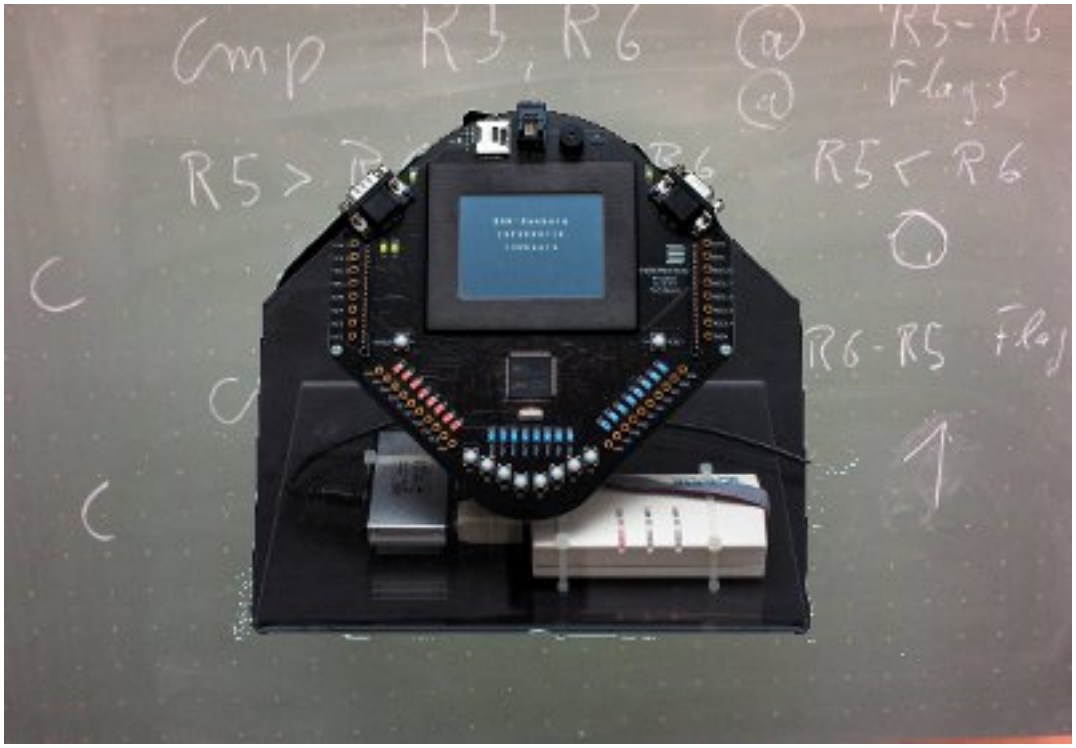
GT/GS SS 14 WS 14/15



Prof. Dr. Reinhard Baran

14. November 2014

Termine:
22.9. Vorlesung 1
29.9. Vorlesung 2
6.10. Vorlesung 3
10.10. Praktikum 1 (3)
13.10. Vorlesung 4
17.10. Praktikum 1 (2,1)
20.10. Vorlesung 5
24.10. Praktikum 2 (1,3)
31.10. Praktikum 3 (3) Praktikum 2 (2)
7.11. Praktikum 3 (1,2)
10.11. Vorlesung 6 Praktikum 4 (2) (14:00)
14.11. Praktikum 4 (1,3)
17.11. Vorlesung 7 Praktikum 5 (3) (14:00)
28.11. Praktikum 5 (1,2)
5.12. Praktikum 6 (1,3)
12.12. Praktikum 7 (3) Praktikum 6 (2)
19.12. Praktikum 7 (1,2)
12.1. Vorlesung 8 (Klausurvorbesprechung)
16.1. Praktikum 8 (1,3)
23.1. Praktikum 8 (2)

Inhaltsverzeichnis

Kapitel 1

Architektur

1.1 Programmierbare Systeme

Beispiele: Zeitschaltuhr, Brotbackautomat, Waschmaschine.

1.1.1 Genereller Aufbau von programmierbaren Rechnersystemen

Eine Gemeinsamkeit von programmierbaren Rechnersystemen ist, dass ihr Verhalten durch Programme bestimmt werden, die in einem schnellen Speicher abgelegt sind. Auch wenn wir in vielen Fällen, wenn wir von Computern sprechen an PCs denken, gibt es wesentlich mehr Rechner, die in Geräte eingebaut sind und dort ihre Funktion erfüllen. Es gibt aber auch Großrechenanlagen, wie sie in Forschungszentren stehen, die ihrerseits einen anderen Zweck haben. Bevor man versucht sie zu klassifizieren, sollte man sich darüber im Klaren werden, welche Bestandteile ihnen gemeinsam sind. Praktisch jeder Computer enthält ein Rechenwerk, Speicher, Ein-Ausgabe Geräte und ein Steuerwerk. Zusätzlich verfügen praktisch alle Computer interne Register sowie Busse, über die Daten und Befehle zwischen den einzelnen Elementen transferiert werden.

Rechenwerk

Das Rechenwerk im Computer ist die Einheit, die arithmetische und logische Operationen durchführt. Häufig wird es auch „ALU" (Arithmetic Logical Unit) genannt. Es berechnet arithmetische und logische Funktionen. Typischerweise kann es mindestens folgende Operationen durchführen:

Arithmetisch
- Addition
- Subtraktion
- Multiplikation

Logisch
- Negation
- Konjunktion (AND)
- Disjunktion (OR)
- Kontravalenz (XOR)

Häufig liegen jedoch auch noch zusätzliche Operationen vor, die sich mit Bitmanipulation beschäftigen. Einige dieser Operationen werden auf mehrfaches Ausführen einfachere Operationen nachgebildet. Dann übernimmt das Steuerwerk die Organisation der Abarbeitung. Die meisten ALUs verarbeiten ganze Zahlen. Eine ALU kann meistens zwei Binärwerte mit gleicher Stellenzahl (n) miteinander verknüpfen. Man spricht von n-Bit ALUs. Typische Werte für n sind 8, 16, 32 und 64. Analog werden z.B. die Begriffe 32-Bit oder 64-Bit-CPU verwendet, wenn über den Prozessor gesprochen wird, in denen diese ALUs zum Einsatz kommen.

