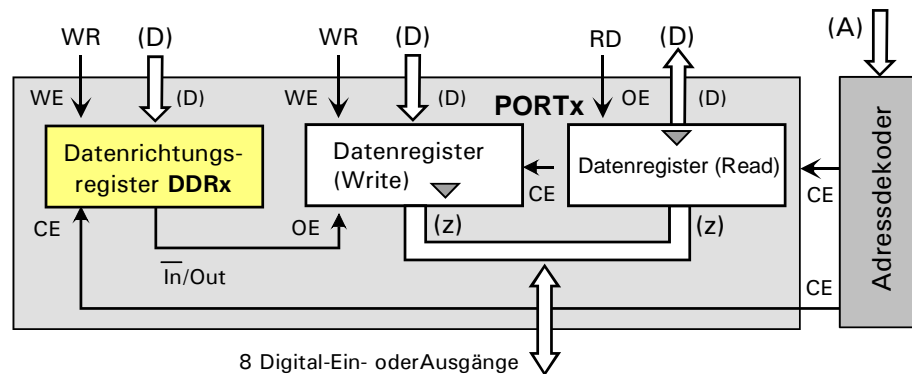
these pins can be used as general purpose I/O. Data Direction Register B (DDRB) determines the primary direction of each pin. DDRB also determines the source of data for a read of PORTB.

This register is not in the on-chip map in expanded and peripheral modes.

CAUTION:

To ensure that you read the value present on the PORTB pins, always wait at least two cycles after writing to the DDRB register before reading from the PORTB register.

Read and write: anytime (provided this register is in the map).



DDRB — Port B Data Direction Register

Address Offset: \$0003

	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	0	0	0	0	0	0	0	0

This register controls the data direction for Port B. When Port B is operating as a general purpose I/O port, DDRB determines the primary direction for each Port B pin. A "1" causes the associated port pin to be an output and a "0" causes the associated pin to be a high-impedance input. The value in a DDR bit also affects the source of data for reads of the corresponding PORTB register. If the DDR bit is zero (input) the buffered pin input is read. If the DDR bit is one (output) the output of the port data latch is read.

This register is not in the on-chip map in expanded and peripheral modes. It is reset to \$00 so the DDR does not override the three-state control signals.

Read and write: anytime (provided this register is in the map).

DDRB7–0 — Data Direction Port B

0 = Configure the corresponding I/O pin as an input

1 = Configure the corresponding I/O pin as an output

Analog: PORT J

Datenregister PTJ bei Adresse \$0268

Datenrichtungsregister DDRJ bei Adresse \$026A

2.2 Hello Embedded World

Schritt 3: Informationen über die Entwicklungsumgebung, Entwurf und Codierung

Wie schreibe und übersetze ich ein Programm?

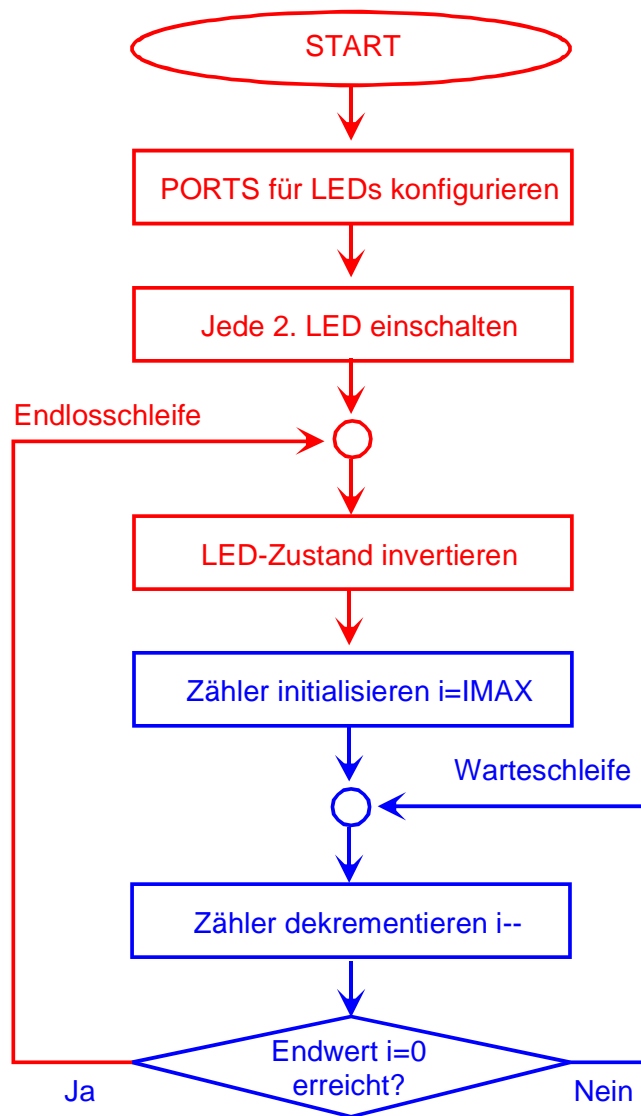
- Installation und Bedienung der Entwicklungsumgebung siehe **Anhang CodeWarrior**
- Damit der Programmierer nicht mit den hexadezimalen Adressen direkt umgehen muss, definiert die Entwicklungsumgebung in Include-Dateien Symbole für den Zugriff auf die Register bzw. auf die einzelnen Bits, z.B.

<i>Symbole in Include-Dateien</i>	<i>für C-Programme^{*1}</i> mc9s12dp256.h	<i>für Assembler-Programme</i> mc9s12dp256.inc
Port B: gesamter Port	#define PORTB (*(char*) 0x0001)	PORTB: equ \$0001
PORTB Bit 0	#define PORTB_BIT0 PORTB.Bits.BIT0	
...
DDRB	#define DDRB (*(char*) 0x0003)	DDRB: equ \$0003
DDRB Bit 0	#define DDRB_BIT0 DDRB.Bits.BIT0	
Port J: gesamter Port	#define PTJ (*(char*) 0x0268)	PTJ: equ \$0268
PTJ Bit 0	#define PTJ_PTJ0 PTJ.Bits.PTJ0	
...
DDRJ	#define DDRJ (*(char*) 0x026A)	DDRJ: equ \$026A
DDRJ Bit 0	#define DDRJ_DDRJ0 DDRJ.Bits.DDRJ0	

^{*1} Vereinfachte Darstellung, in Wirklichkeit sind die Ports über Strukturen und Unions von Bitfeldern bzw. Byte- oder Word-Datentypen definiert. Die Namensgebung der Ports ist leider nicht einheitlich (**PORTB** aber **PTJ**) und z.T. auch zwischen C und Assembler unterschiedlich.

2.2 Hello Embedded World

Entwurf



Codierung in C (CodeWarrior-Projekt **BlinkingLeds.mcp**)

```
#include <hidef.h>          //Common defines
#include <mc9s12dp256.h>    //CPU specific defines

#pragma LINK_INFO DERIVATIVE "mc9s12dp256b"

#define IMAX  200000L      //Delay count

long i;                   //Counter variable

void main(void)
{
    EnableInterrupts;      //Allow for debugger

    DDRJ_DDRJ1 = 1;        //Port J.1 as output
    PTJ_PTJ1   = 0;        //J.1=0 --> Activate LEDs

    DDRB = 0xFF;           //Port B as outputs
    PORTB = 0x55;          //Turn on any other LED

    for(;;)
    {
        PORTB = ~PORTB;    //Toggle LEDs

        for (i=IMAX; i > 0; i--)
            //Delay loop
    }
}
```

Beim Testen stellt man fest, dass das LED-7-Segment-Display ebenfalls an PORT B angeschlossen ist und abgeschaltet werden muss, wenn es nicht mitblinken soll: Port P.3...0 Ausgänge, P3...0= 1111_B

2.2 Hello Embedded World

Schritt 4: Informationen über die Testumgebung

Wie teste („debugge“) ich das Programm? → siehe **Anhang CodeWarrior**

The screenshot displays the True-Time Simulator & Real-Time Debugger interface. The main window shows the C source code for `main.c` at line 27. The code defines a `main` function that enables interrupts, configures I/O pins, and toggles LEDs in a loop. The assembly window shows the corresponding assembly code for the `main` function. The CPU Register window shows the current state of the CPU registers, including the Program Counter (PC) at address C058. The Memory window shows the current memory contents. The Data window shows the current state of the variables, including the `PORT` and `DDR` registers. The IO_Led window shows the current state of the LEDs, with the `PORT=AA` and `DDR=FF` configuration.

