

Computerarchitektur

3

Skript zur Vorlesung
Prof. Dr. Jörg Friedrich

Dieses Skript „Computerarchitektur 3“ darf in seiner Gesamtheit nur zum privaten Studiengebrauch benutzt werden. Das Skript ist in seiner Gesamtheit urheberrechtlich geschützt. Folglich sind Vervielfältigungen, Übersetzungen, Mikroverfilmungen, Scan-Vervielfältigungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen unzulässig. Ein darüber hinaus gehender Gebrauch ist zivil- und strafrechtlich unzulässig.

Professoren und Lehrbeauftragte an Hochschulen und Fachhochschulen sind eingeladen, dieses Werk in Teilen oder in seiner Gesamtheit für Zwecke der Lehre auch ohne Angabe des Ursprungs zu verwenden. Gerne wird auf Anfrage der FrameMaker-Quelltext zur Verfügung gestellt.

Inhalt

■ Einführung

1.1 Geschichtliches	1
1.2 Kategorien von Rechnern	2
1.3 „Hello World“	2
1.4 Lernziele	6
1.5 Ein Blick unter die Haube	6
1.5.1 Was passiert beim Kompilieren?	9
1.5.2 Allgemeiner Aufbau von Rechnern	14
1.5.3 Was passiert nach dem Einschalten?	15
1.5.4 Wie kommt der Buchstabe auf den Bildschirm?	16
1.5.5 Wie schalten wir unsere LED ein und aus?	17
1.6 Speicher	17
1.6.1 Speicherhierarchie	17
1.6.2 Breiteianhänger oder Spitzeanhänger?	18
1.6.3 Zwei Herzen im Dreivierteltakt.	18

■ Instruction Set Architecture

2.1 Rechnerarchitekturen Übersicht	22
2.2 Klassifikation von ISAs	24
2.3 Programmiermodell des 68HCS12	25
2.4 Datentypen	26
2.4.1 Datentypen-Übersicht	27
2.4.2 Abbildung von C- auf Assembler-Datentypen	27
2.5 Die Speicherbelegung (Memory Map)	29
2.6 Assembler: die Sprache der Hardware	31
2.6.1 Einfaches Beispielprogramm	32
2.6.2 Label	33
2.6.3 Befehle	33
2.6.4 Operanden	33
2.6.5 Assemblerdirektiven	34
2.7 Adressierung der Operanden	35
2.7.1 Implizit	36
2.7.2 Unmittelbar	36
2.7.3 Direkt	37

2.7.4 Extended	38
2.7.5 Relativ	39
2.7.6 Indiziert, 5-Bit Offset	40
2.7.7 Indiziert, 9-Bit Offset	41
2.7.8 Indiziert, 16-bit Offset	41
2.7.9 Indiziert, indirekter 16-Bit Offset	42
2.7.10 Indiziert, prä-dekrementiert	42
2.7.11 Indiziert, prä-inkrementiert	43
2.7.12 Indiziert, post-dekrementiert	44
2.7.13 Indiziert, post-inkrementiert	45
2.7.14 Indiziert, Akkumulator Offset	45
2.7.15 Indiziert-indirekt, D Akkumulator Offset	46
2.8 Befehlstypen	47
2.9 Assemblerbefehle des 68HCS12	48
2.9.1 Daten bewegen	48
2.9.2 Rechnen	56
2.9.3 Testen und Vergleichen	60
2.9.4 Logische und Bit-Operationen	62
2.9.5 Verzweigung und Schleifen	67
2.9.6 Tabellen und Arrays	78
2.9.7 Der Stack und Unterprogramme	79
2.10 Interrupts, Traps und Resets	85
2.10.1 Vorbereitung und Ablauf eines Interrupts	88
2.10.2 Interrupt-Serviceroutine in Assembler	89
2.10.3 Interrupt-Serviceroutine in C	93
2.10.4 Verhalten nach einem Reset	95
■ Ein- und Ausgabeprogrammierung	
3.1 Überblick	97
3.2 Polling und Interruptbetrieb	99
3.3 Parallele Schnittstellen	100
3.3.1 Mehrfach verwendete Anschlüsse	103
3.3.2 Kernmodul-Ports	103
3.4 Zeitgeber (Timer)	104
3.4.1 Freilaufender Zähler	106
3.4.2 Zeitmessung von Eingangssignalen (Input Capture)	109
3.4.3 Zeitabhängige Ausgangssignale (Output Compare)	109

3.5 Pulsweitenmodulator	111
3.6 Serielle Schnittstellen	114
3.7 Analog-Digitalwandler	114
3.7.1 Initialisierung	114
3.7.2 Kanalselektion	116

■ Rechenleistung

4.1 Definition von Rechenleistung	117
4.2 Der Kern des Prozessors: Datenpfad und Steuerung	119
4.2.1 Single Cycle Datenpfad	119
4.2.2 Multi Cycle Datenpfad	119
4.3 Pipelining	121
4.3.1 Überblick	121
4.3.2 Pipeline Hazards	121
4.3.3 Multiple Issue	122

■ Anhang B

5.1 Hardware für die Laborübungen	125
5.2 Allgemeine Hinweise	125
5.3 Anlegen eines neuen Projekts	126
5.4 Peripherie	127
5.4.1 LED-Zeile	127
5.4.2 Schalter und Taster	133
5.4.3 Siebensegment-Anzeige	133
5.4.4 LCD 16x2	134
5.4.5 A/D-Wandler und Potentiometer	135
5.4.6 Lautsprecher	137
5.5 Codewarrior-Simulator	137

■ Index

Einführung

1.1 Geschichtliches

Elektronische Datenverarbeitung begann vor ca. 65 Jahren. Der Fortschritt auf diesem Gebiet bis heute ist gewaltig. Hätte die Automobiltechnik Vergleichbares geleistet, könnte man heute von Oslo bis Alicante in einer Sekunde für ein paar Cent reisen.

Viele Anwendungen, die vor 30 Jahren noch „Science Fiction“ waren, sind heute Realität:

- Geldautomaten
- Mobiltelefone
- PCs und Laptops
- Human Genome-Projekt
- World Wide Web

Die Verbreitung von Rechnern hat in den letzten Jahrzehnten die Art und Weise, wie wir leben, dramatisch verändert. Allein in den letzten Jahren ist z.B. mit der Einführung des Word Wide Webs der Zugang zu Information wesentlich leichter geworden. So können Studenten jetzt Teile ihrer Diplomarbeiten einfach aus dem Internet herunterladen, und Professoren können das ohne große Aufwand feststellen. In der Vorlesung kann man sein Mobiltelefon vor sich auf den Tisch legen, und damit seinen Status dokumentieren. Das ging vor ein paar Jahren wenn überhaupt nur über die Kleidung und das Benehmen, denn das Auto konnte man ja schlecht in den Hörsaal schieben. In seinem PDA-Terminkalender kann man alle wichtigen Termine notieren, wie z.B. das morgendliche Zahneputzen oder das Mittagessen. Mit Hilfe von ESP im Auto kann man Kurven fahren wie Michael Schumacher, weil der Rechner die Fahrzeugkontrolle übernimmt.

1.2 Kategorien von Rechnern

Wir unterscheiden vier Kategorien von Rechnern:

- **Server** sind Computer, die für größere Programme und viele Benutzer eingesetzt werden. Der Zugriff erfolgt typischerweise über ein Netzwerk. Kosten: zwischen 2000 und 1 Mio. Euro, je nach Zuverlässigkeit. Beispiele: Portale im WWW, Gebührenberechnung der Telekommunikationsbetreiber, Datev-Rechenzentren der Steuerberater
- **Desktop Computer (PCs)** sind für den Gebrauch durch eine einzelne Person entwickelt und besitzen typischerweise eine Tastatur, eine grafische Anzeige und eine Maus. Kosten: zwischen 500 und 3000 Euro.
- **Supercomputer** sind besondere Rechner, die Höchstleistungen bei höchsten Kosten (Millionen von Euro) erzielen. Sie werden z.B. für die Wettervorhersage oder Simulationen in der Produktentwicklung eingesetzt. Ähnliche Rechenleistungen werden heute auch durch Zusammenschaltung von Servern erzielt (Cluster), allerdings bei kleinerer Zuverlässigkeit.
- **Embedded Computer** (eingebettete Systeme) sind Rechner, die in einem umfassenderen System enthalten sind und oft vom Benutzer gar nicht wahrgenommen werden. Auf ihnen läuft oft nur eine spezielle, dedizierte Software. Beispiel: ABS im Auto, Waschmaschinensteuerung, Mobiltelefon, Maus. Kosten: zwischen 1 und 1000 Euro.

Mit Abstand die größte Anzahl von Rechnern sind embedded Computer, gefolgt von Desktop-Systemen. Abbildung 1-2 zeigt die Aufteilung auf verschiedene Rechnerarchitekturen (IA-32 ist der typische PC, ARM wird in vielen Mobiltelefonen und Spielekonsolen eingesetzt).

Im Labor beschäftigen wir uns mit einem Rechner, der viel im Automobilbereich eingesetzt wird (Freescale 68HCS12). Der gleiche Rechner wird auch in der Vorlesung „Echtzeitsysteme“ verwendet.

1.3 „Hello World“

Jeder C-Programmierer kennt das Programm „hello.c“:

```
#include <stdio.h>
```

```
main()
{
    printf("hello, world\n");
}
```

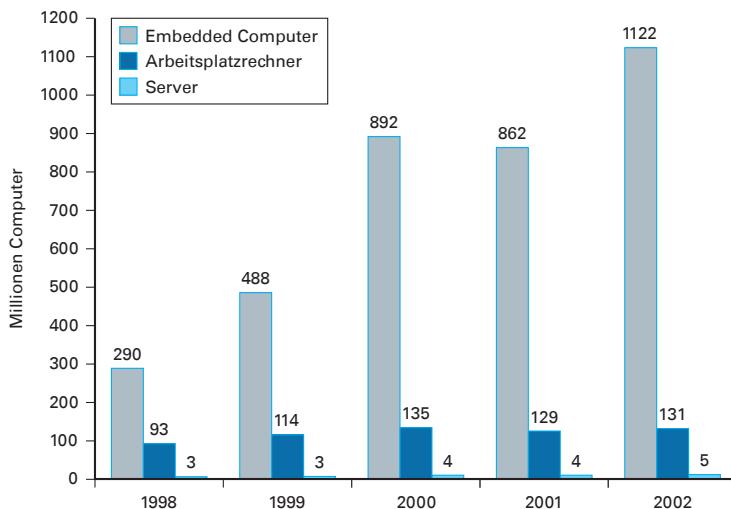


Abbildung 1-1: Anzahl zwischen 1998 und 2002 verkaufter Prozessoren

Jeder C-Programmierer weiß, dass das Programm kompiliert werden muss:

```
cc hello.c
```

Danach kann man es mit

```
a.out
```

ausführen. Großartig! Was passiert denn dabei? Um die Vorgänge übersichtlich zu halten, besccheiden wir uns mit einer Alternative zu „Hello World“, dem „Blinklicht“. Wir bemühen auch nicht gleich einen Supercomputer, sondern nehmen unser kleines Freescale-Laborsystem. Zuvor haben wir natürlich sichergestellt, dass unsere Entwicklungsumgebung funktionsfähig ist. Unser Programm sieht so aus wie in Abbildung 1-3. Eine entsprechende Vorlage finden Sie in der Materialsammlung zur Vorlesung, Ordner „Beispiele\01_Blinklicht“.

Wir müssen zuerst die Hardware initialisieren. Der Hauptteil unseres Programms ist eine kleine Schleife, die abwechselnd eine „1“ oder eine „0“ auf einen mysteriösen „Port“ schreibt. Dieser Port ist elektrisch mit der LED PB0 verbunden. Man kann sich den „Port“ als ein 8-Bit breites Register vorstellen.

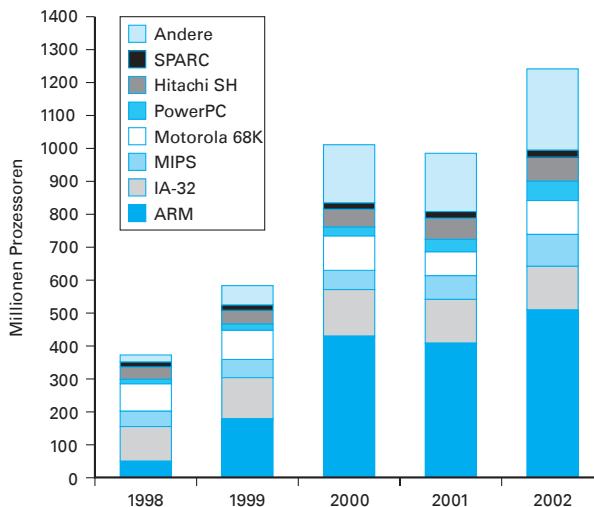


Abbildung 1-2: Anzahl zwischen 1998 und 2002 verkaufter Prozessoren nach Architektur

Natürlich darf man aus mindestens zwei Gründen in der rauen Wirklichkeit so nicht programmieren:

- Die Blinkfrequenz hängt von der Taktfrequenz des Rechners und der Optimierfähigkeit des Compilers ab. Das Programm ist also nicht portierbar.
- Während des Blinkens verbraucht der Rechner Rechenleistung nur fürs Warten bzw. Hochzählern einer Warteschleife. Damit wird mehr Rechenleistung und elektrische Leistung verbraucht, als notwendig ist. Elektrische Leistung ist besonders bei mobilen, aus Batterien gespeisten Geräten ein wichtiger Designparameter (Fahrradtacho, Mobiltelefon, drahtlose Maus).

The screenshot shows a Windows-style window titled "main.c". The file path is displayed as "D:\Daten\Fried\Metrowerks\HCS12\Blinklicht\Sources\main.c". The code itself is a C program for an MC9S12DP256 microcontroller. It includes comments explaining the purpose of the code, authorship, and copyright information. The code initializes the Data Direction Registers (DDR) for Port P and Port J, sets up Port B as an output, and then enters a loop where it toggles the LED at PB0. A constant delay of 400,000 units is used.

```
/*  
 * Ein einfaches C-Programm, um die LED PB0 auf dem Labor-Board  
 * zum Blinken zu bringen.  
 *  
 * Autor: Joerg Friedrich  
 * Copyright: FHT Esslingen  
 * Letzte Änderung: 5.9.2005  
 */  
  
#include <hiddef.h>      /* Einige Makros und Defines */  
#include <mc9s12dp256.h>  /* Prozessorspezifischen Defin. */  
  
#pragma LINK_INFO DERIVATIVE "mc9s12dp256b"  
  
void main(void) {  
    unsigned long int i;  
    const unsigned long int delay = 400000;  
  
    EnableInterrupts; // Das brauchen wir fuer den Debugger  
  
    // Deaktiviere die 7-Segment Anzeige  
    DDRP = DDRP | 0xf; // Data Direction Register Port P  
    PTP = PTP | 0x0f; // Schalte alle vier Segmente aus  
  
    // Aktiviere die LEDs  
    DDRJ_DDRJ1 = 1; // Data Direction Register Port J  
    PTJ_PTJ1 = 0; // Schalte LED-Zeile ein  
  
    // Schalte Port B als Ausgang  
    DDRB = 0xFF; // Data Direction Register Port B  
  
    // Schalte LED PB0 ein und aus  
    PORTB = 0x01;  
  
    for(;;){  
        PORTB = ~PORTB & 0x01;  
        for (i=0; i < delay; ++i) {  
            /* do nothing */  
        }  
    }  
}
```

Abbildung 1-3: Blinklicht.c

Ändern Sie die Blinkfrequenz.

Aufgabe 1-1

Aufgabe 1-2

Verändern Sie das obige Programm so, dass abwechselnd die LEDs PB0 und PB7 blinken, d.h. wenn die eine eingeschaltet ist, soll die andere ausgeschaltet sein. Compilieren Sie das Programm, laden es auf das Board und führen es aus.

Aufgabe 1-3

Erstellen Sie ein Lauflicht. Alle LEDs sollen von rechts nach links eingeschaltet werden, bis alle leuchten. Dann sollen sie von rechts nach links ausgeschaltet werden, bis alle dunkel sind. Die Laufrichtung soll durch eine Variable umkehrbar sein.

1.4 Lernziele

Folgende Fragen sollen in dieser Vorlesung besprochen werden:

- Wie werden Computerprogramme aus C oder Java übersetzt in eine für die Hardware verständliche Sprache?
- Wie führt die Hardware das resultierende Programm aus?
- Wie sieht die Schnittstelle zwischen Software und Hardware aus, und wie veranlasst die Software die Hardware, entsprechende Funktionen auszuführen?
- Wie programmiert man Ein- und Ausgabeelemente?
- Was bestimmt die Leistungsfähigkeit eines Rechners?

1.5 Ein Blick unter die Haube

Praktisch jedes Rechnersystem kann man sich wie in Abbildung 1-4 aufgebaut vorstellen. Die verschiedenen Schichten dienen zur Abstraktion; niemand möchte bei der Programmierung seiner Finanzsoftware mehr die Hardwareregister im Computer bedienen, und ein SQL-Befehl sollte nicht abhängig vom verwendeten Betriebssystem sein.

Um mit der Hardware kommunizieren zu können, bedarf es elektrischer Signale. Das in heutigen Rechnern fast ausschließlich verwendete System verwendet zwei Signalzustände, „1“ und „0“ oder „low“ und „high“, die zwei elektrischen Signalpegeln (z.B. 3,3 V und 0 V) entsprechen. Das Alphabet der Hardware besteht also aus nur zwei Symbolen, so wie das deutsche Alphabet aus 26 Symbolen (ohne Umlaute) besteht. So wie man mit diesen 26 Symbolen nahezu beliebig viele Wörter und ihre Bedeutung definieren und damit hunderttausende umfangreicher Bücher schreiben kann, kann man

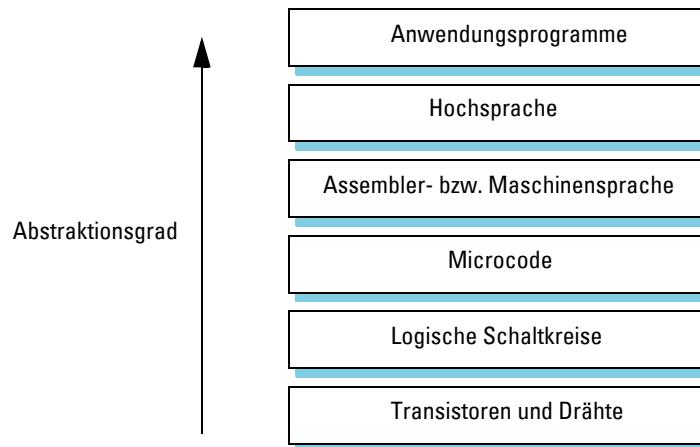


Abbildung 1-4: Schichtenmodell eines Rechnersystems

mit den zwei Symbolen der Hardware viele verschiedene Befehle (Maschinenbefehle) definieren und riesige Softwarepakete schreiben.

Wir definieren ein solches Zeichen der Hardware als ein binäres Zeichen oder **Bit** (von binary digit). Die Befehle, die die Rechnerhardware versteht, sind Zusammenstellungen von Bits. Unser Freescale-Rechner z.B. weiß, dass die Bitfolge 00011000 00001100 bedeutet, dass er den Inhalt der zwei Register A und B (eine Zusammenstellung von jeweils 8 Flip Flops) addieren soll. Diese Bitfolgen kann man sich auch als Binärzahlen vorstellen. Es ist eine der Grundlagen aller Rechner, dass sie sowohl Befehle als auch jegliche Form von Daten als Binärzahlen verarbeiten, nicht zuletzt, weil sie gar nichts anderes kennen. Ob es sich um eine Zahl oder einen Buchstaben oder einen Befehl handelt, wird erst aus dem Zusammenhang deutlich (bzw. für die Hardware aus dem Zustand, in dem sie sich gerade befindet).

Die ersten Rechner wurden tatsächlich so programmiert, dass die Befehle binär über Schalter, später dann über Lochstreifen in den Computer eingegeben wurden. Der Computer war also zunächst so schnell, wie der Programmierer die Schalter umlegen konnte, zumindest beim Starten. Das Programmieren war ein mühsames Geschäft und es gab Freaks, die die kryptische Folge von Einsen und Nullen als Programm verstehen konnten. Diese kryptische Darstellung eines Programms nennt man **Maschinencode**.

Schnell kam man auf die Idee, ein Programm für die Umsetzung einer für den Menschen besser lesbaren Form einer Software in die kryptische Maschinendarstellung zu erstellen. Der Programmierer würde z.B. schreiben:

ADD A, B

und dieses Programm könnte diesen Text übersetzen in den Maschinencode

00011000 00001100

Diese Anweisung sagt dem Rechner, dass er die beiden in den Registern A und B enthaltenen Zahlen addieren soll und das Ergebnis in A ablegen. Dummerweise haben unsere Freescale-Ingenieure sich für diesen Befehl nicht die schöne obige Form einfallen lassen, sondern etwas, das nicht viel besser als die Binärdarstellung ist:

ABA

Assemblersprache

Na ja, wenn man es einmal weiß, kann man es sich schon merken. Vom Merken kommt der Name „**Mnemonic**“ für die Befehlsdarstellung, weil man sich an diese unvergleichlich charmanten Buchstabenfolgen so gerne erinnert. Das Programm, das die Übersetzung vornimmt, nennt man **Assembler**. Programme, die der Assembler versteht, nennt man **Assemblerprogramme**. Das Programm selbst ist in der **Assemblersprache** geschrieben.

In dieser Vorlesung unterhalten wir uns mit dem Rechner z.T. in dieser Assemblersprache, um ihn gut zu verstehen. In der Industrie wird nur noch in Ausnahmefällen Assembler programmiert, aber um zu verstehen, wie Rechner funktionieren, gibt es nichts besseres. Ein wichtiges Ziel dieser Vorlesung ist es, den Zusammenhang zwischen einem in C geschriebenen Programm und dem daraus resultierenden Assemblercode klarzumachen.

Assemblersprache ist natürlich nicht der letzte Schrei in Bezug auf einen Produktivitätsfortschritt beim Programmieren, weil man eine eins-zu-eins Umsetzung eines Assemblerbefehls in einen Hardwarebefehl hat. Und wer möchte seine Datenbankabfrage schon mit Befehlen wie ABA schreiben?

Der nächste Schritt war konsequenterweise die Entwicklung eines Programms, das noch abstraktere Darstellungen von Befehlen in Maschinensprache umsetzen konnte. Die Sprache, die dieses Programm verstehen konnte, nennt man **Hochsprache**. Jetzt kann man schreiben:

A = A + B;

was unserer mathematischen Symbolsprache schon sehr nahe kommt. Der Compiler setzt diese Anweisung dann in diesem Fall in den Maschinenbefehl

00011000 00001100

um. Abbildung 1-5 zeigt einen etwas komplexeren Fall; hier erkennt man, dass die Umsetzung nicht mehr eins zu eins stattfindet, sondern dass eine Hochsprachenanweisung in mehrere Maschinenan-

weisungen heruntergebrochen werden kann. Dadurch steigt die Produktivität eines Programmierers drastisch (typisch Faktor 6 bis 10 für C verglichen mit Assembler).

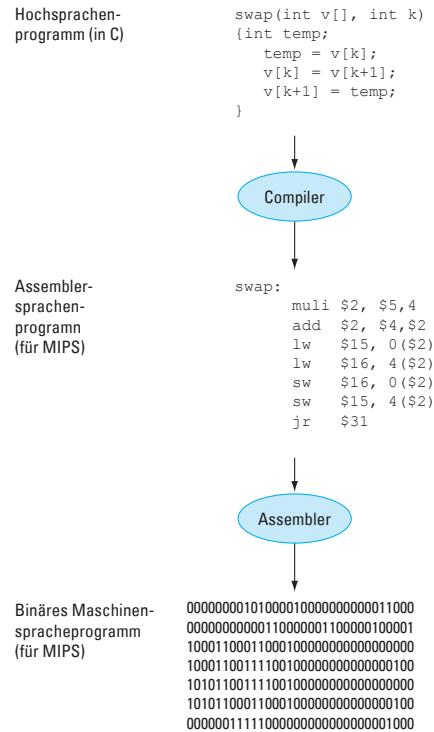


Abbildung 1-5: Umsetzung eines C-Programms in Maschinencode

1.5.1 Was passiert beim Kompilieren?

Wir wissen schon, dass man dem Computer in Form von Textdateien beschreibt, was er tun soll. Diese Textdateien nennt man auch „Quelltext“ oder „Source Code“. Sie können in verschiedenen Programmiersprachen erstellt werden, z.B. C, C++, Pascal, Fortran, Algol, Cobol, Modula-2, Oberon, Pearl oder Java. Der Compiler liest eine solche Datei und erzeugt daraus Maschinencode, der

allerdings so noch nicht von der Hardware ausführbar ist. In dieser vom Compiler erzeugten Objektdatei stehen zusätzliche Informationen für das spätere Zusammenfügen mit anderen Objektdateien, also Informationen über veröffentlichte Symbole und ihre Adressen als auch benötigte Symbole aus anderen Dateien mit den Stellen, an denen sie eingesetzt werden müssen.

Linking

Das Verbinden (*engl. linking*) der verschiedenen Objektdateien geschieht durch ein weiteres Programm, den Linker. Der Linker versucht, alle noch fehlenden Symbole in den verschiedenen Objektdateien aufzulösen und eine ausführbare Datei zu erstellen (*engl. executable*).

Loading

Die ausführbare Datei enthält ebenfalls nicht nur Maschinencode, sondern noch zusätzliche Informationen zum Debuggen und Relokieren. Diese Relokation des Maschinencodes wird bei einem Desktop- oder Serversystem durch ein weiteres Programm vorgenommen, den Lader (*engl. loader*). Auf einem Multitaskingsystem können viele verschiedene Programme laufen, und man kann nicht zur Kompilier- oder Linkzeit festlegen, an welcher Adresse das Programm laufen soll. So müssen vom Lader alle Adressen an den dann zum Ablauf des Programms vorgesehenen Platz verschoben werden.

Locating

Bei einem eingebetteten System befindet sich das Programm zumeist in einem Festwertspeicher. Es liegt meistens auch fest, wie viele Programme ablaufen müssen und wo im Speicher sie laufen sollen. Aus diesem Grunde zieht man die Aufgabe des Laders vor und legt mit einem Lokator (*engl. locator*) fest, an welcher Stelle das Programm ablaufen soll. Die entstehende Datei ist ein Abbild des mit dem Programm geladenen Speichers (Memory Image) und enthält nur noch Maschinencode und Konstanten.

Intel-Hex-Format, Motorola-S-Record

Für diese sogenannte Image-Datei gibt es verschiedene Formate. Die populärsten sind das Intel-Hex-Format (es gibt verschiedene davon) und das Motorola S-Record-Format. Das letztere kann man sich in der Metrowerks-Entwicklungsumgebung für den eigenen Code einmal anschauen.

Zur Veranschaulichung betrachten wir nun ein kleines Programm, das zwei Zahlen addieren soll. Wir schreiben es in der Programmiersprache C (Abb. 1-7). Diese Programm kompilieren wir nun, und schauen uns das Ergebnis so an, dass der Zusammenhang zwischen unserem C-Code und dem vom Compiler erzeugten Assemblercode sichtbar wird. Dies kann man mit der Funktion „disassemble“ erreichen (rechte Maustaste auf der zu disassemblierenden Datei gedrückt halten). Das Ergebnis ist in Abb. 1-8 dargestellt. Man sieht jeweils die Zeile des C-Quelltextes und darunter den erzeugten Assemblercode. Ganz links neben dem Assemblercode steht der zugehörige Maschinencode in Hexasdezimalform.

Man kann sehen, dass es zu jedem Assemblerbefehl genau eine Zeile Maschinencode gibt, die aus einem bis vier Byte besteht. Ein C-Befehl kann aber in mehrere Assemblerbefehle übersetzt werden. In unserem Beispiel ist das noch nicht gravierend; einen deutlicheren Unterschied sieht man bei komplexeren C-Ausdrücken wie Entscheidungsstrukturen oder Operationen auf Arrays.

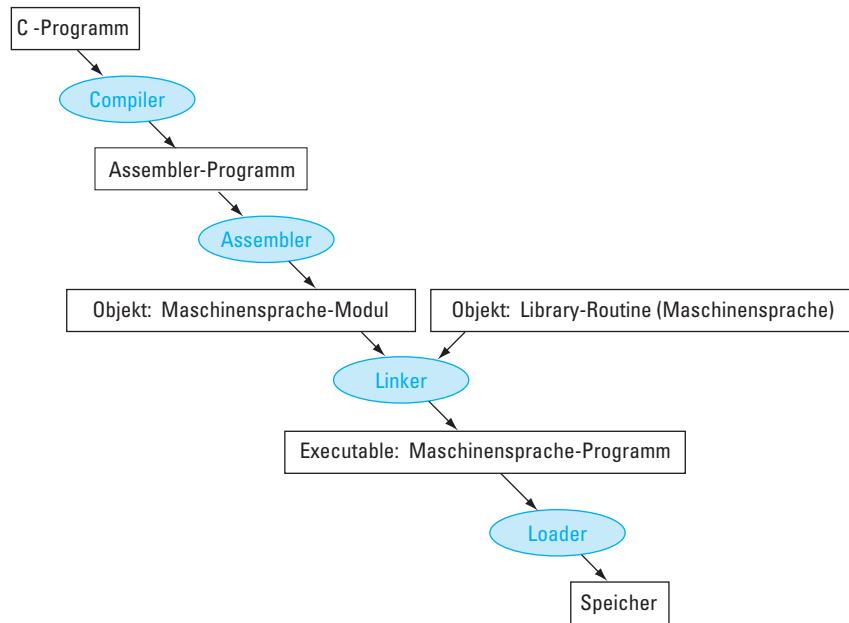
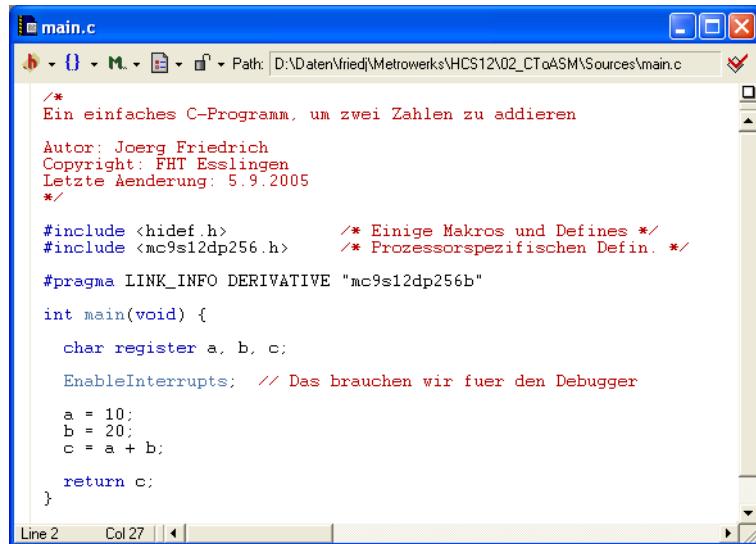


Abbildung 1-6: Compile - Link - Load - Zyklus



The screenshot shows a window titled "main.c" containing C code. The code is a simple program to add two numbers (10 and 20) and return the result. It includes comments explaining the purpose, author, copyright, and last modification date. The code uses register variables and enables interrupts for the debugger. The file path is listed as D:\Daten\friedj\Metrowerks\HCS12\02_CToASM\Sources\main.c.

```
/*
Ein einfaches C-Programm, um zwei Zahlen zu addieren
Autor: Joerg Friedrich
Copyright: FHT Esslingen
Letzte Änderung: 5.9.2005
*/
#include <hidef.h>          /* Einige Makros und Defines */
#include <mc9s12dp256.h>      /* Prozessorspezifischen Defin. */
#pragma LINK_INFO DERIVATIVE "mc9s12dp256b"
int main(void) {
    char register a, b, c;
    EnableInterrupts; // Das brauchen wir fuer den Debugger
    a = 10;
    b = 20;
    c = a + b;
    return c;
}
```

Abbildung 1-7: Einfaches C-Programm, um zwei Zahlen zu addieren

The screenshot shows a window titled "main.c.lst" containing assembly code. The code is generated from a C program that adds two integers. The assembly code uses the MC9S12DP256 processor's instruction set, including LEAS, LDAB, STAB, ASLB, ADDA, and STA instructions. Registers A and D are used for calculations, and the stack pointer SP is managed. The code includes comments explaining the purpose of certain instructions like EnableInterrupts.

```
1: /*
2: Ein einfaches C-Programmum zwei Zahlen zu addieren
3:
4: Autor: Joerg Friedrich
5: Copyright: FHT Esslingen
6: Letzte Änderung: 5.9.2005
7: */
8:
9: #include <hidef.h>           /* Einige Makros und Defines */
10: #include <mc9s12dp256.h>      /* Prozessorspezifischen Defin. */
11:
12: #pragma LINK_INFO DERIVATIVE "mc9s12dp256b"
13:
14: int main(void) {
Function: main
    0000 1b9d      LEAS  -3,SP
    15:
    16:     char register a, b, c;
    17:
    18:     EnableInterrupts; // Das brauchen wir fuer den Debugger
    0002 10ef      CLI
    19:
    20:     a = 10;
    0004 c60a      LDAB  #10
    0006 6b82      STAB  2,SP
    21:     b = 20;
    0008 58        ASLB
    0009 6b80      STAB  0,SP
    22:     c = a + b;
    000b 860a      LDAA  #10
    000d ab80      ADDA  0,SP
    000f 6a81      STA   1,SP
    23:
    24:     return c;
    0011 b704      SEX   A,D
    25: }
    0013 1b83      LEAS  3,SP
    0015 3d        RTS
    26:
Line 38 Col 41
```

Abbildung 1-8: Der vom Compiler erzeugte Assemblercode

Ersetzen Sie für die Addition im eben gezeigten Beispiel die VariablenTypen durch int und long. Schauen Sie sich den vom Compiler erzeugten Maschinen- bzw. Assemblercode an und vergleichen Sie ihn mit dem in Abbildung 1-8. Erläutern Sie die Unterschiede.

Aufgabe 1-4

1.5.2 Allgemeiner Aufbau von Rechnern

Praktisch jeder Rechner besteht aus folgenden fünf Komponenten:

- Eingabe (Input)
- Ausgabe (Output)
- Speicher (Memory)
- Datenpfad (Datapath)
- Steuerung (Control)

Man kann jedes Teil eines beliebigen Rechners, egal ob alt oder neu, einem dieser Blöcke zuordnen. Die Eingabe erfolgt über Eingabegeräte (Input devices), z.B. eine Tastatur oder eine Maus, oder auch einen Analog-Digitalwandler bei einem Prozessrechner. Die Ausgabe erfolgt über Ausgabegeräte (output devices), z.B. einen Bildschirm oder Leuchtdioden, aber auch Magnetventile oder piezoelektrische Wandler.

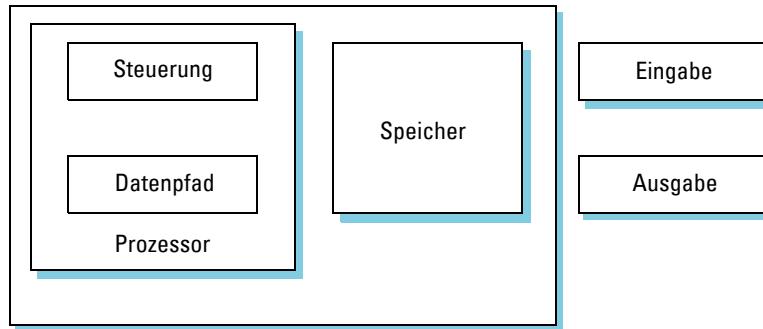


Abbildung 1-9: Die fünf Komponenten eines jeden Rechners

Bus

Eine etwas andere, mehr hardware-orientierte Darstellung ist in Abbildung 1-10 zu sehen. Hier erkennt man noch ein verbindendes Element, den sogenannten „Bus“, der die einzelnen Komponenten des Rechners miteinander verbindet. Moderne Rechner haben häufig mehr als einen Bus. Der Bus lässt sich logisch aufteilen in

- Datenbus
- Adressbus
- Steuerbus

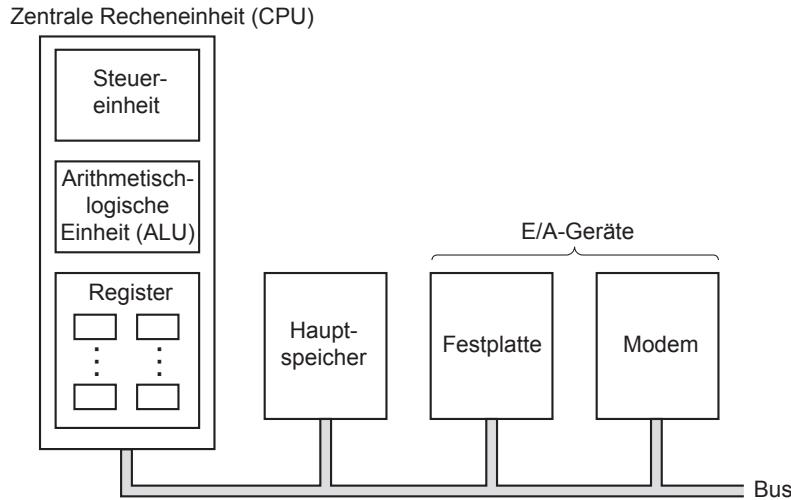


Abbildung 1-10: Alternative Sicht auf eine allgemeine Rechnerarchitektur

Mehrere Bussysteme werden verwendet, um unterschiedliche Anforderungen gerecht zu werden, z.B. einem hohen Datendurchsatz bei einer Grafikkarte oder möglichst einfachen Anschlüssen bei Peripheriegeräten.