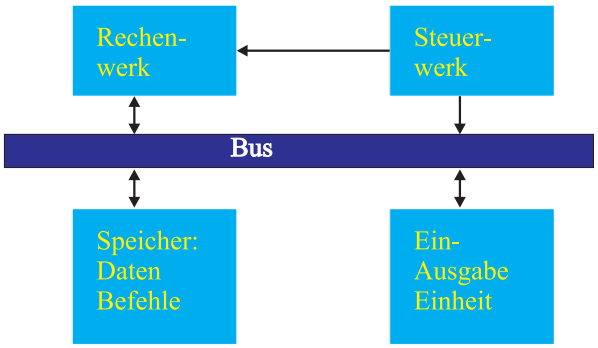


Abbildung 1.1: Aufbau eines von Neumann Rechners

Die n-Bit ALU ist meist aus einzelnen 1-Bit ALUs zusammengesetzt, die jeweils an die höherwertige ALU ein Carry-Bit weiterreichen, mit dem ein Übertrag an der jeweiligen Stelle gekennzeichnet wird. Um die in Reihe geschalteten 1-Bit ALUs in die geforderte Funktionsart umzuschalten, hat jede 1-Bit ALU zusätzlich zu den Eingängen für die zu verknüpfenden Werte und das Carry-Bit noch einen Eingang für einen Steuervektor (Op.-Code), der aus dem Steuerregister (Operationsregister, OR) gelesen wird.

Die Gesamte n-Bit ALU hat außer dem Ausgangsvektor für das Ergebnis noch einen Ausgangsvektor um ihren Zustand zu signalisieren, der im Statusregister (auch Condition Code Register) abgelegt wird. Dieses Register enthält meistens vier Statusbits (in Form von Flipflops). Die einzelnen Werte des Statusregisters können in Computerprogrammbefehlen weiterverwendet werden (z.B. für bedingte Sprünge).

Speicher

Das Steuerwerk holt aus dem Arbeitsspeicher Befehle und Daten. Die Befehle werden im Steuerwerk interpretiert und die Daten im Rechenwerk verarbeitet. Man unterscheidet Speicher, den man lesen und beschreiben kann (RAM = Random Access Memory besonders für Daten) von dem Speicher, den man nur lesen kann (ROM = Read Only Memory besonders für Befehle). Manchmal ist es nützlich, dass RAM Speicher seine Daten auch behält, wenn die Stromversorgung ausfällt (CMOS RAM). (Beispiel Waschmaschine im Schongang). Genauso kann es hilfreich sein Programmspeicher zu löschen. (EPROM) Festplattenspeicher gehören nicht zum Speicherwerk sondern zum Eingabe-/Ausgabewerk, denn die Daten werden von der Festplatte zunächst in den Arbeitsspeicher geladen, bevor sie verarbeitet werden können.

Register

In Prozessoren eines Computers sind sehr schnelle Speicher mit speziellen Funktionen direkt eingebaut. Man nennt sie Register. Die Gesamtheit aller Register eines Prozessors bezeichnet man als Registersatz. Register werden dort zum Zwischenspeichern von Befehlen, Speicheradressen und Rechenoperanden benutzt. Die Registersätze verschiedener Arten von Prozessoren unterscheiden sich in der Art, der Anzahl und der Größe der zur Verfügung stehenden Register.

1.1.2 Von Neumann Architektur

Die Von-Neumann-Architektur benannt nach John von Neumann ist ein Schaltungskonzept zur Realisierung universeller Rechner (Von-Neumann-Rechner, VNR), welches folgende Komponenten enthält:

ALU - Rechenwerk Sie führt Rechenoperationen und logische Verknüpfungen aus.

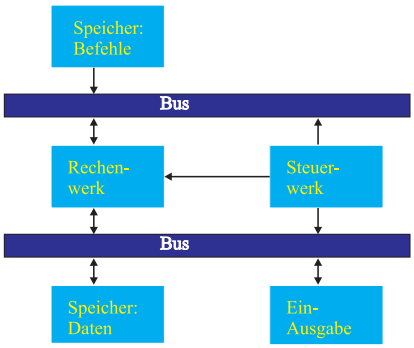


Abbildung 1.2: Aufbau eines Harvard Rechners

Memory - Speicherwerk Es speichert sowohl Programme als auch Daten, welche für das Rechenwerk zugänglich sind.

Control Unit - Steuerwerk Es interpretiert die Anweisungen eines Programmes und steuert die Ausführung dieser Befehle.

I/O Unit - Eingabe-/Ausgabewerk Es steuert die Ein- und Ausgabe von Daten.

Bus - Verbindungssystem Es verbindet alle Komponenten.

Das Rechen- und das Steuerwerk (manchmal auch Leitwerk genannt) bilden die sogenannte Zentraleinheit, den Prozessor. Dieses Konzept, das von Neumann 1945 in dem zunächst unveröffentlichten Papier „First Draft of a Report on the EDVAC" im Rahmen des Baus der EDVAC beschrieb, war zur Zeit seiner Entwicklung revolutionär. Zuvor entwickelte Rechner waren im Allgemeinen an ein festes Programm gebunden, das entweder hardwaremäßig verschaltet war oder über Lochkarten eingelesen werden musste. Mittels der Von-Neumann-Architektur war es nun möglich, Änderungen an Programmen sehr schnell durchzuführen oder in kurzer Folge ganz verschiedene Programme ablaufen zu lassen, ohne Veränderungen an der Hardware vornehmen zu müssen.

Die meisten der heute gebräuchlichen Computer basieren auf dem Grundprinzip der Von-Neumann-Architektur. Die einfache Einteilung der verschiedenen Schaltwerke hat im Laufe der Zeit freilich zahlreiche Veränderungen erfahren. Der Prozess der Befehlsverarbeitung erfolgt in einem Von-Neumann-Rechner mittels so genannter Von-Neumann-Zyklen. Von-Neumann-Rechner gehören nach dem Klassifizierungsschema von Michael J. Flynn zur Klasse der SISD-Architekturen (Single Instruction, Single Data), die keine Parallelverarbeitung vorsieht. Viele Ideen der sogenannten Von-Neumann-Architektur waren schon lange vorher 1936 von Konrad Zuse ausgearbeitet, in zwei Patentschriften von 1937 dokumentiert und größtenteils bereits 1938 in der Z1 mechanisch realisiert worden. Es ist aber unwahrscheinlich, dass von Neumann Zuses Arbeiten kannte, als er 1946 seine Architektur vorstellte.