C-Programm

```
void main(void)
{
    EnableInterrupts;           //Allow interrupts for

    DDRJ_DDRJ1 = 1;             //Port J.1 as output
    PTJ_PTJ1 = 0;               //J.1=0 --> Activate LEDs

    DDRB = 0xFF;                //Port B.7...0 as outputs
    PORTB = 0x55;               //Turn on every second LED

    for(;;)
    {
        PORTB = ~PORTB;        //Toggle LEDs
    }
}
```

Assembler Programm

```
main
C056 ANDCC #239
C058 BSET 0x026A, #2
C05C BCLR 0x0268, #2
C060 LDAB #255
C062 STAB 0x03
C064 LDAA #85
C066 STAA 0x01
C068 BRA ++40 ; abs = C090
C06A COM 0x0001
```

CPU Register

HC12		CPU Cycles: 79		Auto	
D	0	A	0	B	0
IX	C099	IY	0		
IP	C058	PC	C058	PPAGE	0
SP	1101	CCR	SXHINZVC		

Variablen und Visualisierung

Speicher Inhalt

Address	Value	Comment
000000	00 AA 00 FF 00 00 00 00
000008	8F 00 00 00 90 00 01 00
000010	00 00 00 00 00 00 00 00
000018	00 00 00 10 00 00 40 F2
000020	00 00 00 00 00 00 00 00
000028	00 00 00 00 00 00 00 00
000030	00 00 00 00 00 00 00 00
000038	00 00 00 00 00 00 00 00
000040	00 00 00 00 00 00 00 00

IO_Led

PORT=AA DDR=FF

2.2 Hello Embedded World

Speicherplatzbedarf des C-Programms (siehe Datei `simulator.map`)

Summary of section sizes per section type:

READ_ONLY (R):	9B (dec: 155)	← ROM: Programmcode + konstante Daten
READ_WRITE (R/W):	104 (dec: 260)	← RAM: Variable Daten 4 Byte (+ Stack 256 Byte ^{*1})
NO_INIT (N/I):	23D (dec: 573)	← Peripherieregister (fest, unabhängig vom Programm)

Sieht man sich das Maschinenprogramm an, das der C-Compiler erzeugt hat, stellt man fest, dass bei besserer Optimierung ein kleineres und schnelleres Programm möglich wäre (siehe nächste Seite):

Summary of section sizes per section type:

READ_ONLY (R):	2A (dec: 42)	← ROM: Programmcode + konstante Daten
READ_WRITE (R/W):	100 (dec: 256)	← RAM: Variable Daten 0 Byte (+ Stack 256 Byte ^{*1})

. . . .

Ausführungsgeschwindigkeit (gemessen mit dem Simulator)

<i>Laufzeit in CPU-Takten</i>	<i>C-Programm</i>	<i>Assembler-Programm</i>
CPU Reset bis Programmzeile Toggle LEDs	99 Takte	19 Takte
Durchlauf der Schleife Toggle LEDs bis Toggle LEDs (für IMAX=1)	47 Takte	17 Takte

^{*1} Der Stack-Bereich könnte beim C-Programm verkleinert werden, beim Assemblerprogramm könnte er ganz wegfallen, da er hier überhaupt nicht genutzt wird.

→ Zum **Verständnis, wie Hochsprachenprogramme** in Embedded Systemen **optimiert werden können/müssen** (und zur Fehlersuche in Embedded System Compilern), **ist die Kenntnis der Assembler-Programmebene notwendig.**

2.2 Hello Embedded World

Blinkende Leuchtdioden in optimiertem Assembler (CodeWarrior-Projekt **BlinkingLedsAsm.mcp**)

```
XDEF      Entry, main          ; Export symbols
XREF      __SEG_END_SSTACK     ; Import symbols: End of stack
INCLUDE   'mc9s12dp256.inc'    ; include derivative specific macros

IMAX: EQU 2048                  ; Symbolic constant: Delay count

.data:    SECTION              ; RAM: Variable data section (not used in this program)

.const:    SECTION             ; ROM: Constant data (not used in this program)

.init:     SECTION             ; ROM: Code section
main:      ; Begin of the program
Entry:     LDS  #__SEG_END_SSTACK ; Initialize stack pointer
           CLI                      ; Enable interrupts, needed for debugger

           BSET DDRJ, #2          ; Bit Set:          Port J.1 as output
           BCLR PTJ,  #2          ; Bit Clear:        J.1=0 --> Activate LEDs

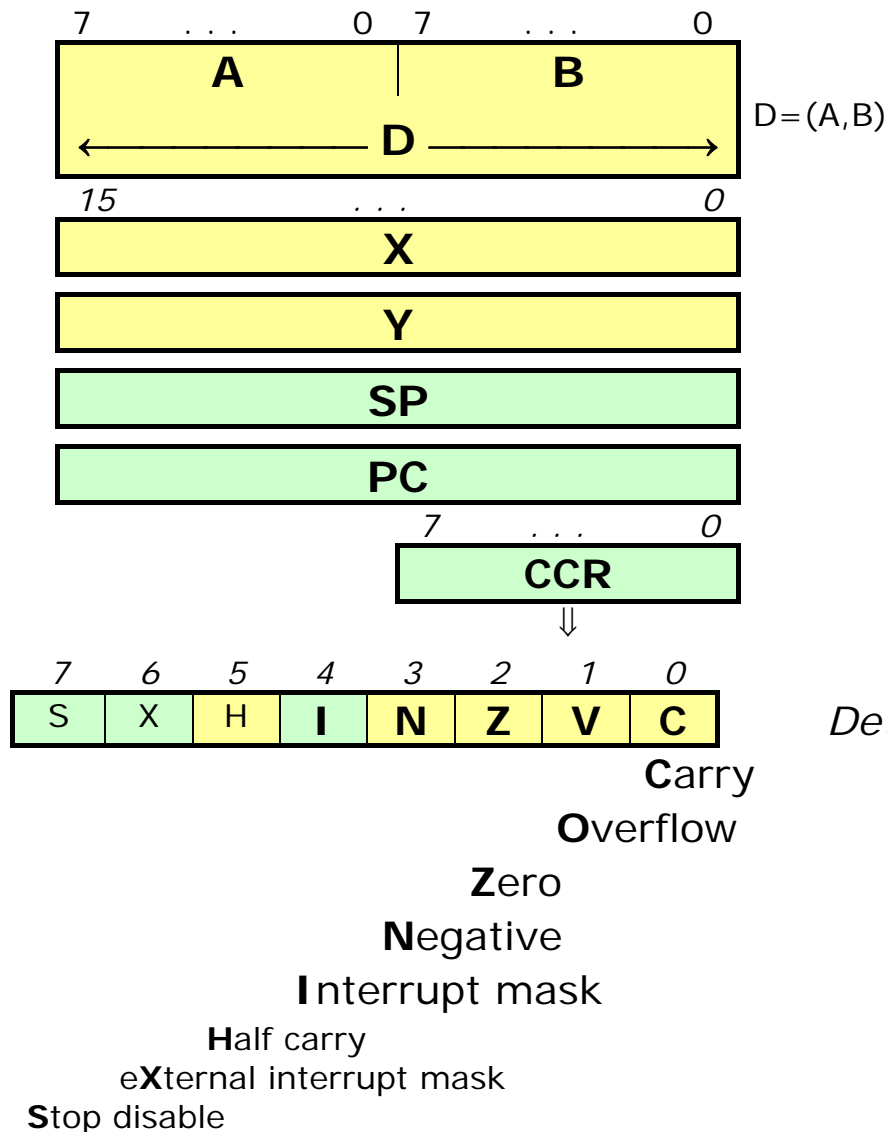
           MOVB #$FF, DDRB        ; $FF -> DDRB:      Port B.7...0 as outputs (LEDs)
           MOVB #$55, PORTB       ; $55 -> PORTB:    Turn on every other LED

loop:      COM  PORTB            ; Complement Port B:Toggle LEDs

           LDX  #IMAX             ; Delay loop to control toggle Frequency
wait0:     LDY  #IMAX            ; (Uses two nested counter loops with registers X and Y)
waitI:     DBNE Y, waitI         ; --- Decrement Y and branch to waitI if not equal to 0
           DBNE X, wait0         ; --- Decrement X and branch to wait0 if not equal to 0
           BRA  loop             ; Branch to loop
```

2.3 Registermodell, Datentypen, Adressierungsarten

2.3 Registermodell (für den Programmierer sichtbare Register, siehe [3.1 Kap. 2])



Akkumulator

Als 16 bit Register D oder als zwei 8 bit Register A, B für arithmetische und logische Operationen

Indexregister X

Daten und deren Adressierung

Indexregister Y

Daten und deren Adressierung

Stackpointer SP

Adressierung des Stacks

Programmzähler PC

Adressierung des Codes

Condition Code Register

Statusbits bei arithmetischen Operationen und Steuerbits

Details zum CCR-Register

Übertrag bei Operationen mit BCD-Zahlen
Sperren externer Unterbrechungssignale (Reset: X=1)
Ignorieren des Stop-Befehls (nach Reset: S=1)