1.5.3 Was passiert nach dem Einschalten?

Nach dem Einschalten holt die Steuerung von einer durch die Hardware vorbestimmten Stelle im Speicher entweder einen Befehl oder eine Adresse, an der der erste Befehl zu holen ist. Diese Stelle im Speicher nennt man den **Reset-Vektor**, weil der Instruktionszeiger nach einem Reset auf diese Stelle zeigt. Der Reset-Vektor steht typischerweise entweder ganz unten oder ganz oben im Speicher-Raum. Mancher Rechner, wie unser Freescale 68HCS12, unterscheiden noch die Art des Resets:

- Power-on reset (POR)
- Watchdog-Reset (Computer Operating Properly, COP)

Der Watchdog ist eine kleine Schaltung vergleichbar mit der „Toter Mann“-Schaltung in Eisenbahnen. Software muss regelmäßig eine bestimmte Befehlsfolge an eine Adresse im Speicher schreiben, um den Watchdog zu beruhigen. Wird an die Stelle längere Zeit nicht diese Folge geschrieben, führt

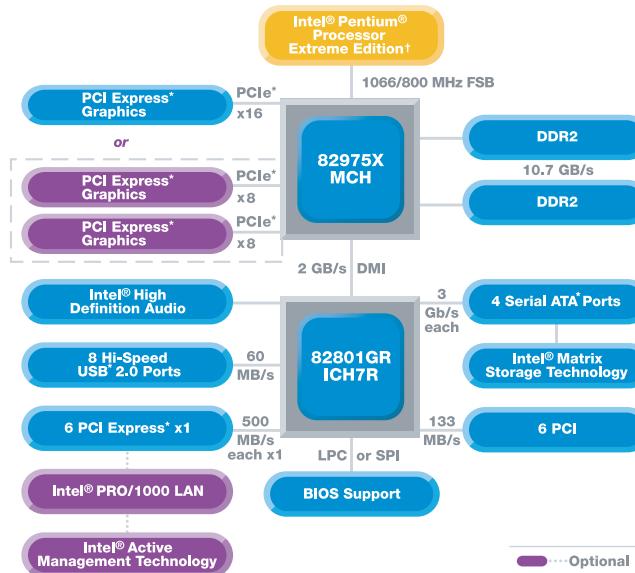


Abbildung 1-11: Moderne Intel975X Systemarchitektur mit verschiedenen Bussen

der Watchdog einen Reset aus. Damit kann man verhindern, dass sich der Rechner „aufhängt“. Watchdog-Schaltkreise überprüfen oft auch noch den Pegel der Versorgungsspannung und führen einen Reset durch, wenn diese sich kurzfristig unter einen minimalen Wert begeben hat.

Bei unserem Freescale-Rechner liegt der Power-on Reset-Vektor an der Adresse \$FFFE-\$FFFF. Dort muss die Adresse für den ersten abzuarbeitenden Befehl abgelegt sein (nicht der Befehl selbst).

1.5.4 Wie kommt der Buchstabe auf den Bildschirm?

Wird in der Vorlesung besprochen.

1.5.5 Wie schalten wir unsere LED ein und aus?

Dazu schauen wir uns zunächst den Schaltplan unseres Laborsystems an (siehe Anlagen). Danach nehmen wir die Beschreibung der Ports für unseren Freescale-Rechner zur Hand (Datei 003-S12DP256PIMV2-Port Integration Module.pdf in den Anlagen). Den Rest besprechen wir in der Vorlesung.

1.6 Speicher

1.6.1 Speicherhierarchie

Es gibt verschiedene Möglichkeiten, Daten innerhalb eines Rechnersystems zu speichern. Sie unterscheiden sich hauptsächlich durch die Zugriffsgeschwindigkeit, mit der die Daten der Recheneinheit zur Verfügung gestellt werden können, und der Kapazität, d.h. der Menge an aufnehmbaren Daten. Je mehr Daten gehalten werden können, desto kleiner wird der Preis pro Bit, und desto langsamer wird der Zugriff. Je schneller auf die Daten zugegriffen werden kann, desto höher wird der Preis pro Bit und um so kleiner wird die zur Verfügung stehende Kapazität.

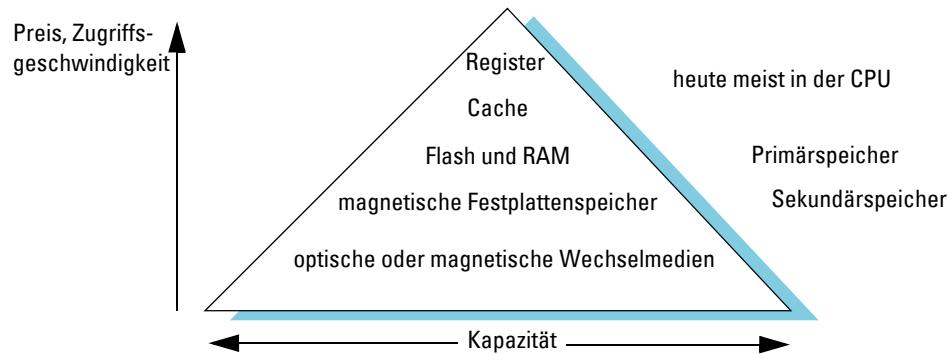


Abbildung 1-12: Speicherhierarchie: schnell oder viel?

Das Ziel beim Entwurf eines hierarchisierten Speichers ist es, eine Rechenleistung basierend auf der Geschwindigkeit der schnellsten Speicher zu Kosten und mit einer Speicherkapazität basierend auf der Kapazität der billigsten Speicher zu erzielen.

1.6.2 Breiteianhänger oder Spitzeianhänger?

In Jonathan Swifts Roman „Gullivers Reisen“ wird ein Disput beschrieben zwischen Liliputanern, die ihr gekochtes Ei am spitzen Ende geköpft haben (Little Endians) und solchen, die es für richtig hielten, die Köpfung am breiten Ende (Big Endians) vorzunehmen. Dieser Disput endete darin, dass mehr als elftausend Personen zu Tode kamen, viele hundert bändereiche Abhandlungen über das Thema geschrieben wurden, und den Breiteianhängern die Fähigkeit zur Bekleidung öffentlicher Ämter aberkannt wurde.

Einen ähnlich wichtigen Disput kann man führen, wenn festzulegen ist, in welcher Reihenfolge man Zahlenfolgen mit mehr als 8 Bit Länge im Speicher ablegen soll: die niedrigerwertigeren Stellen an der niedrigeren Speicheradresse oder an der höherwertigen Speicheradresse. Die Frage wurde nie ganz geklärt, und es gibt Anhänger beider Methoden.

Die „Big Endians“ sind die Freescale-Anhänger, die „Little Endians“ sind die Intel-Anhänger, und es gibt ein paar salomonische Prozessorhersteller, die es dem Benutzer überlassen, ob er den Rechner lieber als Little oder Big Endian betreiben möchte. Auf unserem Freescale-Board finden wir deshalb eine „Big Endian“-Darstellung, in unserem PC mit Intel Pentium-Prozessor eine „Little Endian“-Darstellung.



Abbildung 1-13: Little Endian und Big Endian

Merken kann man sich die Anordnung mit einer Affenbrücke bezogen auf Abbildung 1-13: der Affe ist klein.

1.6.3 Zwei Herzen im Dreivierteltakt

Jeder Computer arbeitet schrittweise („getaktet“), da seinem Steuerwerk eine Zustandsmaschine zugrunde liegt. Als Taktfrequenz wird meist der Takt angegeben, mit dem der Mikrocode abgearbeitet wird (z.B. Pentium 4GHz). RISC-Maschinen können meist pro Takt einen einfachen Befehl abarbeiten. Komplexere Befehle benötigen bei allen Architekturen mehrere Taktzyklen.

Durch die sehr hohen Taktfrequenzen moderner Prozessoren und die technologischen Grenzen, die Quartzoszillatoren gesetzt sind (ca. hundert Megahertz bedingt durch die notwendige Genauigkeit und die Abmessungen) werden die Taktsignale heute meist auf dem Prozessor mit Hilfe einer Frequenzvervielfacher-Schaltung (mit Phased Locked Loop, PLL) aus einem niedrigeren externen Takt erzeugt. Auch auf unserem Freescale-Prozessor wird so vorgegangen, allerdings hauptsächlich aus Kosten- und Flexibilitätsgründen. Dadurch, dass sich die Taktfrequenz über eine Vervielfacherschaltung ableitet, kann sie programmiert werden. Das ist z.B. wünschenswert, um die elektrische Leistungsaufnahme und die erforderliche Rechenleistung gegeneinander auszubalancieren. Niedrigere Taktfrequenzen führen zu niedrigerem Stromverbrauch.

Die Taktfrequenz sagt nur in ihrer Größenordnung etwas über die Leistungsfähigkeit eines Rechners aus. Aufgrund unterschiedlichen internen Schaltkreisdesigns kann ein Rechner mit einem internen Takt von 1 GHz durchaus leistungsfähiger sein als einer mit 2 GHz Taktfrequenz (vergleiche z.B. AMD- und Intel-Angaben zur Taktfrequenz).

Für unseren Freescale-Prozessor ist die Funktion und Programmierung der Takterzeugung durch das sogenannte Clock and Reset Generator Modul in einem eigenen Dokument beschrieben (004-S12CRGV2-Clock&Reset-Generator.pdf in der Anlage).

Arbeiten Sie sich durch die Beschreibung des CRG-Moduls. Welche Werte müssen Sie in welche Register (Adressen) schreiben, um für das Laborsystem eine Taktfrequenz von 20 MHz zu erzielen? Ziehen Sie zur Lösung auch den Schaltplan zu Rate, denn Sie müssen ja wissen, mit welcher Frequenz der Referenzoszillator läuft, oder?

Aufgabe 1-5

Instruction Set Architecture

Es wäre für den Programmierer eines Rechners mühsam, wollte er sich die ganze Zeit Gedanken über die dem Rechner zugrunde liegende Struktur machen. Selbst für den Programmierer eines Compilers wäre es anstrengend, wenn er sich von Version zu Version eines Mikroprozessors immer wieder mit Implementierungsdetails der Hardware beschäftigen müsste.

Aus diesem Grund hat man an der Schnittstelle zwischen Software und Hardware die **Instruction Set Architecture** (ISA) definiert. Sie beschreibt, was der (Assembler-)Programmierer oder Compilerbauer vom Rechner sieht bzw. sehen muss. Sie beschreibt

- die Register, die der Programmierer sieht (das **Programmiermodell**)
- die Assemblerbefehle (das **Instruction Set**)

Solange der Hardwaredesigner diese Schnittstelle bei neueren Prozessoren unangetastet hält bzw. nur erweitert, können existierende Maschinenspracheprogramme weiterhin auf einer solchen Architektur ablaufen. Dies war das Erfolgsrezept von IBM und Intel.

Ein und dieselbe ISA kann verschiedene Hardwareimplementierungen haben (z.B. 16-Bit (DOS)-Modus von 80386, 80486, Pentium). Eine ISA kann sogar als Software implementiert werden (z.B. die Java Virtual Machine oder die virtuellen Maschinen der VMWare oder der in vielen CPUs existierende Microcode).

Die ISA unseres Freescale-Rechners 68HCS12 ist eine Weiterentwicklung der ISA des 68HC11. Programme, die für den 68HC11 geschrieben wurden, laufen ebenfalls auf dem 68HCS12, obwohl der 68HCS12 einen deutlich erweiterten Funktionsumfang besitzt. Im Folgenden beschreiben wir beispielhaft die ISA des 68HCS12-Rechners.

2.1 Rechnerarchitekturen Übersicht

von-Neumann-Architektur

Man unterscheidet Rechnerarchitekturen nach verschiedenen Kriterien. Abbildung 2-1 zeigt eine Aufteilung nach der *Anordnung von Daten und Befehlen im Speicher*. Die heute bei sehr vielen Rechnern gebräuchliche **von-Neumann-Architektur** hält Daten und Programmbefehle im gleichen Adressraum. Der Adressraum kann dabei aufgeteilt sein in Festwertspeicher und flüchtigen Speicher, aber es gibt insgesamt für jede Adresse nur genau eine Speicherzelle. Unser Freescale-Rechner verwendet diese Architektur (abgesehen vom Paging, aber das betrachten wir hier nicht).

Harvard-Architektur

Alternativ werden in der **Harvard-Architektur** Daten und Programmbefehle in getrennten Adressräumen gehalten. Unter einer Adresse \$8C07 kann also einmal ein Befehl oder ein Datum liegen. Eine spezielle Leitung vom Steuerwerk definiert, welche der beiden möglichen Speicherzellen gemeint ist. Diese Architektur findet sich z.B. in Mikroprozessoren des Typs 8051 und vielen digitalen Signalprozessoren.

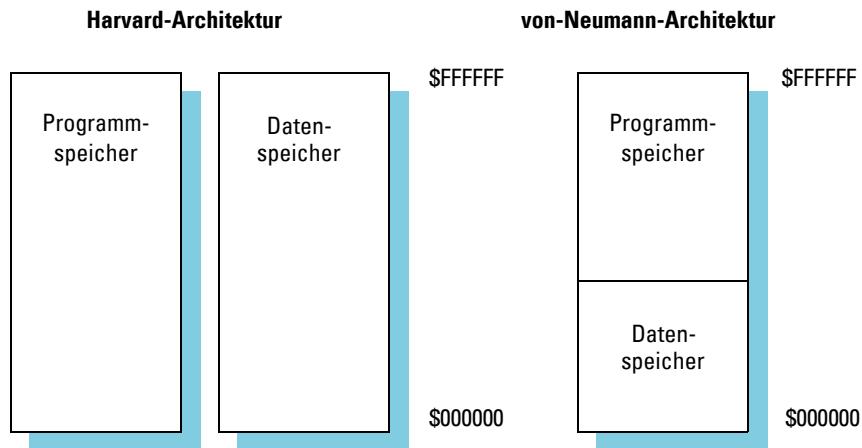


Abbildung 2-1: Rechnerarchitekturen nach Speicherraumanordnung

CISC

Das zweite Unterscheidungsmerkmal für Rechnerarchitekturen liegt in der *Art des Befehlssatzes* begründet. Es gibt Rechner, die einen umfangreichen Befehlssatz mit mächtigen Befehlen besitzen. Die komplexeren Befehle wie z.B. eine kombinierte Multiplizier-Akkumulieroperation, wie sie in digitalen Filtern häufig verwendet wird, benötigen dabei u.U. viele Maschinenzyklen, sind also entspre-

chend langsam. Diese Architekturen nennt man „Complex Instruction Set Computer“ oder CISC-Architektur.

Alternativ hat sich seit ca. 1985 eine Architektur durchgesetzt, die mit möglichst wenigen, aber sehr effizienten und schnellen Befehlen auskommt. Diese „Reduced Instruction Set Computer“ (RISC) benötigen aufgrund ihres einfacheren Aufbaus häufig weniger elektrische Leistung bei gleicher Rechenleistung als ihre CISC-Kollegen. Sie haben meist einen umfangreichen und symmetrischen Registersatz.

RISC

Unseren Freescale-Rechner kann man als CISC bezeichnen. Der Intel-Pentium ist eine Mischform zwischen RISC und CISC. Im Kern dieses Mikroprozessors arbeitet eine RISC-Maschine; über ein eigenes Programm wird diese RISC-Maschine so programmiert, dass sie nach außen hin als CISC-Maschine mit einem komplexen Befehlssatz erscheint. Dadurch können häufig benötigte Operationen direkt in Hardware realisiert werden und den Rechner damit schnell machen, ohne auf den Komfort komplexerer, aber damit auch langsamerer Befehle verzichten zu müssen.

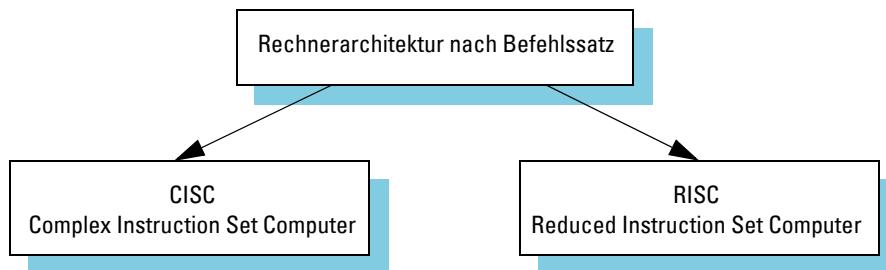


Abbildung 2-2: RISC und CISC

Bei der Klassifizierung eines Rechners spielt die Breite des Datenpfades (typischerweise Anzahl der Bit pro Register) eine große Rolle. Besitzen z.B. die wesentlichen Arbeitsregister eine Breite von 8 Bit, spricht man von einem „8-Bit-Rechner“ oder einer „8-Bit-Architektur“. Unser Freescale-Rechner ist eine Mischung aus 8-Bit und 16-Bit-Rechner. Die zwei wesentlichen 8-Bit Register lassen sich für viele Befehle zu einem gemeinsamen 16-Bit-Register zusammenfassen. Deshalb verkauft Freescale den Rechner als 16-Bit-Rechner.

2.2 Klassifikation von ISAs

Ein wesentliches Unterscheidungsmerkmal in der Klassifizierung von Instruction Set Architectures beruht auf der Art der Datenspeicherung innerhalb des Zentralprozessors. Man unterscheidet

- Stack-Architektur
- Akkumulator-Architektur
- Vielzweck-Register-Architektur

Es gibt daneben noch Mischformen; manche Zentralprozessoren haben mehr Register als einen Akkumulator, aber sie schränken den Verwendungszweck der Register ein. Solche Architekturen, zu denen auch unser Laborrechner sowie der Intel-Pentium-Rechner zählen, nennt man „Erweiterte Akkumulator-Architekturen“ oder „Spezialzweck-Register-Architekturen“.

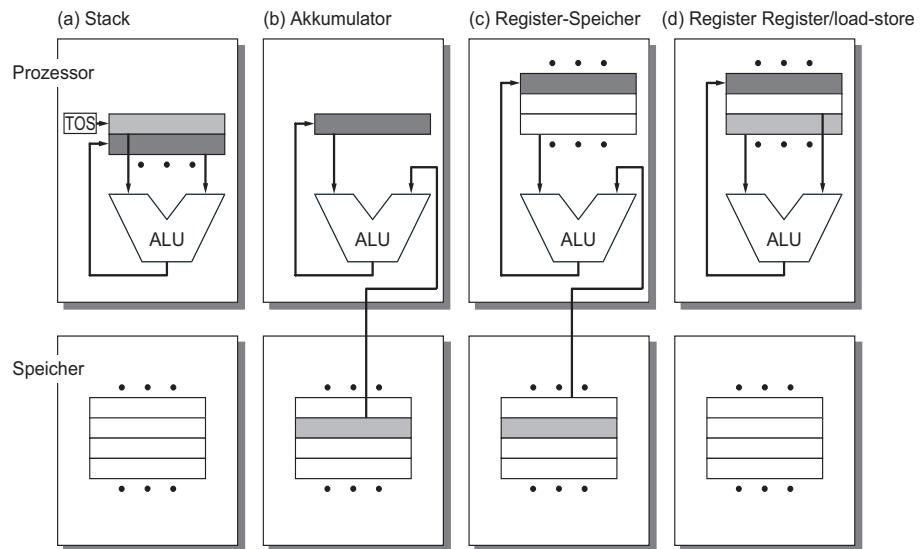


Abbildung 2-3: Klassifikation von Rechnerarchitekturen

Praktisch alle nach ca. 1980 entwickelten Architekturen folgen dem Prinzip der „Load-Store-“ bzw. Vielzweck-Register-Architektur.

Tabelle 2-1: Code-Folge für C=A+B für vier Klassen von Instruktionsarchitekturen

Stack	Akkumulator	Register (Memory)	Register (Load-Store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

2.3 Programmiermodell des 68HCS12

Das Programmiermodell des 68HCS12 ist in Abbildung 2-4 dargestellt. Es ist üblich, allen Registern einen symbolischen Namen zu geben, der so im Assemblercode verwendet werden kann.

Die CPU besitzt zwei allgemein nutzbare 8-Bit-Akkumulatoren **A** und **B**, die für bestimmte Befehle zu einem 16-Bit-Register **D** zusammengefasst werden können.

Weiterhin gibt es

- zwei 16-Bit Index-Register **X** und **Y**
- einen 16-Bit Stack-Zeiger (Stack Pointer **SP**)
- einen 16-Bit Instruktionszeiger (Progam Counter **PC**)
- ein 8-Bit Statusregister (Condition Code Register **CCR**)

Befehle, die nur auf Registern arbeiten, sind in der Regel die schnellsten. Zugriffe auf den Hauptspeicher erfordern zusätzliche Taktzyklen.

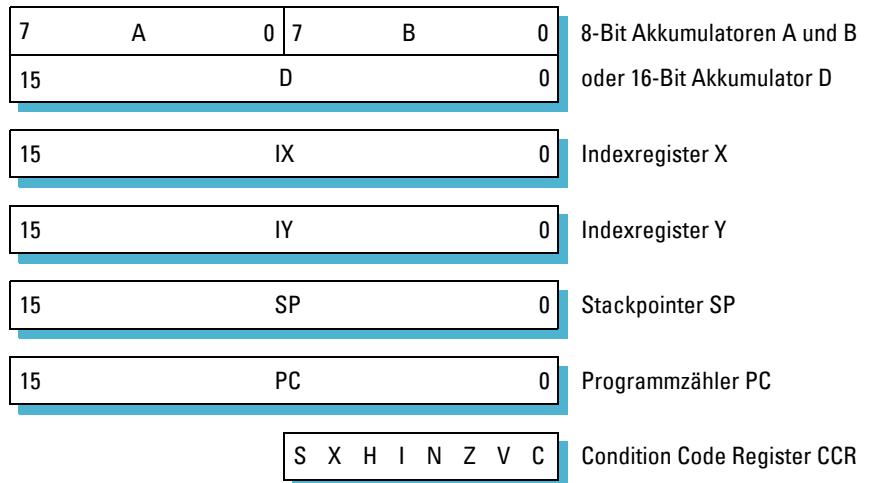


Abbildung 2-4: Programmiermodell des 68HCS12

Aufgabe 2-1

Machen Sie sich bitte mit den Funktionen der verschiedenen Register vertraut, indem Sie Abschnitte 2.2.1 bis 2.2.5.8 im S12CPUV2 Reference Manual durcharbeiten.

2.4 Datentypen

Jede Programmiersprache definiert eingebaute Datentypen. So gibt es in der Programmiersprache C z.B. die Datentypen `char`, `int`, `long`, `double`, `float` und Zeiger (Pointer), z.T. noch in ihrer nicht vorzeichenbehafteten Version (`unsigned`). In den meisten höheren Programmiersprachen kann der Programmierer auch eigene Datentypen definieren, die sich aus den eingebauten Datentypen zusammensetzen (in C mit `typedef` und `struct`). Praktisch alle Programmiersprachen bieten eine Unterstützung für Matrizen (Arrays) an.

Die Datentypen der Programmiersprachen müssen in die Datentypen der Hardware abgebildet werden. Die Datentypen, die die Hardware kennt, werden bestimmt durch die Art der Register bzw. bei

der Adressierung die Art der Zeiger. Ein Datum in Assembler muss immer in irgendeiner Form in einem Register untergebracht werden können oder direkt durch einen Befehl bearbeitbar sein.

Da Datentypen der Programmiersprache und der Rechnerhardware oft nicht direkt zueinander passen, muss eine Abbildung erfolgen. So ist es z.B. kein Problem, komplexe Datentypen auf einem 4-Bit-Rechner zu bearbeiten, es dauert nur entsprechend lange. Dagegen können z.B. leistungsfähige Prozessoren 64-Bit long-Werte direkt auf ihre Register abbilden und bearbeiten. Dies geht jedoch auf Kosten der Chipfläche (Preis) und des Leistungsbedarfs (Problem bei mobilen Geräten).

2.4.1 Datentypen-Übersicht

Unser Freescale-Rechner kennt folgende Datentypen:

- Bits
- 5-Bit vorzeichenbehaftete Integer (nur für indizierte Zeigeroperationen)
- 8-Bit vorzeichenbehaftete oder nicht vorzeichenbehaftete Integer
- 8-Bit, 2-stellige BCD-Zahlen
- 9-Bit vorzeichenbehaftete Integer (nur für indizierte Zeigeroperationen)
- 16-Bit vorzeichenbehaftete oder nicht vorzeichenbehaftete Integer
- 16-Bit effektive Adressen (Zeiger)
- 32-Bit vorzeichenbehaftete oder nicht vorzeichenbehaftete Integer

Andere Rechner können andere eingebaute Datentypen haben. So ist es z.B. bei leistungsfähigeren Rechnern nicht üblich, einzelne Bits (außer vielleicht im Status-Register) adressieren zu können. Bei einem Mikrocontroller wie unserem Freescale-Prozessor sind Bit-Befehle recht nützlich, um z.B. einzelne Flip Flops wie zur Steuerung einer LED ansprechen zu können.

Vorzeichenbehaftete Integer werden in den meisten Rechnern in 2er-Komplementform dargestellt. 16-Bit effektive Adressen sind bei unserem Freescale-Prozessor das Ergebnis einer Adressberechnung, z.B. aus Basisadresse plus einem Index oder Offset.

2.4.2 Abbildung von C- auf Assembler-Datentypen

In C ist nicht festgelegt, wie Datentypen in der Rechnerhardware zu realisieren sind. Es ist abgesehen von einigen Minimalwerten noch nicht einmal festgelegt, welchen Zahlenbereich sie abdecken müssen.

So kommt es, dass ein C-Datentyp `int` auf einem Rechner einen 8-Bit-Wert darstellt (Zahlenbereich -128 bis 127), auf einem anderen einen 16-Bit-Wert (Zahlenbereich -32768 bis 32767), oder sogar einen 32-Bit-Wert (Zahlenbereich -2147483648 bis 2147483648). Selbst der Wert eines `char` ist im C-Standard nicht definiert; praktisch jeder Compiler setzt ihn aber einem 8-Bit-Wert gleich. Aus diesem Grund und wegen der Big Endian/Little Endian Problematik ist der Datentyp `char` der einzige, der an einer Rechner-Rechner-Schnittstelle verwendet werden darf.

Die Abbildung von C-Datentypen auf Datentypen der Maschine ist also maschinenabhängig. Damit ist die Verwirrung allerdings noch nicht komplett. Viele Compilerhersteller, auch der unserer Entwicklungsumgebung, erlauben es dem Programmierer, die Abbildung beim Compilieren durch Konfigurationsschalter zu beeinflussen. Das kann dazu führen, dass durch Setzen eines einzigen Schalters in einem kleinen Menü irgendwo tief in der Oberfläche ein Programm ein völlig anderes Zeitverhalten bekommt oder sogar gar nicht mehr funktioniert, nur weil z.B. ein `int` nun statt 8 Bit 16 Bit benötigt. Und wie schnell klickt man aus Versehen irgendwo rum. Das ist der Grund, warum Integrierte Entwicklungsumgebungen (IDE) für die Erzeugung von Produktcode gefährlich sind. Stattdessen sollte man lieber unter Versionskontrolle gehaltene Skripte verwenden, z.B. mit `make` oder `ant`.

Die voreingestellte Abbildung von C-Datentypen auf Hardware-Darstellung für unsere Entwicklungsumgebung und unseren Prozessor ist in Tabelle 2-2 für integrale und in Tabelle 2-3 für Gleitkommazahlen dargestellt..

Tabelle 2-2: Codewarrior C-Compiler Abbildung ganzzahliger Datentypen auf 68HCS12

Typ	Default-Format	Wertebereich min.	Wertebereich max.
char (signed)	8 Bit	-128	127
signed char	8 Bit	-128	127
unsigned char	8 Bit	0	255
signed short	16 Bit	-32768	32767
unsigned short	16 Bit	0	65535
signed int	16 Bit	-32768	32767
unsigned int	16 Bit	0	65535
enum	16 Bit	-32768	32767

Tabelle 2-2: Codewarrior C-Compiler Abbildung ganzzahliger Datentypen auf 68HCS12

Typ	Default-Format	Wertebereich min.	Wertebereich max.
signed long	32 Bit	-2147483648	2147483647
unsigned long	32 Bit	0	4294967295

Tabelle 2-3: Codewarrior C-Compiler Abbildung Gleitkomma-Datentypen auf 68HCS12

Typ	Default-Format	Wertebereich min.	Wertebereich max.
float	IEEE32	-1.17549435E-38F	3.402823466E+38F
double	IEEE32	-1.17549435E-38F	3.402823466E+38F
long double	IEEE32	-1.17549435E-38F	3.402823466E+38F

Zeiger belegen grundsätzlich zwei Byte (wenn man ohne Paging im „Small“- oder „Default“-Modell arbeitet, was wir im Rahmen dieser Vorlesung tun). Damit kann man 64 kByte adressieren.

2.5 Die Speicherbelegung (Memory Map)

Neben dem Wissen um die Register und der Kenntnis des Befehlssatzes ist die Kenntnis der Speicherbelegung die dritte Voraussetzung, um auf einem Rechnersystem hardwarenah programmieren zu können.

Die Speicherbelegung wird entweder in Form von Tabellen oder bei weniger komplexen Systemen in einer Form wie in Abb. 2-5 gezeigt dargestellt. Auch Mischformen (Bild zur Übersicht, Tabelle für die Details) sind üblich.

Es ist nicht festgelegt, wo im Speicherraum „oben“ ist. Meine bevorzugte Sicht ist, die höherwertigen Adressen auch „höher“ darzustellen. Für unseren Motorola-Prozessor liegt die für ihn höchstmögliche Adresse \$FFFF also oben in der Abbildung.

Die niedrigste Adresse ist praktisch immer \$0 (es gibt keine „negativen“ Adressen im Adressraum). Die höchstwertige Adresse in unserem System ist \$FFFF; es gibt also insgesamt 65536 Adressen in diesem Raum.

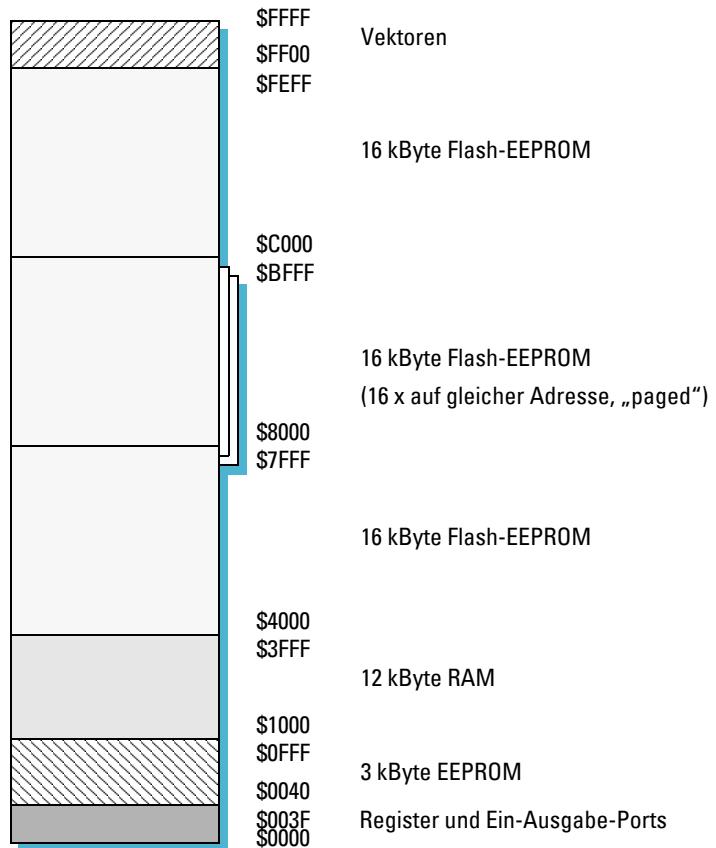


Abbildung 2-5: Die Memory-Map unseres 68HCS12-Laborrechners

Es gibt verschiedene Arten von Speicherzellen bzw. Registern. Für unseren speziellen Prozessortyp liegen von \$0000 bis \$03FF die verschiedenen Register für die Prozessorkonfiguration (z.B. Taktrate) und die Peripherie (z.B. das Flip Flop zum Einschalten unserer LED). Zwischen \$0400 und \$0FFF liegen 3 KByte EEPROM, in dem man nichtflüchtige Daten ablegen kann. Der Schreib-/

Lesespeicher (RAM) belegt den Raum von \$1000 bis \$3FFF; das sind insgesamt 12 kByte Hauptspeicher, die für Variablen und Rücksprungadressen verwendet werden können. Zwischen \$4000 und \$FFFF liegt Flash-Speicher, in dem das Maschinenprogramm einschließlich konstanter Daten sowie im oberen Bereich die Interruptvektoren (siehe "Was passiert nach dem Einschalten?" auf Seite 15.) untergebracht sind.

Für jedes Register des Bereiches von \$0000 bis \$03FF ist genau dokumentiert, was es in der Hardware bewirkt. Wir werden bei Gelegenheit das eine und andere Register genauer betrachten.

2.6 Assembler: die Sprache der Hardware

Ein Programm in Assembler zu schreiben entspricht der Vorgehensweise beim Schreiben einer Software in einer Hochsprache. Das Programm wird in einer oder mehreren Textdateien erstellt und von einem Assembler in den Maschinencode übersetzt. Der Assembler-Quelltext muss einer bestimmten Form genügen, damit er übersetzbare ist.

Jede Zeile eines Assemblerprogramms, wenn sie nicht leer ist, muss eines oder mehr der folgenden Felder enthalten:

- Label (Sprungmarke)
- Befehl (Maschinenbefehl oder Assemblerdirektive)
- Operand(en)
- Kommentar

Assemblerdirektiven oder **Pseudobefehle** sind Anweisungen an den Assembler selbst, nicht an den Prozessor. Assemblerdirektiven tauchen im Maschinenprogramm nicht mehr auf und benötigen deshalb im Speicher auch keinen Platz. Assemblerbefehle führen eins zu eins zu Maschinenbefehlen; sie sind nur eine für den Programmierer besser lesbare Art der Maschinensprache als die reinen Zahlen der Maschinenbefehle.

[Assemblerdirektive](#)

Bei der Übersetzung des Assemblerprogramms bildet sich der Assembler einen fiktiven Adresszähler (current location counter, CLC), der normalerweise mit \$0 beginnt. So kann der Assembler z.B. bei Sprüngen die Sprungadresse oder -Distanz bestimmen. Mit der Assemblerdirektive ORG kann der Programmierer den CLC auf einen von ihm gewünschten Wert setzen. Beispiel:

```
ORG  $1000
var1: dc.b „Dies ist eine Zeichenkette“ ; die an der Adresse $1000 beginnt
var2: ds.b $47      ; an welcher Adresse steht diese Variable jetzt?
                  ; gut, dass wir uns darum nicht mehr kümmern müssen!
```

legt die Zeichenkette und den Wert \$47 ab der Adresse \$1000 im Hauptspeicher ab.

Für die Übersetzung eines Assemblerprogramms benötigt der Assembler zwei Durchläufe (Two-Pass-Assembler): im ersten Durchlauf kann er die Adressen von Vorwärtsreferenzen (Label, die noch nicht definiert sind) noch nicht auflösen.

Zeichen dürfen groß oder kleingeschrieben werden. Nur bei Symbolen wird zwischen Groß- und Kleinschreibung unterschieden, andere Elemente wie z.B. Befehle können sowohl groß als auch klein geschrieben werden. Eine vollständige Zeile sähe so aus:

```
Label1: LDAA $F03B ; ein Kommentar
```

Meist ordnet man die Befehle, Operanden und Kommentare zur Verbesserung der Übersichtlichkeit ausgerichtet in eigenen Spalten an, also z.B. so:

```
GanzLangesLabel:           ; der Assembler merkt sich den CLC an dieser Stelle
    LDS #DATENENDE ; hier startet immer der Kommentar
        ldaa #$ff   ; hier steht kein Kommentar
; eine reine Kommentarzeile
        ; eine ausgerichtete Kommentarzeile
```

2.6.1 Einfaches Beispielprogramm

Unser erstes Assembler-Beispielprogramm addiert den Inhalt der Speicherzelle an der Adresse \$1000 zu dem Inhalt der Speicherzelle an der Adresse \$1001 und speichert das um eins reduzierte Ergebnis in der Speicherzelle mit der Adresse \$1002. Vorher stellen wir sicher, dass sich in den Speicherzellen \$1000 und \$1001 definierte Werte befinden:

```
ORG  $1000 ; setze den CLC auf $1000
      ; damit werden die folgenden Speicherplaetze an die
      ; Adresse $1000 im Hauptspeicher (RAM) gelegt
var1: dc.b $25 ; der erste Summand bei $1000
var2: dc.b $47 ; der zweite Summand bei $1001
res:  ds.b 1  ; ein Byte reservieren für das Ergebnis bei $1002

ORG  $4000 ; setze den CLC auf $4000 (Flash-EEPROM), damit ab dort
      ; die folgenden Maschinenbefehle gespeichert werden
      ; koennen
ldaa var1 ; lade Register A mit Wert in var1
adda var2 ; addiere den Wert in var2 zu Register A
deca      ; dekrementiere Register A
staa res  ; Speichere Ergebnis an Speicherstelle $1002
```

```
end ; sagt dem Assembler, dass hier das Programm zu Ende ist
```

2.6.2 Label

Label stehen immer am Anfang einer Zeile, beginnen nicht mit einer Zahl und enden mit einem Doppelpunkt.