1.1.3 Harvard

Eine der wichtigsten Modifikationen ist dabei die Aufteilung von Befehls- und Datenspeicher gemäß der Harvard-Architektur. Einzelne Elemente der Harvard-Architektur fließen seit den 1980er Jahren verstärkt wieder in die üblichen Von-Neumann-Rechner ein, da eine klarere Trennung von Befehlen und Daten die Betriebssicherheit erfahrungsgemäß deutlich erhöht. Besonders die gefürchteten Pufferüberläufe, die für die meisten Sicherheitslücken in modernen Systemen verantwortlich sind, werden bei stärkerer Trennung von Befehlen und Daten besser beherrschbar.

1.1.4 Befehlsablauf im Von Neumann Rechner

Die Durchführung eines Befehls im Von Neumann Rechner ist in folgende Teilschritte zerlegt:

Befehl holen In das Befehls-Register, das zusammen mit Steuerwerk und Rechenwerk (ALU: Arithmetic Logical Unit) die CPU darstellt, wird aus RAM- oder ROM-Speicher der nächste zu bearbeitende Befehl geholt.

Entschlüsseln Der Befehl wird durch das Steuerwerk in Schaltinstruktionen für das Rechenwerk aufgelöst (übersetzt).

Operanden holen Aus RAM oder ROM werden nun die Operanden geholt: die Werte, die durch den Befehl verändert werden sollen bzw. die als Parameter verwendet werden.

Durchführen Die Operation wird vom Rechenwerk ausgeführt. An dieser Stelle wird, so vom Programm gewünscht, auch der Befehlszähler verändert (Sprungbefehl).

Befehlszähler aktualisieren Der Befehlszähler wird erhöht. Beim „Durchführen" kann der Befehlszähler wieder verändert werden (Sprungbefehl).

Jetzt kann der Zyklus von vorn beginnen.

1.1.5 Befehlsablauf im Harvard Rechner

Im Harvard Rechner können gleichzeitig Befehle und Operanden geholt werden.

1.1.6 Zahlensysteme

Ein Zahlensystem wird zur Darstellung von Zahlen verwendet. Eine Zahl wird dabei nach den Regeln des Zahlensystems als Folge von Ziffern dargestellt. Man unterscheidet im Wesentlichen zwischen Additionssystemen und Stellenwertsystemen (Positionssystemen). In einem Additionssystem wird eine Zahl als Summe der Werte ihrer Ziffern dargestellt. Ein Beispiel sind die römisch-etruskischen Zahlen mit den Ziffern

I (1), V (5), X (10), L (50), C (100), D (500) und M (1000)

Die Ziffern werden mit abnehmender Wertigkeit geschrieben und addiert. 2002 wird zum Beispiel als MMII dargestellt. Da solche Zahlen sehr lang werden können, wurde das System später modifiziert, so dass Ziffern nur dreimal hintereinander auftreten dürfen. Eine kleinere Ziffer die vor einer größeren steht, wird von dieser abgezogen. So wurde VIIII zu IX. Abweichend von dieser Regel (und dem heute weit verbreiteten Gebrauch) wurde die 4 von den Römern nicht als IV, sondern als IIII geschrieben (auf Uhren ist diese Schreibweise bis heute üblich), da die Zeichenfolge IV als Kürzel für den höchsten Gott Jupiter reserviert war. Das römische Zahlensystem wurde bis ins 15. Jahrhundert allgemein in Europa verwendet. Ein großer Nachteil ist vor allem, dass sich keine 0 darstellen lässt. Das Unärsystem wird gerne auf Bierdeckeln eingesetzt (die Zahl n dezimal wird durch n Striche dargestellt). Das Unärsystem braucht für die Darstellung großer Zahlen jedoch viel Platz.

Stellenwertsysteme

In einem Stellenwertsystem (Positionssystem) impliziert die Stelle (Position) den Wert der jeweiligen Ziffer. Die 'niederwertigste' Position steht dabei im Allgemeinen rechts.

Andere Zahlensysteme

Das Duodezimalsystem hat als Basis die 12. Wir finden es in der Rechnung mit Dutzend und Gros und im angelsächsischen Maßsystem (1 Shilling = 12 Pence). Auch die Stundenzählung hat in diesem System ihren Ursprung. In vielen polytheistischen Religionen gab es 12 Hauptgötter, die sich z. B. im alten Ägypten in 3 oberste Götter und 3*3 zugeordnete Götter aufteilten. (Die 3 galt als perfekte Zahl, vgl. Dreifaltigkeit). Die Babylonier benutzten ein Zahlensystem mit einer Basis von 60. Bei einigen Naturvölkern sind auch noch Zahlensysteme zu anderen Basen gefunden worden. Vergleichsweise weit verbreitet ist das System zur Basis 20. Bei diesen Völkern werden in

der Regel zum Zählen neben den Fingern auch noch die Füße verwendet. Das analog zu erwartende Zahlensystem zur Basis fünf bei Völkern, die nur eine Hand zum Zählen benutzen, wurde aber bisher nirgendwo entdeckt. In Neuseeland war hingegen das System zur Basis 11 üblich und einige Völker benutzen das System zur Basis 18.

Übung Wie viele Sekunden sind seit Christi Geburt Mitternacht bis zu Beginn der Vorlesung vergangen (ohne Schaltjahre/ Sekunden)? Lösung:

$$\begin{aligned} t/sec = J \cdot 365 \cdot 24 \cdot 60 \cdot 60 \\ +(31 + 28 + dat) \cdot 24 \cdot 60 \cdot 60 \\ +h \cdot 60 \cdot 60 \\ +m \cdot 60 \end{aligned} \tag{1.1}$$

Dass wir bei unserer Zeitrechnung so merkwürdige Gewichtung den einzelnen Stellen haben, liegt an den astronomischen Gegebenheiten.

Zahlendarstellung durch Polynome

In Stellenwertsystemen stellt sich das Ganze wesentlich einfacher dar. Ein Stellenwertsystem hat eine Basis b, sowie Ziffern s, die von 0 bis b-1 laufen. Die Ziffernposition hat einen Wert, der einer Potenz der Basis entspricht. Für den Wert W einer $n+1$- stelligen Zahl in einem Stellenwertsystem mit Basis b und den Stellen $s_0, s_1, \cdots, s_n$ gilt dann:

$$W = \sum_{i=0}^{n} b^i \cdot s_i \tag{1.2}$$

Diesen Ausdruck kennen wir als Polynom, auch wenn üblicherweise die Basis mit x statt b und die Koeffizienten mit a_i statt s_i bezeichnet werden.