2.3 Registermodell, Datentypen, Adressierungsarten

Datentypen

	<i>HCS12</i>		<i>80x86</i>
	<i>Assembler</i> ^{*1}	<i>CodeWarrior C</i> ^{*1}	<i>Visual C++</i>
<ul style="list-style-type: none"> Betragszahlen: Natürliche Zahlen (unsigned) 2er-Komplementzahlen: Ganze Zahlen (signed) 			
8bit -128 ... +128 0 ... 255	DC.B, DS.B ^{*3}	char unsigned char	
16bit -32768 ... +32768 0 ... 65535	DC.W, DS.W ^{*3}	short, int unsigned short, unsigned int	short unsigned short
32bit -2147483648 ... +2147483647 0 ... 4294967295	DC.L, DS.L ^{*3}	long unsigned long	int, long unsigned int, long
<ul style="list-style-type: none"> Gleitkommazahlen 			
IEEE 32bit	-	float, double ^{*1}	float
IEEE 64bit	-	(double) ^{*1}	double
<ul style="list-style-type: none"> Adressen/Pointer (unabhängig vom Datentyp) 	16bit (near pointer)	16bit (near pointer)	32bit
Bitfelder	1bit	8, 16 oder 32bit	32bit
Enumerationen	-	16bit	32bit
Array	^{*3}	datentyp name[anzahl]	
Strukturen, Unions	-	struct, union	

2.3 Registermodell, Datentypen, Adressierungsarten

Darstellung von Zahlen und Zeichenkonstanten

	<i>HCS12 Assembler</i>	<i>C</i>
Dezimalzahl (Basis 10)	-34, 127	
Hexadezimalzahl (Basis 16)	\$3F8A , -\$3F	0x3F8A
Oktalzahl (Basis 8)	@7345	
Dualzahl (Basis 2)	%10101001	0b10101001
Gleitkommazahl	-	3.14159 , 1.6e-19
ASCII-Zeichen	'Z'	'Z'
ASCII-Z-String ^{*4}	"Dies ist ein String", 0	"Dies ist ein String"

*1 Beim Metroworks C-Compiler lässt sich die Bitgröße der Datentypen durch Compiler-Optionen einstellen.

*2 In Assembler wird bei den Datentypen nicht nach **signed** und **unsigned** unterschieden.

*3 Variablen im RAM-Speicher: **name: DS.B anzahl**
Reserviert eine oder mehrere 8bit Variable im RAM-Speicher, die mit dem Variablennamen **name** angesprochen werden kann. Die Variable wird **nicht** initialisiert. Wenn **anzahl** > 1 ist, wird ein Array reserviert. Mit **DS.W** bzw. **DS.L** werden 16bit bzw. 32bit Variable bzw. Arrays reserviert.

Konstante im ROM-Speicher: **name: DC.B wert**
Definiert eine 8bit Konstante im ROM-Speicher, die mit dem Namen **name** angesprochen werden kann und mit dem Wert **wert** initialisiert wird. Mit **DC.W** bzw. **DC.L** werden 16bit bzw. 32bit Konstante reserviert.

Mit **name: DCB.B anzahl, wert**
wird ein kompletter Konstantenblock mit **anzahl** Bytes definiert und jedes Byte mit **wert** initialisiert. Analog **DCB.W** bzw. **DCB.L**.

*4 In C wird an den String automatisch ein 0-Byte angehängt (ASCII Zero String), um das Ende des Strings zu kennzeichnen. In Assembler muss das 0-Byte gegebenenfalls explizit angegeben werden.

2.3 Registermodell, Datentypen, Adressierungsarten

Adressierungsarten (siehe [3.1 Kap. 3], [1.4 Kap. 4], [2.1 Kap. 2.7])

Der HCS12 ist eine **Zwei-Adress-CPU**, d.h. ein Maschinenbefehl kann maximal zwei Operanden haben, wobei einer der beiden Operanden (der Ziel-Operand) durch das Ergebnis der Operation überschrieben wird:

Register-Operand

(Explizite) Register-Adressierung	INST reg Register werden explizit als Operanden angegeben. Wird nur bei wenigen Befehlen verwendet, meist werden Register implizit adressiert.	
Beispiel:		Kopiere Inhalt von Reg. D nach Reg. X

Implizite (Register) Adressierung Freescale-Bezeichnung: Inherent INH	INST Der Operand (meist eines der Register A, B, D, X, Y, SP) wird nicht explizit angegeben, sondern ist implizit in der Bezeichnung des Befehls enthalten.	
Beispiel:		Inkrementiere den Inhalt von X

Konstante als Operand

Unmittelbare Adressierung Immediate IMM	INST const Operand steht als Konstante unmittelbar im Befehl. Wird bei Freescale durch #... markiert, z.B. #20, #-20, #\$0A, #%01101011	
Beispiel:		Lade Konstante B010 _h in Register D

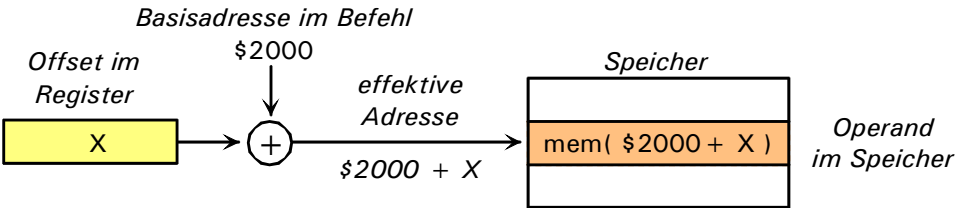
2.3 Registermodell, Datentypen, Adressierungsarten

Speicher-Variable als Operand

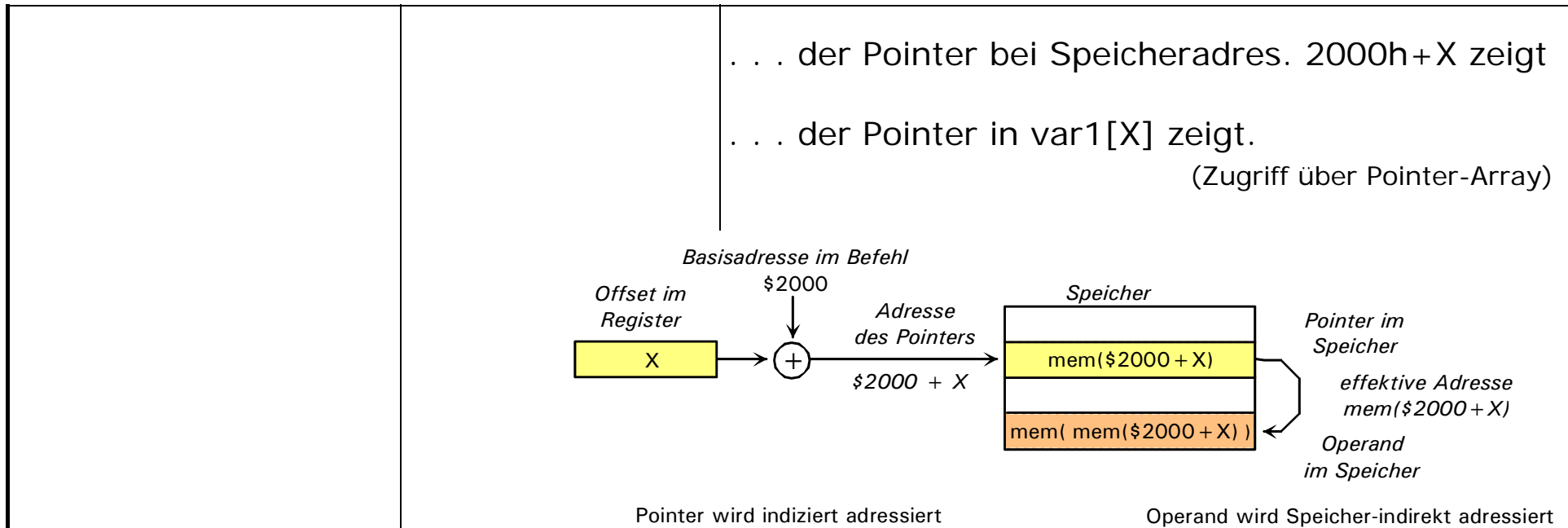
Direkte Adressierung DIR 8bit, EXT 16bit Adresse	INST adresse Speicheradresse steht direkt im Befehl. In der Regel wird beim Programmieren statt der Adresse der Name einer Variablen angegeben.	
Beispiele:		Lade Y mit dem Inhalt der Speicheradresse 2000 _h Lade Y mit dem Inhalt der Variable var1 Lade Y mit dem Adresse der Variable var1