Label sind bei der Definition eines Symbols (mit SET oder EQU) erforderlich. Sie erhalten dann den Wert des Ausdrucks im Operandenfeld.

Label vor anderen Assemblerdirektiven, Befehlen oder einem Kommentar wird der Stand des CLC zugewiesen.

2.6.3 Befehle

Befehle folgen einem Label und sind von diesen durch Leerzeichen oder Tabs getrennt. Befehle dürfen nicht in der ersten Spalte beginnen, da sie sonst von einem Label nicht unterscheidbar wären. Befehle müssen sein

- ein Befehlsmnemonic (z.B. LDAA)
- eine Assemblerdirektive (z.B. ORG)
- ein Makro (das behandeln wir nicht)

2.6.4 Operanden

Operanden folgen den Befehlen und sind von diesen durch Leerzeichen oder Tabs getrennt. Benötigt ein Befehl mehrere Operanden, sind diese durch Kommata getrennt. Das Format der Operanden hängt vom vorhergehenden Befehl und der Art der Adressierung ab. Da es für ein Verständnis der Assemblersprache wie auch des Zusammenhangs zwischen C und Assembler äußerst wichtig ist, wie die Zentraleinheit an die Operanden herankommt, besprechen wir zunächst die verschiedenen Adressierungsarten.

In den meisten Rechnern werden in einem Befehl maximal zwei Operanden miteinander verknüpft. In diesem Fall sind neben dem Befehl noch drei weitere Angaben notwendig:

- Adresse des ersten Operanden
- Adresse des zweiten Operanden
- Adresse des Resultats

Fügt man diese Angaben zu einem Maschinenbefehl zusammen, erhält man einen Dreiadressbefehl. Dies führt zu einer großen Befehslänge, einem hohen Speicherplatzbedarf und einem langsameren Programmablauf (z.B. 8 Bit für den Befehl selbst, und drei mal 16 Bit für die Adressen ergibt zusammen 56 Bit oder 7 Byte). Deshalb reduziert man die Befehslänge durch

- implizite Adressierung
- verdeckte Adressierung
- Reduzierung der Adressbits

Unser 68HCS12 nutzt alle drei Möglichkeiten. Implizite Adressierung bedeutet, dass im Befehl selbst schon z.B. der Zieloperand und einer oder beide Quelloperanden enthalten ist. Beispiel:

ABA ; vollständig implizite Adressierung: addiere A zu B und
; speichere Ergebnis in A

Verdeckte Adressierung bedeutet, dass eine Quelladresse mit der Zieladresse übereinstimmt. Beispiel:

ABA ; verdeckte Adressierung: A ist Quelle und Ziel zugleich

Eine Reduktion der Adressbits erreicht man z.B. dadurch, dass man nur Register verwendet (wird bei RISC-Architekturen gerne gemacht) oder z.B. bei unserem 68HCS12 spezielle Befehle zur Verfügung stellt, die nur auf den untersten 256 Byte im Adressraum arbeiten (8-Bit Adressen).

2.6.5 Assemblerdirektiven

Die meisten Assembler verstehen eine ganze Reihe von Befehlen, die nicht zum Sprachumfang der Maschinensprache gehören, sondern dem Assembler selbst Anweisungen geben, wie er die Übersetzung durchführen soll. Die wichtigsten sind hier kurz beschrieben.

Tabelle 2-4: Assemblerdirektiven

Direktive	Beschreibung	Beispiel
DS [.<size>] <count>	Define Space, rückt den CLC um soviel Bytes weiter; size kann B (Byte), W (Wort) oder L (Langwort) sein	DS.B 7 ; reserviert 7 Byte DS.W 1 ; reserviert 2 Byte
END	markiert Ende des Quelltextes	END

Tabelle 2-4: Assemblerdirektiven

Direktive	Beschreibung	Beispiel
DC [.<size>] <expression>	Define Constant; size kann B, W oder L sein.	DC.B "Ein String", 0
DCB [.<size>] <count>, <value>	Define Constant Block, kopiert <value> in eine Reihe von Bytes (size = B), Wörter (W), oder Langwörter (L)	DCB.B 100, \$FF ; 100 Bytes mit \$FF initialisiert
EQU <expression>	wie #define in C. Expression darf nur schon definierte Symbole enthalten	PI: EQU 3141592
ORG <expression>	Setzt den CLC auf <expression>	ORG \$4000
<name>: SECTION [SHORT]	definiert einen relokierbaren Abschnitt	CODE: SECTION
XDEF <label>[,<label>]...	publiziere Symbole für andere Dateien	XDEF Code, main ; Code != code!
XREF <symbol>[,<symbol>]...	importiere Symbole aus anderen Dateien	XREF subroutine1, subroutine2
XREFB <symbol>[,<symbol>]...	importiere Symbole aus der Zero-Page aus anderen Dateien	XREFB port1, port2

2.7 Adressierung der Operanden

Viele Maschinenbefehle erwarten Operanden, z.B. für die Addition oder bei Transportbefehlen. Das Steuer- bzw. Rechenwerk kann auf diese Operanden auf verschiedene Art zugreifen. Die Art des Zugriffs beschreibt man mit „Adressierungsart“, d.h. wie dem Steuer- bzw. Rechenwerk die Adresse des oder der notwendigen Operanden mitgeteilt wird.

Tabelle 2-5: Adressierungsarten des 68HCS12

Typ	Beschreibung	Beispiel
implizit	keine Operanden	CLRA
direkt	<8-Bit Adresse>	STAA \$30, STAA var1
extended	<16-Bit Adresse>	LDAA \$FFFF

Tabelle 2-5: Adressierungsarten des 68HCS12

Typ	Beschreibung	Beispiel
relativ	<Offset zum CLC, 8-Bit oder 16-Bit, 2er-Komplement>	BRA loopStart
unmittelbar	#<immediate 8-Bit Ausdruck> oder #<immediate 16-Bit Ausdruck>	LDAA #\$64
indiziert, 5-Bit Offset	<5-Bit Offset> zu X, Y, SP, oder PC	LDAA 3,X
indiziert, prä-dekrementiert	<3-Bit Offset> zu -X, -Y, -SP	ADDD 5,-X
indiziert, prä-inkrementiert	+<3-Bit Offset> zu +X, +Y, +SP	
indiziert, post-dekrementiert	-<3-Bit Offset> zu X-, Y-, SP-	
indiziert, post-inkrementiert	+<3-Bit Offset> zu X+, Y+, SP+	
indiziert, Akkumulator-Offset	A, B, oder D + X, Y, SP, oder PC	LDAA B,X
indiziert, 9-Bit Offset	<9-Bit Offset> zu X, Y, SP, oder PC	STAA 18, Y
indiziert, 16-Bit Offset	<16-Bit Offset> zu X, Y, SP, oder PC	STAA \$140, Y
indiziert, indirekt 16-Bit Offset	[<16-Bit Offset> zu X, Y, SP, oder PC]	LDAA [4,X]
indirekt indiziert, D-Offset	[D+ X, Y, SP oder PC]	JMP [D, PC]

2.7.1 Implizit

Befehle mit dieser Adressierungsart benötigen keine Operanden oder alle Operanden sind in internen Registern gespeichert. Die CPU muss nicht auf den Hauptspeicher zugreifen, um den Befehl auszuführen. Beispiel:

NOP	; Befehl ohne Operand
CLRA	; Der Operand befindet sich im Register A

2.7.2 Unmittelbar

Der Wert des Operanden folgt direkt auf den Befehl. Damit ist er Teil des Maschinencodes und kann sich normalerweise zur Laufzeit nicht mehr ändern (Konstante). Beispiel:

```
main: LDAA #$64
```

```
LDX    #$1100
BRA    main
```

In diesem Beispiel wird der hexadezimale Wert \$64 (Schreibweise in C: 0x64) in das Register A geladen. Da das Register A 8 Bit breit ist, erwartet der LDAA-Befehl einen 8-Bit Operanden. Das Register X ist 16 Bit breit, folglich erwartet der Befehl LDX einen 16-Bit Operanden.

Mit der immediate Adressierungsart kann man auch die Adresse eines Symbols referenzieren. Beispiel:

```
ORG    $1000
var1:  DC.B  $16, $10
ORG    $4000
main:
LDX    #var1
BRA    main
```

In diesem Beispiel wird die Adresse der Variablen 'var1' (\$1000) ins X-Register geladen.

Achtung:

Einer der häufigsten Programmierfehler ist das Vergessen des #-Zeichens. Dies führt dazu, dass der Assembler den Ausdruck als eine Adresse anstatt eines konstanten Wertes interpretiert. Beispiel:

```
LDAA  $40
```

bedeutet, dass der Akkumulator mit dem Wert, der an der Adresse \$40 gespeichert ist (irgendein Port-Register), geladen wird, und nicht mit dem Wert \$40.

2.7.3 Direkt

Diese Adressierungsart erlaubt den schnellen Zugriff auf Speicherzellen, die mit einem Byte adressiert werden können (der „direkten Seite“, engl. „direct page“ oder „zero page“, des Hauptspeichers). Man kann also alle Speicherzellen zwischen \$0000 und \$00FF damit erreichen.

Zugriff auf diesen Speicherbereich ist schneller und erfordert weniger Programmspeicherplatz als die „extended“ Adressierungsart, unterscheidet sich aber ansonsten nicht von dieser. Früher hat man Programme dadurch schneller gemacht, dass man häufig benutzte Daten in diesen Bereich gelegt hat. Beispiel:

```
ORG  $30
```

```

daten:      DS.B 1

MyCode:     SECTION
Entry:
            LDS   #$3FFE      ; initialisiere Stack-Zeiger
            LDAA #$01
main:       STAA daten
            BRA  main

```

In diesem Beispiel wird der Wert aus Register A in der Variablen „data“ abgelegt, die sich an der Adresse \$30 befindet.

Beispiel:

```

MyData:    SECTION SHORT
daten:     DS.B 1
           XREFB externeDaten
MyCode:    SECTION
Entry:
            LDS   #$3FFE      ; initialisiere Stack-Zeiger
            LDAA daten
main:      STAA externeDaten
            BRA  main

```

Der Datenbereich „daten“ ist einem sogenannten relokierbaren Abschnitt untergebracht, d.h. zum Zeitpunkt der Übersetzung des Programms stehen die Adressen dieses Abschnitts noch nicht fest. Um dem Assembler trotzdem mitteilen zu können, dass sich diese Daten in der „zero page“ befinden werden, wird die Assemblerdirektive „SHORT“ nach dem Schlüsselwort SECTION benutzt.

Das Symbol „externeDaten“ wird importiert. XREF.B bedeutet, dass es sich um ein Bytedatum handelt, das irgendwo extern in einer anderen Datei definiert ist. Dadurch, dass es ebenfalls in der SHORT SECTION untergebracht ist, weiß der Assembler, wie der richtige Maschinenbefehl aussehen muss.

2.7.4 Extended

In dieser Adressierungsart kann der gesamte Adressraum von 64 kByte angesprochen werden. Beispiel:

```

XDEF Entry
ORG  $1100
data:  DS.B 1
MyCode: SECTION

```

```

Entry:
    LDS    #$3FFE          ; initialisiere Stack-Zeiger
    LDAA   #$01
main:  STAA  data
    BRA   main

```

In diesem Beispiel wird der Wert aus Register A in die Variable "datum" gespeichert. Diese befindet sich an der Adresse \$1100 im Speicher.

2.7.5 Relativ

Diese Adressierungsart wird für die Bestimmung der Zieladresse bei einer Verzweigung verwendet. Damit der Code einfach verschiebbar bleibt, wird die Adresse dabei nicht absolut, sondern als Distanz (Offset) zum Verzweigungspunkt angegeben. Ein Verzweigungsbefehl testet bestimmte Bits im CCR. Befinden sich die Bits in einem entsprechenden Zustand, wird der Offset zur Adresse des auf den Verzweigungsbefehl folgenden Befehls hinzugefügt und die Ausführung des Programms an dieser Stelle fortgesetzt.

Kurzverzweigungsbefehle (BRA, BEQ, ...) erwarten einen ein Byte langen Offset. Der maximale Sprungbereich beträgt für diese Befehle [-128..127]. Beispiel:

```

main:
    NOP
    NOP
    BRA  main

```

In diesem Beispiel verzweigt die Anwendung wieder an den Anfang, nachdem zwei NOP-Befehle ausgeführt worden sind. Man sieht hier, dass man als Assemblerprogrammierer den Offset nicht ausrechnen muss, sondern eine Sprungmarke angeben kann. Der Assembler errechnet den Offset aus den Werten des CLC für das Label "main" und dem momentanen Wert des CLC selbst.

Langverzweigungsbefehle (LBRA, LBEQ, ...) erwarten einen zwei Byte langen Offset. Der maximale Sprungbereich beträgt deshalb [-32768..32767].

Der Assembler kennt ein spezielles Symbol für den momentanen Stand des CLC, den Stern (*). Mit diesem Symbol kann man ebenfalls den Offset für eine Verzweigung angeben. Der * bezieht sich auf den Anfang des Befehls, in dem er verwendet wird. Beispiel:

```

main:
    NOP
    NOP
    BRA  * - 2

```

In diesem Beispiel verzweigt die Anwendung um 2 Byte vor die BRA-Anweisung, d.h. auf das Label "main".

Innerhalb eines absolut adressierten Abschnitts (mit ORG festgelegte SECTION) dürfen folgende relative Adressierungsausdrücke vorkommen:

- ein in einem absoluten Abschnitt definiertes Label
- ein in einem relokierbaren Abschnitt definiertes Label
- ein externes Label (definiert mit XREF)
- ein absolutes EQU oder SET Label

Innerhalb eines relokierbaren Abschnittes dürfen folgende relative Adressierungsausdrücke vorkommen:

- ein in einem absoluten Abschnitt definiertes Label
- ein in einem relokierbaren Abschnitt definiertes Label
- ein externes Label (definiert mit XREF)

2.7.6 Indiziert, 5-Bit Offset

In dieser Adressierungsart wird zu einem Basis-Indexregister ein 5-Bit Offset addiert, um die für diesen Befehl gültige Adresse (effektive Adresse) zu bestimmen. Der gültige Bereich für den Offsetwert ist [-16..15]. Als Basis-Indexregister kann man X, Y, SP, oder PC verwenden.

Mit dieser Adressierungsart kann man gut Elemente in einer n-Element-Tabelle adressieren, wenn die Tabelle weniger als 16 Byte groß ist. Beispiel:

```

ORG $1000
CST_TBL: DC.B $5, $10, $18, $20, $28, $30
          ORG $1200
DATA_TBL: DS.B 10

main:
        LDX #CST_TBL
        LDAA 3,X

        LDY #DATA_TBL
        STAA 8,Y

```

Der Akkumulator A wird mit dem Byte-Wert der Speicherzelle an Adresse \$1003 (\$1000 + 3) geladen. Der Wert des Akkumulators A wird dann in der Speicherzelle mit Adresse \$1208 (\$1200 + 8) gespeichert.

2.7.7 Indiziert, 9-Bit Offset

Diese Adressierungsart ist bis auf den möglichen Offsetbereich identisch zu indizierten 5-Bit Offset-adressierung. Auch hier wird zu einem Basis-Indexregister ein 9-Bit Offset addiert, um die für diesen Befehl gültige Adresse (effektive Adresse) zu bestimmen. Der gültige Bereich für den Offsetwert ist [-256..255]. Als Basis-Indexregister kann man X, Y, SP, oder PC verwenden.

Mit dieser Adressierungsart kann man gut Elemente in einer n-Element-Tabelle adressieren, wenn die Tabelle weniger als 256 Byte groß ist. Beispiel:

```

ORG    $1000
CST_TBL:DC.B $5, $10, $18, $20, $28, $30, $38, $40, $48
          DC.B $50, $58, $60, $68, $70, $78, $80, $88, $90
          DC.B $98, $A0, $A8, $B0, $B8, $C0, $C8, $D0, $D8
ORG    $1200
DATA_TBL:
DS.B   40
main:
LDX    #CST_TBL
LDAA  20,X

LDY    #DATA_TBL
STAA  18, Y

```

Der Akkumulator A wird mit dem Byte-Wert der Speicherzelle an Adresse \$1014 (\$1000 + 20) geladen. Der Wert des Akkumulators A wird dann in der Speicherzelle mit Adresse \$1212 (\$1200 + 18) gespeichert.

2.7.8 Indiziert, 16-bit Offset

In dieser Adressierungsart wird zu einem Basis-Indexregister ein 16-Bit Offset addiert, um die für diesen Befehl gültige Adresse (effektive Adresse) zu bestimmen. Den 16-Bit Offset kann man sich entweder als signed oder unsigned Wert vorstellen (\$FFFF darf man als -1 oder 65535 interpretieren). Als Basis-Indexregister kann man X, Y, SP, oder PC verwenden. Beispiel:

```

main:
LDX    #$600

```

LDAA \$300,X

LDY #\$1000
STAA \$140,Y

Der Akkumulator A wird mit dem Byte aus der Speicherzelle mit Adresse \$900 (\$600 + \$300) geladen. Dann wird der Wert aus dem Akkumulator A an die Speicherzelle mit Adresse \$1140 (\$1000 + \$140) geschrieben.

2.7.9 Indiziert, indirekter 16-Bit Offset

Dies ist die wohl komplizierteste Adressierungsart für den Freescale 68HCS12-Prozessor. Wie bei der indizierten Adressierung mit 16-Bit Offset wird zu einem Basis-Indexregister ein 16-Bit Offset addiert. Die resultierende Adresse ist aber nicht die effektive Adresse für den Befehl, sondern sie zeigt nur auf die beiden Speicherstellen, in denen die effektive Adresse gespeichert ist (darum „indirekt“).

Den 16-Bit Offset kann man sich entweder als signed oder unsigned Wert vorstellen (\$FFFF darf man als -1 oder 65535 interpretieren). Als Basis-Indexregister kann man X, Y, SP, oder PC verwenden. Beispiel:

```
ORG $1000
CST_TBL1: DC.W $1020, $1050, $2001
ORG $2000
CST_TBL2: DC.B $10, $35, $46

ORG $4000
main:
LDX #CST_TBL1
LDAA [4,X]
```

Der Offset '4' wird zum Wert in Register X (\$1000) addiert, um einen Adresszeiger auf die Adresse \$1004 zu erstellen. Dann wird ein Adresszeiger mit dem Wert \$2001 von der Speicherstelle \$1004 gelesen. Der Akkumulator wird mit dem Wert an der Speicherstelle \$2001 geladen, also \$35.

2.7.10 Indiziert, prä-dekrementiert

In dieser Adressierungsart kann der Wert des verwendeten Basis-Indexregisters um den angegebenen Wert dekrementiert werden, bevor sein Wert zur Bildung der effektiven Adresse genutzt wird. Der

Dekrementierwert kann [1..8] sein. Als Basis-Indexregister können X, Y oder SP verwendet werden.
Beispiel:

```
ORG $1000
CST_TBL:    DC.W $5, $10, $18, $20, $28, $30
END_TBL:    DC.B $0
main:
        CLRA
        CLR B
        LDX #END_TBL

loop:
        ADDD 1, -X
        CPX #CST_TBL
        BNE loop
```

Das Basis-Indexregister X wird mit der Adresse des Elements geladen, das auf Tabelle CST_TBL folgt (\$100C).

Das Register X wird um 1 dekrementiert, sein Wert ist nun \$100B. Der Wert an dieser Adresse (\$30) wird zu Register D addiert. Der Wert in X ist nicht gleich dem Wert von CST_TBL (\$1000), also wird X wieder dekrementiert und der nächste Wert (\$00, MSB von \$30) zu D addiert.

Diese Schleife wird solange ausgeführt, bis alle Werte in CST_TBL in D aufsummiert sind.

Funktioniert das obige Programm auch noch, wenn die Werte in Tabelle CST_TBL größer als \$ff werden? Wenn nicht, wie könnte man das Programm korrigieren?

Aufgabe 2-2

2.7.11 Indiziert, prä-inkrementiert

In dieser Adressierungsart kann der Wert des verwendeten Basis-Indexregisters um den angegebenen Wert inkrementiert werden, bevor sein Wert zur Bildung der effektiven Adresse genutzt wird. Der Inkrementierwert kann [1..8] sein. Als Basis-Indexregister können X, Y oder SP verwendet werden.
Beispiel:

```
ORG $1000
CST_TBL:    DC.W $15, $27, $16, $41, $39, $fe
END_TBL:    DC.B $0
main:
        CLRA
        CLR B
```

```

        LDX  #CST_TBL
loop:
ADDD 2,+X
CPX  #END_TBL
BNE  loop

```

Das Basisregister X wird mit der Adresse der Tabelle CST_TBL (\$1000) geladen. Das Register X wird um 2 inkrementiert, sein Wert ist dann \$1002. Der Wert an dieser Adresse (\$27) wird zu Register D addiert.

Register X hat zunächst nicht den Wert von END_TBL (\$100C), also wird wieder inkrementiert und der Inhalt an der Speicherstelle \$1004 zu D addiert. Die Schleife wird so lange ausgeführt, bis X den Wert von END_TBL erreicht hat.

2.7.12 Indiziert, post-dekrementiert

In dieser Adressierungsart kann der Wert des verwendeten Basis-Indexregisters um den angegebenen Wert dekrementiert werden, nachdem sein Wert zur Bildung der effektiven Adresse genutzt wurde. Der Dekrementierwert kann [1..8] sein. Als Basis-Indexregister können X, Y oder SP verwendet werden. Beispiel:

```

ORG  $1000
CST_TBL:
DC.W $5, $10, $18, $20, $28, $30
END_TBL:
DC.W $0

main:
CLRA
CLRB
LDX  #END_TBL
loop:
ADDD 2,X-
CPX  #CST_TBL
BNE  loop

```

Das Basisregister X wird mit dem Wert des Elements geladen, das auf CST_TBL folgt (also \$100C). Der Wert an der Adresse \$100C (\$0) wird zu Register D addiert und X wird anschließend um 2 dekrementiert. Der Wert von X ist nun \$100A.

Register X hat nicht den Wert von CST_TBL (\$1000), also wird wieder an den Anfang der Schleife gesprungen. Der Wert in \$100A (\$30) wird zu D addiert und X wird wieder um 2 dekrementiert. Sein Wert beträgt nun \$1008.

Die Schleife wird solange wiederholt, bis X auf den Anfang der Tabelle CST_TBL zeigt.

2.7.13 Indiziert, post-inkrementiert

In dieser Adressierungsart kann der Wert des verwendeten Basis-Indexregisters um den angegebenen Wert inkrementiert werden, nachdem sein Wert zur Bildung der effektiven Adresse genutzt wurde. Der Dekrementierwert kann [1..8] sein. Als Basis-Indexregister können X, Y oder SP verwendet werden. Beispiel:

```
ORG    $1000
CST_TBL:DC.W $5, $10, $18, $20, $28, $30
END_TBL:DC.B $0
main:
        CLRA
        CLRB
        LDX    #CST_TBL
loop:
        ADDD  1,X+
        CPX    #END_TBL
        BNE    loop
```

Das Basisregister X wird mit der Adresse der Tabelle CST_TBL (\$1000) geladen. Der Wert an der Stelle \$1000 (\$0) wird zu Register D addiert; danach wird X um eins inkrementiert. Sein Wert beträgt nun \$1001.

Register X hat nicht den gleichen Wert wie END_TBL (\$100C), also wird wieder an den Anfang der Schleife gesprungen. Der Wert an der Stelle \$1001 (\$5) wird zu D addiert, und danach Register X um eins inkrementiert. Sein Wert beträgt nun \$1002.

Die Schleife wird solange wiederholt, bis alle Werte in CST_TBL in D aufaddiert worden sind.

2.7.14 Indiziert, Akkumulator Offset

Diese Adressierungsart addiert den Wert in einem der Akkumulatoren zu einem Basis-Indexregister, um die für diesen Befehl gültige effektive Adresse zu erstellen. Als Basis-Indexregister können X, Y, SP oder PC dienen. Als Akkumulatoren können A, B oder D dienen. Beispiel:

```
main:
```

```

LDAB  #$20
LDX   #$600
LDAA B,X
LDY   #$1000
STAA $140,Y

```

Der Wert in B (\$20) wird zum Wert in X (\$600) addiert, um die effektive Adresse zu erstellen (\$620). Der Wert aus Speicherstelle \$620 wird in den Akkumulator A geladen und in die Speicherstelle \$1140 kopiert.

2.7.15 Indiziert-indirekt, D Akkumulator Offset

Dies ist neben der indiziert-indirekten Adressierung mit 16-Bit Offset die komplexeste Adressierungsart des 68HCS12-Prozessors. Hier wird der Wert im Akkumulator D zu einem Basis-Indexregister addiert, um einen Zeiger auf eine Speicherstelle zu erhalten, die die effektive Adresse für diesen Befehl enthält. Als Basis-Indexregister können X, Y, SP oder PC dienen. Beispiel:

```

entry1: NOP
        NOP
entry2: NOP
        NOP
entry3: NOP
        NOP
main:
        LDD #2
        JMP [D, PC]
goto1: DC.W entry1
goto2: DC.W entry2
goto3: DC.W entry3

```

Dieses Beispiel realisiert eine Sprungtabelle (computed goto). Ab Label "goto1" liegen die Zieladressen als Datenwerte im Speicher. Wird **JMP [D, PC]** ausgeführt, zeigt PC auf "goto1" (immer auf die Adresse des nächsten auszuführenden Befehl, selbst wenn es keiner ist). D enthält den Wert 2.

Der **JMP**-Befehl addiert die Werte von D und PC und erhält damit die Adresse von goto2. Die CPU liest den an goto2 gespeicherten Wert, interpretiert ihn als Adresse und springt nach entry2.

2.8 Befehlstypen

Die meisten Befehle einer Instruction Set Architecture können wie in Tabelle 2-6 gezeigt kategorisiert werden.

Tabelle 2-6: Klassifizierung von Befehlen

Befehlstyp	Beispiel
Datentransfer	Register laden und abspeichern, Register vertauschen
Arithmetisch und logisch	Integer-Arithmetik, addieren, subtrahieren, und, oder
Steuerung	Verzweigungen, Sprünge, Return, Unterprogramme
System	Betriebssystemaufrufe, Management des virtuellen Speichers, Low Power
Fließkomma	Fließkomma-Befehle, addieren, subtrahieren, vergleichen
Dezimal	Dezimal addieren, subtrahieren, vergleichen
String	Zeichenkette verschieben, vergleichen, suchen
Grafik	Pixel- und Linienoperationen, Kompression, Dekompression

Tabelle 2-7 gibt eine Übersicht über die Häufigkeit der Benutzung verschiedener Befehlstypen für den SPECint92-Benchmark für einen Intel 80x86-Prozessor.

Tabelle 2-7: Die zehn am häufigsten verwendeten Befehle des 80x86 (SPECint92)

Rang	80x86-Befehl	% Häufigkeit des Aufrufs
1	laden	22%
2	bedingte Verzweigung	20%
3	Vergleich	16%
4	speichern	12%
5	addieren	8%

Tabelle 2-7: Die zehn am häufigsten verwendeten Befehle des 80x86 (SPECint92)

Rang	80x86-Befehl	% Häufigkeit des Aufrufs
6	and	6%
7	subtrahieren	5%
8	Register-zu-Register-Transfer	4%
9	call	1%
10	return	1%
Insgesamt		96%

2.9 Assemblerbefehle des 68HCS12

2.9.1 Daten bewegen

Diese Klasse von Befehlen ist die am meisten verwendete. Damit werden Daten zwischen Speicherzellen, Registern oder Speicherzellen und Registern transferiert. Die häufige Verwendung ist deshalb notwendig, weil Prozessoren nur eine kleine Anzahl von Registern besitzen, und Programme eine große Anzahl von Variablen haben.

Tabelle 2-8: Load- und Store-Befehle

Mnemonik	Funktion	Operation
Load-Befehle		
LDAA	lade A	$(M) \Rightarrow A$
LDAB	lade B	$(M) \Rightarrow B$
LDD	lade D	$(M:M+1) \Rightarrow A:B$
LDS	lade SP	$(M:M+1) \Rightarrow SP_H:SP_L$
LDX	lade Indexregister X	$(M:M+1) \Rightarrow X_H:X_L$
LDY	lade Indexregister Y	$(M:M+1) \Rightarrow Y_H:Y_L$

Tabelle 2-8: Load- und Store-Befehle

Mnemonik	Funktion	Operation
LEAS	lade SP mit effektiver Adresse	effektive Adresse \Rightarrow SP
LEAX	lade X mit effektiver Adresse	effektive Adresse \Rightarrow X
LEAY	lade Y mit effektiver Adresse	effektive Adresse \Rightarrow Y
Store-Befehle		
STAA	Speichere A	(A) \Rightarrow M
STAB	Speichere B	(B) \Rightarrow M
STD	Speichere D	(A) \Rightarrow M, (B) \Rightarrow M+1
STS	Speichere SP	(SP _H ;SP _L) \Rightarrow M:M+1
STX	Speichere X	(X _H ;X _L) \Rightarrow M:M+1
STY	Speichere Y	(Y _H ;Y _L) \Rightarrow M:M+1

Alle Datentransportbefehle mit der Ausnahme solcher, bei den Register A oder B involviert sind, beziehen sich auf 16-Bit Werte. Mit der Ausnahme der Austauschbefehle verändern die Befehle nicht den Inhalt der Quelle. Selbst wenn der Befehl „move“ heißt, wird hier lediglich kopiert, und der Wert in der Quelle kann weiterverwendet werden.

Register D ist nur eine andere Sicht auf die Register A und B. Man kann also nicht gleichzeitig Register D und A oder B verwenden. Im Speicher wird der niederwertige Teil des Wortes und die höhere Adresse geschrieben („Big Endian“). Als Adresse eines 16-Bit Wortes gibt man immer die Adresse des höherwertigen Bytes (most significant byte, MSB) an.

Der Rechner arbeitet am schnellsten, wenn das MSB eines 16-Bit-Wertes an einer geraden Adresse abgelegt wird. Man nennt diese Art der Speicherung „ausgerichtet“ (*engl. „aligned“*). Man kann mit der Assemblerdirektive

```
ORG (*+1) &$ffff
```

den Current Location Counter auf eine gerade Adresse zwingen.

8-Bit Akkumulator Ladebefehle

Die 8-Bit Akkumulatorregister A und B werden mit den Anweisungen

LDAA

LDAB

geladen. Diese Anweisungen haben einen Operanden. Diese Anweisungen verändern das CCR, d.h. im CCR ist ohne weitere Befehle erkennbar, ob der geladene Wert null, negativ oder positiv ist. Anstatt 0 zu laden, kann man die Befehle CLRA und CLRB verwenden, die etwas effizienter sind.

Beispiele:

```
LDAA #'A' ; lade Akkumulator A mit ASCII-Wert von Zeichen "A" (65)
LDAB $760 ; lade B mit Wert in Speicherzelle $760
LDAA #-6 ; lade A mit -6
LDAB 2,X ; lade B mit Inhalt der Speicherzelle, auf die der Wert in
           ; X + 2 zeigt (geschrieben als (X) + 2)
LDAA var1 ; lade A mit Wert in Speicherzelle var1. var1 wird mit
           ; einer DS, DB oder EQU-Direktive definiert.
```

8-Bit Akkumulator Speicherbefehle

Die Inhalte der 8-Bit Akkumulatorregister A und B können mit folgenden Anweisungen in den Hauptspeicher geschrieben werden:

STAA

STAB

Diese Anweisungen haben einen Operanden, der die Speicherzelle angibt. Auch diese Anweisungen beeinflussen das CCR, so dass folgende Verzweigungsbefehle vorher keinen Vergleich mehr ausführen müssen.

Beispiele:

```
STAA $30 ; Speichere Wert in A nach Speicherzelle $30 (zero page)
STAB $1200 ; Speichere Wert in B nach Speicherzelle $1200
STAB var1 ; Speichere Wert in B an die durch var1 bezeichnete
           ; Speicherzelle. var1 kann mit DS, DB oder EQU definiert
           ; werden
STAA 25,SP ; Speichere Wert von A an die Speicherstelle mit der Adresse
           ; (SP) + 25, wobei (SP) der Inhalt des Stackpointer-Registers
           ; ist.
```

16-Bit Lade- und Speicherbefehle

Die Anweisungen zum Laden und Speichern der 16-Bit Register entsprechen denen der 8-Bit Register. Der erste Buchstabe des Register findet sich im dritten Buchstaben des Befehls, also z.B LDS für das Laden des SP-Registers oder STY für das Speichern des Y-Registers. Beispiele:

```

LDX #0815      ; lade X mit der Konstanten 815 (dezimal)
LDD #0101      ; lade D mit der binären Konstanten 000000000000101
LDX #-28000    ; lade X mit der Konstanten -28000 (dezimal)
LDX 0,Y        ; lade Indexregister X mit dem Inhalt des Wortes an der
                ; Speicherstelle (Y), wobei (Y) den Inhalt des Registers
                ; Y bedeutet. Steht in Y z.B. $2008 und an der Speicher-
                ; stelle $2008 der Wert $0816, steht anschließend in X der
                ; Wert $0816
LDY #ABCD      ; Lade Indexregister Y mit der Adresse von Speicherstelle
                ; ABCD, wobei ABCD mit einer DB, DW, DS oder EQU-Direktive
                ; definiert worden ist.
STD ABCD       ; speichert den Wert von D in Speicherzelle ABCD
STD 0,Y        ; speichert den Wert von D an die Speicherstelle, auf die
                ; Y zeigt, in unserem Fall also auch an ABCD (aufgrund des
                ; Befehls LDY #ABCD weite oben)

```

Laden der effektiven Adresse

Es gibt Befehle, um eine effektive Adresse in die Register X, Y oder SP zu laden. Das CCR wird durch diese Operationen nicht beeinflusst. Beispiele:

```

LEAX 0,PC ; lade Indexregister X mit der Adresse des nächsten
            ; Befehls
LEAY B,Y ; lade Y mit (Y) + (B), wobei (ZZ) der Wert im Register ZZ
            ; bedeutet
LEAX 10,X ; lade X mit (X) + 10, wobei (X) der Wert im Register X
            ; bedeutet

```

Mit Hilfe der LEAS-Anweisung kann man auf dem Stack Platz für lokale Variable belegen und wieder freigeben. Diese Vorgehensweise besprechen wir später detaillierter.