Das bekannteste und verbreitetste Zahlensystem ist das Dezimalsystem (oder 10er-System) mit Basis 10, und den Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8 und 9. In ihm entspricht jeder Ziffernposition eine Zehnerpotenz. Beispielsweise bedeutet die Ziffernfolge 6857, dass
die 7 mit $10^0 = 1$,
die 5 mit $10^1 = 10$,
die 8 mit $10^2 = 100$ und
die 6 mit $10^3 = 1000$
gewichtet wird, so dass man 6000 + 800 + 50 + 7 erhält.

Das Dezimalsystem stammt ursprünglich aus Indien. Der persische Mathematiker Muhammad ibn Musa al-Chwarizmi verwendete es in seinem Arithmetikbuch, das er im 8. Jahrhundert schrieb. Bereits im 10. Jahrhundert wurde das System in Europa eingeführt, damals noch ohne Null. Durchsetzen konnte es sich in Europa jedoch erst im 12. Jahrhundert mit der Übersetzung des genannten Arithmetikbuchs ins Lateinische.

Übung $W(102030_5)$, $W(102030_7)$

Horner Schema

Für die Umrechnung zwischen den einzelnen Zahlensystemen ist das Hornerschema sehr nützlich. Dabei wird die Darstellung des Polynoms aus Gleichung 5.4 ersetzt durch eine Darstellung ohne Potenzen:

$$W = \sum_{i=0}^{n} b^i \cdot s_i = s_0 + b(s_1 + b(s_2 + \cdots b(s_{n-2} + b(s_{n-1} + b \cdot s_n)) \cdots)) \tag{1.3}$$

Übung Bitte zeigen Sie für N=4, dass beide Darstellungen identisch sind.

Die Darstellung der Zahl im Hornerschema hat einen großen Vorteil für die Betrachtung von Zahlen in Stellenwertsystemen: da sowohl b, als auch alle s_i natürliche Zahlen sind und $b > s_i$, kann man die s_i sukzessive durch Division mit Rest ermitteln:

$$\begin{aligned} W/b &= \underbrace{s_1 + b(s_2 + \cdots b(s_{n-2} + b(s_{n-1} + b \cdot s_n))\cdots)}_{W_1} \text{ Rest } s_0 \\ W_1/b &= \underbrace{s_2 + \cdots b(s_{n-2} + b(s_{n-1} + b \cdot s_n)\cdots)}_{W_2} \text{ Rest } s_1 \\ &\cdots \\ W_{n-2}/b &= \underbrace{s_n}_{W_{n-1}} \text{ Rest } s_{n-1} \\ W_{n-1}/b &= 0 \text{ Rest } s_n \end{aligned} \tag{1.4}$$

Wir können nun recht einfach Zahlen aus dem Dezimalsystem in andere Zahlensysteme umrechnen. Wir dividieren die Zahl aus dem Dezimalsystem so lange durch die Basis des anderen Systems, bis wir fertig sind und notieren dabei den Rest. Beispiel: 2595_{10} umrechnen in das neuseeländische Zahlensystem mit Basis 11 (die Ziffer mit der Wertigkeit 10 nennen wir A):

$$\begin{aligned} 2595_{10}/11 &= 235_{10} \text{ Rest } 10_{10} && 10_{10} = \mathrm{A}_{11} \\ 235_{10}/11 &= 21_{10} \text{ Rest } 4 \\ 21_{10}/11 &= 1_{10} \text{ Rest } 10_{10} && 10_{10} = \mathrm{A}_{11} \\ 1_{10}/11 &= 0 \text{ Rest } 1 \\ 2595_{10} &= 1\mathrm{A}_{11}4\mathrm{A}_{11} \end{aligned} \tag{1.5}$$

1.1.7 Zahlensysteme mit 2^n

Dualsystem

Im 17. Jahrhundert führte der Mathematiker Gottfried Wilhelm Leibniz mit der Dyadik das Dualsystem (ein binäres Zahlensystem), also das Stellenwertsystem mit der Basis 2 und den Ziffern 0 und 1, ein. Dieses wird vor allem in der Informationstechnik verwendet, da in diesen System viele Berechnungen einfacher auszuführen sind als in anderen Systemen. Die Werte der Stellen sind dann

$2^0 = 1$,
$2^1 = 2$,
$2^2 = 4$,
$2^3 = 8$,
$2^4 = 16$ und
$2^5 = 32$ u.s.w.

Dieses Zahlensystem ist sehr wichtig in der Informatik, da sich mit logischen Schaltungen direkt arithmetische Verknüpfungen realisieren lassen. Wenn wir beispielsweise zwei einstellige Zahlen a und b zu der einstelligen Summe s und den Übertrag C addieren möchten, erreichen wir es durch die logischen Operationen:

$$\begin{aligned} s = a + b &= (a \wedge \bar{b}) \vee (\bar{a} \wedge b) \\ C &= a \wedge b \end{aligned} \tag{1.6}$$

Diese Thematik wird in der Digitaltechnik vertieft.

Wir können uns überzeugen, dass das Addieren genauso funktioniert, wie wir es im Dezimalsystem gewohnt sind: Man addiert zwei Zahlen stellenweise und berücksichtigt in der darauffolgenden

Addition den Übertrag der Vorhergehenden. Bei zwei n- stelligen Dualzahlen A mit den Ziffern a_i und B mit den Ziffern b_i gilt für die Ziffern s_i der Summe S (mit Zuhilfenahme der Überträge C_i):

$$\begin{aligned} s_0 &= (a_0 \wedge \bar{b_0}) \vee (\bar{a_0} \wedge b_0) \\ C_0 &= a_0 \wedge b_0 \\ s_i &= (a_i + b_i) \\ C_i &= ((a_i \vee b_i) \wedge c_{i-1}) \vee (a_i \wedge b_i) \quad i > 0 \end{aligned} \tag{1.7}$$

Der Übertrag der letzten Addition wird im Allgemeinen separat in dem sogenannten „Carry" Bit gespeichert. Beispiel: $00101110_2 + 01011111_2$

	0	0	1	0	1	1	1	0
+	0	1	0	1	1	1	1	1
=	1	0	0	0	1	1	0	1

Übung :
$11110000_2 + 00001111_2$
$01010101_2 + 01010111_2$
Die Subtraktion werden wir im Zusammenhang mit den negativen Zahlen besprechen. Multiplikation und Division werden in der Digitaltechnik behandelt.