Indirekte Adressierung in verschiedenen Varianten (Motorola/Freescale-Bezeichnung: Indexed)		
Register-indirekt ... Indexed IDX	INST 0, reg_{X,Y,SP} Speicheradresse steht im Register X, Y, SP, d.h. Register als Pointer	
Beispiel:		Lade das Register D mit dem Inhalt der Speicheradresse in X (indirekte Adressierung)
... mit Prä- oder Post-Inkrement bzw. -Dekrement Auto Increment IDX	INST const 1,...,+8, {+ -}reg_{X,Y,SP} INST const 1,...,+8, reg_{X,Y,SP}{+ -} Der Pointer im Register X, Y, SP kann vor oder nach der Berechnung der Adresse um einen konstanten Wert im Bereich 1, ... , 8 inkrementiert oder dekrementiert werden.	
		Lade den Inhalt der Speicheradresse in X ins Register D dekrementiere X vorher um 2 ... , inkrementiere X danach um 4

2.3 Registermodell, Datentypen, Adressierungsarten

<p>... mit Index (Offset)</p> <p>Indexed</p> <p>IDX 5bit-Konstante</p> <p>IDX1 9bit-Konstante</p> <p>IDX2 16bit-Konstante</p>	<p>INST const, reg_{X,Y,SP,PC} Adresse = const + reg_{X,Y,SP,PC}</p> <p>INST reg_{A,B,D}, reg_{X,Y,SP,PC} Adresse = reg_{A,B,D} + reg_{X,Y,SP,PC}</p> <p>Speicheradresse wird aus einer Konstante bzw. dem Inhalt des Registers A, B oder D plus dem Inhalt des Registers X, Y, SP oder PC (Index) gebildet.</p>
<p>Beispiel:</p>	<p>Lade Y mit dem Inhalt der Adresse 2000_h + X</p>  <p>Lade Y mit dem Inhalt von var1[X], d.h. dem Inhalt der Adresse von var1 + X (Zugriff auf Array)</p> <p>Lade Y mit dem Inhalt der Speicheradresse D + X</p>
<p>Speicher-indirekt ... mit Index</p> <p>Indexed-Indirect</p> <p>[IDX2]</p>	<p>INST [const, reg_{X,Y,SP,PC}]</p> <p>INST [D, reg_{X,Y,SP,PC}]</p> <p>Speicheradresse des Operanden steht in einem Pointer im Speicher. Die Adresse des Pointers wird aus einer Konstanten bzw. dem Inhalt des Registers D (Achtung A, B hier nicht erlaubt!) plus dem Inhalt des Registers X, Y, SP oder PC gebildet.</p>
<p>Beispiel:</p>	<p>Lade Y mit dem Inhalt der Adresse, auf die . . . der Pointer bei Adresse D+X zeigt</p>

2.3 Registermodell, Datentypen, Adressierungsarten



Bei Verzweigungsbefehlen (Sprung, Branch) wird die sogenannte **relative Adressierung** (Motorola/ Freescale-Bezeichnung REL) verwendet. Dabei wird die Zieladresse des Sprungs aus dem augenblicklichen Wert des Programmzählers und einem im Befehl als Konstante enthaltenen Offset gebildet, d.h. faktisch handelt es sich um eine indizierte Register-indirekte Adressierung. Der Programmierer muss sich damit nicht direkt auseinandersetzen, sondern verwendet als Zieladresse einen symbolische Marke (Label):

```
start: . . .  
      BRA start
```

2.4 Grundbefehlssatz

2.4 Übersicht der wichtigsten Maschinenbefehle (siehe [2.1 Kap. 2.9], [3.1 Kap.5, 6])

- Daten-Transportbefehle (inklusive Stack)
- Arithmetische und logische Befehle
- Vergleiche und Programmverzweigungen (inklusive Software-Interrupts)
- Sonstige Befehle

Abkürzungen

reg_{A,B,D} . . .	Eines der Register A , B , D , . . .
mem	Operand im Speicher mit beliebiger Speicheradressierung (direkt, indiziert, indiziert-indirekt)
imm	unmittelbarer Operand
mem_i	mem oder imm
adr	Adresse im Programmcode, relativ zum PC adressiert
LD{AA AB ... S}	Abkürzung für LDAA , LDAB oder LDS
8bit bzw. 16bit	Als Index: Größe eines Operanden

Soweit nicht anders angegeben, setzen alle folgenden Maschinenbefehle die Statusbits N, Z, V, C im CCR-Register in Abhängigkeit vom Ergebnis des Befehls, so dass bedingte Sprungbefehle häufig direkt ohne zusätzlichen Vergleichsbefehl verwendet werden können.

2.4 Grundbefehlssatz

Transportbefehle

(nur die LD... und ST... Befehle beeinflussen die Statusbits N, Z, V, C)

LD{AA AB D X Y S} mem_i	mem_i → reg_{A,B,D,X,Y,SP}	LoaD register from memory A, B werden mit einem 8bit, D, X, Y, SP mit einem 16bit Wert geladen
ST{AA AB D X Y S} mem	reg_{A,B,D,X,Y,SP} → mem	STore register to memory
TFR reg_{A,B,D,X,Y,SP,CCR} , reg_{A,B,D,X,Y,SP,CCR} *1	reg → reg	TransFeR register to register Wenn das Quellregister 8bit, das Zielregister 16bit breit ist, wird das MS-Byte mit dem Vorzeichen aufgefüllt (Sign Extension). Umgekehrt wird nur das LSByte kopiert.
EXG reg_{A,B,D,X,Y,SP,CCR} , *1 reg_{A,B,D,X,Y,SP,CCR}	reg ↔ reg	EXchanGe register Austauschen der Registerinhalte
TAB, TBA TSX, TSY, TXS, TYS TAP, TPA XGDX, XGDY	A → B bzw. B → A SP → X , SP → Y , X → SP , Y → SP A → CCR , CCR → A D ↔ X , X ↔ D	Alternativen zu TFR bzw. EXG (kürzere Befehle)
MOVB mem_i, mem MOVW mem_i, mem *1	mem_i → mem 8bit mem_i → mem 16bit (Adressierungsart [IDX] nicht möglich)	MOVE Byte MOVE Word Kopieren im Speicher
SEX reg_{A,B,CCR}, reg_{D,X,Y,SP} *1	reg_{A,B,C} → reg_{D,X,Y,SP}	Sign EXtension Copy Vorzeichenrichtige Erweiterung von 8bit auf 16bit für 2er-Komplement-Zahlen (wie TFR)

*1 Diese Befehle beeinflussen die Statusbits N, Z, V, C nicht.

2.4 Grundbefehlssatz

Berechnen einer indizierten oder indirekten Adresse (effektive Adresse)

LEA {X Y S} mem *1	Adresse von mem → reg_{X,Y,SP} (Adressierungsart DIR nicht sinnvoll)	Load Effective memory Address into register Berechnung einer Adresse zur Laufzeit und Laden in ein Register
------------------------------	---	--

Stack (siehe Kap 2.6)

PSH [A B C] *1	SP-1→SP, reg_{A,B,CCR}→Stack	PuSH register to stack Registerinhalt auf den Stack kopieren
PSH {D X Y} *1	SP-2→SP, reg_{D,X,Y}→Stack	
PUL [A B C] *1	Stack→reg_{A,B,CCR}, SP+1→SP	Pull register from stack Registerinhalt vom Stack kopieren
PUL {D X Y} *1	Stack→reg_{D,X,Y}, SP+2→SP	

*1 Diese Befehle (außer **PULC**) beeinflussen die Statusbits N, Z, V, C nicht.

Hinweis:

Die **PSH...** und **PUL...** Befehle können mit **ST...** und **LD...** „simuliert“ werden:

z.B.:

STAA 1, -SP	=	PSHA
STD 2, -SP	=	PSHD
LDAA 1, SP+	=	PULA
LDD 2, SP+	=	PULD

Häufig eingesetzt, um den Stackpointer **SP** zu verändern, ohne Daten zu kopieren:

LEAS +n, SP	„Abräumen“ von n Byte vom Stack
LEAS -n, SP	Platz reservieren für n Byte auf dem Stack