Register zu Register Transferbefehle

Um Daten von einem Register in ein anderes zu transferieren, gibt es den folgenden Befehl:

```
tfr Quellregister,Zielregister
```

Alle weiteren Transfer-Befehle existieren nur aus Gründen der Aufwärtskompatibilität mit Vorgängerversionen des 68HCS12-Prozessors. Sie bewirken im Prinzip das gleiche wie der tfr-Befehl, beeinflussen aber im Gegensatz zu diesem das CCR. Beispiele:

```
tfr    A, B ; kopiere Inhalt von A nach B (Zieloperand immer rechts)
tab   ; kopiere Inhalt von A nach B und setze CCR
tfr    X, B ; kopiere das LSB von X nach B
tfr    A, D ; kopiere von A nach D mit Vorzeichenerweiterung. Akkumulator
          ; B enthält den ursprünglichen Wert von A, und A enthält $FF
          ; $00, abhängig vom Vorzeichen des ursprünglichen Wertes in A
sex   A, D ; gleiche Wirkung wie tfr A, D
```

Tabelle 2-9: Transfer- und Austauschbefehle

Mnemonik	Funktion	Operation
Transfer-Befehle		
TAB	transferiere A nach B	$(A) \Rightarrow B$
TAP	transferiere A nach CCR	$(A) \Rightarrow CCR$
TBA	transferiere B nach A	$(B) \Rightarrow A$
TFR	transferiere Register nach Register	$(A, B, CCR, D, X, Y, \text{ oder } SP) \Rightarrow A, B, CCR, D, X, Y, \text{ oder } SP$
TPA	transferiere CCR nach A	$(CCR) \Rightarrow A$
TSX	transferiere SP nach X	$(SP) \Rightarrow X$
TSY	transferiere SP nach Y	$(SP) \Rightarrow Y$
TXS	transferiere X nach SP	$(X) \Rightarrow SP$
TYS	transferiere Y nach SP	$(Y) \Rightarrow SP$
Austausch-Befehle		
EXG	Tausche Register mit Register	$(A, B, CCR, D, X, Y, \text{ oder } SP) \Leftrightarrow A, B, CCR, D, X, Y, \text{ oder } SP$
XGDX	Tausche D mit X	$(D) \Leftrightarrow (X)$

Tabelle 2-9: Transfer- und Austauschbefehle

Mnemonik	Funktion	Operation
XGDY	Tausche D mit Y	$(D) \Leftrightarrow (Y)$
Vorzeichenerweiterung		
SEX	Erweiterte Vorzeichen für 8-Bit Operand	Vorzeichenerweiterung (A, B oder CCR) \Rightarrow D,X,Y oder SP

Austauschbefehle

Es gibt einen Austauschbefehl, bei dem Registerwerte gegenseitig ersetzt werden:

```
EXG Register1, Register2
```

Die beiden Befehle XGDX und XGDY existieren nur aus Gründen der Aufwärtskompatibilität mit Vorgängerversionen des 68HCS12. Es ist möglich aber davon abzuraten, 8-Bit Register mit 16-Bit Registern zu vertauschen. Beispiele:

```
EXG X, Y      ; tausche Werte in den Registern X und Y
EXG Y, X      ; gleich wie EXG X Y
EXG A, B      ; tausche Werte in den Register A und B
```

Speicher zu Speicher Move-Befehle

Zwei Befehle existieren, um Werte direkt von einem Platz im Speicher zu einem anderen zu schieben, ohne dabei ein Register zu benutzen:

```
MOVB Quelle, Ziel; kopiert ein Byte von Quelle nach Ziel
```

```
MOVW Quelle, Ziel; kopiert ein Wort von Quelle nach Ziel
```

Beispiele:

```
MOVB #25, $2000    ; setze Speicherzelle $2000 auf Wert 25 dezimal
MOVW var1, var2    ; setze Wort var2 auf den Wert von Wort var1
MOVB 1,X+, 1,Y+    ; kopiert den Wert, auf den X zeigt in die Speicher-
                   ; zelle, auf die Y zeigt, und inkrementiert danach
                   ; X und Y um eins
```

Tabelle 2-10: Move-Befehle

Mnemonik	Funktion	Operation
MOVB	Kopiere Byte (8-Bit)	$(M_1) \Rightarrow M_2$
MOVW	Kopiere Wort (16 Bit)	$(M:M+1_1) \Rightarrow M:M+1_2$

Deklaration von Tabellen und Arrays

Alle höheren Programmiersprachen kennen komplexere Datenstrukturen wie z.B. Tabellen (für konstante Werte) oder Arrays (für variable Werte). In diesen Datenstrukturen wird auf jedes Element über einen Index zugegriffen. In C würde man ein Array von 10 unsigned Byte so deklarieren:

```
unsigned char array1[10];
```

Eine Tabelle mit 5 initialisierten Integerwerten sähe so aus:

```
const int tabelle1[5] = {1, 3, 5, 7, 9};
```

Rechnerhardware kennt keine Tabellen oder Arrays, außer dem großen Byte-Array des Hauptspeichers. Mit Hilfe der vorhandenen Befehle muss die Funktion einer Tabelle oder eines Arrays nachgebildet werden.

In unserem 68HCS12 Assembler benutzen wir die Assembldirektiven DS (define space), um Speicherplatz für Arrays zu reservieren. Als Argument wird dabei die Anzahl der Byte (DS.B), Worte (DS.W) oder Langworte (DS.L) angegeben, die zu reservieren sind. Der Assembler macht dabei nichts anderes, als dass er den CLC um die entsprechende Anzahl Byte inkrementiert.

Die Direktive DC.B (define constant byte) oder DC.W (define constant word) wird benutzt, um Konstanten zu definieren. Werden mehrere Konstanten hintereinander gelegt, erhält man eine Tabelle. In eingebetteten Systemen werden Konstanten typischerweise im ROM oder EEPROM abgelegt, da die Programme nicht von einer Festplatte geladen werden können. Beispiel:

```
ORG $1000      ; Beginn des RAM
array1: DS.B 10    ; unsigned char array1[10]

ORG $4000      ; Beginn des Flash-EEPROM
tabelle1:
DC.W 1,3,5,7,9 ; const int[5]={1,3,5,7,9}
```

Tabellen oder Arrays von Zeichen nennt man auch Zeichenketten (Strings). Diese lassen sich wie von C gewohnt definieren:

```
theString:DC.B "A nice null terminated string", 0
```

Zugriff auf Tabellen und Arrays

Man greift auf Elemente einer Tabelle oder eines Arrays entweder sequentiell oder wahlfrei zu. In beiden Fällen wird ein Index verwendet, um ein Element zu adressieren. In der Softwaretechnik hat das erste Element einer Tabelle stets den Index 0. Im folgenden Beispiel wird jedes Byte des Array array1 auf den Wert 32 gesetzt (ASCII-Wert für Leerzeichen):

```
LDX #array1      ; Definition siehe oben, Adresse von Element 0
LOOP: MOVB #' ', 1,X+ ; speichere Leerzeichen an momentaner Adresse und
      ; inkrementiere Zeige um ein Byte
      CPX #array1+10
      BNE LOOP
```

Es ist keine gute Vorgehensweise, die Größe einer Datenstruktur verstreut über eine Software einzuprogrammieren. Es wäre besser gewesen, wenn wir das Ende der Struktur markiert hätten und mit der Differenz gearbeitet hätten:

```
ORG $1000      ; Beginn des RAM
array1: DS.B 10      ; unsigned char array1[10]
array1Size EQU * - array1
      LDX #array1      ; Definition siehe oben, Adresse von Element 0
LOOP: MOVB #' ',1,X+ ; speichere Leerzeichen an momentaner Adresse und
      ; inkrementiere Zeige um ein Byte
      CPX #array1+array1Size
      BNE LOOP
```

Alternativ kann man auch so arbeiten:

```
ORG $1000      ; Beginn des RAM
array1Size EQU 10      ; Definition der Größe
array1: DS.B arraySize ; unsigned char array1[10]
      LDX #array1      ; Definition siehe oben, Adresse von Element 0
LOOP: MOVB #' ',1,X+ ; speichere Leerzeichen an momentaner Adresse und
      ; inkrementiere Zeige um ein Byte
      CPX #array1+array1Size
      BNE LOOP
```

Um wahlfrei auf einzelne Elemente eines Arrays zugreifen zu können, muss zunächst die Adresse ausgerechnet werden. In C nimmt uns das der Compiler ab:

```
a = tabelle1[3]; /* wir brauchen über die Adresse nicht nachzudenken */
```

In Assembler ist es etwas aufwändiger. Die Adresse des i -ten Elementes ist die Basisadresse + $i \cdot (\text{Elementgröße in Byte})$. Um zum Beispiel den Wert in array1[1] zum Wert in array1[5] zu addieren und in array1[8] zu speichern würden wir wie folgt vorgehen:

```
LDAA array1+1 ; array1[1]
ADDA array1+5 ; + array1[5]
STAA array1+8 ; nach array1[8]
```

Das ist nicht soviel anders als in der C-Zeigerarithmetik. Interessanter wird es, wenn wir nicht ein Array von Byte bearbeiten müssen, sondern ein Wortarray (oder noch größere Elemente wie structs), und wenn wir das Element zur Laufzeit auswählen können müssen. Oben war es ja fest im Code eingeprограмmiert. Beispiel in C:

```
unsigned char i;
... /* b wird berechnet */
res = tabelle1[i]; /* i kann jeden Wert haben, tabelle ist eine Worttabelle ! */
```

In Assembler können wir so vorgehen:

```
LDX #tabelle1
LDAB i           ; Index i holen
ASLB            ; Index mit 2 multiplizieren (ein Wort = 2 Byte)
MOVW b,x,res    ; Index = 2*i + x
```

2.9.2 Rechnen

Der Prozessor kann Zahlen addieren, multiplizieren und dividieren. Diese Operationen finden in den Registern A, B und D statt. Die Ergebnisse einer Multiplikation und Division werden z.T. in den dafür zweckentfremdeten Indexregistern X und Y gehalten, da die anderen Register alleine nicht groß genug sind, um die Ergebnisse zu fassen. Es gibt Inkrement- und Dekrement-Befehle, um Speicher- oder Registerinhalte um einen konstanten Wert zu erhöhen oder erniedrigen.

Weiterhin wird für die Ermittlung von Adressen Zeigerarithmetik unterstützt.

Tabelle 2-11: Additions- und Subtraktionsbefehle

Mnemonik	Funktion	Operation
Additionsbefehle		
ABA	Addiere B zu A	$(A) + (B) \Rightarrow A$
ABX	Addiere B zu X	$(B) + (X) \Rightarrow X$
ABY	Addiere B zu Y	$(B) + (Y) \Rightarrow X$
ADCA	Addiere mit Carry zu A	$(A) + (M) + C \Rightarrow A$
ADCB	Addiere mit Carry zu B	$(B) + (M) + C \Rightarrow B$
ADDA	Addiere ohne Carry zu A	$(A) + (M) \Rightarrow A$
ADDB	Addiere ohne Carry zu B	$(B) + (M) \Rightarrow B$
ADDD	Addiere zu D	$(A:B) + (M:M+1) \Rightarrow A:B$
Subtraktionsbefehle		
SBA	Subtrahiere B von A	$(A) - (B) \Rightarrow A$
SBCA	Subtrahiere mit Borrow von A	$(A) - (M) - C \Rightarrow A$
SBCB	Subtrahiere mit Borrow von B	$(B) - (M) - C \Rightarrow B$
SUBA	Subtrahiere Speicher von A	$(A) - (M) \Rightarrow A$
SUBB	Subtrahiere Speicher von B	$(B) - (M) \Rightarrow B$
SUBD	Subtrahiere Speicher von D (A:B)	$(D) - (M:M+1) \Rightarrow D$

Tabelle 2-12: Multiplikations- und Divisionsbefehle

Mnemonik	Funktion	Operation
Multiplikationsbefehle		
EMUL	16 x 16 Bit Multiplikation (unsigned)	$(D) \times (X) \Rightarrow Y:D$
EMULS	16 x 16 Bit Multiplikation (signed)	$(D) \times (X) \Rightarrow Y:D$
MUL	8 x 8 Bit Multiplikation (unsigned)	$(A) \times B \Rightarrow A:B$
Divisionsbefehle		
EDIV	32 durch 16 Bit Division (unsigned)	$(Y:D) \div (X) \Rightarrow Y, Rest \Rightarrow D$
EDIVS	32 durch 16 Bit Division (signed)	$(Y:D) \div (X) \Rightarrow Y, Rest \Rightarrow D$
FDIV	16 durch 16 Bit Division (fractional)	$(D) \div (X) \Rightarrow X, Rest \Rightarrow D$
IDIV	16 durch 16 Bit Ganzzahldivision (unsigned)	$(D) \div (X) \Rightarrow X, Rest \Rightarrow D$
IDIVS	16 durch 16 Bit Ganzzahldivision (signed)	$(D) \div (X) \Rightarrow X, Rest \Rightarrow D$

Tabelle 2-13: Dekrement- und Inkrementbefehle

Mnemonik	Funktion	Operation
Dekrementbefehle		
DEC	Dekrementiere Speicherzelle	$(M) - 1 \Rightarrow M$
DECA	Dekrementiere A	$(A) - 1 \Rightarrow A$
DECB	Dekrementiere B	$(B) - 1 \Rightarrow B$
DES	Dekrementiere SP	$(SP) - \$0001 \Rightarrow SP$
DEX	Dekrementiere X	$(X) - \$0001 \Rightarrow X$

Tabelle 2-13: Dekrement- und Inkrementbefehle

Mnemonik	Funktion	Operation
DEY	Dekrementiere Y	(Y) - \$0001 \Rightarrow Y
Inkrementbefehle		
INC	Inkrementiere Speicherzelle	(M) + 1 \Rightarrow M
INCA	Inkrementiere A	(A) + 1 \Rightarrow A
INCB	Inkrementiere B	(B) + 1 \Rightarrow B
INS	Inkrementiere SP	(SP) + \$0001 \Rightarrow SP
INX	Inkrementiere X	(X) - \$0001 \Rightarrow X
INY	Inkrementiere Y	(Y) + \$0001 \Rightarrow Y

Tabelle 2-14: Clear-, Einerkomplement- und Zweierkomplementbefehle

Mnemonik	Funktion	Operation
CLC	Setzte C-Bit in CCR auf 0 (clear C)	0 \Rightarrow C
CLI	Setzte I-Bit in CCR auf 0 (clear I)	0 \Rightarrow I
CLR	Setze Speicherzelle auf 0	\$00 \Rightarrow M
CLRA	Setze A auf 0 (clear A)	\$00 \Rightarrow A
CLRB	Setze B auf 0 (clear B)	\$00 \Rightarrow B
CLV	Setze V-Bit in CCR auf 0 (clear V)	0 \Rightarrow V
COM	1er-Komplement Speicherzelle	\$FF - (M) \Rightarrow M oder (\bar{M}) \Rightarrow M
COMA	1er-Komplement A	\$FF - (A) \Rightarrow A oder (\bar{A}) \Rightarrow A
COMB	1er-Komplement B	\$FF - (B) \Rightarrow B oder (\bar{B}) \Rightarrow B
NEG	2er-Komplement Speicherzelle	\$00 - (M) \Rightarrow M oder (\bar{M}) + 1 \Rightarrow M

Tabelle 2-14: Clear-, Einerkomplement- und Zweierkomplementbefehle

Mnemonik	Funktion	Operation
NEGA	2er-Komplement A	\$00 - (A) \Rightarrow A oder $(\bar{A}) + 1 \Rightarrow A$
NEGB	2er-Komplement B	\$00 - (B) \Rightarrow B oder $(\bar{B}) + 1 \Rightarrow B$

Clear-Befehle

Es gibt einige Befehle, mit denen Register und Speicherzellen effizient auf 0 gesetzt werden können:

CLR
CLRA
CLRB

Das CCR wird bei dieser Operation beeinflusst. Register D wird am effizientesten durch die Folge CLRA, CLRB auf 0 gesetzt. Beispiele:

```
CLRA      ; Akkumulator A wird auf 0 gesetzt
CLR       $995 ; Speicherzelle an Adresse $995 enthält anschließend 0
```

2.9.3 Testen und Vergleichen

Ein Vergleichsbefehl vergleicht zwei Werte und wird zusammen mit einer bedingten Verzweigung für Kontrollstrukturen (z.B. if-then-else, while, switch-case) verwendet. Ein Testbefehl prüft einen Wert daraufhin, ob er null ist. All diese Befehle setzen Bits im CCR, die von den Verzweigungsbefehlen für ihre Entscheidung benutzt werden.

Tabelle 2-15: Vergleichs- und Testbefehle

Mnemonik	Funktion	Operation
Vergleichsbefehle		
CBA	Vergleiche A mit B	(A) - (B)
CMPA	Vergleiche A mit Speicherzelle	(A) - (M)
CMPB	Vergleiche B mit Speicherzelle	(B) - (M)
CPD	Vergleiche D mit Speicherzelle	(A:B) - (M:M+1)

Tabelle 2-15: Vergleichs- und Testbefehle

Mnemonik	Funktion	Operation
CPS	Vergleiche SP mit Speicherzelle	(SP) - (M:M+1)
CPX	Vergleiche X mit Speicherzelle	(X) - (M:M+1)
CPY	Vergleiche Y mit Speicherzelle	(Y) - (M:M+1)
Testbefehle		
TST	Teste Speicher auf 0 oder minus	(M) - 0
TSTA	Teste A auf 0 oder negativ	(A) - 0
TSTB	Teste B auf 0 oder negativ	(B) - 0

Beispiele:

```

CMPA      #9 ; vergleiche A mit dem konstanten Wert 9
CBA       ; subtrahiere B von A, setze CCR abhängig von Ergebnis,
           ; Ergebnis wird aber nirgendwo abgespeichert, d.h. A und
           ; B behalten ihren ursprünglichen Wert
CPX  var1 ; vergleiche Indexregister X mit dem Inhalt an der Speicher-
           ; zelle mit der Adresse var1
CPX  #var1 ; vergleiche die Adresse in X mit der Adresse von var1
TST  var2 ; vergleiche Wert an der Stelle var2 mit 0
CMPB 5,Y  ; vergleiche den Inhalt von B mit dem Inhalt der Speicher-
           ; zelle an der Adresse (Y)+5, wobei (Y) der Inhalt von Y ist

```

Es ist wichtig, dass beide Operanden beim Vergleich entweder signed oder unsigned sind, ansonsten funktioniert der Vergleich nicht über den gesamten Wertebereich. Möchte man signed und unsigned Bytes über deren gesamten Bereich vergleichen, muss man die Operanden zuvor auf Words erweitern.

2.9.4 Logische und Bit-Operationen

Die Befehle zur Booleschen Logik und Bitbefehle sind in den beiden folgenden Tabellen zusammengefasst.

Tabelle 2-16: Befehle zur Booleschen Logik

Mnemonik	Funktion	Operation
ANDA	UND A mit Speicherzelle	$(A) \bullet (M) \Rightarrow A$
ANDB	UND B mit Speicherzelle	$(B) \bullet (M) \Rightarrow B$
ANDCC	UND CCR mit Speicherzelle (einzelne Bit in CCR löschen)	$(CCR) \bullet (M) \Rightarrow CCR$
EORA	Exklusiv-ODER A mit Speicherzelle	$(A) \oplus (M) \Rightarrow A$
EORB	Exklusiv-ODER B mit Speicherzelle	$(B) \oplus (M) \Rightarrow B$
ORAA	ODER A mit Speicherzelle	$(A) + (M) \Rightarrow A$
ORAB	ODER B mit Speicherzelle	$(B) + (M) \Rightarrow B$
ORCC	ODER CCR mit Speicherzelle (einzelne Bit in CCR setzen)	$(CCR) + (M) \Rightarrow CCR$

Tabelle 2-17: Bit-Befehle

Mnemonik	Funktion	Operation
BCLR	Setze Bits in Speicherzelle auf 0	$(M) \bullet (\overline{mm}) \Rightarrow M$
BITA	teste Bits in A	$(A) \bullet (M)$
BITB	teste Bits in B	$(B) \bullet (M)$
BSET	Setze Bits Speicherzelle auf 1	$(M) + (mm) \Rightarrow M$

Wird einer dieser Befehle mit unmittelbarer Adressierung eingesetzt, nennt man den Operanden die **Maske**. Alle Befehle beeinflussen die N- und Z-Bits je nach dem Ergebnis der Operation, setzen das V-Bit auf 0 und lassen die anderen Bits unverändert.

Die Bit-Test-Befehle entsprechen den AND-Befehlen, nur wird das Ergebnis nicht gespeichert. Dies entspricht der Beziehung zwischen den Compare-Befehlen und den Subtraktionsbefehlen.

Für das Manipulieren einzelner Bits ist es bequem, entsprechende „Bitmasken“ zu definieren. Beispiel:

```
Bit0    EQU  %00000001
Bit1    EQU  %00000010
Bit2    EQU  %00000100
Bit3    EQU  %00001000
Bit4    EQU  %00010000
Bit5    EQU  %00100000
Bit6    EQU  %01000000
Bit7    EQU  %10000000
```

Bitoperationen sind immer nur auf Byte-Daten möglich. In den folgenden Beispielen verwenden wir die oben definierten Bitmasken.

Bit setzen (auf "1")

Mit den OR-Befehlen kann man einzelne Bit in einem Byte auf "1" setzen. Das ist besonders für das Setzen von Bits in den beiden Akkumulatoren interessant, da hier die BSET-Befehle nicht zur Verfügung stehen. Beispiel:

```
ORAA #Bit4      ; setze Bit 4 in Akkumulator A
ORAB #Bit2      ; setze Bit 2 in Akkumulator B
```

Wenn wir Bits im Speicher setzen wollen, bieten sich die BSET-Befehle an:

```
BSET $3000, #B1      ; setze Bit 1 in der Speicherzelle bei $3000
BSET var1, #B0 + B7  ; setze das unterste und oberste Bit in var1
```

Bit zurücksetzen (auf "0")

Die Vorgehensweise beim Zurücksetzen von Bits entspricht der beim Setzen von Bits, nur dass hier der AND-Befehl bzw. BCLR-Befehl verwendet wird. Beispiel:

```
ANDA #~Bit4        ; setze Bit 4 im Akkumulator A auf "0"
ANDB #~Bit1        ; setze Bit 1 im Akkumulator B auf "0"
```

Wenn Bit im Speicher zurückgesetzt werden sollen, kann die BCLR-Anweisung verwendet werden:

```
BCLR var1, #Bit5    ; setze Bit 5 in Speicherstelle var1 auf "0"
```

```
BCLR $3000, #Bit2           ; setze Bit 2 in Speicherstelle $3000 auf "0"
```

Bit komplementieren

Mit der Exklusiv-Oder-Funktion kann ein Bit negiert werden. Beispiel:

```
EORA #Bit2                 ; toggle Bit 2 in Akkumulator A
EORB #Bit7                 ; toggle Bit 7 in Akkumulator B
```

Möchte man Bits im Hauptspeicher negieren, müssen diese zuerst in einen Akkumulator geladen werden, und nach der Negation wieder in den Hauptspeicher:

```
LDAA $3000                 ; lade das Byte in den Akkumulator
EORA #Bit4                 ; negiere Bit 4
STAA $3000                 ; und wieder zurück in die Speicherzelle
```

Auf bestimmte Bit testen

Weitere Beispiele:

```
BITA #%1000      ; verzweige nach loop1, wenn Bit 3 (von rechts mit
                  ; 0 beginnend gezählt) im Akkumulator A nicht 0 ist
BNE  loop1
ANDA #~8         ; lösche Bit 3 im Akkumulator A
EORA #$01        ; negiere das unterste Bit in Akkumulator A
ORAB var2        ; Verknüpfe den Inhalt von B mit dem Inhalt an der
                  ; Speicherzelle var2. Dies setzt in B die Bits, die in
                  ; var2 auch gesetzt sind
```

Es gibt zusätzliche Bitbefehle, um einzelne Bit im CCR zu setzen bzw. zurückzusetzen. Beispiele:

```
CLI             ; setze Interrupt-Flag zurück
SEC             ; setze Carry-Bit auf 1
```

Bit-Befehle bei der Ein-und Ausgabeprogrammierung

Die Bit-Setz- und Testbefehle sind besonders bei der Programmierung der Ein- und Ausgabeausteine hilfreich. Zum Ein- und Ausschalten der LEDs auf unserem Labor-Board kann man diese Befehle gut gebrauchen, da jede LED mit einem Bit im Prozessor verbunden ist. Beispiele:

```
BSET PORTB,#%10000001  ; Schalte die oberste und unterste LED ein
BCLR PORTB,#%11111111  ; Schalte alle LEDs aus
```

Bits schieben und rotieren

Für den 68HCS12 gibt es für die Akkumulatoren A und B sowie Speicherzellen Schiebe- und Rotierbefehle. Alle Befehle schieben das höchstwertige Bit durch das Carry-Bit, um längere Schiebefehle zu ermöglichen. Da logische und arithmetische Linksschiebeoperationen identisch sind, gibt es keine eigenen logischen Linksschiebeoperationen; die unterschiedlichen Mnemoniks repräsentieren den gleichen Maschinencode.

Tabelle 2-18: Schiebe- und Rotierbefehle

Mnemonik	Funktion	Operation
Logische Schiebefehle		
LSL LSLA LSLB	Bits in Speicherzelle nach links schieben Bits in A nach links schieben Bits in B nach links schieben	
LSLD	Bits in D nach links schieben	
LSR LSRA LSRB	Bits in Speicherzelle nach rechts Bits in A nach rechts schieben Bits in B nach rechts schieben	
LSRD	Bits in D nach rechts schieben	
Arithmetische Schiebefehle		
ASL ASLA ASLB	Bits in Speicherzelle nach links schieben Bits in A nach links schieben Bits in B nach links schieben	
ASLD	Bits in D nach links schieben	
ASR ASRA ASRB	Bits in Speicherzelle nach rechts Bits in A nach rechts schieben Bits in B nach rechts schieben	
Rotierbefehle		

Tabelle 2-18: Schiebe- und Rotierbefehle

Mnemonik	Funktion	Operation
ROL ROLA ROLB	rotiere Bits in Speicherzelle links durch Carry rotiere Bits in A links durch Carry rotiere Bits in B links durch Carry	
ROR RORA RORB	rotiere Bits in Speicherzelle rechts durch Carry rotiere Bits in A rechts durch Carry rotiere Bits in B rechts durch Carry	

Bit-Manipulation in C

C ist eine Programmiersprache, die das Manipulieren von Bit ermöglicht. C unterstützt auch Bitfelder. Allerdings sind die einzelnen Positionen eines Bit in einem Feld nicht spezifiziert, so dass diese Funktionalität für die hardwarenahe Programmierung nicht direkt brauchbar ist. Beispiel in C und Assembler:

```

18:      #define Bit0 (0x1)
19:      #define Bit1 (0x2)
20:      #define Bit2 (0x4)
21:      #define Bit3 (0x8)
22:
23:      #define var1 *((unsigned char *) 0x3000)
24:      unsigned char var2;
29:
30:      var1 |= Bit2; /* setze Bit 2 von var1 auf 1 */
0003 1c300004      BSET  $3000,#4
31:      var1 &= ~Bit0; /* setze Bit 0 von var1 auf 0 */
0007 1d300001      BCLR  $3000,#1
32:      var1 ^= Bit1; /* Toggle Bit 1 von var 1 */
000b f63000      LDAB   $3000
000e c802        EORB   #2
0010 7b3000      STAB   $3000
33:
34:      if (var1 & Bit3) {
0013 1f30000802    BRCLR $3000,#8,*+7 ;abs = 001a
35:          var2 = var1; /* nur wenn Bit 3 in var1 gesetzt ist */
0018 6b80        STAB   0,SP
36:      }
37:

```

```
38: /* extrahiere Bit 3 bis 6 von var1 und speicher in var2 */
39: var2 = (var1 >> 3) & 0xff;
001a f63000      LDAB   $3000
001d 54          LSRB
001e 54          LSRB
001f 54          LSRB
0020 6b80      STAB   0,SP
```

2.9.5 Verzweigung und Schleifen

Zwei der wichtigsten Programmierstrukturen sind Verzweigungen (z.B. *if... then ... else*) und Schleifen (z.B. *do ... while*). Diese von den Hochsprachen her bekannten Konstrukte müssen in der Maschinensprache abbildbar sein. Jede Rechnerhardware stellt dafür Befehle zur Verfügung, die man als „bedingte Verzweigungen“ bezeichnet. Die Verzweigungsbedingungen werden dabei durch Bits in einem Statusregister, im Falle unseres 68HCS12 des CCR gebildet.

Vorhergehende Befehle setzen Bits im CCR (z.B. kann ein Test- oder Vergleichsbefehl das Z-Bit setzen, um anzugeben, dass das Ergebnis 0 war). Die bedingte Verzweigung reagiert nun auf dieses Bit und springt abhängig von seinem Wert zu einer Sprungmarke, oder macht mit dem nächsten Befehl weiter.

Von den acht Bit im CCR werden vier für bedingte Verzweigungen benutzt:

- Z-Bit wird gesetzt, wenn das Ergebnis der vorhergegangenen Operation 0 war, z.B. bei einer Addition, Subtraktion, einem Ladebefehl oder einem Vergleich.
- N-Bit wird gesetzt, wenn das Ergebnis der vorhergegangenen Operation negativ war.
- C-Bit wird gesetzt, wenn zur korrekten Darstellung der vorhergegangen Operation ein Carry oder Borrow aufgetreten ist. Bei einer Addition oder Subtraktion von unsigned Zahlen zeigt das C-Bit oder Carry-Flag einen Überlauf an. Das C-Bit wird von fast allen arithmetischen Befehlen beeinflusst, mit Ausnahme der Inkrement- und Dekrement-Befehle.
- V-Bit entspricht dem C-Bit bei signed Werten. Tritt bei einer Operation ein Überlauf auf, wird dieses Bit gesetzt. Das V-Bit wird von fast allen arithmetischen Befehlen beeinflusst, mit Ausnahme der Inkrement- und Dekrement-Befehle.

Bei der Programmierung ist es hilfreich, im *S12CPUV2 Reference Manual* nachzuschlagen, welche Bit im CCR durch einen Befehl beeinflusst werden. Die Darstellung dort ist wie folgt (Beispiel LDS-Befehl):

	S	X	H	I	N	Z	V	C
CCR Details:	—	—	—	—	Δ	Δ	0	—

N: Set if MSB of result is set; cleared otherwise
 Z: Set if result is \$0000; cleared otherwise
 V: 0; cleared

Das Dreieck-Symbol kennzeichnet, das der Wert des Bits abhängig von der Operation ist. Die Art der Abhängigkeit ist unter der Grafik beschrieben. Im obigen Beispiel wird das N-Bit gesetzt, wenn das MSB des Ergebnisses, d.h. das MSB des Stackpointers SP nach dem Laden auf 1 steht; ansonsten wird das N-Bit auf 0 gesetzt.

Es gibt zwei Klassen von Verzweigungsbefehlen: solche mit kurzem Offset [-128...+127] und solche mit langem Offset [-32768...32767]. Bis auf dieses Merkmal sind die Befehle funktional identisch und unterscheiden sich in ihrem Mnemonik nur durch das vorangestellte L bei den Langbefehlen.

Tabelle 2-19: Verzweigungsbefehle Kurzform

Mnemonik	Funktion	Operation
Unbedingte Verzweigungen		
BRA	Branch always, immer verzweigen	1=1
BRN	Branch never, nie verzweigen	1=0
Bedingte Verzweigungen		
BCC	Branch if carry clear, verzweige wenn Carry-Bit auf 0	C=0
BCS	Branch if carry set, verzweige wenn Carry-Bit auf 1	C=1
BEQ	Branch if equal, verzweige wenn Z-Bit gesetzt (Zero Bit)	Z=1
BMI	Branch if minus, verzweige wenn kleiner null, N-Bit gesetzt	N=1
BNE	Branch if not equal, verzweige wenn Z-Bit auf 0	Z=0

Tabelle 2-19: Verzweigungsbefehle Kurzform

Mnemonik	Funktion	Operation	
BPL	Branch if plus, verzweige wenn größer oder gleich null	N=0	
BVC	Branch if overflow clear	V=0	
BVS	Branch if overflow set	V=1	
Arithmetische unsigned Verzweigungen			
		Relation	
BHI	Branch if higher	R > M	C + Z = 0
BHS	Branch if higher or same	R ≥ M	C = 0
BLO	Branch if lower	R < M	C = 1
BLS	Branch if lower or same	R ≤ M	C + Z = 1
Arithmetische signed Verzweigungen			
BGE	Branch if greater than or equal	R ≥ M	N ⊕ V = 0
BGT	Branch if greater than	R > M	Z + (N ⊕ V) = 0
BLE	Branch if less or equal	R ≤ M	Z + (N ⊕ V) = 1
BLT	Branch if less than	R < M	N ⊕ V = 1

Tabelle 2-20: Verzweigungsbefehle Langform

Mnemonik	Funktion	Operation
Unbedingte Verzweigungen		
LBRA	Branch always, immer verzweigen	1=1
LBRN	Branch never, nie verzweigen	1=0
Bedingte Verzweigungen		
LBCC	Branch if carry clear, verzweige wenn Carry-Bit auf 0	C=0

Tabelle 2-20: Verzweigungsbefehle Langform

Mnemonik	Funktion	Operation	
LBCS	Branch if carry set, verzweige wenn Carry-Bit auf 1	$C=1$	
LBEQ	Branch if equal, verzweige wenn Z-Bit gesetzt (Zero Bit)	$Z=1$	
LBMI	Branch if minus, verzweige wenn kleiner null, N-Bit gesetzt	$N=1$	
LBNE	Branch if not equal, verzweige wenn Z-Bit auf 0	$Z=0$	
LBPL	Branch if plus, verzweige wenn größer oder gleich null	$N=0$	
LBVC	Branch if overflow clear	$V=0$	
LBVS	Branch if overflow set	$V=1$	
Arithmetische unsigned Verzweigungen			
		Relation	
LBHI	Branch if higher	$R > M$	$C + Z = 0$
LBHS	Branch if higher or same	$R \geq M$	$C = 0$
LBLO	Branch if lower	$R < M$	$C = 1$
LBLS	Branch if lower or same	$R \leq M$	$C + Z = 1$
Arithmetische signed Verzweigungen			
LBGE	Branch if greater than or equal	$R \geq M$	$N \oplus V = 0$
LBGT	Branch if greater than	$R > M$	$Z + (N \oplus V) = 0$
LBLE	Branch if less or equal	$R \leq M$	$Z + (N \oplus V) = 1$
LBLT	Branch if less than	$R < M$	$N \oplus V = 1$

Die BRN (Branch-Never)-Anweisung macht gar nichts und funktioniert wie eine zwei Byte lange NOP-Anweisung. Sie wird hauptsächlich für Zeitverzögerungen benutzt.

Alle Verzweigungsbefehle kennen nur die relative Adressierung, im Gegensatz zu den JMP-Befehlen, die deutlich mehr Adressierungsarten unterstützen. Nun ein paar Beispiele:

```

CPX    var1 ; vergleiche Inhalt von X mit Inhalt an der Speicherzelle mit der
       ; Adresse var1
BEQ    S1   ; verzweige nach S1 wenn X = var1. Test ist der gleiche für signed
       ; und unsigned
BHI    S2   ; verzweige nach S2, wenn X > var1. X und var1 sind unsigned

CMPA   #$24 ; vergleiche Inhalt von A mit $24
BGE    S3   ; verzweige nach S3, wenn A >= $24. A wird als
       ; signed interpretiert
BHS    S4   ; verzweige nach S4, wenn A >= $24. A wird als
       ; unsigned interpretiert

CMPB   #0   ; vergleiche B mit 0
BLT    S5   ; verzweige nach S5, wenn B < 0. B wird als signed
       ; interpretiert
BLO    S6   ; verzweige nach S6, wenn B < 0. Befehl ist unsinnig, da B als
       ; unsigned interpretiert wird und nie < 0 werden kann. Die Ver-
       ; zweigung wird tatsächlich nie ausgeführt

```

In höheren Programmiersprachen wie C (na ja) lassen sich bei Verzweigungsbedingungen auch unterschiedliche Datentypen miteinander vergleichen, da der Compiler Code generiert, der vor dem Vergleich eine Typumwandlung vornimmt. Die schöne strukturierte Art des Programmierens z.B. mit *if ... then ... else* kann man in Assembler nicht mehr darstellen; ohne Sprünge (Go To's) ist hier nichts zu machen. Wir schauen uns im Folgenden an, wie Kontrollstrukturen in Assembler programmiert werden können.

Die if ... then Struktur

Die einfachste Auswahl-Kontrollstruktur ist die *if... then* Struktur. Wir wollen z.B. eine maximal zulässige Geschwindigkeit überwachen. Wird sie überschritten, soll eine Vollbremsung eingeleitet werden (neues Leistungsmerkmal für die Motorsteuerung, gerade eben vom Produktmanager erfunden).

In Erkans und Stefans C-Programm würde das so aussehen:

```
#define MAXSPEED 250
if (currentSpeed > MAXSPEED) {
    krasseBremsung();
}
```

In unserem distinguierten Assemblerprogramm sähe das so aus:

```
MAXSPEED      EQU 250
LDAA currentSpeed ; currentSpeed ist vorher mit ds.b definiert worden
```

```

        CMPA #MAXSPEED      ; vergleiche mit MAXSPEED
        BLS  keinStress     ; verzweige, wenn MAXSPEED NICHT überschritten
        JSR  krassBremsung ; Routine woanders definiert
keinStress>:
    ...

```

Erkan und Stefan würden irgendwann merken, dass sie ein fettes Problem haben, wenn sie die Motorsteuerung für den neuesten Bugatti mit 400 km/h Höchstgeschwindigkeit verwenden wollen.

Die if ... then ... else Struktur

Schauen wir uns einmal an, wie der C-Compiler diese Struktur in Maschinensprache umsetzt. Als Datentypen nehmen wir der Einfachheit halber unsigned char:

```

...
unsigned char register a, b, c;
...

if ( a < b ) {
    c = a;
}
else {
    c = b;
}
...

```

Und nun den dazu gehörenden Assemblercode, wie ihn unser Compiler erzeugt:

```

...
LDAB 2,SP      ; Variabe a liegt auf dem Stack, kommt nach B
CMPB 1,SP      ; a (B) wird mit b direkt auf dem Stack verglichen
BHS  WEITER1   ; wenn a >= b dann nach WEITER1
STAB 0,SP      ; ansonsten a in Variable c auf den Stack legen
BRA  WEITER2   ; überspringe die nächsten Befehle

WEITER1:
LDAB 1,SP      ; hole b vom Stack
STAB 0,SP      ; speichere b nach c auf dem Stack

WEITER2:
LDAB 0,SP      ; Rückgabewert immer in Register B
...

```

Wenn der Compiler richtig warmgelaufen ist (d.h. wir haben die Optimierung nicht ausgeschaltet wie im vorhergehenden Listing), generiert er allerdings einen etwas anderen Code:

```
LDAB  2,SP          ; kennen wir schon, ist aber auch c = a!!!
CMPB  1,SP          ; kennen wir schon
BCC   WEITER1:      ; kennen wir schon
SKIP2                 ; Opcode für CPS!!!!!, schmutziger Trick
WEITER1:
LDAB  1,SP          ; kennen wir schon
STAB  0,SP          ; kennen wir schon
```

Der Compiler lässt den Prozessor auf den CPS-Befehl laufen, der hier keinerlei Bedeutung hat. Allerdings liest dieser Befehl die folgenden zwei Byte als Operanden, also das LDAB 1, SP. Es wird damit „aufgegessen“ ohne dass sonst irgendetwas passiert. Dies ist eine trickreiche Art, zwei Byte weiterzuspringen.