Oktalsystem

Da große binäre Zahlen unübersichtlich lang sind, werden zur Darstellung oft Oktalzahlen verwendet, die mit der Basis 8 arbeiten. Oktale Zahlen und binäre Zahlen lassen sich leicht ineinander umwandeln, da 3 Stellen einer binären Zahl gerade einer Stelle einer oktalen Zahl entsprechen. Auch hier können wir uns überzeugen, dass das Addieren genauso funktioniert, wie wir es im Dezimalsystem gewohnt sind: Man addiert zwei Zahlen stellenweise und berücksichtigt in der darauffolgenden Addition den Übertrag der Vorhergehenden. Beispiel: $3456_8 + 2345_8$

	3	4	5	6
+	2	3	4	5
=	6	0	2	3

Übung :
$7701_8 + 0077_8$
$2776_8 + 1002_8$

Hexadezimalsystem

Häufiger als das Oktalsystem wird das Hexadezimalsystem verwendet, das mit der Basis 16 (und den Ziffern 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E und F) arbeitet. Auch hexadezimale Zahlen und binäre Zahlen lassen sich leicht ineinander umwandeln, da 4 Stellen einer binären Zahl gerade einer Stelle einer hexadezimalen Zahl entsprechen. In der Computertechnik werden das Binärsystem, das Oktalsystem und das Hexadezimalsystem verwendet. Auch hier können wir uns überzeugen, dass das Addieren genauso funktioniert, wie wir es im Dezimalsystem gewohnt sind: Man addiert zwei Zahlen stellenweise und berücksichtigt in der darauffolgenden Addition den Übertrag der Vorhergehenden. Beispiel: $BCDE_{16} + 2345_{16}$

	B	C	D	E
+	2	3	4	5
=	E	0	2	3

Übung :
$FF01_{16} + 00FF_{16}$
$2FFE_{16} + 1002_{16}$

Umrechnungen

Hier ein Beispiel zur Umwandlung:

$$
\begin{aligned}
0_{10} &= 00\ 000\ 000_2 &&= 000_8 &&= 0000\ 0000_2 = 0_{16}\\
1_{10} &= 00\ 000\ 001_2 &&= 001_8 &&= 0000\ 0001_2 = 1_{16}\\
2_{10} &= 00\ 000\ 010_2 &&= 002_8 &&= 0000\ 0010_2 = 2_{16}\\
3_{10} &= 00\ 000\ 011_2 &&= 003_8 &&= 0000\ 0011_2 = 3_{16}\\
4_{10} &= 00\ 000\ 100_2 &&= 004_8 &&= 0000\ 0100_2 = 4_{16}\\
5_{10} &= 00\ 000\ 101_2 &&= 005_8 &&= 0000\ 0101_2 = 5_{16}\\
6_{10} &= 00\ 000\ 110_2 &&= 006_8 &&= 0000\ 0110_2 = 6_{16}\\
7_{10} &= 00\ 000\ 111_2 &&= 007_8 &&= 0000\ 0111_2 = 7_{16}\\
8_{10} &= 00\ 001\ 000_2 &&= 010_8 &&= 0000\ 1000_2 = 8_{16}\\
9_{10} &= 00\ 001\ 001_2 &&= 011_8 &&= 0000\ 1001_2 = 9_{16}\\
10_{10} &= 00\ 001\ 010_2 &&= 012_8 &&= 0000\ 1010_2 = A_{16}\\
11_{10} &= 00\ 001\ 011_2 &&= 013_8 &&= 0000\ 1011_2 = B_{16}\\
12_{10} &= 00\ 001\ 100_2 &&= 014_8 &&= 0000\ 1100_2 = C_{16}\\
13_{10} &= 00\ 001\ 101_2 &&= 015_8 &&= 0000\ 1101_2 = D_{16}\\
14_{10} &= 00\ 001\ 110_2 &&= 016_8 &&= 0000\ 1110_2 = E_{16}\\
15_{10} &= 00\ 001\ 111_2 &&= 017_8 &&= 0000\ 1111_2 = F_{16}\\
16_{10} &= 00\ 010\ 000_2 &&= 020_8 &&= 0001\ 0000_2 = 10_{16}\\
&\dots\\
62_{10} &= 00\ 111\ 110_2 &&= 076_8 &&= 0011\ 1110_2 = 3E_{16}\\
&\dots\\
132_{10} &= 10\ 000\ 100_2 &&= 204_8 &&= 1000\ 0100_2 = 84_{16}\\
&\dots\\
255_{10} &= 11\ 111\ 111_2 &&= 377_8 &&= 1111\ 1111_2 = FF_{16}
\end{aligned}
\tag{1.8}
$$

Übung Bitte versuchen Sie sich klar zumachen, wie man eine binäre Multiplikation auf Addition und Verschiebung von Dualzahlen zurückführen kann.

1.2 Der Arm Cortex M3

1.2.1 Historie

Die 1978 gegründete britische Firma Acron , stellte 1985 eine CPU vor, die sehr einfach aufgebaut und stromsparend war. Sie hatte einen sehr eingeschränkten Befehlssatz (Reduced Instruction Set Computer = RISC). Diese CPU nannten sie „Acron RISC Machine" oder abgekürzt ARM1. 1990 wurde die Firma Advanced RISC Machines zur Vermarktung der ARM Architektur auf Basis eines Lizenzmodells gegründet. Die Gründungsmitglieder waren:

APPLE hatte Interesse an einem besonders stromsparenden Prozessors (1993 Newton PDA)

Acron war an der Weiterentwicklung seiner Workstations interessiert

VLSI (später Philips, heute NXP) wollte primär Prozessoren und Microkontrollerchips fertigen und vermarkten.