2.4 Grundbefehlssatz

Arithmetische und logische Befehle

Addieren, Subtrahieren, Inkrementieren, Vorzeichenumkehr

AB{A X Y} SBA	$B + A \rightarrow A, B + X \rightarrow X, B + Y \rightarrow Y$ $A - B \rightarrow A$	ADD/SuBtrahiere (A, B are loaded with a 8bit, D, X, Y are loaded with a 16bit value)
ADD{A B D} mem_i SUB{A B D} mem_i	$\text{reg}_{A,B,D} + \text{mem_i} \rightarrow \text{reg}_{A,B,D}$ $\text{reg}_{A,B,D} - \text{mem_i} \rightarrow \text{reg}_{A,B,D}$	ADD 8bit \pm 8bit oder 16bit \pm 16bit SUBtract
ADC{A B} mem_i _{8bit} SBC{A B} mem_i _{8bit} (ADC, SBC nicht mit D möglich)	$\text{reg}_{A,B} + \text{mem} + C \rightarrow \text{reg}_{A,B}$ $\text{reg}_{A,B} - \text{mem} - C \rightarrow \text{reg}_{A,B}$	ADd with Carry 8bit SuBtract with Carry 8bit
INC mem _{8bit} IN{CA CB X Y S} *1 DEC mem _{8bit} DE{CA CB X Y S} *1 (INC, DEC nicht mit D möglich)	$\text{mem} + 1 \rightarrow \text{mem}$ $\text{reg}_{A,B,X,Y,S} + 1 \rightarrow \text{reg}_{A,B,X,Y,S}$ $\text{mem} - 1 \rightarrow \text{mem}$ $\text{reg}_{A,B,X,Y,S} - 1 \rightarrow \text{reg}_{A,B,X,Y,S}$	INCrement memory 8bit INcrement register DECrement memory 8bit DEcrement register
CLR mem _{8bit} CLR{A B} (CLR nicht mit D möglich)	$0 \rightarrow \text{mem}$ $0 \rightarrow \text{reg}_{A,B}$	CLeaR byte (Laden mit 0)
NEG mem _{8bit} NEG{A B} (NEG nicht mit D möglich)	$-\text{mem} \rightarrow \text{mem}$ $-\text{reg}_{A,B} \rightarrow \text{reg}_{A,B}$	NEGate byte 2er-Komplement-Bildung (Multiplikation mit -1, Vorzeichenumkehr)

*1 INS und DES beeinflussen die Statusbits N, Z, V, C nicht.

2.4 Grundbefehlssatz

Bitweise logische Operationen

COM $\text{mem}_{8\text{bit}}$ COM{A B}	$\text{/mem} \rightarrow \text{mem}$ $\text{/reg}_{A,B} \rightarrow \text{reg}_{A,B}$	COMplement 1er-Komplement-Bildung (Bitweises NOT)
AND{A B} $\text{mem_i}_{8\text{bit}}$ ANDCC $\text{imm}_{8\text{bit}}$	$\text{reg}_{A,B} \text{ AND } \text{mem_i} \rightarrow \text{reg}_{A,B}$ $\text{CCR AND } \text{imm} \rightarrow \text{CCR}$	Bitweises AND
ORA{A B} $\text{mem_i}_{8\text{bit}}$ ORCC $\text{imm}_{8\text{bit}}$	$\text{reg}_{A,B} \text{ OR } \text{mem_i} \rightarrow \text{reg}_{A,B}$ $\text{CCR OR } \text{imm} \rightarrow \text{CCR}$	Bitweises OR
EOR{A B} $\text{mem_i}_{8\text{bit}}$	$\text{reg}_{A,B} \text{ XOR } \text{mem_i} \rightarrow \text{reg}_{A,B}$	Bitweises Exclusive OR

Bitbefehle

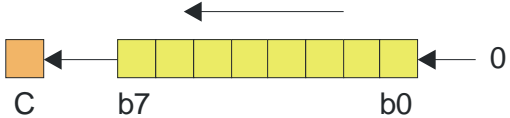
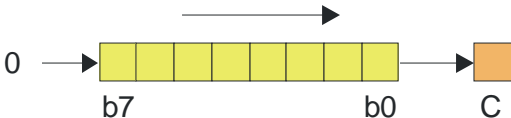
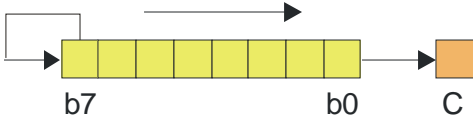
CLC, SEC CLV, SEV	$0 \rightarrow C, 1 \rightarrow C$ $0 \rightarrow V, 1 \rightarrow V$	CLeaR/SEt Carry bit in CCR CLeaR/SEt oVerflow bit in CCR
BCLR $\text{mem}_{8\text{bit}}, \text{imm}$ BSET $\text{mem}_{8\text{bit}}, \text{imm}$	$\text{mem AND /imm} \rightarrow \text{mem}$ $\text{mem OR imm} \rightarrow \text{mem}$	Bit CleaR 8bit Bit SET 8bit

Multiplizieren, Dividieren

MUL EMUL, EMULS	$A \times B \rightarrow D$ unsigned $D \times Y \rightarrow (Y, D)$ unsigned/signed	MULTiPLY 8bit x 8bit \rightarrow 16bit 16bit x 16bit \rightarrow 32bit
IDIV, IDIVS EDIV, EDIVS FDIV	$D / X \rightarrow X, \text{Rest in } D$ $(Y, D) / X \rightarrow Y, \text{Rest in } D$ unsigned/signed $D \times 2^{16} / X \rightarrow X, \text{Rest in } D$	DIVide 16bit / 16bit \rightarrow 16bit 32bit / 16bit \rightarrow 16bit "Pseudo 32bit" / 16bit \rightarrow 16bit

2.4 Grundbefehlssatz

Schieben und Rotieren

LSL $\text{mem}_{8\text{bit}}$ LSL {A B D} ASL $\text{mem}_{8\text{bit}}$ ASL {A B D}	$\text{mem} \ll 1 \rightarrow \text{mem}$ 8bit $\text{reg}_{A,B,D} \ll 1 \rightarrow \text{reg}_{A,B,D}$ 	Logical Shift Left Arithmetic Shift Left Linksschieben für Betrags oder 2er-Komplement-Zahlen um 1bit MSB landet im Carry-Bit des CCR LSB wird mit 0 gefüllt
LSR $\text{mem}_{8\text{bit}}$ LSR {A B D}	$\text{mem} \gg 1 \rightarrow \text{mem}$ 8bit $\text{reg}_{A,B} \gg 1 \rightarrow \text{reg}_{A,B,D}$ 	Logical Shift Right Rechtsschieben für Betragszahlen um 1bit. LSB landet im Carry-Bit des CCR MSB wird mit 0 gefüllt
ASR $\text{mem}_{8\text{bit}}$ ASR {A B} (ASR nicht mit D möglich)	$\text{mem} \gg 1 \rightarrow \text{mem}$ 8bit $\text{reg}_{A,B} \gg 1 \rightarrow \text{reg}_{A,B,D}$ (MSB=Vorzeichen bleibt unverändert) 	Arithmetic Shift Right Rechtsschieben für 2er-Komplement-Zahl um 1bit. LSB landet im Carry-Bit des CCR. MSB wird kopiert.
ROL $\text{mem}_{8\text{bit}}$ ROL {A B} (ROL nicht mit D möglich)	$\text{mem} \ll 1 \rightarrow \text{mem} + c$ 8bit $\text{reg}_{A,B} \ll 1 \rightarrow \text{reg}_{A,B} + c$	ROtate Left Rotiere links. LSB wird mit Carry-Bit gefüllt, MSB landet im Carry-Bit.
ROR $\text{mem}_{8\text{bit}}$ ROR {A B} (ROR nicht mit D möglich)	$\text{mem} \gg 1 \rightarrow \text{mem} + c * 8$ $\text{reg}_{A,B} \gg 1 \rightarrow \text{reg}_{A,B} + c * 8$ (nicht mit D möglich!)	ROtate Right Rotiere rechts um 1bit. MSB wird mit Carry-Bit gefüllt, LSB landet im Carry-Bit.

2.4 Grundbefehlssatz

Vergleiche und Programmverzweigungen

Vergleiche

CBA CMP {A B} mem_i _{8bit} CP {D X Y S} mem_i _{16bit}	Berechnet A - B Berechnet reg_{A,B} - mem_i Berechnet reg_{D,X,Y,SP} - mem_i	Compare Setzt Bits in CCR
TST mem _{8bit} TST {A B}	Berechnet mem - 0 Berechnet reg_{A,B} - 0	Test ob Operand null oder negativ ist, setzt Bits in CCR
BIT {A B} mem _{8bit}	Berechnet reg_{A,B} AND mem_i	Bit Test Wie AND, setzt aber nur Bits in CCR

Unbedingte und bedingte Sprungbefehle

(Sprungbefehle ändern die Statusbits N, Z, V, C nicht)

JMP mem	mem → PC	JuMP Wie {L}BRA, aber Ziel auch indirekt/indiziert adressierbar
{L}BRA adr	adr → PC	BRanch Always
{L}BRN adr	kein Sprung, d.h. No OPeration	BRanch Never
{L}BCC adr	adr → PC , wenn C=0	Branch if Carry Clear
{L}BCS adr	... wenn C=1	Branch if Carry Set
{L}BNE adr	... wenn Z=0	Branch if Not Equal
{L}BEQ adr	... wenn Z=1	Branch if Equal
{L}BPL adr	... wenn N=0	Branch if Plus (positive)
{L}BMI adr	... wenn N=1	Branch if Minus (negative)
{L}BVC adr	... wenn V=0	Branch if Overflow Clear
{L}BVS adr	... wenn V=1	Branch if Overflow Set