Die if ... then Struktur mit komplexer Bedingung

Manchmal sind die Bedingungen eines *if* nicht einfache Ausdrücke, sondern setzen sich veknüpfen über logische AND oder OR Operationen aus mehreren einfachen Ausdrücken zusammen. Beispiel:

```
if (a > 15 && b > 25) {
    a = a + b;
}
```

In Assemblersprache könnte das so aussehen:

```
LDAA    a    ;
CMPA    #15 ;
BLS     doNothing;
LDAA    b    ;
CMPA    #25 ;
BLS     doNothing
ADDB    a    ;
STAB    a    ;
doNothing:
...
...
```

Die AND-Verknüpfung wird einfach durch die Hintereinanderschaltung der einzelnen Bedingungen realisiert. Ähnlich kann man bei einer OR-Verknüpfung vorgehen:

```
if (a > 15 || a < 5) {
```

```
a = a + b;
}
```

In Assemblersprache könnte das so aussehen:

```
LDAA a      ;
CMPA #15   ;
BHI doAdd;
CMPA #5    ;
BHS doNothing
doAdd:
ADDB b      ;
STAB a      ;
doNothing:
...
```

Die switch-case Struktur

Die zuvor besprochenen *if... then* Strukturen reichen aus, um alle anderen Entscheidungsstrukturen darzustellen. In der Praxis hat sich gezeigt, dass Auswahlstrukturen besser in der *switch-case* Struktur beherrscht werden können. Das folgende Beispiel zeigt eine solche Struktur zusammen mit dem vom Compiler generierten Assembler- und Maschinencode. Der Wert des Current Location Counters CLC ist ganz links dargestellt, gefolgt vom Maschinencode:

```
21:    switch (a) {
0003 e681        LDAB  1,SP
0005 c103        CMPB  #3
0007 221c        BHI   *+30 ;abs = 0025
0009 53          DECB
000a 87          CLRA
000b 160000       JSR   _CASE_CHECKED_BYTE
000e 03          DC.B  3
000f 15          DC.B  21
0010 03          DC.B  3
0011 09          DC.B  9
0012 0f          DC.B  15
22:
23:    case 1:   c = 1;
0013 c601        LDAB  #1
0015 6b80        STAB  0,SP
24:          break;
0017 200e        BRA   *+16 ;abs = 0027
25:
26:    case 2:   c = 4;
```

```

0019 c604      LDAB   #4
001b 6b80      STAB   0,SP
27:           break;
001d 2008      BRA    *+10 ;abs = 0027
28:
29:           case 3:  c = 8;
001f c608      LDAB   #8
0021 6b80      STAB   0,SP
30:           break;
0023 2002      BRA    *+4  ;abs = 0027
31:
32:           default: c = 0;
0025 6980      CLR    0,SP
33:
34:       }
35:
36:       return c;
0027 e680      LDAB   0,SP
0029 87        CLRA

```

Der Rückgabewert c wird auf dem Stack an Stelle 0 zwischengespeichert und zum Schluss im Register B an den Aufrufer zurück gegeben. Die Variable a ist auf dem Stack an der Stelle 1 abgelegt. Der Clou an dieser Struktur ist nicht direkt sichtbar: ein berechneter JMP springt auf den richtigen Fall. Dazu dienen die Subroutine _CASE_CHECKED_BYTE sowie die anschließenden Konstanten.

Ein Assemblerprogrammierer könnte hier ähnlich arbeiten, wenn er symbolische Sprungmarken verwenden würde.

Schreiben Sie das obige C-Programm in besser lesbaren Assembler um. Verwenden Sie nicht die Subroutine, aber benutzen Sie ebenfalls das Prinzip eines berechneten Sprungs zur Fallunterscheidung. Achten Sie darauf, dass die Struktur leicht erweiterbar bleibt und Fallunterscheidung an einer Stelle erfolgt, nicht verstreut über den gesamten Code.

Aufgabe 2-3

Die „Für Immer“-Schleife

Die „Für Immer“-Schleife ist die einfachste Iteration. Diese Schleife führt Aktionen wiederholt aus, bis das Programm abgebrochen wird bzw. der Rechner abgeschaltet wird. In C sieht sie folgendermaßen aus:

```

/* ... Initialisierungscode */
for (;;) {

```

```
    /* hier kommen die wiederholt auszuführenden Befehle hin */
}
```

Eine alternative Darstellung in C ist diese:

```
/* ... Initialisierungscode */
while (1) {
    /* hier kommen die wiederholt auszuführenden Befehle hin */
}
```

Diese Art von Schleife findet sich in eingebetteten Systemen häufig in der main-Routine oder, bei Einsatz eines Betriebssystems, in Tasks.

Die Realisierung in Assembler sieht z.B. so aus:

```
; Initialisierungscode
TopOfLoop:

; hier kommen die Befehle hin, maximal 125 Byte Maschinencode

BRA TopOfLoop; bei mehr als 125 Byte: benutze JMP statt BRA
```

Schleife mit nachgeschalteter Abbruchbedingung

In der Programmiersprache C wird eine Schleife mit nachgeschalteter Abbruchbedingung durch eine *do ... while* Struktur beschrieben. Zum Beispiel:

```
unsigned int n, m;

do {
    ++n;
    ++m;
} while ( m < 149 );
```

Wenn wir das von Hand in Assembler kodieren wollen, können wir z.B. zunächst die beiden Variablen n und m in die beiden Register x und y stecken. Dann sähe der Assemblercode so aus:

```
TopOfLoop:
    INX ; inkrementiere n und m
   INY
    CPY #149 ; Springe zurück, wenn Y < 149
    BLT TopOfLoop
```

In diesem Beispiel finden wir den häufigen Fall, dass die Abbruchbedingung durch den Wert eines Zählers bestimmt wird, der in der Schleife verändert wird. Die Konstruktion findet sich so oft in Pro-

grammen, dass der 68HCS12, wie auch die meisten anderen Rechner, dafür eigene Befehle zur Verfügung stellt. Bei diesen Befehlen besteht der mögliche Sprungbereich von -256 bis +255 Byte (9-Bit

Tabelle 2-21: Schleifenbefehle

Mnemonik	Funktion	Operation
DBEQ	Dekrementiere Zähler und verzweige bei 0 (Zähler A, B, D, X, Y oder SP)	(Zähler) - 1 \Rightarrow Zähler if (Zähler) == 0 then verzweige; else mache mit nächstem Befehl weiter
DBNE	Dekrementiere Zähler und verzweige bei !=0 (Zähler A, B, D, X, Y oder SP)	(Zähler) - 1 \Rightarrow Zähler if (Zähler) != 0 then verzweige; else mache mit nächstem Befehl weiter
IBEQ	Inkrementiere Zähler und verzweige bei 0 (Zähler A, B, D, X, Y oder SP)	(Zähler) + 1 \Rightarrow Zähler if (Zähler) == 0 then verzweige; else mache mit nächstem Befehl weiter
IBNE	Inkrementiere Zähler und verzweige bei !=0 (Zähler A, B, D, X, Y oder SP)	(Zähler) + 1 \Rightarrow Zähler if (Zähler) != 0 then verzweige; else mache mit nächstem Befehl weiter
TBEQ	Teste Zähler und verzweige bei 0 (Zähler A, B, D, X, Y oder SP)	if (Zähler) == 0 then verzweige; else mache mit nächstem Befehl weiter
TBNE	Teste Zähler und verzweige bei !=0 (Zähler A, B, D, X, Y oder SP)	if (Zähler) != 0 then verzweige; else mache mit nächstem Befehl weiter

Offset).

Diese Befehle fassen die Inkrement- bzw. Dekrementoperation eines Zählers, der sich in einem Register befindet, den Vergleichs- und Verzweigungsbefehl zusammen. Beispiel:

```

LDY #100
LDX #99
TopOfLoop:
    DECX
    DBNE Y, TopOfLoop; springe zurück, solange Y != 0
Continue:
    ...

```

Aufgabe 2-4

Welchen Wert hat X im obigen Beispiel, wenn der Rechner die Schleife verlassen hat?

Schleife mit vorgeschalteter Abbruchbedingung

In der Programmiersprache C wird eine Schleife mit vorgeschalteter Abbruchbedingung durch eine *while*-Struktur oder eine *for*-Struktur beschrieben. Zum Beispiel:

```
unsigned char a;
unsigned int *x, *y;
... /* initialisiere a, x, y */
while (a != 0) {
    *x++ = *y++;
    --a;
}
```

Aufgabe 2-5

Erläutern Sie, was das Programm genau macht.

In Assembler könnten wir dieses kleine Programm so realisieren:

```
TopOfLoop:
    TSTA             ; ist A != 0?
    BEQ  ExitLoop
    MOVW 2,Y+,2,X+ ; kopiere Wort von Adresse, auf die Y zeigt an die Stelle,
                     ; auf die X zeigt. Danach inkrementiere die beiden Zeiger
                     ; um jeweils 2 Byte
    DECA ; --A
    BRA  TopOfLoop ; und noch einmal
ExitLoop:
    ...
```

Einen kleinen Schönheitsfehler hat dieses Programm: wir haben zwei Verzweigungsbefehle.

Aufgabe 2-6

Schreiben Sie das obige Programm so um, dass die Schleife nur noch aus zwei anstatt fünf Maschinbefehlen besteht. Eventuell ist ein zusätzlicher Befehl außerhalb der Schleife erforderlich, der aber nur einmal ausgeführt werden muss.

2.9.6 Tabellen und Arrays

Wird in der Vorlesung besprochen.

2.9.7 Der Stack und Unterprogramme

Ein Stack oder Stapspeicher ist eine Datenstruktur, die sich gut zum Speichern temporärer Daten eignet. Auf die Elemente eines Stacks wird nach dem LIFO-Prinzip zugegriffen: last in, first out. Wie bei einem Stapel liegt ganz oben immer das zuletzt abgelegte Element.

Praktisch jeder Rechner unterstützt die Verwaltung eines Stacks mit Hilfe eines Stackpointers. Der Stackpointer zeigt typischerweise auf das letzte abgelegte Element, oder auf den nächsten freien Platz. Leistungsfähigere Rechnerarchitekturen stellen z.T. mehrere Stackpointer zur Verfügung, um auf diese Weise Multitasking besser zu unterstützen.

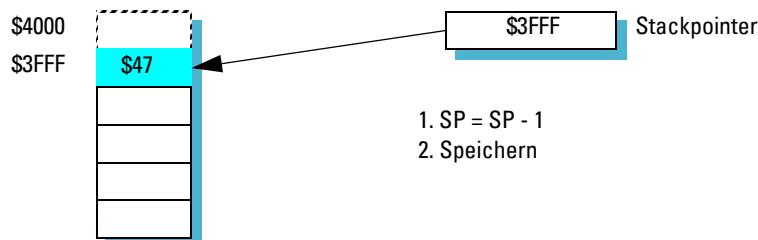
Stackoperationen

Die Stackverwaltung kann so aufgebaut sein, dass der Stack sich in Richtung niedrigerer Adressen erweitert, oder in Richtung höherer Adressen. Unser 68HCS12 hat sich für die erste Variante entschieden; der Stack wächst in Richtung niedrigerer Adressen oder nach unten. Aus diesem Grund wird der Stackpointer mit der höchsten RAM-Adresse + 1 initialisiert. Der Stack würde dann in Richtung der Anfangsadresse des RAMs wachsen, bis er irgendwann mit den dort abgelegten Daten kollidiert. Für unseren Laborrechner würden wir nach einem kurzen Blick auf die Memory Map also die Adresse \$4000 als Anfangswert für den Stackpointer festlegen, da die letzte RAM-Adresse \$3FFF ist:

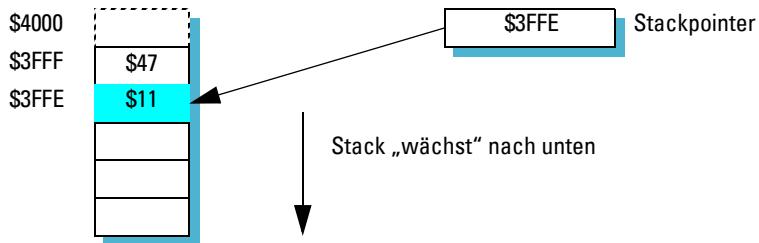
```
LDS $4000 ; initialisiere Stackpointer
```

Zeigt der Stackpointer auf diese Adresse, dann ist der Stack leer.

Das Speichern eines Datums auf dem Stack nennt man eine Push-Operation (*engl. push: schieben*). Dabei wird zuerst der Stackpointer dekrementiert, und dann der Wert (in unserem Beispiel \$47) gespeichert.

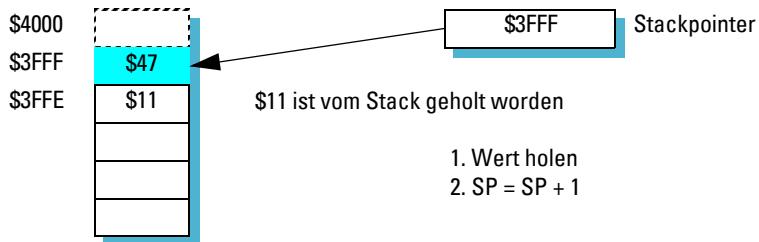


Die nächste Push-Operation wiederholt den Vorgang, hier mit dem Wert \$11:



Mit jedem Element, das auf den Stack gelegt wird, wächst der vom Stack belegte Speicherplatz. Der Stack kann also nicht größer werden, als Platz im RAM ist, in unserem Fall 12 KByte.

Das Entfernen von Elementen vom Stack nennt man eine „Pull“-Operation (*engl. pull*: ziehen). In unserer Darstellung wird immer das unterste Element entfernt, indem der Stackzeiger inkrementiert wird.



Der Wert \$11 an der Stelle \$3FFE wird zwar geholt, aber nicht gelöscht. Würden wir jetzt einen neuen Wert auf den Stack legen, würde die \$11 überschrieben. Würden wir auch die \$47 noch holen, wäre der Stack wieder leer, die Werte stünden aber trotzdem noch in den Speicherzellen.

Der 68HCS12 kann sowohl Byte-Werte als auch Wort-Werte auf den Stack legen. Wenn Wort-Werte mit 16 Bit auf den Stack gelegt werden, wird der Stackzeiger um 2 dekrementiert. Entsprechend wird beim Holen verfahren.

Mit Hilfe der Push- und Pull-Operationen kann man recht bequem den Wert von Registern zwischenspeichern, wenn diese kurzzeitig für eine andere Aufgabe benötigt werden. Nehmen wir z.B. einmal an, alle Register wären belegt und wir müssten Akkumulator D mit einem konstanten Wert multiplizieren. Die Multiplikationsoperation emul erfordert aber die Benutzung von Register Y; sie

funktioniert mit einem unmittelbaren Operanden nicht. Wir können dann Y auf dem Stack zwischenspeichern:

```
PSHY  
LDY #pi  
EMUL  
PULY
```

und schon ist das Problem gelöst.

Wir können Werte auf dem Stack ansehen, ohne sie vom Stack zu holen, indem wir indizierte Adressierung verwenden. Das wird benutzt, um bei Hochspracheprogrammen auf Parameter und lokale Variablen zuzugreifen:

```
LDAB 0,SP ; lade Akkumulator B mit dem obersten Byte des Stack  
           ; (in unserem Bild mit dem niedrigsten, also z.B. $11)  
LDAA 1,SP ; lade Akkumulator A mit dem 2. Byte, also z.B. $47
```

Diese Befehle wären identisch zu der Befehlsfolge

```
PULB  
PULA  
PUSHA  
PUSHB
```

Unterprogramme und GOTO

Ein Hochsprachenprogramm ist ohne Unterprogramme nicht denkbar. Unterprogramme nehmen Anweisungen auf, die immer wieder benutzt werden. Praktisch jede Rechnerhardware unterstützt diese Programmierstruktur durch entsprechende Befehle. Der Aufruf eines Unterprogramms entspricht einer Verzweigung, bei der am Ende aber wieder an den Verzweigungspunkt zurück gekehrt wird. Man muss sich also merken, von wo man verzweigt hat.

Die meisten Rechner merken sich den Verzweigungspunkt dadurch, dass sie die dem Verzweigungspunkt nachfolgende Adresse auf den Stack legen, bevor sie den Sprung ausführen. Der BSR bzw. JSR-Befehl unseres 68HCS12 stellt also eine Kombination aus einer Push-Operation und einem JMP-Befehl dar.

Manche Rechner sichern bei einem Unterprogrammaufruf automatisch noch zusätzlich einige oder alle Register. Bei unserem Rechner müssten wir das programmieren, wenn wir sicherstellen wollen, dass Registerwerte über den Sprung hinweg erhalten bleiben.

Tabelle 2-22: Sprung- und Unterprogrammbefehle

Mnemonik	Funktion	Operation
BSR	Branch to subroutine ($[-128 \dots +127]$)	$SP - 2 \Rightarrow SP$ $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)}$ Unterprogramm-Adresse $\Rightarrow PC$
CALL	Call to subroutine in expanded memory, Verwenden wir nicht	
JMP	Jump	Adresse $\Rightarrow PC$
JSR	Jump to subroutine (gesamter Adressraum)	$SP - 2 \Rightarrow SP$ $RTN_H:RTN_L \Rightarrow M_{(SP)}:M_{(SP+1)}$ Unterprogramm-Adresse $\Rightarrow PC$
RTC	Return from call in expanded memory, verwenden wir nicht	
RTS	Return from subroutine	$M_{(SP)}:M_{(SP+1)} \Rightarrow PC_H:PC_L$ $SP + 2 \Rightarrow SP$

Durch die verschiedenen Adressierungsmöglichkeiten des JMP-Befehls lassen sich gut berechnete GOTOS realisieren. Wie schon vorher gezeigt, kann man diese z.B. bei der Realisierung eines switch-case-Statements verwenden. In der Hochsprache sind GOTOS verpönt, weil sie zu fehleranfälligem Code führen. In Assembler kann man ohne GOTOS (d.h. Verzweigungsbefehle und JMP) nicht programmieren. Daraus folgt zwingend, dass Assemblerprogramme fehleranfälliger und weniger wartungsfreundlich sind als Hochsprachenprogramme.

Parameter- und Ergebnisübergabe

Eine Prozedur in einer Hochsprache sieht typischerweise so aus:

```
void testProzedur(int param1, char param2) {
    /* hier kommt der Code */
}
```

d.h. es können Parameter übergeben werden und es gibt keinen Rückgabewert. Eine Funktion unterscheidet sich von einer Prozedur dadurch, dass sie einen Rückgabewert besitzt:

```
int testFunktion(int param1, char param2) {  
    /* hier kommt der Code */  
}
```

In Assembler gibt es drei Möglichkeiten, Parameter und Rückgabewerte zwischen aufrufendem Programm und Unterprogramm zu transportieren:

- über Register
- über den Stack
- über Speicher

Alle drei Möglichkeiten werden auch verwendet, abhängig von der Rechnerarchitektur und den Festlegungen des Compiler-Herstellers.

Wenn nur wenige Parameter mit einem entsprechend speicherplatzsparenden Datentyp zu übergeben sind, ist die effizienteste Methode die Übergabe in Registern. Passen die Parameter nicht in die Register, wird in der Regel der Stack als Übergabespeicher verwendet. Die Übergabe in einem reservierten Speicherbereich ist fehleranfällig und wird nur auf sehr kleinen Rechnern mit minimalem Stack benutzt (z.B. verwendet der Keil-Compiler für die 8051-Architektur dieses Verfahren).

Das Ergebnis einer Funktion wird fast immer in Registern an den Aufrufer zurückgegeben. In der Regel muss man sich als Hochsprachenprogrammierer nicht darum kümmern, wie genau der Compilerhersteller die Parameterübergabe und Funktionswertrückgabe implementiert hat. Das wird erst dann interessant, wenn man eigene Assemblerroutinen von C aus verwenden möchte, oder der seltenere Fall, dass C-Routinen von Assembler aus verwendet werden sollen. Interessant wird es auch beim Debuggen von hardwarenaher Software, wenn man beobachten möchte, wie Parameter an die Hardware transportiert werden.

Der Metrowerks C-Compiler benutzt die sogenannte Pascal-Aufruf-Konvention für Funktionen mit einer feststehenden Anzahl von Parametern. Die Parameter werden von links nach rechts auf den Stack geschoben; der Aufrufer muss die Parameter wieder vom Stack entfernen. Ist der letzte Parameter von einem einfachen Typ (int, char, etc.), wird er nicht auf den Stack geschoben, sondern in einem Register übergeben.

[Pascal-Aufrufkonvention](#)

[C-Aufrufkonvention](#)

Bei Funktionen mit einer variablen Anzahl von Parametern wird die sogenannte C-Aufruf-Konvention benutzt. In diesem Fall schiebt der Aufrufer die Parameter von rechts nach links auf den Stack.

Tabelle 2-23: Parameterübergabe

Größe des letzten Parameters	Beispiel-Datentyp	Register
1 Byte	char	B
2 Byte	int	D
3 Byte	far data pointer	X(L) B(H)
4 Byte	long	D(L) X(H)

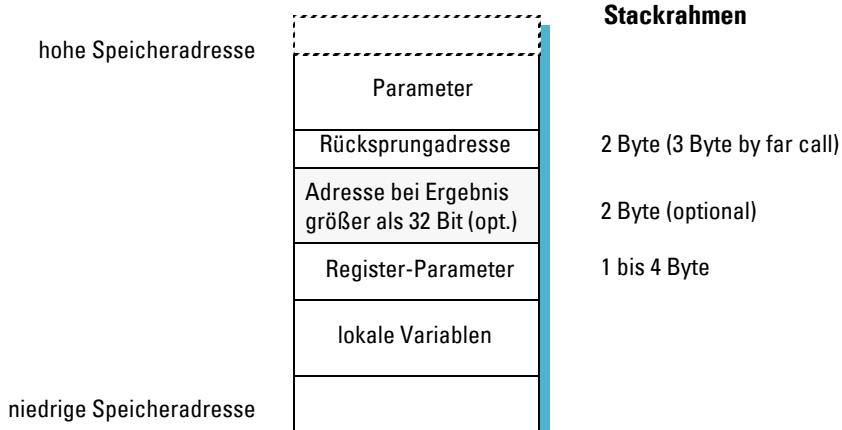
Rückgabewerte von Funktionen werden in Registern gehalten, außer der Rückgabewert ist länger als ein Langwort (32 Bit). Die verwendeten Register hängen vom Datentyp des Rückgabewertes ab. Bei Rückgabewerten größer als 32 Bit wird ein zusätzlicher Parameter auf dem Stack übergeben, der einen Zeiger auf die Stelle darstellt, an der der Rückgabewert erwartet wird.

Tabelle 2-24: Übergabe des Rückgabewertes

Größe des letzten Parameters	Beispiel-Datentyp	Register
1 Byte	char	B
2 Byte	int	D
3 Byte	far data pointer	X(L) B(H)
4 Byte	long	D(L) X(H)

Stackrahmen

Funktionen besitzen einen sogenannten Stackrahmen, in dem sie ihre lokalen Daten halten. Der Compiler benutzt den Stackpointer als Basisadresse, um auf die lokalen Daten zuzugreifen. Dieser Stackrahmen sieht wie folgt aus:



Die Register-Parameter werden erst nach dem Einsprung in die Unterroutine auf dem Stack gesichert; danach werden je nach Bedarf die lokalen Variablen auf dem Stack angelegt. Die Unterroutine ist dafür verantwortlich, bis auf die Parameter (nicht die Register-Parameter) alle anderen Werte wieder vom Stack zu entfernen, d.h. nach dem Rücksprung zeigt der Stackzeiger auf den zuletzt gepushten Parameter. Der Aufrufer ist dann dafür verantwortlich, den Stackzeiger wieder auf den Wert vor dem Aufruf zu bringen. Kurz gesagt muss also jeder selbst das aufräumen, was er auf den Stack gelegt hat, mit der Ausnahme der Adresse für Rückgabewerte, die größer als 32 Bit sind.

2.10 Interrupts, Traps und Resets

Wir haben im ersten Kapitel gelernt, dass der Rechner ein Programm so abarbeitet:

1. Befehl aus Speicherzelle holen, auf die der Programmzähler zeigt
2. Programmzähler auf nächsten Befehl stellen
3. Befehl dekodieren
4. Befehl ausführen

Der Programmzähler wird mit jedem Befehl erhöht und würde nach dem Erreichen der höchsten Adresse wieder bei Adresse \$0000 beginnen, wenn es keine Verzweigungen gäbe. Mit Verzweigungsstrukturen wie Schleifen und bedingten Verzweigungen kann der Programmierer bestimmen, welcher von mehreren möglichen Befehlen als nächster Befehl abgearbeitet werden sollte.

Die meisten Rechner haben über die Möglichkeit der bedingten und unbedingten Verzweigungen hinaus noch einen weiteren Mechanismus, den Programmzähler auf einen anderen Wert zu stellen. Das Verstellen des Programmzeigers stellt dabei für den normalen, linearen bzw. programmatisch verzweigten Ablauf eine Ausnahme dar (*engl. exception*). Man kann drei Arten von Ausnahmen unterscheiden:

1. Interrupts (Unterbrechungen)
2. Traps (Fallen)
3. Reset

Interrupts sind zum normalen Programmablauf asynchrone Ereignisse, die meistens von Peripheriegeräten und Zeitgebern verursacht werden.

Traps sind durch Software oder die interne Rechnerhardware angestoßene Ereignisse. Dazu zählt man z.B. den Versuch, einen unbekannten Befehl zu dekodieren, oder bei unserem Rechner die SWI (Software Interrupt) Anweisung. Bei manchen Rechnern wie z.B. den Freescale PowerPC führt auch der nicht ausgerichtete Zugriff auf Speicher zum Auslösen einer Trap.

Diese beiden Arten von Ausnahmen springen in eine spezielle Subroutine, aus der sie normalerweise nach kurzer Zeit wieder zurückkehren. Das unterscheidet sie vom **Reset**, bei dem eine Rückkehr nicht vorgesehen und auch nicht möglich ist. Ein Reset wird durch Einschalten der Spannungsversorgung, durch einen speziellen Pin am Rechner oder durch den Watchdog ausgelöst.

Im allgemeinen Sprachgebrauch unterscheidet man häufig nicht zwischen Ausnahmen und Interrupts, sondern verwendet für alle Ausnahmen den Begriff „Interrupt“.

Fast alle Rechner, die Ausnahmen unterstützen, definieren eine Interrupt-Vektortabelle. Dies ist eine bestimmte Region im Adressraum, in der für jede mögliche Interruptursache die Adresse der zugehörigen Interrupt-Behandlungsroutine abgelegt ist.

Tabelle 2-25: Interrupt-Vektortabelle (Auszug)

Vektor-adresse	Interruptquelle	CCR-Maske	Enable	HPRIO-Wert
\$FFFE,\$FFFF	Reset	-	-	-
\$FFFA,\$FFFB	COP failure reset	-	COP rate select	-
\$FFF8,\$FFF9	Unimplemented instruction trap	-		-
\$FFF6,\$FFF7	SWI	-		-
\$FFF4,\$FFF5	XIRQ	X-Bit		-
\$FFF2,\$FFF3	IRQ	I-Bit	IRQCR(IRQEN)	\$F2
\$FFF0,\$FFF1	Real Time Interrupt	I-Bit	CRGINT(RTIE)	\$F0
\$FFEE,\$FFEF	Enhanced Capture Timer channel 0	I-Bit	TIE(C0I)	\$EE
\$FFEC,\$FFED	Enhanced Capture Timer channel 1	I-Bit	TIE(C1I)	\$EC
\$FFE0,\$FFE1	Enhanced Capture Timer channel 7	I-Bit	TIE(C7I)	\$E0
\$FFDE,\$FFDF	Enhanced Capture Timer overflow	I-Bit	TSRC2(TOF)	\$DE
\$FFD2,\$FFD3	ATD0	I-Bit	ATD0CTL2(ASCIE)	\$D2
\$FFD0,\$FFD1	ATD1	I-Bit	ATD1CTL2(ASCIE)	\$D0
\$FFCE,\$FFCF	Port J	I-Bit	PTJIF(PTJIE)	\$CE
\$FFCC,\$FFCD	Port H	I-Bit	PTHIF(PTHIE)	\$CC
\$FF8E,\$FF8F	Port P	I-Bit	PTPIF(PTPIE)	\$8E

Die Interrupts sind priorisiert. Die Interrupts mit der höheren Priorität stehen weiter oben in der Tabelle. Der Reset ist also der am höchsten priorisierte Interrupt. Die Beziehung zwischen Interruptvektor und zugehörigem Auslösemechanismus ist durch die Rechnerhardware vorgegeben. Bei manchen Rechnern besitzt der Zentralprozessor nur wenige Interrupteingänge, die durch vorgesetzte

Interrupt-Kontroller aufgefächert werden. Der Prozessor muss dann mit dem Interrupt-Kontroller kommunizieren, um die Interruptursache zu erfahren und um den zugehörigen Interruptvektor bestimmen zu können.

Die Priorität der nicht „maskierbaren“ Interrupts (solche mit einem „-“ in der Spalte „CCR-Maske“) kann nicht verändert werden. Aus der Menge der maskierbaren Interrupts kann man einen auf die höchste Prioritätsstufe verschieben, indem man in das HPPIO-Register den entsprechenden Wert schreibt.

Maskierbare Interrupts können kollektiv durch Setzen des I-Bits im CCR blockiert (maskiert) werden (I-Bit=1). Der XIRQ-Interrupt kann durch das Setzen des X-Bits im CCR blockiert werden (X-Bit=1). Für jeden maskierbaren Interrupt kann an der Quelle über ein Bit bestimmt werden, ob der Interrupt aktiviert ist (Bit = 1) oder deaktiviert ist (Bit = 0).

2.10.1 Vorbereitung und Ablauf eines Interrupts

Damit ein Interrupt verarbeitet werden kann, müssen einige Voraussetzungen erfüllt sein:

- Die Interruptquelle muss einen Interrupt anzeigen. Für die eingebauten Peripheriegeräte geschieht das durch das Setzen eines Interrupt-Flags in einem zugehörigen Statusregister.
- Die Interrupts müssen an der Quelle freigeschaltet sein. Für die eingebauten Peripheriegeräte geschieht das durch Setzen eines Enable-Bits in einem zugehörigen Steuerregister (Spalte „Enable“ in Tabelle 2-25).
- Der Interrupt darf nicht maskiert sein. In den meisten Fällen bedeutet dies, dass das I-Bit im CCR zurückgesetzt sein muss. Nach dem Einschalten des Rechners steht das I-Bit auf 1, d.h. alle maskierbaren Interrupts sind gesperrt. Das Monitorprogramm setzt das I-Bit aber auf 0, da es Interrupts für die serielle Schnittstelle benötigt.

Aufgabe 2-7

Finden Sie heraus, welche Register und Bits innerhalb dieser Register sie abfragen bzw. setzen müssen, um ein Drücken des Tasters SW2 auf dem Laborboard feststellen zu können. Benutzen Sie dazu die Schaltpläne und sonstige notwendige Dokumentation.

Was passiert nun, wenn alle oben erwähnten Bedingungen erfüllt sind und ein Interrupt durchgeleitet worden ist? Der Prozessor geht wie folgt vor:

- Der Programmzähler PC, der auf die nächste auszuführende Anweisung zeigt, die Register Y,X,A,B und das CCR werden in genau dieser Reihefolge auf den Stack gelegt
- Das I-Bit im CCR wird gesetzt; somit sind alle weiteren maskierbaren Interrupts blockiert. Die Interruptroutine kann also nicht von einem weiteren maskierbaren Interrupt unterbro-

chen werden. Im Falle eines XIRQ-Interrupts wird auch noch das X-Bit im CCR gesetzt.

- Der Programmzähler wird mit den zugehörigen Interruptvektor geladen, d.h. der Rechner springt zu der durch den Interruptvektor definierten Adresse und arbeitet den dort liegenden Befehl ab.
- Der Rechner arbeitet die weiteren Befehle der Interrupt-Serviceroutine ab.

Die letzte Anweisung in einer Interrupt-Serviceroutine ist immer die RTI (Return from Interrupt)-Anweisung. Diese Anweisung restauriert alle vor Eintritt in den Interrupt auf den Stack gelegten Register. Da das I-Bit zu diesem Zeitpunkt zurückgesetzt gewesen sein muss, werden durch die RTI-Anweisung die maskierbaren Interrupts wieder freigegeben.

Häufig ist es nicht akzeptabel, dass ein niedrig priorisierte Interrupt die Bearbeitung eines wichtigen Interrupts verhindert. Aus diesem Grund ist es üblich, die Maskierung durch das I-Bit so schnell wie möglich wieder aufzuheben. Dies geschieht entweder durch sehr kurze Interruptroutinen, oder indem man innerhalb der Interruptroutine den CLI-Befehl (Clear Interrupt) ausführt.

Bevor man den CLI-Befehl aufruft, muss sichergestellt sein, dass das Interruptflag zurückgesetzt ist, sonst würde sofort wieder der gleiche Interrupt auftreten.

Es gibt eine spezielle Instruktion WAI, die die oben erwähnten Register schon auf den Stack schiebt, den Takt des Rechners abschaltet und dann nichts anderes macht, als auf einen Interrupt zu warten. Dadurch kann der Rechner bei Auftreten des Interrupts schneller reagieren. Der Stromverbrauch wird deutlich reduziert, da große Teile des Rechners nicht mehr getaktet werden.

Ein in dieser Hinsicht noch weitergehender Befehl ist der STOP-Befehl. Die stop-Anweisung muss man explizit zulassen, indem das S-Bit im CCR zurückgesetzt wird (S-Bit=0). Im Prinzip funktioniert die STOP-Anweisung wie die zuvor erwähnte WAI-Anweisung, nur werden jetzt alle Takte abgeschaltet, also auch die der Zeitgeber. Damit sinkt der Leistungsverbrauch des Rechners fast auf 0 mW. Aus diesem Tiefschlafzustand kann der Rechner allerdings nur durch ein Signal am RESET-Eingang, am XIRQ-Eingang oder am IRQ-Eingang geweckt werden.

2.10.2 Interrupt-Serviceroutine in Assembler

Wir gehen nun zurück auf unser erstes kleines Programm, das Blinklicht. Wir hatten es damals mit einer Schleife programmiert, und die Blinkfrequenz hing davon ab, wieviel der Rechner sonst noch zu tun hatte. Das können wir mit einer Interruptroutine viel besser bewerkstelligen. Wir nehmen einen der vielen Zähler des Rechners, die vom Prozessortakt gesteuert werden, laden diesen mit einem angenehmen Wert und lassen uns immer aufwecken, wenn der Zähler abgelaufen ist. In der Zwischenzeit kann der Rechner entweder nichts oder andere Ding abarbeiten, ohne dass wir befürchten müssen, mit unserer Blinkerei aus dem Takt zu geraten.

Damit das System zuverlässig arbeitet, müssen wir folgende Voraussetzungen schaffen:

- Der Stackzeiger muss initialisiert sein
- Die Interruptvektoren müssen auf die richtigen Adressen verweisen
- Die Peripheriegeräte, von denen wir Interrupts erwarten, müssen richtig initialisiert werden
- Benötigt die Interruptroutine Speicherplatz, muss dieser reserviert sein
- Die von den Peripheriegeräten erwarteten Interrupts müssen bei diesen freigegeben werden
- Die Maskierung über das I-Bit muss durch eine CLI-Anweisung aufgehoben werden

Kommen wir jetzt zu unserem Beispiel (in der Materialsammlung Beispiele\Interrupt-ASM):

```
; Exportierte Symbole
XDEF main, rtiISR
```

Wir müssen den Einsprungpunkt und die Interruptroutine nach außen hin bekannt machen, so dass der Linker sie finden kann. Beide Marken sind Adressen der zugehörigen Interruptvektoren.

Nun lesen wir eine Datei ein, die uns die Entwicklungsumgebung zur Verfügung stellt. Sie enthält eine ganze Reihe von EQU-Anweisungen, die uns den Umgang mit den Registern des Prozessors erleichtern.

```
INCLUDE 'mc9s12dp256.inc'
```

Wir schieben das Programm ins RAM, da wir genügend Platz haben und den Flash-Speicher schonen wollen. Hier definieren wir den Einsprungpunkt.

```
PRSTART EQU $2000
```

Im weiteren Verlauf reservieren wir Speicherplatz für Variablen. Den Zähler verwenden wir allerdings noch nicht.

```
; -----
; hier beginnt der Bereich für Variable und Konstanten
ORG RAMStart ; im Include definiert

zaehler: DS 2 ; Platz fuer einen Zaehler
```

Nun beginnt das Hauptprogramm. Wir plazieren es absolut an der weiter oben definierten Adresse. Man kann die Plazierung auch dem Linker überlassen, aber dann wird das Debugging etwas zäher.

Man müsste zunächst nachschauen, wohin der Linker den Code gesteckt hat. Diese Information stellt der Linker in einer Datei `Monitor.map` bzw. `Simulator.map` zur Verfügung.