1993 wurde der ARM7TDMI vorgestellt, der mit 40-180 MHz betrieben wird und eine dreistufige Pipeline und einen gemeinsamen Datenbus für Instruktionen und Daten (Von Neumann

Architektur) besitzt. Er wird millionenfach in Mobiltelefonen und Spielkonsolen eingesetzt. Die diversen Lizenznehmer entwickelten ARM-Varianten, die sich durch Performance aber auch durch die geringe Leistungsaufnahme auszeichneten. So gab es mit dem StrongARM, entwickelt von Digital Equipment, eine Version die mit 233 MHz getaktet wurde und nur 1 W benötigte. Lizenznehmer sind Analog Devices, Apple, Atmel, Conexant, Freescale (ehemals Motorola), HTC Corporation, HP, IBM, Infineon, Intel (XScale), Luminary Micro, Motorola, NEC, NetSilicon, Nintendo, NXP (ehemals Philips), Oki, Palm, Samsung, Sony, STMicroelectronics, Texas Instruments, Toshiba und weitere.

Der gemeinsam mit DEC entwickelte ARM StrongARM war die erste Abspaltung der ARM-Architektur, die 1995 als SA-110 im Newton 2000 durch einen Stromsparmodus für hohe Akkulaufzeiten sorgte. Der Nachfolger SA-1100 (1997) war mit einer LCD-Schnittstelle, einer MCP-Audio/Touchscreen-Schnittstelle, PCMCIA-Unterstützung, IrDA, USB und DMA-Controller eines der ersten System-on-a-Chip.

Der ARM9 (ARMv5 1997) ist eine Weiterentwicklung der StrongARM- und ARM8-Architektur. Der wesentliche Unterschied des ARM9 gegenüber dem ARM7 ist je ein getrennter Bus für Instruktionen und Daten (Harvard-Architektur). Meist werden diese an separate Caches für Daten und Instruktionen angeschlossen. Außerdem hat der ARM9 eine fünfstufige Pipeline und kann so höhere Taktraten erreichen und weist bessere CPI-Werte (Cycles per Instruction) auf. Wird der ARM9 ohne Caches an einem externen Speicher mit nur einem Datenbus betrieben, schrumpft der Geschwindigkeitsvorteil gegenüber der ARM7-Architektur wegen häufiger Pipeline-Stalls mit einer höheren Penalty durch die längere Pipeline. Ohne Cache kann in einem solchen ungünstigen Szenario ein ARM7 aufgrund seiner kürzeren Pipeline trotz eines deutlich niedrigeren Taktes schneller sein. Allerdings sollte dieser Fall in realen Systemen nicht auftreten, da ein ARM9 teurer ist und nur wegen seiner besseren Performance ausgewählt wird. Kommt es eher auf die Kosten an, so spart man sinnvollerweise nicht am Cache, sondern verwendet einen ARM7.

Die ebenfalls auf dem ARMv5 basierenden XScale-Prozessoren von Intel (802xx, PXA25x, XA263, PXA26x, PXA27x, PXA3xx) sind mit einer Taktfrequenz bis zu 1,250 GHz verfügbar und finden sich in vielen mobilen Geräten (Palm Tungsten, Sony Clié). ARMv6 (2002)

Die ARMv6-Architektur umfasst die Familien ARM11, ARM Cortex-M0, ARM Cortex-M0+ und ARM Cortex-M1. ARMv7 (2004)

Die ARMv7-Architektur ist ? als ARM Cortex-M3, ARM Cortex-M4, ARM Cortex-A und ARM Cortex-R ? in sehr vielen Mobilgeräten nahezu allgegenwärtig, etwa in Prozessoren/SoCs von Apple (A4, A5), nVidia (Tegra), Samsung (Exynos) und Texas Instruments (OMAP). Außerdem wurde sie auch in herstellereigene Designs von Apple (A6), Qualcomm (Snapdragon), und Marvell (MMP, Armada XP) umgesetzt.

1.2.2 Aufbau des Kerns der Cortex M3

Registermodell

Die CPU Architektur

```
Befehl  Zustand   Zustand   Zustand   Zustand   Zustand
  N       Fetch    Decode    Execute   Finished  Finished
  N+1              Fetch     Decode    Execute   Finished
  N+2                        Fetch     Decode    Execute
Zeitachse   ------------------------------------------>
```

Man muss als Programmierer dieses „Pipelining" berücksichtigen, wenn man relativ zum Befehlszähler Speicher adressieren möchte: Während der Befehl `N` ausgeführt wird steht in `r15` schon eine um 8 höhere Adresse. Eine Konsequenz dieser Technik ist, dass jeder Befehl möglichst in einem Zyklus abgearbeitet werden muss.

1.2.3 Das Current Program Status Register

Zusätzlich zu den direkt veränderbaren Registern gibt es das Status-Register (CPSR, Current Program Status Register), das die Statusbits und andere Informationen, wie z. B. den momentanen Ausführungsmodus, enthält. Die zunächst hier interessierenden Statusbits sind zum einen das Carry Flag (siehe oben binäre Additon), das gesetzt wird, wenn bei der Addition von vorzeichenlosen Zahlen der Zahlenbereich überschritten wurde oder wenn beim Subtraktionsbefehl eine kleinere (vorzeichenlose) Zahl von einer größeren Zahl abgezogen wurde. Des weiteren ist im Folgenden das Zero Flag von Interesse, das gesetzt wird, wenn das Ergebnis eines Datenverabeitungsbefehls Null war. Zum Setzen dieser Flags muss bei den Datenverarbeitungsbefehlen das Bit Nummer 20 (s.U.) gesetzt sein. Der Inhalt dieser Flags kann für Bedingungen im Ablauf des Programms genutzt werden. Näheres zur Verwendung dieses Registers findet sich in Abschnitt 2.2.3.

1.2.4 Speicherorganisation beim CORTEX M3

Fast alle Eigenschaften der CORTEX CPU beziehen sich auf 32 Bit Größen, so sind die Register 32 Bit breit, die Datenverarbeitungsbefehle verrechnen 2 32 Bit Operanden zu einem 32 Bit Ergebnis und 32 Bit Werte werden für Adressen benutzt. So kann der CORTEX 2^{32}, also 4.294.967.296 Speichereinheiten verwalten. Man sagt 4 Giga. In SI Systemen bezeichnet man Giga als 10^9, in der Digitaltechnik nennt man 1 Giga 2^{30} was 7 % mehr ist. Bei der Speicherverwaltung ist die kleinste Einheit das Byte (8 Bit). In Texten werden Buchstaben in 8 bit Worten verschlüsselt. Um mit dem CORTEX Textverarbeitung betreiben zu können, müssen also auch Bytes verarbeitet werden können.