2.4 Grundbefehlssatz

{L}BGT adr	adr → PC wenn . . . >	Branch if GreaTer
{L}BGE adr	>=	Branch if Greater or Equal
{L}BEQ adr	=	Branch if Equal
{L}BLE adr	<=	Branch if Less or Equal
{L}BLT adr	<	Branch if Less nach Vergleich von 2er-Komplement-Zahlen
{L}BHI adr	adr → PC wenn . . . >	Branch if Higher
{L}BHS adr	>=	Branch if High or Same
{L}BEQ adr	=	Branch if Equal
{L}BLS adr	<=	Branch if Less or Same
{L}BLO adr	<	Branch if Lower Nach Vergleich von Betragzahlen
BRCLR mem _{8bit} , imm, adr	adr→PC wenn mem & imm=0	BRanch if bits are CLeaRed
BRSET mem _{8bit} , imm, adr	adr→PC wenn /mem & imm=0	BRanch if bits are SET

adr ist hier eine Speicheradresse im Programmcode, die vom Programmierer über eine Marke (Label) adressiert wird. Im Maschinenbefehl selbst steht ein Offsetwert relativ zum aktuellen Programmzählerstand (relative Adressierung). Die Befehle mit {L} können zu einer beliebigen Adresse springen, die Befehle ohne {L} können relativ zum aktuellen Wert von PC nur über eine Distanz von -128, ..., +127 springen.

Bei allen bedingten Sprüngen wird der aktuelle Wert der Statusbits im CCR ausgewertet, die von einem vorherigen Befehl, häufig einem Vergleichsbefehl, gesetzt wurden.

2.4 Grundbefehlssatz

Schleifenbefehle (Schleifenbefehle ändern die Statusbits N, Z, V, C nicht!)

IBEQ $\text{reg}_{A,B,D,X,Y,SP}, \text{adr}$ DBEQ $\text{reg}_{A,B,D,X,Y,SP}, \text{adr}$ IBNE $\text{reg}_{A,B,D,X,Y,SP}, \text{adr}$ DBNE $\text{reg}_{A,B,D,X,Y,SP}, \text{adr}$	$\text{reg}_{A,B,D,X,Y,SP} \pm 1 \rightarrow \text{reg}_{\dots}$ $\text{adr} \rightarrow \text{PC}$ wenn $\text{reg}_{\dots} = 0$ $\text{adr} \rightarrow \text{PC}$ wenn $\text{reg}_{\dots} \neq 0$	Increment/Decrement register and Branch if Equal to 0 ... Branch if Not Equal to 0
TBEQ $\text{reg}_{A,B,D,X,Y,SP}, \text{adr}$ TBNE $\text{reg}_{A,B,D,X,Y,SP}, \text{adr}$	$\text{adr} \rightarrow \text{PC}$ wenn $\text{reg}_{\dots} = 0$ wenn $\text{reg}_{\dots} \neq 0$	Test register and Branch if ...

Unterprogrammaufrufe (Unterprogrammaufrufe ändern die Statusbits N, Z, V, C nicht)

JSR mem	mem \rightarrow PC Speichert Rücksprungadresse auf dem Stack, sh. 2.6	Jump to SubRoutine Unterprogrammaufruf, wie BSR, aber Ziel auch indirekt/indiziert adressierbar
BSR adr	adr \rightarrow PC Speichert Rücksprungadresse auf dem Stack, sh. 2.6	Branch to SubRoutine wie JSR, aber nur relative Adressierung (kürzerer Befehl)
RTS	Holt die Rücksprungadresse vom Stack, sh. 2.6	ReTurn from SubRoutine Rückkehr aus einem Unterprogramm
CALL, RTC	Unterprogrammaufruf und Rücksprung, wenn der erweiterte Programmspeicher > 64KB verwendet wird.	

2.4 Grundbefehlssatz

Softwareinterrupts (siehe Kap. 3) (Softwareinterrupts (ausser RTI) ändern die Statusbits N, Z, V, C nicht)

SWI	Speichert Rücksprungadresse und die Register X, Y, D, CCR auf dem Stack, nicht maskierbar, setzt I=1	SoftWare Interrupt Aufruf der SWI Interrupt Service Routine
TRAP	Wie SWI	TRAP for unimplemented op-codes Aufruf der TRAP Interrupt Service Routine für ungültige Befehle
RTI	Restauriert die Register	ReTurn from Interrupt Rückkehr aus einer Interrupt Service Routine
CLI	0 → I	CLeAr Interrupt mask Freigeben von Interrupts der On-Chip-Peripherie
SEI	1 → I	SeT Interrupt mask Sperren von Interrupts der On-Chip-Peripherie

Sonstige Befehle

NOP	-	No Operation
WAI, STOP	WAIt und STOP Stromsparmmodus: Hält die CPU ohne/mit der On-Chip-Peripherie an, bis ein Interrupt auftritt. Kann nur durch Interruptsignale beendet werden. Sollten in Verbindung mit dem Metroworks Debugger nicht verwendet werden.	
MEM, REV, EMIN..., EMAX..., MIN..., MAX..., ETBL, TBL, ...	Befehle für die Implementierung von Fuzzy Logic, Minimum und Maximum-Operationen und für Tabellen-Zugriffe, sh. [3.1].	

2.5 Programmierbeispiele 1

2.5 Programmierbeispiele 1

- Beispiele zu Transportbefehlen und Adressierungsarten (CodeWarrior-Projekt [AsmIntro.mcp](#))

```
.data: SECTION
var1:   ds.w   1
var2:   ds.b   1
var3:   ds.b   2

.const: SECTION
const1: dc.b   $00, $11, $22, $33

.init:  SECTION
main:   . . .
        LDD   #$1234
        TFR   D, X
        STD   var1
        STAA  var2
        STD   var3

        LDD   const1
        LDD   #const1
```

2.5 Programmierbeispiele 1

Fortsetzung

```
LDY  #$0001
LDX  D, Y
LDX  const1, Y

LDY  #const1
LDAA 1, Y+

LDAA 2, +Y

LDAA 1, -Y

LDAA 1, Y-


LDD  #const1
STD  var1
LDX  #0000
LDD  var1, X
LDD  [var1, X]
```

2.5 Programmierbeispiele 1

Fortsetzung

```
LDD  #$AAAA      ; D = 0xAAAA
LDX  #$5555      ; X = 0x5555
LDAA #$7F        ; A = 0x7F
TFR  A, X
LDAA #$80        ; A = 80h
TFR  A, X
TFR  X, B
```

```
MOVW #$5678,var1
MOVW var1, var2
LDX  #var3      ; X = &var3
MOVB var1, 0, X
MOVB 0, X, 1, X
```

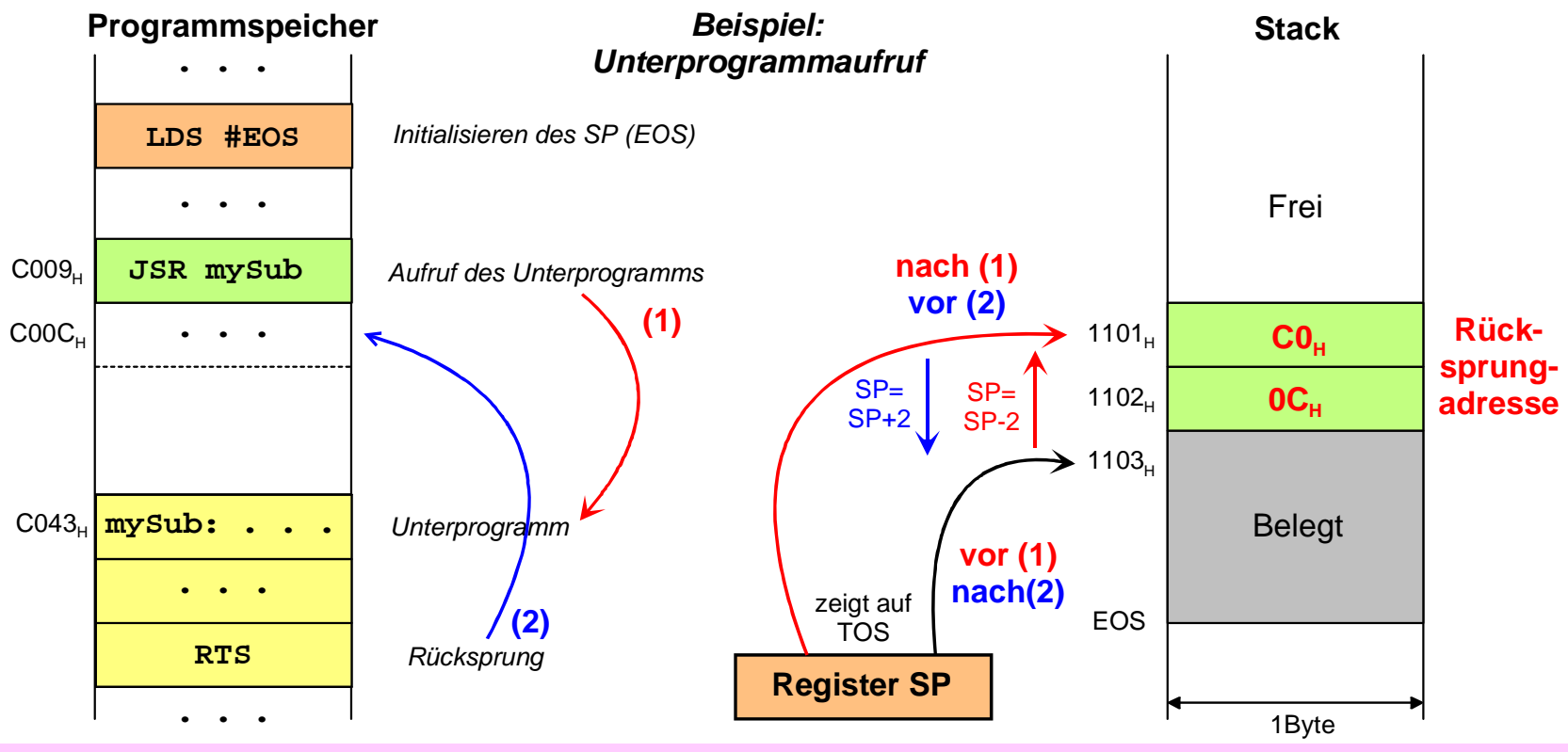
```
LDD  var1
LDD  var1+1
LDD  var1+3
```