Im Hauptprogramm initialisieren wir den Stackzeiger und die Ports, die wir benutzen. Die LED ist am Port B angeschlossen. Port P und Port J müssen wir einstellen, damit unsere Siebensegment-Anzeige nicht mitblinkt. Die LED-Zeile und die Siebensegmentanzeige teilen sich nämlich ein paar Leitungen (siehe Schaltplan).

```
ORG PRSTART
main:           ; Symbol ist auch in Monitor_linker.prm als
                ; Reset-Vektor definiert
    LDS #RAMEnd      ; initialisiere Stack, Symbol ist in Include definiert
    MOVW #0, zaehler

    LDAA #$ff
    STAA DDRB        ; Data Direction Register B auf Ausgabe stellen
    STAA DDRP        ; Data Direction Register P auf Ausgabe stellen
    MOVB #$0f, PTP    ; alle LEDs ausschalten

    BSET DDRJ, #2
    BCLR PTJ, #2

    LDAA #1
    STAA PORTB
```

Jetzt setzen wir den Teilfaktor für den RTI-Zähler. Wir nutzen den maximalen Wert, da wir einen relativ schnellen 4 MHz-Oszillator haben. Anschließend geben wir den RTI frei und setzen die Maskierung aller Interrupts aus.

```
MOVB #$7f, RTICTL ; setze RTI-Rate
BSET CRGINT, #$80 ; gib RTI frei
CLI              ; Interrupt-Maskierung abschalten
```

Wir laufen in eine Endlosschleife, die wir stromsparend programmiert haben. Das kann dazu führen, dass man beim Debugging der Schleife Probleme bekommt. In diesem Fall kann man den `WAI`-Befehl problemlos durch einen `NOP`-Befehl (No Operation) ersetzen.

```
mainLoop:
    WAI                  ; schlafe, mein Prinzchen, schlaf ein...
    BRA mainLoop
```

Jetzt fehlt noch das Kernstück der Übung: die Interrupt-Serviceroutine. Sie sieht aus wie eine ganz normale Funktion, endet aber mit einem `RTI`-Befehl anstelle eines `RTS`- oder `RTC`-Befehls:

```

rtiISR:
    BSET CRGFLG, #$80 ; setze das RTI Interrupt Flag zurueck
    NOP             ; kleiner Bug bei manchen HCS12 ?
    CLI             ; gib fuer andere Interrupts wieder frei
    LDAA PORTB     ; lade Port B
    EORA #1         ; negiere Bit 0 von Port B
    STAA PORTB     ; schreibe zurueck

    LDD  zahler      ; Zaehler laden
    ADDD #1          ; nur so zum Spass ein Zaehler
    STD  zahler

    RTI              ; das war es auch schon

```

Woher weiß der Rechner, dass `rtiISR` eine Interrupt-Serviceroutine für den RTI sein soll? Aus dem Assembler-Quelltext geht das nicht hervor. Diese Angaben müssen wir in der CodeWarrior-Entwicklungsumgebung in einer Befehlsdatei für den Linker machen. Die Datei heißt `Monitor_linker.prm` bzw. `Simulator_linker.prm` und sieht so aus:

```

NAMES END

SEGMENTS
    RAM = READ_WRITE 0x3000 TO 0x3F00;
    STACK = READ_WRITE 0x3F01 TO 0x3FFF;
    /* unbanked FLASH ROM */
    ROM_4000 = READ_ONLY 0x4000 TO 0x7FFF;
    ROM_C000 = READ_ONLY 0xC000 TO 0xFFFF;
END

PLACEMENT
    DEFAULT_ROM INTO ROM_4000;
    DEFAULT_RAM INTO RAM;
    SSTACK INTO STACK;
END

STACKTOP 0x3FFF

VECTOR 0 main /* reset vector: this is the default entry point for an
               assembly application. */
INIT main      /* for assembly applications: that this is as well the
               initialisation entry point */
VECTOR 7 rtiISR /* fuer unsere Echtzeitintervall-Routine */
// alternativ: VECTOR ADDRESS 0xFFFF rtiISR

```

Die entscheidende Anweisung steht ganz unten: hier verbinden wir das Symbol `rtiISR` mit dem Interruptvektor für den Zähler. Das Symbol `main` wird mit dem Reset-Vektor verbunden. Die Vektoren sind durchnumeriert und man kann die Nummer verwenden oder die Adresse angeben.

Erklären Sie anhand der Linker-MAP-Datei, was der Linker genau gemacht hat.

Aufgabe 2-8

Verändern Sie das obige Beispielprogramm so, dass ein Lauflicht über alle acht LEDs entsteht.

Aufgabe 2-9

Schreiben Sie unter Zuhilfenahme der Ihnen zur Verfügung stehenden Dokumentation ein kleines Programm, das bei Betätigung des Tasters SW2 die LED PB0 aufleuchten lässt. Erweitern Sie dann dieses Programm so, dass nach Loslassen des Tasters die LED PB0 noch ca. zwei Sekunden nachleuchtet. Verwenden Sie keine Abfrage- oder Verzögerungsschleifen.

Aufgabe 2-10

2.10.3 Interrupt-Serviceroutine in C

Interrupt-Serviceroutinen lassen sich nicht nur in Assembler sondern auch in C programmieren. Wir nehmen unser Blinklichtprogramm und schauen uns an, wie es in der Programmiersprache C aussieht (in der Materialsammlung unter Beispiele\Interrupt-C):

```
void main(void) {  
  
    EnableInterrupts; // Das brauchen wir fuer den Debugger und den RTI  
  
    // Deaktiviere die 7-Segment Anzeige  
    DDRP = DDRP | 0xf; // Data Direction Register Port P  
    PTP = PTP | 0x0f; // Schalte alle vier Segmente aus  
  
    // Aktiviere die LEDs  
    DDRJ_DDRJ1 = 1; // Data Direction Register Port J  
    PTJ_PTJ1 = 0; // Schalte LED-Zeile ein  
  
    // Schalte Port B als Ausgang  
    DDRB = 0xFF; // Data Direction Register Port B  
  
    // Schalte LED PB0 ein und aus  
    PORTB = 0x01;
```

Bis hierhin unterscheidet sich das Programm überhaupt nicht von unserem ersten einfachen Blinklichtprogramm. Jetzt jedoch müssen wir den Teiler für den Echtzeitzähler einstellen und den Echtzeit-Interrupt aktivieren:

```
// Stelle Teiler für RTI ein
RTICTL = 0x7f;
CRGINT = CRGINT | 0x80; /* schalte RTI frei */
```

In dieser Schleife machen wir gar nichts mehr:

```
for(;;) {
    /* Daemchen drehen ... */
}
```

Was übrig bleibt, ist die Interrupt-Serviceroutine. Es gibt in der Programmiersprache C kein standardisiertes Schlüsselwort, um eine Routine als Interruptroutine zu kennzeichnen. Der Compiler muss dies jedoch wissen, da er sonst statt eines RTI-Befehls am Ende einen RTS- oder RTC-Befehl generieren würde.

Fast alle Compiler für eingebettete Systeme kennzeichnen Unterbrechungsroutinen durch das Schlüsselwort „*interrupt*“ oder eine compilerspezifische pragma-Anweisung (im CodeWarrior-Compiler heißt sie z.B. #pragma TRAP_PROC und wird direkt vor die Routine gesetzt). Wir benutzen lieber „*interrupt*“ und schreiben:

```
interrupt void rtiISR (void) {

    CRGFLG = CRGFLG | 0x80;      /* setze Interrupt-Flag zurueck */
    PORTB = ~PORTB & 0x01;       /* das war's auch schon */
}
```

Ja, so einfach ist das. Allerdings muss man wie im Assembler auch hier dem Linker mitteilen, dass rtiISR eine Interrupt-Serviceroutine für einen bestimmten Interrupt ist. Das geschieht analog zur Vorgehensweise im Assembler. Alternativ kann man diese Information schon bei der Definition der Routine angeben:

```
interrupt 7 void rtiISR (void) {

    CRGFLG = CRGFLG | 0x80;      /* setze Interrupt-Flag zurueck */
    PORTB = ~PORTB & 0x01;       /* das war's auch schon */
}
```

Das hat den Vorteil, das man die Information nicht über zwei Dateien verstreut, was bei vielen Interrupts zu Wartungsproblemen führen kann.

Schreiben Sie unter Zuhilfenahme der Ihnen zur Verfügung stehenden Dokumentation in der Programmiersprache C ein kleines Programm, das bei Betätigung des Tasters SW2 die LED PB0 aufleuchtet lässt. Erweitern Sie dann dieses Programm so, dass nach Loslassen des Tasters die LED PB0 noch ca. zwei Sekunden nachleuchtet. Verwenden Sie keine Abfrage- oder Verzögerungsschleifen.

Aufgabe 2-11

Programmieren Sie unter Zuhilfenahme der Ihnen zur Verfügung stehenden Dokumentation in der Programmiersprache C ein kleines Programm, das bei Drücken der Taster SW2 bis SW5 vier verschiedene hohe Töne (jeweils einen ganzen Ton Unterschied) auf dem kleinen Lautsprecher des Boards erzeugt.

Aufgabe 2-12

2.10.4 Verhalten nach einem Reset

Ein Reset muss den Prozessor in einen definierten Anfangszustand bringen. Jeder Prozessor hat deshalb eine Reset-Funktion. Der in unserem Labor verwendete Prozessor kennt drei Möglichkeiten, in den Reset-Zustand zu gelangen:

- wenn der RESET-Anschluss auf „0“ geht
- wenn der Watchdog (der hier COP heißt) anschlägt
- wenn der Clock Monitor ein Problem feststellt

In einem Reset geschieht auf unserem Prozessor folgendes:

- Die Betriebsart und die Memory Map werden zurückgesetzt
- Die PLL für den Prozessortakt, der COP und der RTI werden abgeschaltet
- Das HPRIOR-Register wird mit \$F2 initialisiert, so dass der mit dem externen IRQ-Eingang assoziierte Interrupt die höchste Priorität hat
- Analog-Digitalwandler, Zeitgeber und serielle Schnittstellen werden ausgeschaltet
- Alle Ports außer der für das Speicherinterface werden als Eingang geschaltet. Die Pullup-Widerstände sind z.T. gesetzt, z.T. auch nicht.
- Der Programmzähler wird mit dem Reset-Vektor geladen. Die Bits X, I und S im CCR werden gesetzt, alle anderen Register sind unbestimmt.

Auf unserem Board ist im geschützten Bereich des Flash-Speichers eine kleine Monitor-Software installiert, die nach einem Reset die Kontrolle übernimmt. Diese Software kann sich mit dem Debugger der CodeWarrior-Entwicklungsumgebung unterhalten, so dass wir unsere Software auf das Board laden und mit dem Debugger ausführen können.

Ein- und Ausgabeprogrammierung

3.1 Überblick

Jeder Rechner kommuniziert mit seiner Umwelt über das Schreiben und Lesen spezieller Register, deren Ein- bzw. Ausgänge mit der Außenwelt elektrisch oder optisch verbunden sind. Hinter den Registern können sich komplexe Schaltungen wie z.B. Analog-Digitalwandler, Zeitgeber, oder riesige applikationsspezifische Schaltkreise verbergen, oder auch nur ein einfaches Flip Flop, dessen Ausgang nach außen geführt wird.

Wie kann die Zentraleinheit in die Register schreiben bzw. von ihnen lesen? Hier gibt es zwei unterschiedliche Ansätze:

- Register belegen Adressen im normalen Speicher-Adressraum und können mit den für Variablen üblichen Maschinenbefehlen erreicht werden (Memory mapped I/O).
- Register liegen in einem speziell dafür vorgesehenen Adressraum und können nur mit speziellen Assemblerbefehlen erreicht werden (isolierter I/O-Bereich)

Jeder Rechner, der einen speziellen I/O-Bereich mit dafür vorgesehenen Befehlen besitzt, kann selbstverständlich immer auch Peripherieregister in seinem normalen Adressbereich benutzen, wenn der Hardwareentwickler sie dort untergebracht hat (Hybrid-Lösung).

Unser Freescale-Rechner unterstützt nur memory mapped I/O, hat also keinen speziellen I/O-Bereich. Praktisch alle Peripherieregister sind hier im untersten 1 kByte-Adressblock untergebracht, beginnend mit Adresse \$0.

Intel-X86-Rechner haben einen speziellen I/O-Bereich, der in handelsüblichen PCs auch genutzt wird. Da leistungsfähige Rechner eine sehr schnelle Anbindung an den Hauptspeicher benötigen, Pe-

Peripheriegeräte aber oft langsam sein dürfen, haben diese Rechner oft mehrere physikalisch getrennte Adress- und Datenbusse, die aber den gleichen Adressraum belegen. Ein moderner Standard-PC besitzt eine Reihe interner Bussysteme, z.B. PCIe für schnelle Peripherie und Graphik, den Front Side Bus (FSB), der als Verbindung zwischen dem Prozessor und den restlichen Bussystemen ausgeführt ist, einen Speicherbus, einen Bus für SATA zum Anschluss von Festplatten, eine PCI-Bus für langsamere Peripherie, einen oder mehrere USB-Busse. Dazu kommt noch der eigene I/O-Adressbereich.

Steuer-, Status- und Datenregister

Peripherieregister lassen sich immer in drei Klassen aufteilen:

- Steuerregister
- Statusregister
- Datenregister

Steuerregister

Steuerregister werden benutzt, um ein Peripheriegerät zu konfigurieren oder zu steuern. So kann z.B. die Baudrate einer seriellen Schnittstelle eingestellt werden, oder der Analog-Digitalwandler kann angestoßen werden, mit der Wandelung zu beginnen.

Steuerregister müssen immer schreibbar sein, sind aber nicht immer lesbar. Wenn sie lesbar sind, geben sie meist den vorher hineingeschriebenen Wert zurück. Auf unserem Laborsystem kann man alle Steuerregister auch lesen.

Statusregister

Statusregister dienen dazu, den Zustand eines Peripheriegerätes zu beobachten. Zum Beispiel könnte man nachschauen, ob ein Analog-Digitalwandler mit der Umwandlung fertig ist, oder gerade noch damit beschäftigt ist. Bei einer seriellen Schnittstelle könnte man abfragen wollen, ob ein vollständiges Zeichen angekommen ist.

Statusregister sind nicht schreibbar, müssen aber immer lesbar sein. Der Wert der Statusregister wird von der Hardware der Peripherie bestimmt.

Datenregister

Datenregister dienen dazu, Daten zwischen dem Peripheriegerät und dem Zentralprozessor auszutauschen. Wenn wir z.B. die LEDs auf unserem Laborsystem ansteuern wollen, dann speichern wir ein entsprechendes Datum in das Register, an dem die LEDs hängen. Zuvor mussten wir allerdings andere Register so einstellen, dass die gewünschte Peripheriefunktion gewährleistet wird, in unserem Fall durch Schreiben eines Steuerregisters namens „Port Direction Register“.

Für die Register von Peripheriegeräten legt man in der Regel symbolische Namen fest. Häufig liefern die Hersteller des Prozessors oder die Hersteller der Entwicklungsumgebung schon eine Datei mit symbolischen Namen für alle Ein- und Ausgaberegister mit. Diese Datei kann man in seine eigene Software einbinden.

Wenn die Ein-Ausgabegeräte nicht fest in den Mikroprozessor eingebaut sind, berechnet man die Adresse der Register aus der Basisadresse des Gerätes plus einem Offset für das jeweilige Register. So kann man ohne großen Aufwand die Software an unterschiedliche Hardware anpassen.

3.2 Polling und Interruptbetrieb

Ein- und Ausgabegeräte können in zwei Betriebsarten betrieben werden:

- Polling-Betrieb
- Interrupt-Betrieb

Beim Polling fragt der Rechner zyklisch das Ein-Ausgabegerät ab, z.B. ob ein Datum angekommen ist, oder ob der Übertragungspuffer inzwischen leer ist, oder ob der Analog-Digitalwandler die Wandlung beendet hat. Programmiertypisch ist dies ein einfaches Verfahren, das jedoch den Nachteil hat, dass der Rechner entweder sehr viel Rechenzeit mit den Abfragen verbraucht, oder aber die Reaktion des Rechners auf ein Ereignis im Peripheriegerät mit großer Zeitverzögerung (Latenzzeit) erfolgt. Lässt sich der Rechner zuviel Zeit zwischen den Abfragen, kann es zu Fehlern kommen, z.B. indem ein noch nicht abgeholtes Datum schon von einem neuen überschrieben wird (Pufferüberlauf).

Ein Interrupt (Unterbrechung) ist im Wesentlichen ein von der Hardware des Rechners ausgelöster Aufruf einer Unterroutine. Diese Unterroutine nennt man „Interrupt Service Routine“. Statt nun laufend das Peripheriegerät abzufragen, ob irgend etwas passiert ist, meldet sich die Hardware des Peripheriegerätes selbst. Der Rechner besitzt eine Tabelle, in der niedergelegt ist, welche Unterroutine zu welchem Interrupt gehört. Der Programmierer muss dafür sorgen, dass diese Tabelle und die Unterroutinen richtig eingerichtet sind, bevor die Interrupts freigegeben werden.

Die Programmierung von Ein- und Ausgaberoutinen mit Hilfe von Interrupts entlastet den Rechner und auch den Programmierer beträchtlich. Der Programmierer muss sich nicht mehr darum kümmern, immer wieder rechtzeitig das Peripheriegerät abzufragen. Der Rechner kann sich mit sinnvoller Arbeit beschäftigen, statt viele tausend mal vergeblich den Status des Peripheriegerätes abzufragen. In mobilen Geräten schaltet man den Rechner auch gerne in einen Energiesparmodus, und weckt ihn nur durch die Interrupts auf.

3.3 Parallele Schnittstellen

Praktisch alle Mikrocontroller beinhalten allgemein verwendbare Peripherie-Anschlüsse für Ein- und Ausgabe. Über Steuerregister kann programmiert werden, ob ein Anschluss als Ein- oder Ausgang betrieben werden soll. Meistens sind jeweils acht solcher Anschlüsse zu einer Gruppe zusammengefasst. Diese Gruppe nennt man *Port*.

Abbildung 3-1 zeigt die bei unserem Laborrechner vorhandenen Ports. Der Rechner besitzt acht 8-Bit-Ports (A, B, E, H, M, P, S und T) und zwei kleinere Ports (J und K). Diese Ports können für allgemeine Ein-/Ausgabezwecke verwendet werden, wenn sie nicht für interne Peripheriemodule (Ports H, M, P, S, T und J) bzw. für den Anschluss eines externen Speichers (Port A, B, E und K) verwendet werden.

Die meisten Peripheriemodule haben zusätzliche Fähigkeiten wie z.B. A/D-Wandler, Pulsweitenmodulator und CAN-Schnittstelle. Wir betrachten hier die Eigenschaften der Ports H, M, P, S und T für allgemeine Ein- und Ausgabezwecke.

Die Struktur eines einzigen Pins (von acht) eines solchen Ports ist in Abbildung 3-2 gezeigt. Der Port besitzt Steuerungsregister, RDRX $_n$, PPSX und PERX und Datenregister PTX und PTIX. X bezeichnet hier den Port, also H, M, P, S, T oder J. PTIX ist nur lesbar und liest immer den Pegel am Pin selbst.

Jedes Bit in diesem Register entspricht einem Anschluss PX $_n$ in der Weise, dass Bit n im Register zuständig ist für Anschluss PX $_n$ des Ports.

Wenn DDRX $_n$ auf 1 steht, ist der Pin als Ausgang programmiert. Der Wert, der ins Datenregister PTX $_n$ geschrieben wird, bestimmt dann den Ausgangsspegel. Wird RDRX $_n$ auf 1 gesetzt, wird der Treiberstrom von maximal 10 mA auf 2 mA reduziert. Dies reduziert den Stromverbrauch und elektromagnetische Abstrahlung, macht den Ausgang aber auch langsamer. Der Wert in PTIX ergibt den gleichen Wert wie in PTX, wenn der Ausgang nicht kurzgeschlossen ist.

Wird DDRXb auf 0 gesetzt, ist der Pin als Eingang programmiert. Der Wert am Anschluss kann über PTIX $_n$ oder PTX $_n$ gelesen werden; man kann auf PTX schreiben, aber beim Zurücklesen wird nicht der Registerwert, sondern der Wert am Anschluss zurückgegeben.

Wenn der Anschluss als Eingang konfiguriert ist, kann man ihn entweder mit einem Pulldown oder einem Pullup versehen. Um Pullup oder Pulldown zu aktivieren, muss man Register PERX $_n$ mit einer 1 programmieren. In diesem Fall versieht eine 1 in Register PPSX $_n$ den Eingang mit einem Pull-down von ca. 130 µA, und eine 0 mit einem Pullup.

Unbenutzte Anschlüsse sollten immer als Ausgang programmiert werden, wenn sichergestellt ist, dass an ihnen nichts angeschlossen wird. Alternativ kann man sie auch als Eingang programmieren, muss dann aber sicherstellen, dass sie auf der Leiterplatte mit Masse oder der Versorgungsspannung verbunden sind oder die Pullups bzw. die Pulldowns aktiviert sind. Auf keinen Fall darf man An-

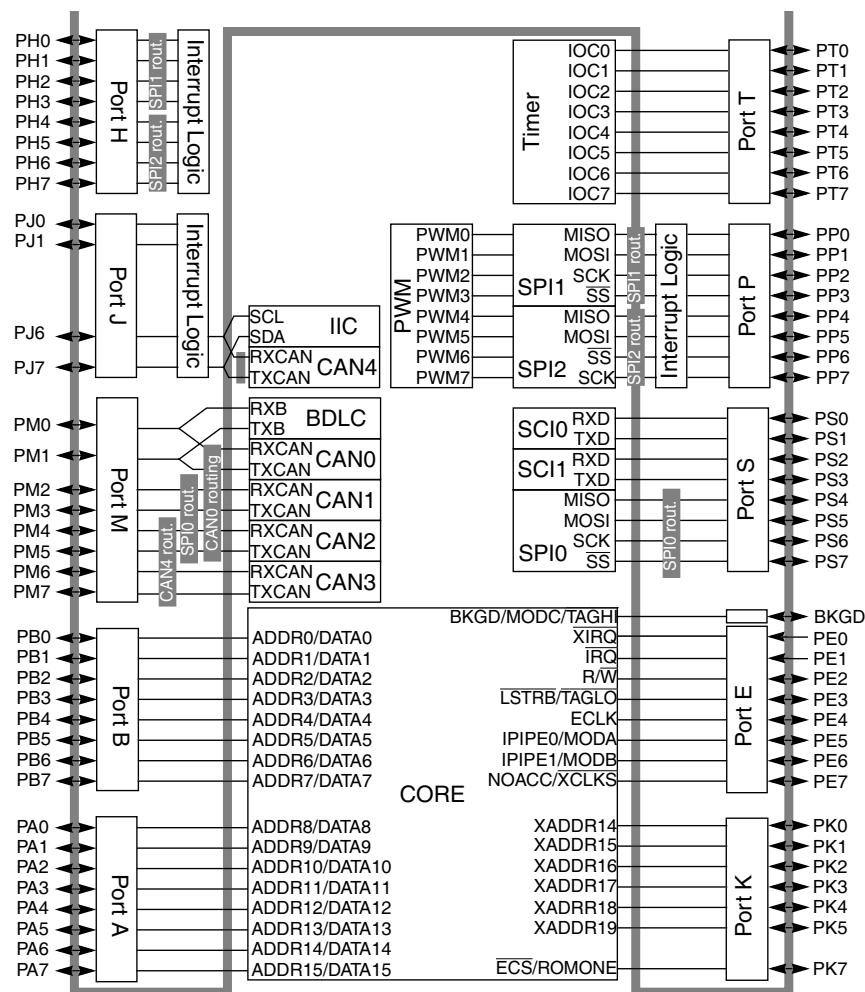


Abbildung 3-1: Überblick über die Ports des 68HCS12 9DP256

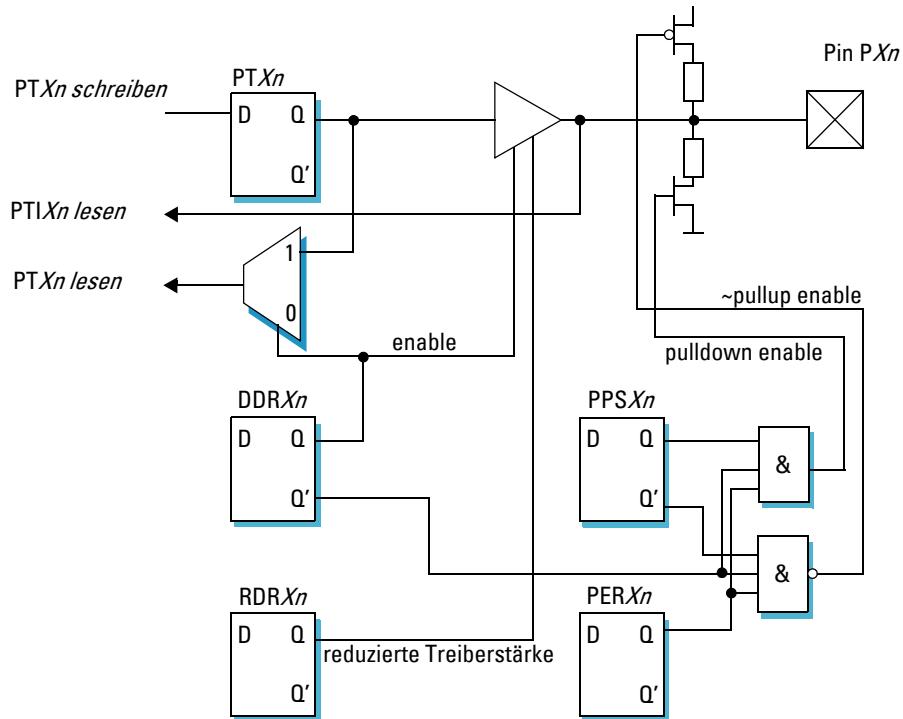


Abbildung 3-2: Schematische Darstellung eines General Purpose I/O-Pins

schlüsse als Eingang einfach offen lassen, da dies zu einem sogenannten Latch-Up-Effekt führen kann, der den Prozessor inaktiviert.

Man kann die Anschlüsse auch unabhängig voneinander verwenden, d.h. einen Teil als Eingang und einen anderen Teil als Ausgang programmieren. Dazu eignen sich besonders die Bit-Manipulationsbefehle. Wollten wir z.B. Port H Pin 2 als Eingang ohne Pullup oder Pulldown verwenden, könnten wir so vorgehen:

```
bclr DDRH,#4
bclr PERH,#4
```

Wir könnten den Wert am Anschluss lesen und wenn dort eine 1 steht, nach *branch1* verzweigen durch

```
brsetPTIH,#4,branch1
```

Wollten wir den Pin3 des Port H als Ausgangspin mit reduziertem Treiberstrom betreiben, könnten wir so vorgehen:

```
bset DDRH,#8  
bset RDRH,#8
```

Wir würden den Anschluss auf 1 setzen mit

```
bset PTH,#8
```

und auf 0 mit

```
bclr PTH,#8
```

Jeder Pin kann als Tristate-Pin verwendet werden, indem man ihn über Register DDRH entweder hochohmig, d.h. als Eingang konfiguriert, oder als Ausgang.

3.3.1 Mehrfach verwendete Anschlüsse

Einige Anschlüsse sind neben ihrer allgemeinen Funktion als Vielzweck-Anschlüsse mit speziellen Peripheriemodulen assoziiert, wie z.B. Analog-Digital-Wandlern, seriellen Schnittstellen oder Zeitgebern. Sobald diese speziellen Module aktiviert sind, überschreiben sie die Funktion des allgemein verwendbaren Anschlusses wie er weiter oben beschrieben worden ist. Einige Funktionen sind allerdings immer noch möglich, wie in Abbildung 3-3 gezeigt.

Das Modul steuert die Richtung des Anschlusses (Ein- oder Ausgabe) und die Treiberstärke. Allerdings lässt sich die Treiberstärke bzw. die Pulldown- bzw. Pullup-Konfiguration weiterhin separat einstellen. Den Wert am Anschlusspin kann man immer über PTIX lesen. Das PTX-Register hat keine Wirkung, aber man kann darein schreiben, und wenn das DDRX-Bit gesetzt ist, auch daraus lesen.

3.3.2 Kernmodul-Ports

Die Ports A, B, E und K sind Teil des sogenannten Kernmoduls (CPU Core-Moduls), d.h. es gibt sie in allen Ausführungen dieses Prozessors. Diese Ports haben eine einfache Struktur. Es gibt keine Pulldown-Möglichkeit, und die Pullups können nur gemeinsam für den gesamten Port über das

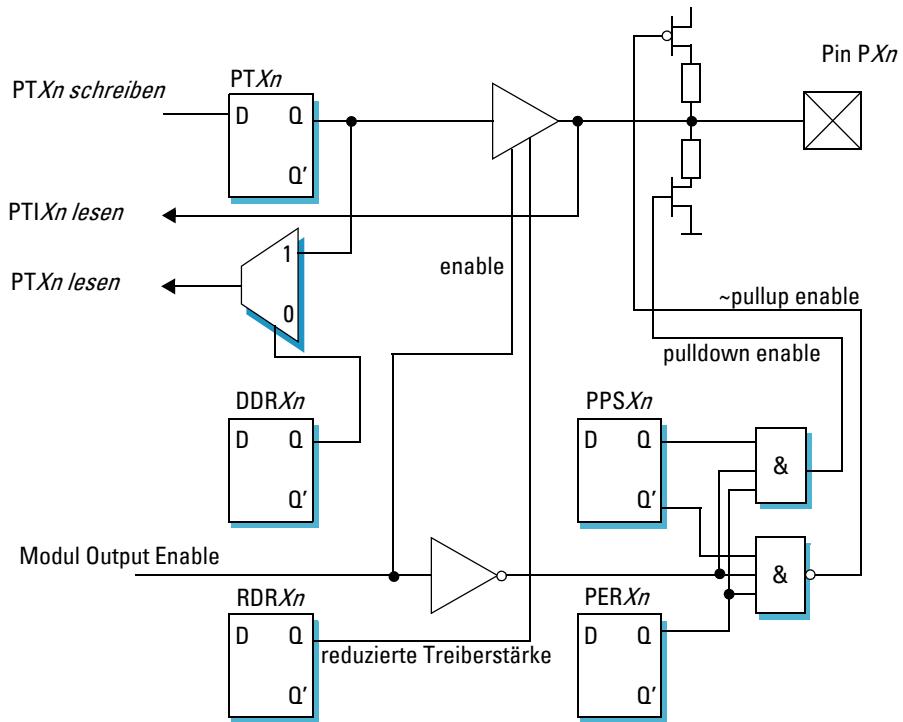


Abbildung 3-3: Überschriebene Funktion eines General Purpose I/O-Pins

PUCR-Register konfiguriert werden. Die Treiberstärke kann auch nur insgesamt für den gesamten Port über das RDRV-Register eingestellt werden. Ports E und K haben eine spezielle Bedeutung und sollten nicht für allgemeine Ein-Ausgabezwecke verwendet werden.

3.4 Zeitgeber (Timer)

In vielen Anwendungen benötigen Rechner ein Konzept von Zeit. Dies gilt insbesondere für Echtzeitsysteme, aber auch z.B. für Computerspiele, die in einem bestimmten Tempo ablaufen sollen. Aus diesem Grund besitzen die meisten Rechner Zeitgeber, die so konfiguriert werden können, dass

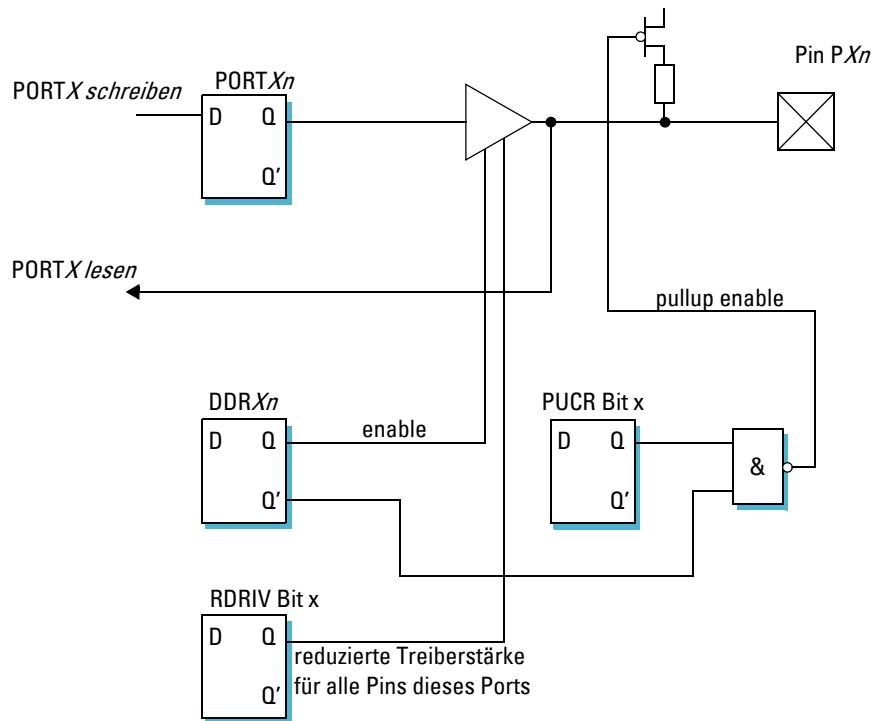


Abbildung 3-4: Ports des Kernmoduls (A, B, E und K)

bei Ablauf einer Zeit ein Interrupt erzeugt wird. Kontroller für eingebettete Anwendungen besitzen z.T. sehr umfangreiche Zeitgebermodule, um z.B. Aktoren wie Servomotoren oder Magnetventile zu steuern, und Zeiten und Impulsfrequenzen zu messen, wie es z.B. in vielen Anwendungen im Automobil erforderlich ist.

Der von uns eingesetzte Mikrocontroller besitzt ein sehr umfangreiches Zeitgebermodul, das hier nur z.T. besprochen werden kann. Wir können seine Funktionalität in vier Klassen aufteilen:

- Freilaufender Zähler
- Zeitmessung bei Eingangssignal (Input Capture)
- Erzeugung zeitabhängiger Ausgangssignale (Output Compare)

- Impulszähler

Der Kontroller besitzt einen durch einen Oszillator getriebenen 16-Bit-Zähler. Dieser Zähler kann für die Messung relativer Zeiten benutzt werden. Daneben stellt der Kontroller 8 Zeitgeber-Kanäle zur Verfügung, die verschieden konfiguriert werden können.

Im „Input Capture“-Modus wird der Wert des 16-Bit-Zählers in ein Register übernommen, wenn am zugeordneten Eingangspin ein Signalübergang detektiert worden ist. Damit kann man z.B. Frequenzen oder Periodendauern eines externen Signals messen.

Ein Kanal kann auch im „Output Compare“-Modus betrieben werden. Hier wird der zugeordnete Ausgangspin auf einen bestimmten Pegel gesetzt oder verändert, sobald der 16-Bit-Zähler den im zugeordneten Vergleichsregister eingestellten Wert erreicht hat. Damit kann man zeitlich genau definierte Ausgangssignale z.B. für die Steuerung von Servomotoren oder auch die Ansteuerung eines kleinen Lautsprechers erzeugen. Allerdings bietet der Kontroller noch ein eigenständiges Modul für die Erzeugung von pulsweitenmodulierten Signalen an.

Es steht ein 16-Bit Impulszähler zur Verfügung, mit dessen Hilfe man Eingangsimpulse summieren kann. Zusammen mit dem Zeitgeber kann man so z.B. Frequenzmessungen vornehmen oder auch sehr lange Zeiten messen.

Dem Zeitgebermodul ist der Port T zugeordnet. Jedem Zeitgeberkanal ist einer der 8 Anschlüsse des Ports T zugeordnet.

Der Haupttakt für das Zeitgebermodul leitet sich aus dem Prozessor-Bustakt ab. Dieser ist bei unserem Rechner in der Betriebsart als Evaluation Board (EVB) auf 24 MHz eingestellt. Im Boot-Mode muss der Anwendungsprogrammierer die Frequenz selbst einstellen; der voreingestellte Wert ist deutlich niedriger als 24 MHz.

3.4.1 Freilaufender Zähler

Der freilaufende Zähler des Zeitgebermoduls hat eine zentrale Bedeutung. Die acht Zeitgeberkanäle benutzen den Wert dieses Zähler zur Realisierung ihrer Funktionen. Der eigentliche Zähler ist über das Register TCNT erreichbar, er ist nur lesbar. Der Zähler wird nicht direkt vom Bustakt getaktet, sondern über einen Vorteiler (Prescaler). Dieser ist nach einem Reset auf einen Teilkoeffizienten von 1 eingestellt. Über das Register TSCR2, Bits 0 bis 2 (PR2, PR1, PR0) lassen sich Teilkoeffizienten von 2^n einstellen, wobei n der Wert der aus PR2, PR1 und PR0 gebildeten Binärzahl ist. Der maximale Teilkoeffizient ist also 128, der minimale 1.