Adressbus

Auf den Adressbus legt die CORTEX CPU die in 32 Bit binär verschlüsselte Adresse des Speicherelements, das sie ansprechen möchte. Die Speicherbausteine vergleichen die am Adressbus anliegenden Signale mit ihrer eigenen Adresse. Wenn die Adressen übereinstimmen, dann ist dieses Speicherelement angesprochen.

Steuerbus

Auf dem Steuerbus gibt es neben den Signalen für die Synchronisation auch ein Signal, das steuert, ob der Inhalt des angesprochenen Elements in die CPU kopiert werden soll oder ob ein Datum von der CPU in das angesprochene Element kopiert werden soll.

Datenbus

Je nach Transferrichtung legt die CPU oder das angesprochene Speicherelement das zu transferierende Datum bitweise auf die Datenleitungen. Das können 8, 16 oder 32 Bit sein.

Big und Little Endian

Wenn eine 32 Bit große Zahl in vier Byte gespeichert ist, muss festgelegt werden in welcher Reihenfolge die vier Byte gespeichert werden. Von den 24 möglichen Reihenfolgen haben sich deren zwei durchgesetzt. Nehmen wir an, ein zweiunddreißig Bit Wort besteht aus den Bits $d_{31}d_{30}d_{29}\cdots d_2d_1d_0$, wobei d_{31} das höchstwertigste Bit ist und d_0 das Niederwertigste. Diese 32 Bit werden wie folgt auf 4 Bytes aufgeteilt: `b0`=$d_7d_6d_5d_4d_3d_2d_1d_0$, `b1`= $d_{15}d_{14}d_{13}d_{12}d_{11}d_{10}d_9d_8$, `b2`=$d_{23}d_{22}d_{12}d_{20}d_{19}d_{18}d_{17}d_{16}$ und `b3`=$d_{31}d_{30}d_{29}d_{28}d_{27}d_{26}d_{25}d_{24}$ 32 Bit Worte werden im CORTEX immer an durch 4 teilbaren Adressen gespeichert, die zwei niederwertigsten Bits der Adresse sind also immer Null, eine Wortadresse hat also immer die Form `xx--xx00`. In dem sogenannten „Little-endian storage" -Muster werden auf dieser und den drei folgenden Adressen diese 4 Bytes wie folgt angeordnet:

```
Adresse:  xx--xx00 xx--xx01 xx--xx10 xx--xx11
Inhalt      b0       b1       b2       b3
```

In dem sogenannten „Big-endian storage" Muster werden auf dieser und den drei folgenden Adressen diese 4 Bytes wie folgt angeordnet:

```
Adresse:  xx--xx00 xx--xx01 xx--xx10 xx--xx11
Inhalt      b3       b2       b1       b0
```

Beispiel: Die Dezimalzahl 2952665384_{10} soll an der Adresse 256(=100 hex) gespeichert werden. Diese Zahl ist in Hexadezimal $affe1928_{16}$ Im „Little-endian storage" Muster werden diese 4 Bytes wie folgt angeordnet:

```
Adresse(hex):  100 101 102 103
Inhalt (hex)    28  19  fe  af
```

Im „Big-endian storage" Muster werden diese 4 Bytes wie folgt angeordnet:

```
Adresse(hex):  100 101 102 103
Inhalt (hex)    af  fe  19  28
```

Kapitel 2

Assembler Programmierung

2.1 IDE

Unter einer IDE (Integrated Development Environment) versteht man ein Programmpaket, das die Entwicklung eines Softwareprodukts von der Idee bis zum Integrationstest unterstützt. Die IDE, die im TI Labor für die Ausbildung benutzt wird ist das Paket μVision der Firma Keil. Ein Überblick über die Kompontenten dieses Paketes ist in Abbildung 2.1 gegeben.

2.1.1 Lader

Heutzutage benutzt man dafür sogenannte Ladeprogramme, die ausführbare Programme in den Speicher des Rechners schreibt, der dieses Programm ausführen soll. Bei der Keil IDE übernimmt das der USB JTAG Adapter. (Joint Test Action Group bezeichnet den IEEE-Standard 1149.1).

2.1.2 Debugger

Der Debugger ist ein Programm zum Testen und Fehlerlokalisieren. In der professionellen Softwareentwicklung ist er eines der wichtigsten Werkzeuge. Bei der Keil Entwicklungsumgebung kann der Debugger wahlweise die zu entwickelnden Programme zum Test über den JTAG Adapter in den Seicher des Zielsystems laden oder das Ziesystem simulieren.

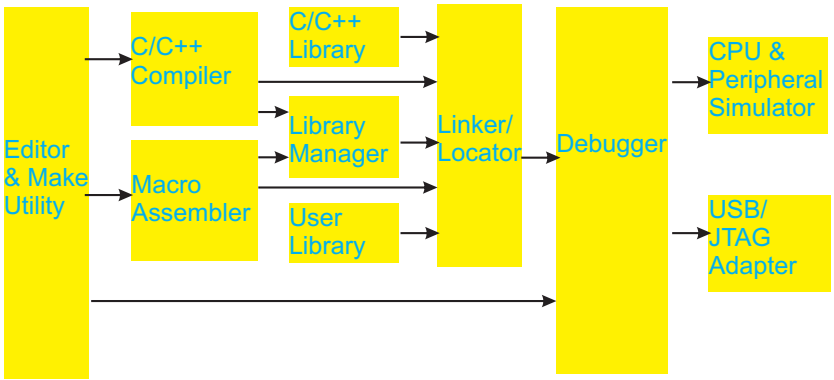


Abbildung 2.1: Toolchain der μVision IDE

Abbildung 2.2: Bedienfeld der PDP8

Einzel-Schritt-Modus

Im Single Step Modus führt die CPU nach Betätigen einer Taste einen Befehl aus und zeigt den Zustand des Rechners nach diesem Befehl an.

Breakpoints

Möchte man einen kompletten Programmabschnitt testen, dann kann man einen Befehl am Ende des Abschnitts markieren. Betätigt man dann die Abspieltaste, werden alle Befehle des Abschnitts ausgeführt und der Debugger zeigt den Zustand des Rechners nach Ausführen der Befehle des Abschnitts an.