2.6 Stack

2.6 Stack (Stapelspeicher)

Aufgabe: Speicherbereich im RAM zum Zwischenspeichern von Registern und Rücksprungadressen von Unterprogrammen

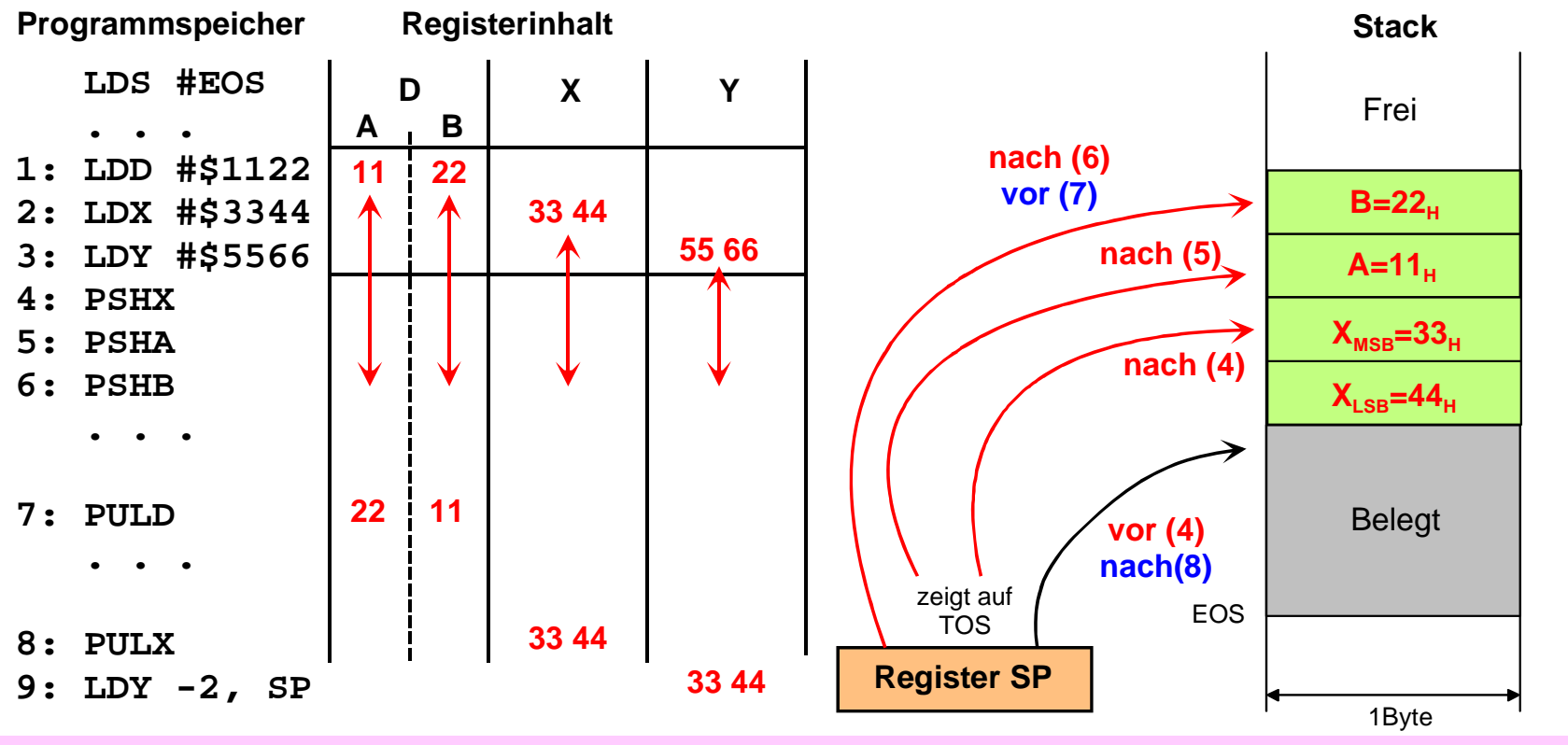
Prinzip: Last-In-First-Out-(LIFO) Speicher, wird vom Ende her (End of Stack EOS) belegt
Zugriff Register-indirekt über den Stackpointer SP.
SP zeigt auf das als letztes auf den Stack gelegte Byte (Top of Stack TOS)



2.6 Stack

Beispiel: Sichern von Registern auf dem Stack

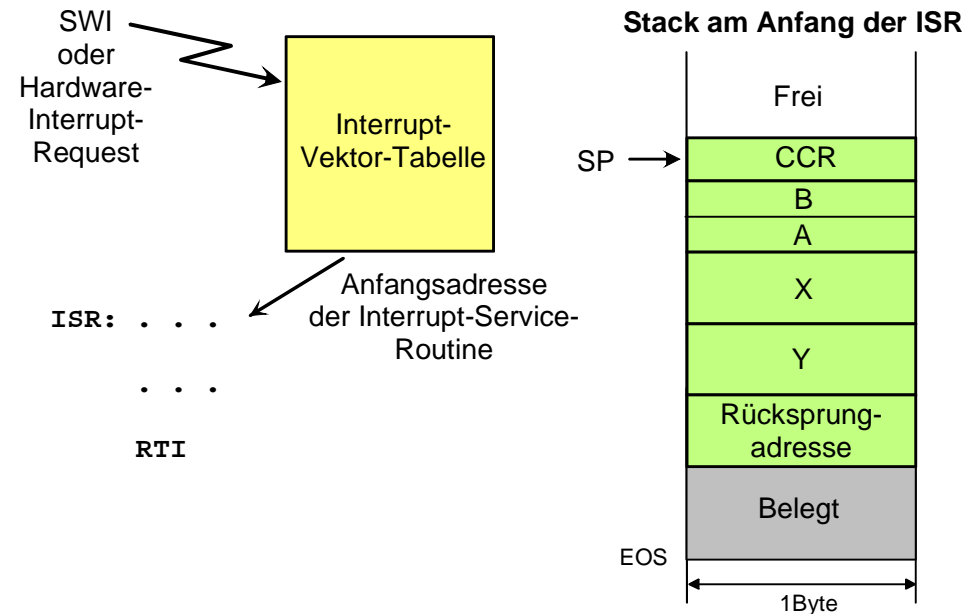
(CodeWarrior-Projekt **AsmIntro.mcp**)



- Stackbereich muss im RAM-Speicher reserviert werden, wird vom Linker erledigt
- Stackpointer muss am Programmbeginn initialisiert werden, hier **LDS #EOS** ^{*1}
- Stackpointer wird automatisch inkrementiert/dekrementiert
- Anzahl der Bytes, die auf den Stack gelegt und wieder geholt werden, muss balanciert sein

2.6 Stack

- Beim Aufruf von Interrupt-Service-Routinen (Unterprogramme, die durch den Befehl SWI oder durch ein Interrupt-Request-Signal der Hardware ausgelöst werden, siehe Kapitel 3), sichert die CPU automatisch den Inhalt sämtlicher Register auf den Stack:



*1 Die CodeWarrior-HCS-Entwicklungswerkzeuge definieren die Größe des Stacks in den Linker-Steuerdateien **simulator_Linker.prm** bzw. **Monitor_Linker.prm** defaultmässig mit **STACKSIZE 0x100** und stellen ein Symbol **__SEG_END_SSTACK** bereit, das auf die Endadresse des Stacks zeigt. Im Assembler-Quellprogramm wird der Stack-Pointer damit folgendermassen initialisiert:

```
; Import symbols
XREF __SEG_END_SSTACK          ; End of stack

; Begin of program code
main:    LDS    #__SEG_END_SSTACK ; Initialize stack pointer
```

2.7 Programmierbeispiele 2

2.7 Programmierbeispiele 2

- Arithmetik und Logik (CodeWarrior-Projekt [AsmIntro2.mcp](#))

<i>C-Programm</i>	<i>Äquivalentes Assembler-Programm</i>
<pre>char a08 = 1, c08 = 3; int a16 = 1, b16 = 2, c16 = 3; long a32 = 1, b32 = 2, c32 = 3; unsigned char cu08= 3; unsigned int cu16= 3; void main(void) { c16 = a16 + b16; //Addition 16bit c32 = a32 + b32; //Addition 32 bit c08 = (char) c16; //signed 16 → 8bit cu08 = (unsigned char) cu16; //unsigned 16 → 8bit</pre>	<pre>LDD a32+2 ADDD b32+2 STD c32+2 LDD a32 ADCB b32+1 ADCA b32 STD c32 LDAB cu16+1 STAB cu08</pre>

2.7 Programmierbeispiele 2

<i>C-Programm</i>	<i>Äquivalentes Assembler-Programm</i>
<code>c16 = c08; //signed 8 → 16 bit</code>	
<code>cu16 = cu08; //unsigned 8 → 16 bit</code>	
<code>cu16= cu16 >> 2; //Shift right unsigned</code>	
<code>c16= c16 >> 2; //Shift right signed</code>	<code>LDD c16</code> <code>ASRA</code> <code>RORB</code> <code>ASRA</code> <code>RORB</code> <code>STD c16</code>
<code>c08 = c08 0x81; //Set bits 7 and 0 to one</code>	
<code>a08 = a08 & 0x7E; //Set bits 7 and 0 to zero</code>	

2.7 Programmierbeispiele 2

<i>C-Programm</i>	<i>Äquivalentes Assembler-Programm</i>
<code>c16 = a16 ^ b16; //Bitwise Exclusive OR</code>	
<code>c16 = a16 & b16; //Bitwise AND</code>	
<code>c16 = a16 && b16; //Logical AND</code>	<code>LDD a16 CPD #0 BEQ L1 LDD b16 CPD #0 BNE L2 L1: LDY #0 BRA L3 L2: LDY #1 L3: STY c16</code>

2.7 Programmierbeispiele 2

- Verzweigungen und Schleifen (CodeWarrior-Projekt [AsmIntro2.mcp](#))

<i>C-Programm</i>	<i>Äquivalentes Assembler-Programm</i>
<pre>if (c16 <= 32) //if – else { a08 = 4; } else { a08 = 8; } . . . if (cu16 <= 32) //if - else { . . . } . . . for (;;) //endless loop</pre>	<pre>LDD c16 CPD #32 ;Vergleich BGT L1 LDAB #4 STAB a08 BRA L2 L1: LDAB #8 STAB a08 L2: LDD cu16 CPD #32 ;Vergleich L3: . . . BRA *+0</pre>

2.7 Programmierbeispiele 2

<i>C-Programm</i>	<i>Äquivalentes Assembler-Programm</i>
<pre>for (c08=0; c08 < 3; c08++) //for</pre>	<pre>CLR c08 BRA L4</pre>
<pre>{ c16 = c16 + a16;</pre>	<pre>L0: LDD c16 ADDD a16 STD c16</pre>
<pre>}</pre>	<pre>L1: INC c08</pre>
	<pre>L4: LDAB c08 CMPB #3 BLT L0</pre>
<pre>while (c08 <= 32) //while - do</pre>	<pre>BRA L3</pre>
<pre>{ a16++;</pre>	<pre>L2: LDX a16 INX STX a16</pre>
<pre>}</pre>	<pre>L3: LDAB c08 CMPB #32 BLE L2</pre>
<pre>do { ... } while (c08 <= 32) //do - while</pre>	

2.7 Programmierbeispiele 2

<i>C-Programm</i>	<i>Äquivalentes Assembler-Programm</i>
<pre>enum { KEIN, EINS, ZWEI } eVal; . . . switch (eVal) //switch-case { case KEIN: . . . break; case EINS: . . . break; case ZWEI: . . . break; }</pre>	<pre>KEIN: EQU 0 EINS: EQU 1 ZWEI: EQU 2 eVal: DS.W 1 . . . switch: LDD eVal LSLD TFR D, X JMP [swK, X] swK: DC.W caseKEIN swE: DC.W caseEINS swZ: DC.W caseZWEI caseKEIN: . . . BRA endCase caseEINS: . . . BRA endCase caseZWEI: . . . BRA endCase endCase:</pre>

2.7 Programmierbeispiele 2

- Unterprogrammaufruf (CodeWarrior-Projekt [AsmIntro2.mcp](#))

<i>C-Programm</i>	<i>Äquivalentes Assembler-Programm</i>
<pre>int betrag(int x) { return x > 0 ? x : -x; } void main(void) { . . . c16 = betrag(a16); . . . }</pre>	<pre>betrag: CPD #0 BGT L0 NEGA NEGB SBCA #0 L0: RTS main: . . . LDD a16 JSR betrag STD c16</pre>

Einfachste Art der Parameterübergabe: Aufrufparameter in Register(n)
 Rückgabewert(e) in Register(n)

Bei Funktionen mit vielen Parametern: Parameterübergabe über Stack, siehe Kapitel 4.

Hinweis: Je nach Optimierungseinstellungen des CodeWarrior C-Compilers kann der Assembler-Code deutlich von den dargestellten Lösungen abweichen. Für die Programmierbeispiele wurde die Code-Optimierung faktisch abgeschaltet.

2.7 Programmierbeispiele 2

Befehlslänge und Ausführungsgeschwindigkeit von Befehlen

- Opcode-Länge

Der eigentliche Opcode der HCS12-Befehle ist 1 oder 2 Byte lang. Dazu kommen weitere Bytes für die Operandenadresse (bei direkter Adressierung) oder unmittelbare Operanden bzw. Indexwerte, wobei Konstanten z.T. nur mit 5, 9 oder 11bit gespeichert werden, um Platz zu sparen. Insgesamt ergeben sich dadurch Befehle mit einer Länge von 1 bis 6 Byte.

- Taktzyklen

Die Anzahl der Taktzyklen, die für die Ausführung eines Befehls notwendig sind, hängt von der Länge des Befehls ab (Holphase = Lesen des Speichers), dem Ort der Operanden (Lesen/Schreiben von Registern bzw. des Speichers) sowie der eigentlichen Ausführung (in der ALU) ab. Beispiele siehe nächste Seite.

- Die entsprechenden Angaben finden sich sehr detailliert (und damit bezüglich der Anzahl der Taktzyklen sehr unübersichtlich) in der CPU-Dokumentation [3.1, Abschnitt 6.7 bzw. Anhang A]. Die Länge eines Befehls oder Programmabschnitts kann relativ einfach im Disassembly-Listing bzw. Disassemble-Fenster des Debuggers bzw. in der Map-Datei des Linker/Locators bestimmt werden. Die Dauer eines Befehls bzw. eines Programmabschnitts läßt sich am einfachsten mit dem Simulator ‚messen‘ (Angabe CPU Cycles im Register-Fenster).

2.7 Programmierbeispiele 2

- Beispiele für Befehlslängen und Befehlsdauern in Abhängigkeit der Adressierungsart

<i>Adressierungsart</i> ^{*1}		<i>Befehl</i>	<i>Länge in Byte</i>	<i>Dauer in CPU-Taktzyklen</i> ^{*2}
<i>Quelloperanden</i>	<i>Zielloperand</i>			
Unmittelbar (IMM)	Register	LDD #1234	3	2
Register-Indirekt (IDX)	Register	LDD 0, X	2	3
Register-Indirekt mit Inkrement	Register	LDD 2, X+	2	3
Direkt (EXT)	Register	LDD var1	3	3
Register-Indirekt mit Index (IDX2)	Register	LDD var1, X	4	4
Speicher-Indirekt mit Index ([IDX2])	Register	LDD [var1, X]	4	6
Register	Register	TFR D, X	2	1
Register-Indirekt	Register-Indirekt	MOVW 0, X, 0, Y	4	5
Direkt	Direkt	MOVW var1, var2	6	6
Direkt		JMP adresse	3	3
Direkt		JSR adresse	3	7
Implizit		RTS	1	5

^{*1} **var1, var2** ... 16bit Variable im internen ROM/RAM-Speicher

^{*2} Bei $f_{\text{BUSCLK}}=24\text{MHz}$ dauert ein CPU-Takt ca. 42ns