Um eine Zeitdifferenz zu messen, speichern wir zunächst den aktuellen Wert von TCNT als Anfangszeit. Später laden wir wieder den dann aktuellen Wert von TCNT und ziehen davon den Anfangswert ab:

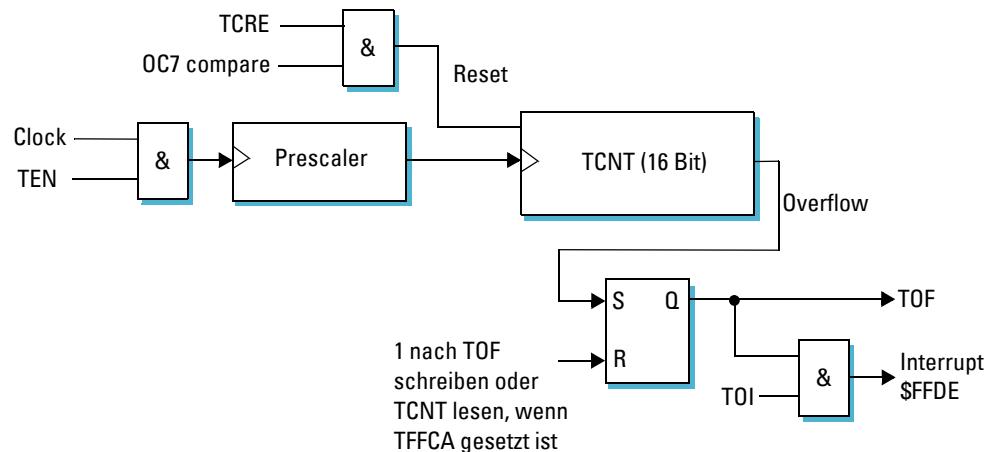


Abbildung 3-5: Konfiguration des freilaufenden Zählers

```

movw TCNT, startZeit ; startZeit ist eine Wort-Variable
; hier kann jetzt etwas passieren, von dem wir die Zeit messen wollen
ldd TCNT           ; hier laden wir die neue Zeit
subd startZeit     ; in D steht jetzt die Zeitdifferenz

```

Der absolute Wert von TCNT ist nicht relevant, und es spielt auch keine Rolle, wenn der Zähler während der Messung überläuft. Die einzige Einschränkung ist, dass die Differenz nicht größer als 65535 Zähleinheiten betragen darf.

Wie groß kann die auf die oben dargestellte Weise zu messende Zeit maximal sein, wenn der Bustakt 24 MHz beträgt? Wie groß ist in diesem Fall die zeitliche Auflösung? Wie groß ist die minimale zeitliche Auflösung (Prescaler steht auf 1)?

Immer wenn der Zähler überläuft, wird das TOF-Status-Bit gesetzt. Man kann das Bit dadurch zurücksetzen, dass man eine 1 in das Register schreibt. Ist das TFFCA-Steuer-Bit gesetzt, wird das TOF

Aufgabe 3-1

durch Lesen des TCNT automatisch zurückgesetzt. In diesem Fall funktioniert das Rücksetzen durch Schreiben einer 1 nach TOF nicht mehr.

Table 3-1: Timer Count Control und Status-Bits

Timer Count Control und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TSCR1	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0
TSCR2	TOI	0	0	0	TCRE	PR2	PR1	PR0
TFLG2	TOF	0	0	0	0	0	0	0

Durch Setzen des TOI-Bits kann man dafür sorgen, dass ein Zählerüberlauf zu einem Interrupt führt. Auf diese Weise kann man den Hardwarezähler um einen Softwarezähler von praktisch beliebiger Länge erweitern. Beispiel für die Erweiterung auf 32 Bit:

```
timerInterruptServiceRoutine:
    ldd timerExtension; liegt irgendwo im Speicher (ds.w 1)
    addd #1
    std timerExtension
    movb #$80,TFLG2; Interrupt-Flag zurücksetzen
    rti
```

Probleme kann es hier beim Lesen des Zählers geben, da sich der Wert während des Lesens ändern kann. Das wird sicher nur selten passieren, aber es kann passieren:

```
movw timerExtension, startZeit
movw TCNT, startZeit+2
```

Der movw-Befehl benötigt 6 Bustaktzyklen für seine Ausführung. In dieser Zeit kann es schon zu einem Zählerüberlauf gekommen sein, so dass z.B. nach der Ausführung des ersten Befehls die Interrupt-Service-Routine angesprungen wird. Sie inkrementiert den Wert in timerExtension, während TCNT wieder bei 0 beginnt. Wenn wir aus der Interrupt-Service-Routine zurückkehren und TCNT gelesen haben, steht in startZeit und startZeit+2 ein inkonsistenter Wert (wir sind in der Zeit zurückgefallen).

Aufgabe 3-2

Überlegen Sie sich ein Verfahren, mit dem Sie das oben geschilderte Problem beheben können. Welche Vor- und Nachteile hat ihre Lösung?

3.4.2 Zeitmessung von Eingangssignalen (Input Capture)

Diese Betriebsart behandeln wir zur Zeit nicht.

3.4.3 Zeitabhängige Ausgangssignale (Output Compare)

Mit Hilfe der „Output Compare“-Funktionalität lassen sich Einzelimpulse oder Impulsfolgen erzeugen. Damit ein Kanal in dieser Betriebsart arbeiten kann, muss der Hauptzeitgeber, der freilaufende Zeitzähler aktiviert sein ($TEN=1$) und das entsprechende Bit IOS_n im Register TIOS muss von seinem voreingestellten Wert von 0 auf 1 gesetzt werden.

Dadurch wird der Wert von TCNT kontinuierlich mit dem Wert des Registers TC_n verglichen, und wenn die beiden Werte übereinstimmen, wird das Flag-Bit CnF im Register TFLG1 gesetzt. Je nach Einstellung der Bits OM_n und OL_n in den Registern TCTL1 bzw. TCTL2 wird in diesem Fall der Ausgangspin PT_n gesetzt, zurückgesetzt, oder sein Wert geändert.

Table 3-2: Timer Output Compare Control und Status-Bits

Timer Output Compare Control und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TIOS	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	IOS0
CFORC	FOC/	FOC6	FOC5	FOC4	FOC3	FOC2	FOC1	FOC0
TSCR1	TEN	TSWAI	TSFRZ	TFFCA	0	0	0	0
TSCR2	TOI	0	0	0	0	0	0	0
TCTL1	OM7	OL7	OM6	OL6	OM5	OL5	OM4	OL4
TCTL2	OM3	OL3	OM2	OL2	OM1	OL1	OM0	OL0
TIE	C7I	C6I	C5I	C4I	C3I	C2I	C1I	C0I
TFLG1	C7F	C6F	C5F	C4F	C3F	C2F	C1F	C0F

Mit Hilfe der Bits in TIE lässt sich für jeden der acht Kanäle konfigurieren, ob bei Gleichstand der TCNT und TC_n -Register ein Interrupt erzeugt wird.

Das folgende kleine Programm erzeugt eine Rechteckschwingung mit einer Frequenz von 10 kHz am Anschluss PT4. Der Kanal 4 des Zeitgebermoduls wird hier im Polling-Betrieb verwendet. Eine Frequenz von 10 kHz entspricht einer Periodendauer von 100 µs; der Pegel muss aber alle 50 µs geändert werden, damit sich eine symmetrische Rechteckschwingung ergibt:

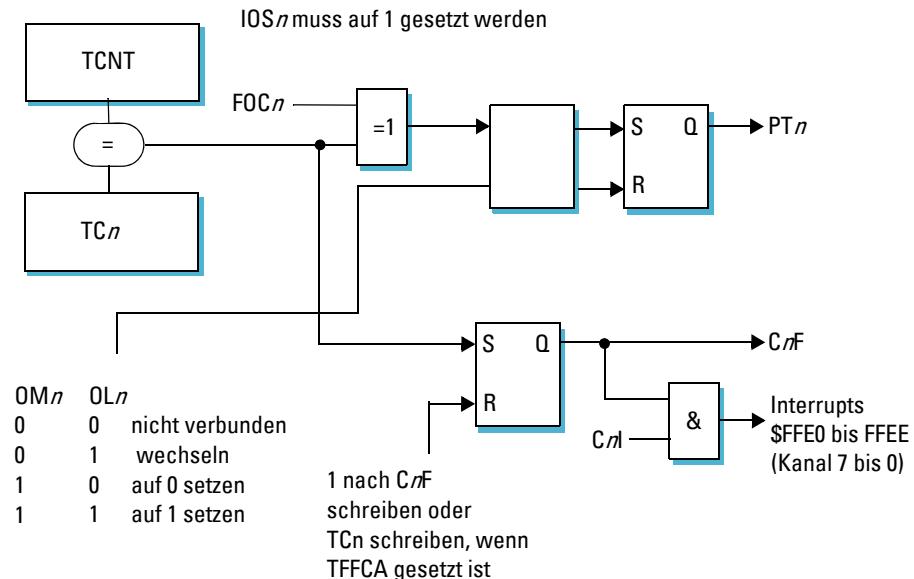


Abbildung 3-6: Zeitgeber im Output Compare-Modus

```

...
        bset TSCR1,#$90; setze TEN und TFFCA-Bits
        bset TIOS, #$10; setze IOS4 (Kanal 4 im Output Compare-Modus)
        bset TCTL1, #1 ; setze OL4 (ändere Ausgang PT4 bei Gleichstand)
        ldd TCNT       ; lade momentanen Zählerstand
        addd #50*24    ; addiere dazu 50 Mikrosekunden
        std TC4        ; speichere diesen Zukunftswert im Vergleichsregister
loop1:
        brclrTFLG1, #$10, loop1; warte, bis C4F-Flag gesetzt ist
        addd #50*24    ; berechne nächsten Zeitpunkt
        std TC4        ; setze neue Zeit und setze Flag zurück
        bra loop1

```

Die ersten drei Befehle initialisieren das Zählermodul. Die Konfiguration ist so gewählt, dass der Zustand des Anschlusses bei jedem erfolgreichen Vergleich wechselt, also von 1 auf 0, und danach von 0 auf 1 usw.

Die auf diese Weise erzielbare maximale Pulslänge erreichen wir, wenn wir zum momentanen Zählerstand 65535 dazu addieren. Die minimale Periode ist dadurch beschränkt, dass die vier Befehle innerhalb der Abfrageschleife in der halben Periode abgearbeitet werden können müssen.

3.5 Pulsweitenmodulator

Pulsweitenmodulation ist eine verbreitete Technik, um digitalisierte Datenwerte über eine einzige Leitung ohne zusätzlichen Takt zu übertragen. Pulsweitenmodulation wird z.B. für die Ansteuerung von Servomotoren, in der Audio-Datenübertragung (Mobiltelefonen) zur Erzeugung von Tönen und Geräuschen sowie zur einfachen Digital-Analogwandlung benutzt.

Bei der Pulsweitenmodulation wird eine kontinuierliche Folge von Rechteckimpulsen mit einer konstanten Frequenz erzeugt. Variiert wird die Zeit zwischen der steigenden und fallenden Flanke eines Impulses, also die „Weite“ des Impulses. Die Weite des Impulses repräsentiert damit einen analogen Wert zwischen 0% einer Taktperiode und 100% einer Taktperiode.

Für die Erzeugung pulsweitenmodulierter Signale benötigt man eine Zähler und zwei Register; in einem Register wird die Pulsweite eingestellt, in dem anderen die Taktperiode. Zu Beginn einer Periode wird der Zähler auf 0 gesetzt und der Ausgang auf 1. Der Zähler wird durch einen Taktgeber erhöht. Sobald der Wert im Zähler dem Wert im ersten Register entspricht, wird der Ausgang auf 0 gesetzt. Sobald der Wert im Zähler den Wert im zweiten Register erreicht, wird er auf 0 gesetzt und ein neuer Zyklus beginnt.

Der Wert im ersten Register muss immer kleiner sein als der Wert im zweiten Register. Die Pulsweite wird dadurch moduliert, dass man den Wert des ersten Registers verändert.

Bei der Pulsdicthtemodulation geht man ähnlich vor; hier wird allerdings die Pulsweite konstant gehalten und die Taktperiode variiert.

Unser im Labor verwendeter Rechner besitzt acht PWM-Kanäle mit jeweils 8-Bit-Zählern und Registern. Diese können aber zu insgesamt vier Kanälen mit 16-Bit Zählern und Registern zusammengefasst werden.

Die PWM-Kanäle sind dem Port P zugeordnet. Dieser Port wird auch vom Serial Peripheral Interface (SPI) verwendet. Jeder Pin kann entweder einem PWM-Kanal oder dem SPI zugeordnet werden. Ist keins von beiden der Fall, steht er als allgemeiner Port-Anschluss zur Verfügung.

Die PWM-Generatoren werden vom Systemtakt getrieben, allerdings nicht direkt, sondern über konfigurierbare Vorteiler (Prescaler). Es gibt zwei Vorteiler-Kanäle, die unterschiedlich eingestellt werden können. Der Vorteiler A wird von den PWM-Kanälen 0, 1 4 und 5 benutzt, während der Vorteiler B von den PWM-Kanälen 2,3,6 und 7 benutzt wird.

Die Teile bestehen aus zwei hintereinandergeschalteten Einheiten. Die erste Einheit teilt durch Zweierpotenzen von 2^0 bis 2^7 . Die zweite Teilerstufe der Kaskade erlaubt eine Teilung durch N, mit $N = 1$ bis 256. Der zweite Teiler ist optional. Wird er nicht benutzt, ist der Teilstufenfaktor 2^M , mit $M = 0$ bis 7. Wird er benutzt, ist der Teilstufenfaktor $2^{M+1} \cdot N$, es findet also noch zusätzlich eine Teilung durch 2 statt.

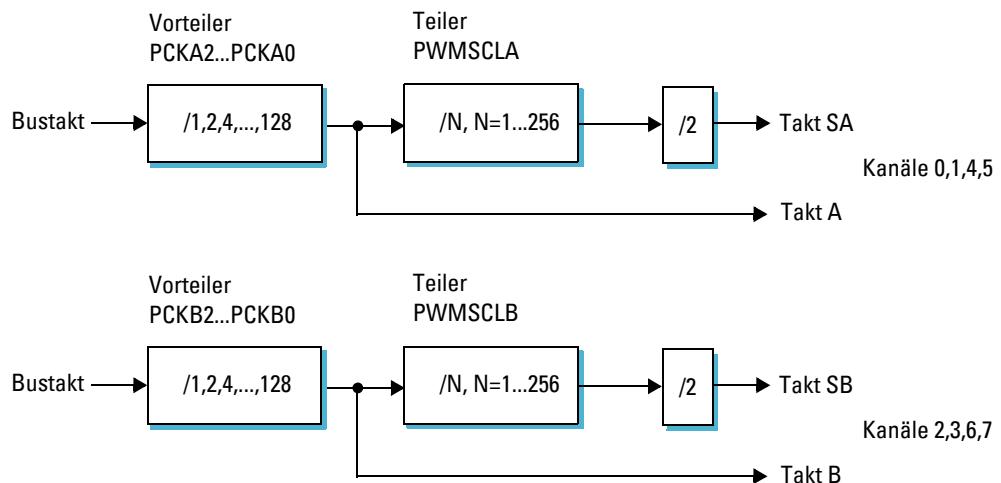


Abbildung 3-7: Konfiguration der PWM-Vorteiler

Ein PWM-Kanal wird durch Setzen des entsprechenden $PWMEn$ -Bits aktiviert. Die Polarität des Ausgangssignals wird durch das entsprechende $PPOLn$ -Bit bestimmt. Ist es auf 1 gesetzt, werden positive Pulse erzeugt (logisch 1). Jeder Kanal kann mit oder ohne zweiten Vorteiler betrieben werden. Soll der Teiler verwendet werden, d.h. der PWM-Kanal durch die Takte SA bzw. SB getrieben werden, muss das entsprechende $PCLKn$ -Bit auf 1 gesetzt werden.

Table 3-3: Pulsweitenmodulator Control- und Status-Bits

Pulsweitenmodulator Control- und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PWME	PWME7	PWME6	PWME5	PWME4	PWME3	PWME2	PWME1	PWME0
PWMPOL	PPOL7	PPOL6	PPOL5	PPOL4	PPOL3	PPOL2	PPOL1	PPOL0
PWMCAE	CAE7	CAE6	CAE5	CAE4	CAE3	CAE2	CAE1	CAE0
PWMCLK	PCLK7	PCLK6	PCLK5	PCLK4	PCLK3	PCLK2	PCLK1	PCLK0
PWMPRCLK	0	PCKB2	PCKB1	PCKB0	0	PCKA2	PCKA1	PCKA0
PWMCTL	CON67	CON45	CON23	CON01	PSWAI	PFRZ	0	0
PWMSCLA	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PWMSCLB	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

Jeder PWM-Kanal hat ein Zählregister PWMCNT_n und zwei Vergleichsregister PWMDTY_n und PWMPER_n . Die Pulsweite (duty cycle) wird über PWMDTY_n eingestellt, die Periode über PWM- PER_n . Schreiben auf PWMCNT_n setzt diesen Zähler zurück auf 0.

Mit Hilfe des PWMCAE-Registers kann man einstellen, dass Pulse zentriert und nicht links ausgerichtet werden. In diesem Fall sind die Pulsweiten und Taktperioden doppelt so hoch. Ein Periode startet immer in der Mitte eines Pulses.

Angenommen, wir wollten 1 μs weite Impulse mit einer Taktperiode von 10 μs erzeugen. Die Bustaktfrequenz beträgt 24 MHz. Um eine Auflösung von 1 μs zu erhalten, benutzen wir die kaskadierten Teiler mit M=0 und N=12. Damit erhält das Taktperiodenregister einen Wert von 10 (10 mal 1 μs) und das Duty Cycle-Register einen Wert von 1. Der folgende Programmausschnitt konfiguriert den Kanal 0 mit Pin 0 an Port P:

```

movb #12, PWMCSCLA; Taktrate von 1  $\mu\text{s}$ 
movb #1, PWMCLK ; nutze Takt SA
movb #1, PWMPOL ; positive Pulse
movb #1, PWMDTY0 ; Pulsweite 1 * 1  $\mu\text{s}$ 
movb #10, PWMPERO ; Periode ist 10*1  $\mu\text{s}$ 
movb #1, PWME ; aktiviere Kanal 0

```

3.6 Serielle Schnittstellen

Wird in der Vorlesung „Bussysteme“ behandelt.

3.7 Analog-Digitalwandler

Viele Anwendungen erfordern eine Anbindung von Rechnern an analoge Sensoren, z.B. Beschleunigungsaufnehmer, Temperaturfühler oder Drucksensoren. Damit solche Werte im Rechner verarbeitet werden können, müssen sie zuerst digitalisiert werden.

Der von uns verwendete Rechner besitzt zwei Analog-Digital-Konverter mit jeweils 10 Bit Auflösung, die nach dem Verfahren der sukzessiven Approximation arbeiten. Jeder der beiden A/D-Wandler kann über einen Analog-Multiplexer auf einen von jeweils acht Eingängen geschaltet werden. Der Wandler ist relativ schnell; er wird mit maximal 2 MHz getaktet und benötigt 14 Taktzyklen für eine komplette Wandlung. Das entspricht einer Wandlungszeit von 7 μ s. Betreibt man den Wandler im 8-Bit-Modus, kann die Anzahl der Wandlungszyklen um 2 reduziert werden, d.h. die minimale Wandlungszeit beträgt dann 6 μ s.

Der A/D-Wandler benötigt zwei Referenzspannungen, VRH und VRL. Die zu messenden Spannungen müssen zwischen diesen beiden Werten liegen. Die Auflösung des A/D-Wandlers beträgt damit $(VRH-VRL)/1024$ Volt. Bei einer Differenz von 5 Volt für $(VRH-VRL)$ beträgt die Auflösung damit ca. 5 mV.

3.7.1 Initialisierung

Der Mikrocontroller besitzt zwei A/D-Wandler. Die Steuerungs- und Statusregister des ersten Wandlers heißen ATD0xxx und liegen im Speicherraum ab Adresse \$80. Die Datenregister heißen ADR0xxx bzw. PORTAD0, wenn die Anschlüsse als Digitaleingänge verwendet werden.

Der Wandler muss vor seiner Verwendung über die Steuerungsregister 2 bis 4 konfiguriert werden. Die Konfigurationsphase wird durch Schreiben in das Steuerungsregister 5 abgeschlossen; damit wird gleichzeitig eine Wandlung gestartet.

Das ADPU-Bit aktiviert das A/D-Wandlermodul. In der Voreinstellung ist das Modul abgeschaltet, da es einen relativ hohen Stromverbrauch hat. Der A/D-Wandler benötigt 10 μ s um betriebsbereit zu sein, nachdem das ADPU-Bit gesetzt worden ist. Alle anderen Bits in ATDxCTL3 können normalerweise auf 0 gesetzt werden. Wird Bit ASCIE gesetzt, wird am Ende einer Wandlung ein Interrupt ausgelöst und das Interrupt-Flag ASCIF wird gesetzt. Durch Schreiben in ATDxCTL5 wird das Flag zurückgesetzt und eine neue Wandlungssequenz wird gestartet.

Die Bits in ATDxCTL3 und ATDxCTL4 müssen richtig gesetzt werden, damit der Wandler wie gewünscht arbeitet. Bits S8C bis S1C bestimmen die Anzahl der Wandlungen pro Befehl. Man darf 1 bis 8 Wandlungen einstellen; andere Werte bedeuten 8 Wandlungen. Die Ergebnisse werden in den Ergebnisregister ADRx0 bis ADRx7 abgelegt, jeweils beginnend mit ADRx0. Ist das FIFO-Bit gesetzt, werden weitere Ergebnisse angehängt. Bei fünf Wandlungen pro Befehl liegen der Ergebnisse also in ADRx0, ADRx1, ADRx2, ADRx3 und ADRx4 für den ersten Wandlungsbefehl, und in ADRx5, ADRx6, ADRx7, ADRx0, ADRx1 für den zweiten Wandlungsbefehl.

Table 3-4: Analog-Digitalwandler Control- und Status-Bits

Analog-Digitalwandler Control- und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ATD0CTL2	ADPU	AFFC	AWAI	ETRIGLE	ETRIGP	ETRIGE	ASCIE	ASCIF
ATD0CTL3	0	S8C	S4C	S2C	S1C	FIFO	FRZ1	FRZ0
ATDOCTL4	SRES8	SMP1	SMP0	PRS4	PRS3	PRS2	PRS1	PRS0
ATD0CTL5	DJM	DSGN	SCAN	MULT	0	CC	CB	CA
ATD0STAT0	SCF	0	ETORF	FIFOR	0	CC2	CC1	CC0
ATD0STAT1	CCF7	CCF6	CCF5	CCF4	CCF3	CCF2	CCF1	CCF0

Die Bits FRZ1 und FRZ0 bestimmen, ob im Debug-Modus die Wandlungen weitergehen, wenn der Prozessor auf einen Breakpoint gelaufen ist.

Der A/D-Wandler darf höchstens mit 2 MHz und muss mindestens mit 500 kHz getaktet werden. Die Taktfrequenz ergibt sich aus dem Bustakt geteilt durch einen Vorteiler. Der Vorteiler wird mit Hilfe der Bits PRS0 bis PRS4 eingestellt. Um z.B. bei einem Bustakt von 24 MHz den maximalen A/D-Wandler-Takt zu erzielen, müsste der Vorteiler auf einen Wert von 12 eingestellt werden. Der Teiler ergibt sich aus der aus PRS0 bis PRS4 gebildeten Dualzahl mal 2. Ein Vorteilerfaktor von 12 wäre also 0010 (PRS4...PRS0), was auch die Voreinstellung ist.

Table 3-5: Analog-Digitalwandler Control- und Status-Bits, konkrete Werte

Analog-Digitalwandler Control- und Status-Bits								
Register	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
ATD0CTL2	1	0	0	0	0	0	0	0

Table 3-5: Analog-Digitalwandler Control- und Status-Bits, konkrete Werte

Analog-Digitalwandler Control- und Status-Bits								
ATD0CTL3	0	0	0	0	1	0	0	0
ATDOCTL4	0	1	0	0	0	1	0	1

Mit SRES8 gesetzt arbeitet der Wandler als 8-Bit-Wandler. SMP1 und SMP0 bestimmen die Anzahl der Taktzyklen für den letzten Abtastzyklus; sie sollten auf ihrem voreingestellten Wert von 10 belassen werden.

Damit sieht eine typische Konfiguration wie in Tabelle 3-5 gezeigt aus.

3.7.2 Kanalselektion

Jeder Analog-Digitalwandler kann mit einem von acht Eingängen verbunden werden. Dazu dienen die Bits CA bis CC im Register ATDxCTL5. Ist das DJM-Bit gesetzt, werden die Wandlungsergebnisse als Integer im Wertebereich 0 bis 1023 zur Verfügung gestellt. Steht dieses Bit auf 0, wird das Ergebnis als binäre Rationalzahl dargestellt (0,1111...). Das erlaubt es, die Auswertung unabhängig davon zu halten, ob man den Wandler mit 8- oder 10-Bit Auflösung betreibt. Außerdem kann man mit Bit DSGN auch eine vorzeichenbehaftete Darstellung einstellen.

Wird das Bit MULT gesetzt, wird automatisch für jeden Analogeingang eine Wandlung vorgenommen, beginnend bei dem mit Ca bis CC eingestellten Eingang, modulo 8.

Wird das Bit SCAN gesetzt, arbeitet der Wandler kontinuierlich, ohne dass man die Wandlungen anstoßen müsste.

Eine Wandlungssequenz wird ansonsten begonnen, indem man in das Register ATDxCTL5 schreibt. Sobald ein Ergebnis in ein Register ADRxx gespeichert worden ist, wird das entsprechende Flag CCFx gesetzt. Die CCx-Bits in Register ATDxSTAT0 zeigen an, wohin das nächste Ergebnis gespeichert werden wird.

Das Bit AFFC in ATD0CTL2 bestimmt den „Fast Flag Clear All“-Modus des A/D-Wandlers. Ist es auf 0 gestellt, wird das CCFx-Bit durch Lesen von ATD0STAT1 und anschließendes Lesen des zugehörigen Datenregisters zurückgesetzt. Steht AFFC auf 1, genügt das Lesen des jeweiligen Datenregisters, um das SCF-Bit und das zugehörige CCFx-Bit zurückzusetzen.

Rechenleistung

4.1 Definition von Rechenleistung

Es gibt mehrere Blickwinkel, unter denen Rechenleistung definiert werden kann. Die beiden wichtigsten sind

- Antwort- oder Ausführungszeit
- Durchsatz

Der Benutzer eines Personal-Computers würde wahrscheinlich sagen, der schnellere Rechner sei der, der die ihm vom Benutzer übertragene Aufgabe in der kürzeren Zeit erledigt. Ihn interessiert die Antwortzeit (engl. response time) oder Ausführungszeit (engl. execution time). Der Betreiber einer Serverfarm würde allerdings wahrscheinlich sagen, dass die schnelleren Rechner diejenigen sind, die pro Tag die meisten Aufgaben erledigt bekommen. Er ist mehr interessiert an dem Durchsatz (engl. throughput).

Wenn wir zukünftig über Rechenleistung sprechen, geht es immer um die Ausführungszeit. Wir definieren:

$$\text{Rechenleistung}_x = \frac{1}{\text{Ausführungszeit}_x}$$

Wir nennen Computer A n mal schneller als Computer B wenn gilt:

$$\frac{\text{Rechenleistung}_A}{\text{Rechenleistung}_B} = \frac{\text{Ausführungszeit}_B}{\text{Ausführungszeit}_A} = n$$

Der Computer, der die meiste Arbeit pro Zeiteinheit erledigen kann, ist also nach dieser Definition der schnellste. Typischerweise geht man bei der Messung von Rechenleistung von einer konstanten Menge zu erledigender Aufgaben aus, einem sogenannten „Benchmark“. Dies sind Programme, die für einen bestimmten Rechner übersetzt und dann dort ausgeführt werden. Die für die Ausführung verwendete Zeit dient als Vergleich für die Rechenleistung eines Computersystems. Daraus wird deutlich, dass nicht nur die Rechnerhardware die so gemessene Rechenleistung bestimmt, sondern auch der Compiler mit seinen Einstellungen und das Betriebssystem, das eventuell bei der Ausführung verwendet wird. Wird die Zeit gemessen, kann man wieder unterschiedliche Zeiten angeben: die Zeit, die das eigentliche Programm auf der Rechnerhardware gelaufen ist, oder die verstrichene Echtzeit (Wanduhrenzeit) oder die insgesamt von der Anwendung und vom Betriebssystem verbrauchte Zeit. Wir konzentrieren uns hier auf die Zeit, die eine Anwendung an Rechenleistung benötigt und lassen, soweit das möglich ist, die in einem Betriebssystem verbrauchte Zeit außer acht. Diese Zeit nennen wir die CPU-Zeit. Wenn wir über Rechenleistung sprechen, meinen wir damit das Reziproke der CPU-Zeit.

Praktisch alle CPUs werden durch einen Takt getrieben, der außer bei Änderung von Betriebszuständen zur Reduktion des Energieverbrauchs in der Regel konstant ist. Damit können wir auch sagen, dass ein bestimmtes Programm eine definierte Anzahl an Taktzyklen benötigt, bis es abgearbeitet ist. So erhalten wir:

$$\text{CPU-Zeit} = \frac{\text{CPU-Taktzyklen}}{\text{Taktrate}}$$

Damit sehen wir, wie Rechenleistung erhöht werden kann: für das gleiche Programm können wir entweder weniger Taktzyklen benötigen oder wir nutzen eine höhere Taktrate, oder beides. Die Anzahl der für ein Programm notwendigen CPU-Taktzyklen ergibt sich aus der Anzahl der (MaschinenSprache-)Anweisungen für ein Programm mal der durchschnittlichen Anzahl der Taktzyklen pro

Maschinenzyklus für dieses Programm (CPI, clock cycles per instruction). Damit sind diese drei Größen von elementarer Bedeutung für die Rechenleistung:

- Anzahl der Maschinenbefehle für ein bestimmtes Programm
- Taktrate
- Anzahl der Takte pro Maschinenbefehl

Manchmal findet man die Bezeichnung „MIPS“ (Million Instructions per Second) als eine Größe für Rechenleistung. Leider ist sie recht unbrauchbar, wenn man bedenkt, dass dabei völlig ignoriert wird, wie viele Maschinenbefehle (Instruktionen) für ein bestimmtes Programm notwendig sind. So ließe sich vielleicht ein sehr einfacher Rechner bauen, der fürchterlich viele Instruktionen pro Sekunde abarbeiten könnte, die aber alle nicht besonders viel leisten. Soll der Rechner dann eine echte Aufgabe erledigen, würde er trotz seiner hohen Geschwindigkeit u.U. länger benötigen als ein Rechner mit mächtigeren Befehlen, die aber im Durchschnitt mehr Taktzyklen benötigen.

4.2 Der Kern des Prozessors: Datenpfad und Steuerung

4.2.1 Single Cycle Datenpfad

Abbildung 4-1 zeigt einen einfachen Prozessor mit einem Datenpfad, der seine Aufgabe in einem Taktzyklus erledigt. Dazu ist eine Trennung von Befehls- und Datenspeicher für die Operanden notwendig. Die meisten realen Prozessoren sind so nicht aufgebaut, sondern haben einen Multi-Cycle Datenpfad.

4.2.2 Multi Cycle Datenpfad

Abbildung 4-2 zeigt einen abstrakten Blick auf einen einfachen Prozessor mit einem Datenpfad, der für die Ausführung einer Instruktion mehr als einen Taktzyklus benötigen kann. Damit wird es möglich, Befehls- und Datenspeicher zusammenzulegen. Hinter jede größere funktionelle Einheit werden Register eingefügt, um den Ausgabewert als Eingabewert für die folgende Stufe bereit zu halten.

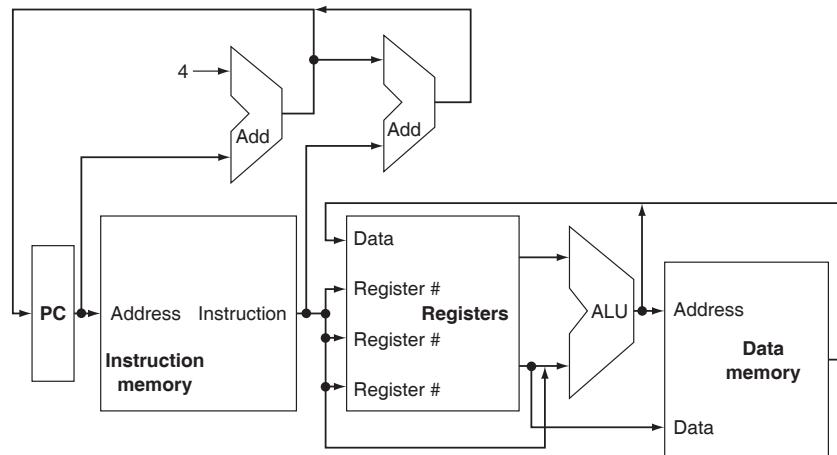


Abbildung 4-1: Single Cycle Datenpfad

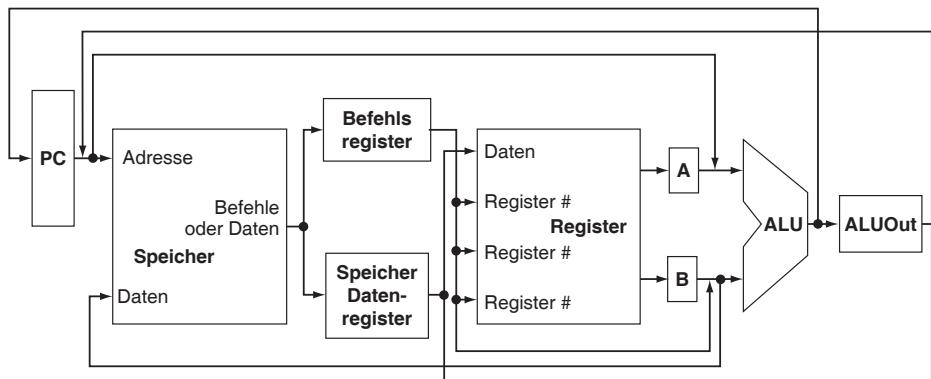


Abbildung 4-2: Multi Cycle Datenpfad

4.3 Pipelining

4.3.1 Überblick

Pipelining ist eine Technik, bei der eine CPU mehrere Befehle zeitlich parallel bearbeiten kann. Neben der mehr technologiebedingten Erhöhung von Taktraten und der Verbreiterung des Datenpfades ist Pipelining eine der wesentlichen Methoden, um Rechenleistung zu erhöhen. Da Pipelining die Möglichkeit ausnutzt, mehrere Instruktionen gleichzeitig abzuarbeiten, gehört es zum sogenannten „Instruction Level Parallelism“. Die Bearbeitung von Befehlen erfolgt typischerweise in diesen wesentlichen Schritten:

1. Befehl aus Befehlsspeicher holen
2. Befehl dekodieren
3. Operanden aus Datenspeicher holen (optional)
4. Befehl ausführen
5. Ergebnis in Datenspeicher ablegen (optional)

Instruction Level Parallelism (ILP)

Eine solche Pipeline besteht also aus fünf Stufen. Wenn alle fünf Stufen gleichzeitig arbeiten könnten, erhielten wir eine Geschwindigkeitssteigerung um den Faktor 5.

Pipelining wird erheblich vereinfacht, wenn alle Befehle gleich lang sind, d.h. in einem Taktzyklus aus dem Befehlsspeicher geholt werden und im direkt darauf folgenden vollständig dekodiert werden können. Es gibt Rechnerarchitekturen, die so entworfen sind; ein Intel IA-32 hat aber z.B. Befehls-längen zwischen 1 und 17 Bytes, was das Pipelining deutlich erschwert. Wenn man genauer hinsieht, übersetzt der IA-32 die Maschinenbefehle zuerst in einfachere, gleich lange Befehle; erst diese werden dann in eine Pipeline geschoben.

4.3.2 Pipeline Hazards

Es ist nicht immer möglich, den nächsten Befehl auch im nächsten Taktzyklus zu bearbeiten. Ein solches Ereignis nennt man „Hazard“. Es gibt drei verschiedene Typen von Hazards.

Strukturelle Hazards

Ein struktureller Hazard entsteht, wenn die Architektur es nicht erlaubt, Funktionen parallel auszuführen. Das könnte z.B. dadurch verursacht sein, dass wir gleichzeitig einen Befehl aus einem Speicher holen wollen, in den wir in diesem Moment auch ein Ergebnis schreiben.

Daten-Hazards

Daten-Hazards entstehen, wenn Operationen von vorhergegangenen Operationen abhängen, die sich noch in der Pipeline befinden. Das kann z.B. passieren, wenn die neuere Operation das Ergebnis der älteren Operation benötigt, diese die Pipeline aber noch nicht verlassen hat, d.h. das Ergebnis noch gar nicht bereit stellen konnte.

Man kann versuchen, Daten-Hazards zu vermeiden, indem man Befehlsfolgen umsortiert, also z.B. einen späteren Befehl vorzieht. Weiterhin kann man das Ergebnis eventuell schon intern zur Verfügung stellen, bevor es z.B. im Register oder Speicher sichtbar wird. Diese Technik nennt man „Forwarding“ oder „Bypassing“.

Steuerungs- oder Verzweigungs-Hazards

Steuerungs- oder Verzweigungs-Hazards entstehen, wenn eine Entscheidung ansteht, diese aber noch nicht gefällt werden konnte. Es müsste nun ein neuer Befehl in die Pipeline gebracht werden, aber es ist noch nicht klar, welcher. Die Pipeline gerät so ins Stocken, bis die Entscheidung klar ist. Man versucht dieses Problem mit einer Vorhersage zu lösen: man nimmt z.B. einfach an, dass eine Verzweigung nie genommen wird („untaken branch“). Die Pipeline gerät dann nur ins Stocken, wenn diese Vorhersage verkehrt war.

Etwas intelligenter Verfahren sehen z.B., dass eine Entscheidung am Fuß einer Schleife liegt. Schleifen werden typischerweise mehrfach durchlaufen, und so ist die vorhergesagte Entscheidung der Sprung zum Kopf der Schleife.

Die leistungsfähigsten Rechner benutzen dynamische Vorhersagen. Für eine bestimmte Verzweigungsstelle wird in einem Puffer aufgezeichnet, wie oft in der Vergangenheit die eine oder andere Entscheidung gefällt wurde. Dadurch werden die Vorhersagen immer genauer, durchschnittlich über 90%.

4.3.3 Multiple Issue

Es gibt zwei grundlegende Methoden, Instruction Level Parallelism (ILP) auszunutzen. Im ersten Fall erhöht man die Tiefe der Pipeline. Vorausgesetzt, dass die Verarbeitung in jeder Stufe in etwa die gleiche Zeit benötigt, kann man dadurch die Taktrate entsprechend erhöhen.

Ein weiterer Ansatz vervielfacht die Anzahl der einzelnen Komponenten im Datenpfad des Computers und bearbeitet damit mehrere Befehle *pro Stufe* gleichzeitig. Diese Technik nennt man „Multiple Issue“, übersetzt vielleicht als „Mehrfacheingabe“. Pro Taktzyklus werden mehrere Befehle parallel in den Datenpfad eingebracht. Mit diesem Ansatz ist es möglich, eine Befehlsrate zu erzielen, die höher ist, als die Taktrate. Anders ausgedrückt kann man ein CPI von kleiner als 1 erzielen.

Moderne Hochleistungsrechner versuchen, zwischen drei und acht Instruktionen pro Taktzyklus einzubringen. Allerdings gibt es viele Randbedingungen, die einzuhalten sind und die dazu führen, dass nicht immer die theoretisch mögliche Verbesserung der Rechenleistung erzielt wird.

Beim Multiple Issue unterscheidet man noch einmal zwischen zwei Arten: statischem Multiple Issue und dynamischem Multiple Issue.

Statisches Multiple Issue

Beim statischen Multiple Issue wird zur Kompilierzeit vom Compiler versucht, Befehle so anzurorden, dass eine parallele Verarbeitung möglich ist. Befehle werden gewissermaßen zu Befehlspaketen zusammengeschnürt, von denen der Compiler weiß, dass die Rechnerhardware sie parallel verarbeiten kann. Man kann sich diese Befehlspakete damit auch als eigene, sehr mächtige Befehle vorstellen; das hat auch zu dem ursprünglichen Namen dieser Technik geführt: Very Long Instruction Word (VLIW). Die Intel IA-64-Architektur benutzt diesen Ansatz, nur heißt er hier „Explicitly Parallel Instruction Computer“ (EPIC). Der Compiler ist beim Statischen Multiple Issue oft auch für die Vermeidung von Steuerungs- und Daten-Hazards zuständig.

Very Long Instruction Word

Der Vorteil des statischen Multiple Issue-Ansatzes ist, dass man die Analyse bezüglich der Parallelisierbarkeit einmal beim Übersetzen eines Programms durchführen kann, so dass tiefgehende Analysen und Optimierungen möglich sind.

Dynamisches Multiple Issue

In Prozessoren mit dynamischem Multiple Issue sorgt die Rechnerhardware zur Laufzeit dafür, dass wenn möglich Befehle auf mehreren Funktionseinheiten parallel ablaufen. Solche Rechner nennt man auch superskalare Rechner. Der vom Compiler erzeugte Code kann wie beim statischen Multiple Issue schon voroptimiert sein; allerdings kann die Hardware nie die Analysen durchführen, die ein Compiler beim Übersetzen eines Programms anwenden kann. Auf der anderen Seite gibt es gegenüber dem statischen Multiple Issue-Ansatz einen deutlichen Vorteil: der Code läuft unabhängig von der Optimierfähigkeit des Prozessors oder seiner tatsächlichen Pipeline-Struktur immer korrekt ab. Bei einigen VLIW-Entwürfen müssen Programme dagegen neu kompiliert werden, wenn man auf ein neues Prozessormodell wechselt.

Superskalare Prozessoren

Anhang B

5.1 Hardware für die Laborübungen

Für die Laborübungen wird eine Hardware verwendet, die mit einem 16-Bit Freescale-Prozessor vom Typ MC9S12DP256B ausgestattet ist. Das reichhaltig ausgestattete Board „Dragon12“ kann unter <http://www.evbplus.com> für ca. 130,00 Euro (USD 139,00 plus 19% Zollgebühr) käuflich erworben werden. Ein Teil der Versuche und Übungen kann auch ohne das Board mit dem Simulator der Entwicklungsumgebung durchgeführt werden.

Zum Betrieb des Boards ist noch ein 9-Volt Netzteil mit mindestens 6 Watt erforderlich. Für die Verbindung mit dem Entwicklungsrechner benötigt man einen RS-232-Anschluss.

Für die Laborversuche haben wir den mitgelieferten Debug-Monitor durch einen Monitor ersetzt, der mit der Metrowerks-Entwicklungsumgebung zusammenarbeitet. Die ursprünglichen Debug-12-Kommandos funktionieren also nicht mehr.

5.2 Allgemeine Hinweise

Das Herunterladen von Software auf den Zielrechner ist in der Regel problemlos möglich. Man muss aber sicher stellen, dass der Debug-Monitor aktiv ist. Es empfiehlt sich deshalb, vor dem Herunterladen den Reset-Knopf des Zielsystems zu betätigen.

Das Board muss im EVB-Modus konfiguriert sein, damit es mit dem Debugger zusammen arbeitet. Das bedeutet, dass SW7 so eingestellt sein muss, dass die linke LED (EVB) rechts unten in der Ecke leuchtet.

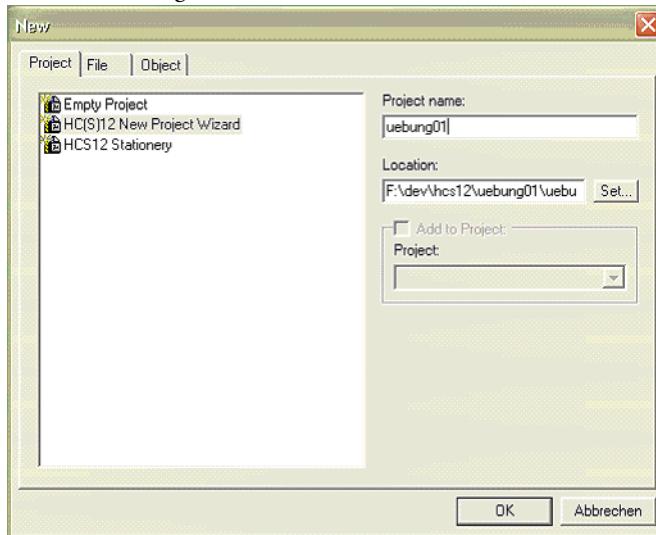
Der Monitor stellt die Bus-Taktfrequenz des Prozessors auf 24 MHz ein. Das sollte auch nicht ge-

ändert werden, da sonst die Anbindung des Debuggers über die serielle Schnittstelle nicht mehr richtig arbeitet.

Hat man keinen Erfolg beim Herunterladen der Software, ohne dass eine Fehlermeldung erscheint, hat man wahrscheinlich statt „Monitor“ „Simulator“ als Debug-Ziel gewählt.

5.3 Anlegen eines neuen Projekts

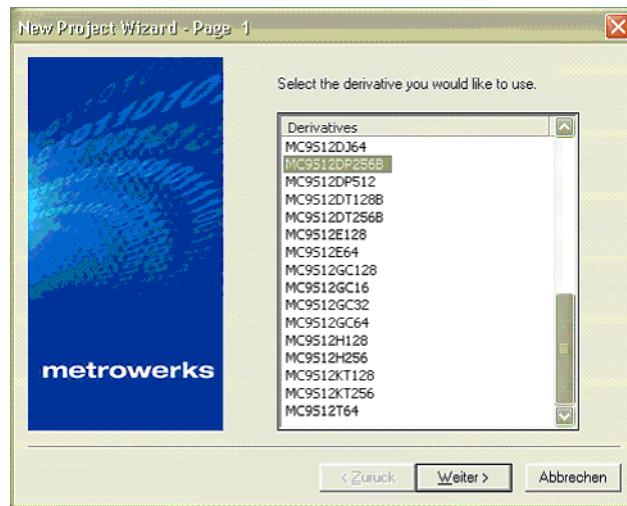
Neue Projekte kann man anlegen, indem man ein existierenden kopiert oder über den Assistenten ein neues erzeugt. Hier ist kurz die Vorgehensweise mit Hilfe des Assistenten beschrieben. Über <File> und <New> wird der Assistent gestartet.



Man vergibt einen geeigneten Projektnamen und einen Ort, wie die Projektdateien gespeichert werden sollen.

Als nächstes ist der richtige Prozessortyp auszuwählen. Auf dem Labor-Board läuft ein MC9S12DP256B.

Als nächstes wählen wir, ob wir ein Assembler-Projekt oder ein C-Projekt anlegen wollen. Es können auch gemischte Projekte angelegt werden. Für die Option „C++“ haben wir leider keine Lizenz. Wir



wählen für unser Beispiel ein Assemblerprojekt, und zwar ein sogenanntes „relokierbares“. Damit können wir Assemblerprogramme schreiben, bei denen über den Linker festgelegt wird, wohin im Speicher sie später gelegt werden. Außerdem können zu dem Projekt so mehrere Assemblerquelldateien gehören, ohne dass wir uns über die Anordnung zu viel Gedanken machen müssen.

5.4 Peripherie

Im Labor wird nur ein Teil der zur Verfügung stehenden Peripherie verwendet. In den folgenden Abschnitten sind die wichtigsten Elemente beschrieben. Weitere Informationen erhält man durch Studium des Schaltbildes, das im Dokumentationsset der Vorlesung zur Verfügung steht.

5.4.1 LED-Zeile

Es steht eine Zeile mit acht roten LEDs zur Verfügung. Diese sind am Port B des Prozessors angeschlossen, und zwar so, dass ein positives Signal auf den Port gegeben werden muss, damit die LEDs eingeschaltet werden. Jede LED ist mit einem Bit des Ports B verbunden; die am weitesten links stehende LED ist mit Bit 0 verbunden.

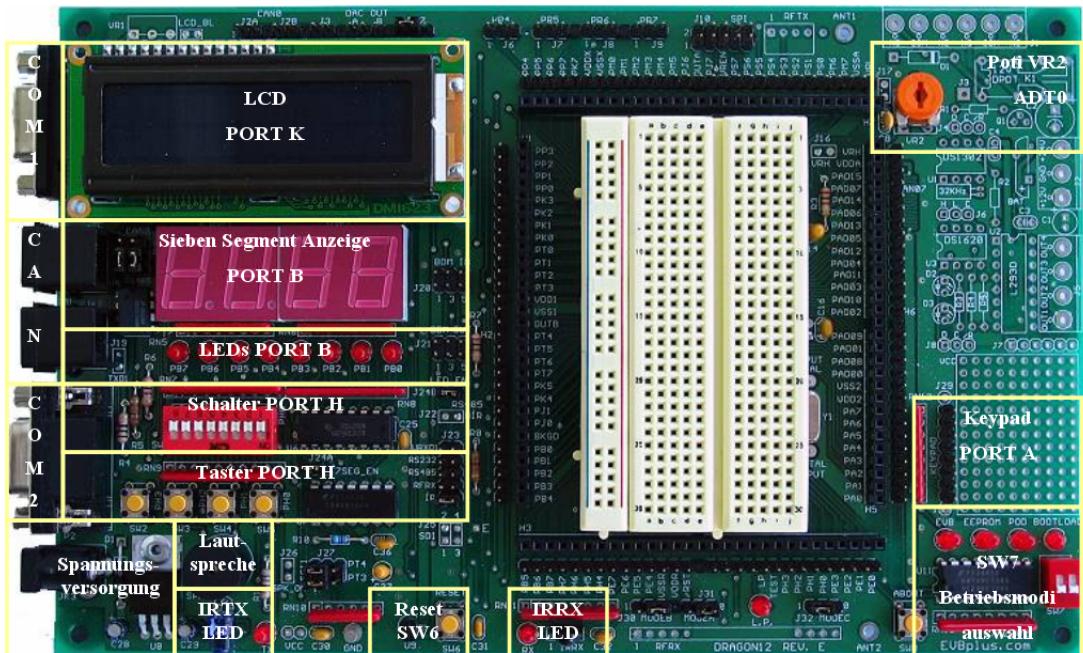


Figure 5-1: Das im Labor verwendete Dragon12-Board von Wytec

Port B wird zweifach verwendet: einmal für die LED-Zeile und dann noch für die LED-Siebensegment-Anzeige. Um die LED-Zeile zu aktivieren und nicht das Siebensegment-Display, muss Bit 1 von Port J auf null gesetzt werden und mit Port P die Siebensegment-Anzeigen ausgeschaltet werden.

Beispiel in Assembler:

```

ldaa #$ff
staa DDRB
clr PORTB           ; port B as output
bset DDRJ,#%00000010 ; turn on LED array
bclr PTJ, #%00000010
bset DDPR,#%00001111 ; turn off 7 segment display
bset PTP, #%00001111

```

Die Namen der Register können in Assembler und C leicht unterschiedlich benannt sein; sie können sich auch etwas von den Namen in der Freescale-Dokumentation unterscheiden. Entscheidend sind bei Assemblerprogrammierung die Namen in der Datei `mc9s12dp256.inc`. Die Basisadressen für die einzelnen I/O-Registerblöcke sind in Tabelle 5-1 aufgeführt. Die Adressen der einzelnen Register innerhalb eines I/O-Blocks ergeben sich aus den Basisadressen plus den im jeweiligen Dokument aufgeführten Register-Offsetadressen.

Alle Register stehen mit ihren Adressen schon in der oben erwähnten Include-Datei, man muss sie also nicht selbst definieren. Der Einfachheit halber folgt hier ein unvollständiger Auszug:

```
; #####  
;      Filename : mc9s12dp256.inc  
; #####  
  
;*** Memory Map and Interrupt Vectors  
;*****  
RAMStart:      equ $00001000  
RAMEnd:        equ $00003FFF  
ROM_C000Start: equ $0000C000  
ROM_C000End:   equ $0000FF7F  
Vportp:        equ $0000FF8E  
Vporth:        equ $0000FFCC  
Vportj:        equ $0000FFCE  
Vatd1:          equ $0000FFD0  
Vatd0:          equ $0000FFD2  
Vtimovf:       equ $0000FFDE  
Vtimch7:       equ $0000FFE0  
Vtimch1:       equ $0000FFEC  
Vtimch0:       equ $0000FFEE  
Vrti:          equ $0000FFF0  
Virq:          equ $0000FFF2  
Vxirq:         equ $0000FFF4  
Vswi:          equ $0000FFF6  
Vtrap:         equ $0000FFF8  
Vcop:          equ $0000FFFA  
Vreset:        equ $0000FFFE  
  
;  
;*** PORTAB - Port AB Register; 0x00000000 ***  
PORTAB:        equ $00000000  
  
;*** PORTA - Port A Register; 0x00000000 ***  
PORTA:         equ $00000000
```

Tabelle 5-1: Device Memory Map

Address	Module	Size (Bytes)
\$0000 - \$0017	CORE (Ports A, B, E, Modes, Inits, Test)	24
\$0018 - \$0019	Reserved	2
\$001A - \$001B	Device ID register (PARTID)	2
\$001C - \$001F	CORE (MEMSIZ, IRQ, HPRIO)	4
\$0020 - \$0027	Reserved	8
\$0028 - \$002F	CORE (Background Debug Mode)	8
\$0030 - \$0033	CORE (PPAGE, Port K)	4
\$0034 - \$003F	Clock and Reset Generator (PLL, RTI, COP)	12
\$0040 - \$007F	Enhanced Capture Timer 16-bit 8 channels	64
\$0080 - \$009F	Analog to Digital Converter 10-bit 8 channels (ATD0)	32
\$00A0 - \$00C7	Pulse Width Modulator 8-bit 8 channels (PWM)	40
\$00C8 - \$00CF	Serial Communications Interface 0 (SCI0)	8
\$00D0 - \$00D7	Serial Communications Interface 0 (SCI1)	8
\$00D8 - \$00DF	Serial Peripheral Interface (SPI0)	8
\$00E0 - \$00E7	Inter IC Bus	8
\$00E8 - \$00EF	Byte Data Link Controller (BDLC)	8
\$00F0 - \$00F7	Serial Peripheral Interface (SPI1)	8
\$00F8 - \$00FF	Serial Peripheral Interface (SPI2)	8
\$0100 - \$010F	Flash Control Register	16
\$0110 - \$011B	EEPROM Control Register	12
\$011C - \$011F	Reserved	4
\$0120 - \$013F	Analog to Digital Converter 10-bit 8 channels (ATD1)	32
\$0140 - \$017F	Motorola Scalable Can (CAN0)	64
\$0180 - \$01BF	Motorola Scalable Can (CAN1)	64
\$01C0 - \$01FF	Motorola Scalable Can (CAN2)	64
\$0200 - \$023F	Motorola Scalable Can (CAN3)	64
\$0240 - \$027F	Port Integration Module (PIM)	64
\$0280 - \$02BF	Motorola Scalable Can (CAN4)	64
\$02C0 - \$03FF	Reserved	320
\$0000 - \$0FFF	EEPROM array	4096
\$1000 - \$3FFF	RAM array	12288
\$4000 - \$7FFF	Fixed Flash EEPROM	16384
\$8000 - \$BFFF	Flash EEPROM Page Window	16384
\$C000 - \$FFFF	Fixed Flash EEPROM	16384

```
;*** PORTB - Port B Register; 0x00000001 ***
PORTB:      equ      $00000001
```

```
;*** DDRAB - Port AB Data Direction Register; 0x00000002 ***
DDRAB:          equ      $00000002
;*** DDRA - Port A Data Direction Register; 0x00000002 ***
DDRA:          equ      $00000002
;*** DDRB - Port B Data Direction Register; 0x00000003 ***
DDRB:          equ      $00000003

;*** PORTK - Port K Data Register; 0x00000032 ***
PORTK:          equ      $00000032

; bit numbers for user in BCLR, BSET, BRCLR and BRSET
PORTK_BIT0:      equ      0          ; Port K Bit 0
PORTK_BIT1:      equ      1          ; Port K Bit 1
PORTK_BIT2:      equ      2          ; Port K Bit 2
PORTK_BIT3:      equ      3          ; Port K Bit 3
PORTK_BIT4:      equ      4          ; Port K Bit 4
PORTK_BIT5:      equ      5          ; Port K Bit 5
PORTK_BIT7:      equ      7          ; Port K Bit 7
; bit position masks
mPORTK_BIT0:     equ      %00000001 ; Port K Bit 0
mPORTK_BIT1:     equ      %00000010 ; Port K Bit 1
mPORTK_BIT2:     equ      %00000100 ; Port K Bit 2
mPORTK_BIT3:     equ      %00001000 ; Port K Bit 3
mPORTK_BIT4:     equ      %00010000 ; Port K Bit 4
mPORTK_BIT5:     equ      %00100000 ; Port K Bit 5
mPORTK_BIT7:     equ      %10000000 ; Port K Bit 7

;*** DDRK - Port K Data Direction Register; 0x00000033 ***
DDRK:          equ      $00000033

;*** SYNR - CRG Synthesizer Register; 0x00000034 ***
SYNR:          equ      $00000034

;*** REFDV - CRG Reference Divider Register; 0x00000035 ***
REFDV:          equ      $00000035

;*** CRGFLG - CRG Flags Register; 0x00000037 ***
CRGFLG:         equ      $00000037

;*** CRGINT - CRG Interrupt Enable Register; 0x00000038 ***
CRGINT:         equ      $00000038

;*** CLKSEL - CRG Clock Select Register; 0x00000039 ***
CLKSEL:         equ      $00000039
```

```
;*** PLLCTL - CRG PLL Control Register; 0x0000003A ***
PLLCTL:           equ      $0000003A

;*** RTICTL - CRG RTI Control Register; 0x0000003B ***
RTICTL:           equ      $0000003B

;*** ARMCOP - CRG COP Timer Arm/Reset Register; 0x0000003F ***
ARMCOP:          equ      $0000003F

;*** TIOS - Timer Input Capture/Output Compare Select; 0x00000040 ***
TIOS:            equ      $00000040

;*** OC7M - Output Compare 7 Mask Register; 0x00000042 ***
OC7M:            equ      $00000042

;*** OC7D - Output Compare 7 Data Register; 0x00000043 ***

;*** PTH - Port H I/O Register; 0x00000260 ***
PTH:             equ      $00000260

;*** PTIH - Port H Input Register; 0x00000261 ***
PTIH:            equ      $00000261

;*** DDRH - Port H Data Direction Register; 0x00000262 ***
DDRH:            equ      $00000262

;*** RDRH - Port H Reduced Drive Register; 0x00000263 ***
RDRH:            equ      $00000263

;*** PERH - Port H Pull Device Enable Register; 0x00000264 ***
PERH:            equ      $00000264

;*** PPSH - Port H Polarity Select Register; 0x00000265 ***
PPSH:            equ      $00000265

;*** PIEH - Port H Interrupt Enable Register; 0x00000266 ***
PIEH:            equ      $00000266

;*** PIFH - Port H Interrupt Flag Register; 0x00000267 ***
PIFH:            equ      $00000267

;*** PTJ - Port J I/O Register; 0x00000268 ***
PTJ:             equ      $00000268

;*** PTIJ - Port J Input Register; 0x00000269 ***
PTIJ:            equ      $00000269
```

```
;*** DDRJ - Port J Data Direction Register; 0x0000026A ***
DDRJ:           equ      $0000026A

;*** RDRJ - Port J Reduced Drive Register; 0x0000026B ***
RDRJ:           equ      $0000026B

;*** PERJ - Port J Pull Device Enable Register; 0x0000026C ***
PERJ:           equ      $0000026C

;*** PPSJ - PortJP Polarity Select Register; 0x0000026D ***
PPSJ:           equ      $0000026D

;*** PIEJ - Port J Interrupt Enable Register; 0x0000026E ***
PIEJ:           equ      $0000026E

;*** PIFJ - Port J Interrupt Flag Register; 0x0000026F ***
PIFJ:           equ      $0000026F
```

5.4.2 Schalter und Taster

Auf dem Board stehen eine Reihe von Schaltern und Tastern zur Verfügung. Die für die Anwendungsprogrammierung interessanten sind am Port H angeschlossen. Die ersten 4 Elemente des DIP-Schalters (PH0 bis PH3) sind mit den vier Tasten SW5 bis SW2 parallel geschaltet. Damit die Taster funktionieren, müssen die DIP-Schalter im Zustand „off“ sein, also in Richtung LCD geschoben.

Port H ist ein weitgehend programmierbarer Port, der auch Interrupts generieren kann. Sind Interrupts aktiviert, muss man auch eine Interrupt-Routine installiert haben, sonst stürzt der Rechner auf Tastendruck hin ab.

Eine umfangreiche Beschreibung der Funktionalität des Ports H findet sich im Dokument „003-Port Integration Module“. In C würde man den Taster SW5 so abfragen:

```
...
DDRH = 0x00; /* nur zur Initialisierung, Port H als Eingang schalten */
...
SW5 = ~(PTH & 0x01); /* Gedrückt, wenn SW5 == 1 */
...
```

5.4.3 Siebensegment-Anzeige

Die Siebensegment-Anzeige ist parallel zu den acht roten LEDs an Port B angeschlossen. Die Ansteuerung erfolgt in einem sogenannten Zeitmultiplexverfahren. Es können immer nur die Segmente

eines Elements angesteuert werden. Über Port P wird eingestellt, welches Element aktiv sein soll. Für das aktive Element muss das zugehörige Bit von Port P auf 0 gesetzt werden. In C würde das so aussehen:

```
...
DDRJ_DDRJ1 = 1; /* LED-Zeile deaktivieren */
PTJ_PTJ1    = 1;

DDRP = 0xff; /* Port P auf Ausgang schalten */

PTP_PTP0 = 0; /* Linke Siebensegment-Anzeige einschalten */
PTP_PTP1 = 1; /* Alle anderen ausschalten */
PTP_PTP2 = 1;
PTP_PTP3 = 1;
...
...
```

Die Anzeigeelemente sind vertauscht montiert. Es ist allerdings durch die Verdrahtung auf dem Board sichergestellt, dass die Segmente gleich angesteuert werden, man muss also die Vertauschung beim Programmieren nicht berücksichtigen. Die Zuordnung der Segmente zu den Bits von Port B für die beiden linken Elemente ist in Abbildung 5-2 dargestellt. Die beiden rechten Elemente kann man genau so ansteuern; Port B Bit 0 ist in diesem Fall z.B. mit Segment D verbunden.

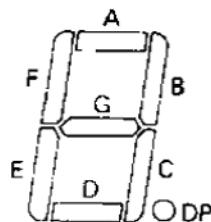
5.4.4 LCD 16x2

Auf dem Board befindet sich ein zweizeiliges Liquid Crystal Display mit einer Zeilenlänge von 16 Zeichen. Das LCD besitzt selbst einen kleinen Mikrocontroller, der mit dem Controller des Laborboards über den Port K kommuniziert. Das LCD wird im 4-Bit-Betrieb benutzt; es sind nur vier Datenleitungen mit dem Labor-Controller verbunden.

Der LCD-Controller ist vom weit verbreiteten Typ HD 44780. Das entsprechende Datenblatt finden Sie in den Unterlagen. Die Verbindung zwischen Labor-Controller und LCD ist in Tabelle 5-2 dargestellt. Es ist zu beachten, dass manche Instruktionen an den LCD-Controller Wartezeiten erfordern.

Tabelle 5-2: Anschluss des LCD an Port K des Controllers

Port K Bit	LCD Funktion
Bit 7	Nicht benutzt
Bit 6	Nicht benutzt



A	= PORTB_BIT0
B	= PORTB_BIT1
C	= PORTB_BIT2
D	= PORTB_BIT3
E	= PORTB_BIT4
F	= PORTB_BIT5
G	= PORTB_BIT6
DP	= PORTB_BIT7

Figure 5-2: Zuordnung von Port B auf die Segmente der Siebensegment-Anzeige

Tabelle 5-2: Anschluss des LCD an Port K des Controllers

Port K Bit	LCD Funktion
Bit 5	DB7 Data Bus
Bit 4	DB6 Data Bus
Bit 3	DB5 Data Bus
Bit 2	DB4 Data Bus
Bit 1	EN - Enable Control Signal
Bit 0	RS - Register select (1=data, 0=instruction)

5.4.5 A/D-Wandler und Potentiometer

Auf dem Labor-Board steht ein Potentiometer zur Verfügung. Dieses ist mit dem Eingang AD07 des Analog-Digitalwandlers ATD0 verbunden. Die Bedeutung der einzelnen Register der A/D-Wandler ist im Dokument „008-AnalogToDigital“ beschrieben.

Die A/D-Wandler können im Interrupt-Modus oder Polling-Modus betrieben werden. Im Interrupt-Modus sieht die Verwendung in C z.B. so aus:

```
unsigned int ADC_Data; /* globale Variable zum Austausch des */
                      /* des gemessenen Wertes */
```

```
/* Die Interrupt-Routine, wird selbständig von Hardware aufgerufen */
interrupt void ADC_ISR(void) {
    ADC_Data      = ATD0DR0;
    ATD0CTL2_ASCIF = 0;
    ATD0CTL5_CA   = 1;
    ATD0CTL5_CB   = 1;
    ATD0CTL5_CC   = 1;
    ATD0CTL5_MULT = 0;
    ATD0CTL5_SCAN = 0;
    ATD0CTL5_DSGN = 0;
    ATD0CTL5_DJM  = 1;
}
```

```
/* Initialisierung des Wandlers */
void ADC_Init(void) {
    ATD0CTL2_ASCIF = 0;
    ATD0CTL2_ASCIE = 1; /* Enable Interrupt */
    ATD0CTL2_ETRIGE = 0;
    ATD0CTL2_ETRIGP = 0;
    ATD0CTL2_ETRIGLE = 0;
    ATD0CTL2_AWAI  = 0;
    ATD0CTL2_AFFC  = 0;
    ATD0CTL2_ADPU  = 1; /* A/D - Wandler Einschalten */
    ATD0CTL3_FRZ0  = 0;
    ATD0CTL3_FRZ   = 0;
    ATD0CTL3_FIFO  = 0;
    ATD0CTL3_S1C   = 1; /* Eine Wandlung pro Sequenz; Ergebnis in ATD0DR0 */
    ATD0CTL3_S2C   = 0;
    ATD0CTL3_S4C   = 0;
    ATD0CTL3_S8C   = 0;
    ATD0CTL4_PRS0  = 1; /* 24MHz Clock */
    ATD0CTL4_PRS1  = 0;
    ATD0CTL4_PRS2  = 1;
    ATD0CTL4_PRS3  = 0;
    ATD0CTL4_PRS4  = 0;
    ATD0CTL4_SMP0  = 0;
    ATD0CTL4_SMP1  = 0;
```

```
ATD0CTL4_SRES8      = 0;
ATD0CTL5_CA         = 1; /* AD7 Eingang - Potentiometer */
ATD0CTL5_CB         = 1;
ATD0CTL5_CC         = 1;
ATD0CTL5_MULT       = 0;
ATD0CTL5_SCAN       = 0;
ATD0CTL5_DSGN       = 0;
ATD0CTL5_DJM        = 1;
}

}
```

5.4.6 Lautsprecher

Auf dem Board ist ein kleiner Lautsprecher montiert. Dieser ist mit dem Bit 5 des Port T verbunden. Durch Hin- und Herschalten mit einer entsprechenden Frequenz lässt sich so ein Ton erzeugen. Port T Bit 5 lässt sich sehr gut durch die Timer-Hardware direkt ansteuern.

5.5 Codewarrior-Simulator

Die Metrowerks-Entwicklungsumgebung beinhaltet einen Simulator für den Freescale-Rechner, der auf dem Dragon12-Board verwendet wird. Der Simulator erlaubt das Ausführen von Programmen ohne die Hardware. Es stehen allerdings nicht alle Ein-/Ausgabeelemente zur Verfügung, und die zur Verfügung stehenden verhalten sich u.U. anders als die Hardware auf dem Dragon12-Board

Die Komponenten können im Simulator über das Menü „Components“ erreicht werden. Hilfreich kann vor allem die Komponente „Led“ sein. Um diese Komponente mit dem Port B zu verknüpfen, muss man mit der rechten Maustaste auf die Komponente klicken und in das Setup-Fenster folgenden String eingeben:

TargetObject.#1

Die „1“ steht dabei für die Adresse des Ports B (0x001). Entsprechend kann man auch andere Ports anschließen.

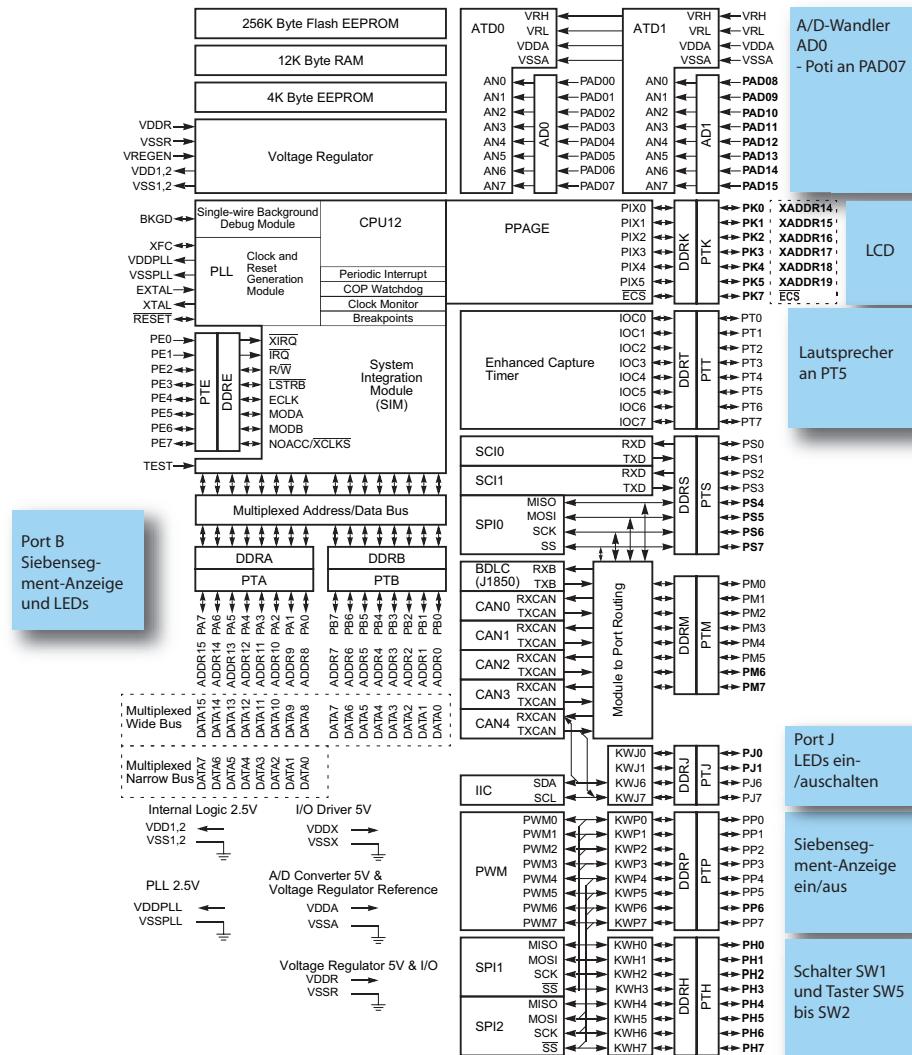


Figure 5-3: Device-Übersicht

Index

A

A/D-Wandler	114
Abstraktion	6
Adressbus	14
Addressierung	35
implizite	34
verdeckte	34
Addressierungsarten	34, 35
Adresszähler	
fiktiver	31
Analog-Digitalwandler	114
Assembler	8
two pass	32
Assembldirektive	
DC	35
DCB	35
DS	34
END	34
EQU	33, 35
ORG	31, 35
SECTION	35
SET	33
XDEF	35
XREF	35
XREFB	35
Assembldirektiven	31, 34
Assemblerprogramm	8
Aufbau	31
Assemblersprache	8
Ausgabe	14

B

Befehle	
---------	--

schnelle	.25
Big Endian	.18
Bus	.14
C	
CCR	.25
C-Datentypen	.28
CISC	.23
CLC	.31
Control	.14
COP	.15
current location counter	.31
D	
Datapath	.14
Datei	
ausführbare	.10
Image-Datei	.10
Datenbus	.14
Datenpfad	.14
Datentypen	.26
Abbildung	.27
Abbildung C nach Assembler	.27
C	.28
Hardware	.26
Hochsprache	.26
DC	.35
DCB	.35
Debug-Information	.10
disassemblieren	.10
Dreiadressbefehl	.34
DS	.34

E

Eingabe	14
einschalten	15
END	34
EQU	33, 35
executable	10

F

Festwertspeicher	10
Format	
Intel-Hex	10
Motorola S-Record	10

H

Harvard-Architektur	22
hello.c	2

I

implizite Adressierung	34
Index-Register	25
Input	14
Instruction Set	21
Instruktionszeiger	25
Intel-Hex	10
ISA	21

K

kompilieren	9
Komponenten	
Aufbau eines Rechners	14
Ausgabe	14
Datenpfad	14
Eingabe	14
Speicher	14
Steuerung	14

L

Label	33
Lader	10
Leistungsverbrauch	4
Linker	10
Linking	10
Little Endian	18
Loader	10
Loading	10
locating	10
Locator	10

M

Maschinencode	7, 9
Memory	14
Memory Map	29
Mnemonic	8
Motorola S-Record	10

O

Object Code	10
Objektdatei	10
Operanden	33
Operandenfeld	33
ORG	31, 35
Output	14

P

PC	25
Port	3
Power-on	16
Program Counter	25
Programmiermodell	21, 25
Programmzähler	25
Prozessor-Register	25

Q

Quelltext 9

R

Rechnerarchitektur

CISC 23

Harvard 22

RISC 23

von-Neumann 22

Rechnerkategorien 2

Rechnerkomponenten 14

Register 25

relokieren 10

Reset 15

Reset-Vektor 15

RISC 23

S

Schichtenmodell 6

SECTION 35

SET 33

Signalzustände 6

Source Code 9

Speicher 14

Speicherbelegung 29

Speicherhierarchie 17

Stack-Pointer 25

Stack-Zeiger 25

Status-Register 25

Steuerbus 14

Steuerung 14

Stromverbrauch 19

superskalar 123

Symbole 10

elektrische 6

system limits 125

T

Taktfrequenz 18

Two Pass Assembler 32

V

verdeckte Adressierung 34

Very Long Instruction Word 123

VLIW 123

von-Neumann-Architektur 22

Vorwärtsreferenzen 32

W

Watchdog 15

X

XDEF 35

XREF 35

XREFB 35