Register Monitor

Im Register Monitor wird der Inhalt der Register der CPU angezeigt.

Variablen Monitor

Zeigt Variablen an. Mehr kommt später.

Speicher Monitor

Im Speicher Monitor kann man sich bestimmte Speicherbereiche anzeigen lassen. Dazu muss man die Adresse eingeben, an welcher man den Speicherinhalt sehen möchte.

2.1.3 Assembler

2.1.4 Weitere Bestandteile der IDE

Zusätzlich gehören zu einer IDE noch Compiler, Linker und Diagramm Editor was später genauer besprochen wird.

2.2 Programmierung des Cortex mit dem KEIL Assembler

2.2.1 Der KEIL Assembler

Der KEIL Assembler, ein Teil der KEIL Toolchain, ist der Assembler, der im TI Labor benutzt wird, um Cortex Assembler Quellprogramme in sogenannte binäre Object Files zu codieren.

Syntax

Ein Assembler Programm ist eine Textdatei mit einer Zeilestruktur. Jede Zeile kann entweder eine Assembler Direktive oder eine Prozessor Instruktion sein.

Prozessor Instruktion Instruktionen für den Prozessor werden als Mnemonics in die Textdatei geschrieben. Ein Mnemonic ist ein Kürzel, das sich besser merken lässt, als der Zahlenwert des Opcodes. So lässt sich das Kürzel `mov r4,80h` besser merken als `F04F0480` für den Befehl die Hexadezimalzahl 80 in das `r4` Register zu kopieren. Zeilen mit Prozessor Instruktionen haben folgende Syntax:

```
MARKE OPCODE OPERAND KOMMENTAR
```

Die Marke und der Kommentar können weggelassen werden. Das Format der Operanden hängt von den jeweiligen Befehlen ab.

Assembler Direktiven Kommandos, die den Ablauf des Assemblierers steuern oder Speicherbereiche definieren, nennt man Assembler Direktiven. Zeilen mit Assembler Direktiven haben folgende Syntax:

```
MARKE DIREKTIVE PARAMETER KOMMENTAR
```

`http://www.keil.com/support/man/docs/armasmref/armasmref_caccehia.htm` Die Marke und der Kommentar können weggelassen werden. Das Format der Parameter hängt von den jeweiligen Direktive ab.

2.2.2 Einige Assembler Direktiven aus Sicht des Programmierers

Speicherinitialisierung

Zur Initialisierung des Speichers stellt der KEIL Assembler eine Reihe von Direktiven bereit.

AREA Die `AREA` Direktive kommuniziert dem Assembler, welchen Speicherbereich die kommenden Anweisungen betreffen. Man unterscheidet den Speicherbereich, der für die Daten gedacht ist (`DATA`), und den, der für die Befehle gedacht ist (`CODE`). Diesen Speicherbereichen kann man Namen geben und angeben, wie der Anfang dieses Bereichs platziert werden soll: Beispiele:
`AREA MyData, DATA, align = 2`
Durch diese Direktive wird der Assembler angewiesen, die folgenden Anweisungen in den Datenspeicher abzulegen. Da es sich beim Cortex M3 um einen Harvard Rechner handelt, sollten hier keine Befehle abgelegt werden. Der Speicherbereich beginnt bei einer durch 4 (2^2) teilbaren Adresse. Bei unserem Entwicklungssystem beginnt der RAM Bereich für die Daten bei der Adresse `20 000 000 h` .
`AREA MyCode, CODE, align = 2`
Durch diese Direktive wird der Assembler angewiesen, die folgenden Anweisungen in den Programmspeicher abzulegen. Obwohl es sich beim Cortex M3 um einen Harvard Rechner handelt, können hier auch Daten abgelegt werden. Der Speicherbereich beginnt bei einer durch 4 (2^2) teilbaren Adresse. Bei unserem Entwicklungssystem beginnt das ROM (bzw Flash RAM) für unsere Befehle bei der Adresse `8 000 000 h` .

Speicher reservieren und initialisieren Die Direktiven `DCB`, `DCW` , `DCD` und `FILL` Speicher zu reservieren und zu initialisieren. Beipiele:
`MeinByte DCB 0xba`
Mit dieser Direktive wird in der zuvor vereinbarten AREA Platz für ein Byte reserviert und dieser Speicher mit dem Byte `0xba` initialisiert. Die Adresse dieser Speicherzelle bekommt den symbolischen Namen `MeinByte`
`MeinWort DCW 0xaffe`

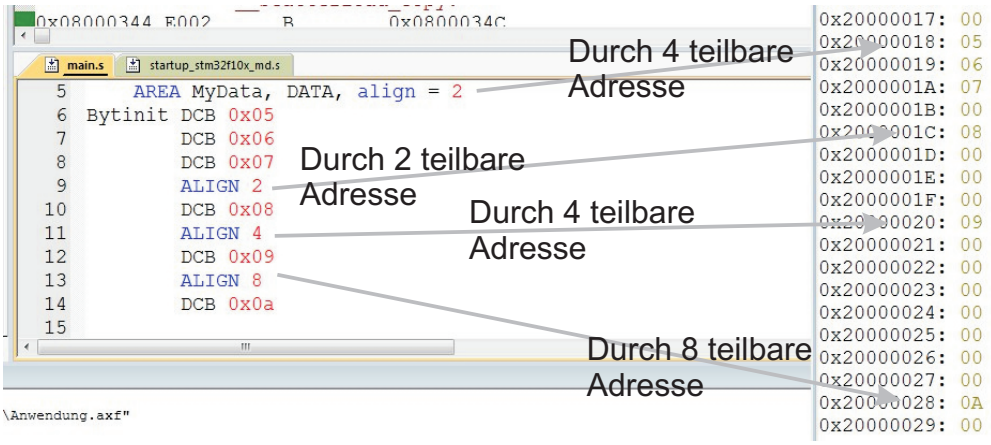


Abbildung 2.3: Zusammenspiel von ALIGN und DCB

Mit dieser Direktive wird in der zuvor vereinbarten AREA Platz für zwei Byte reserviert und dieser Speicher mit den Bytes `0xaf` und `0xfe` initialisiert. Das niederwertige Byte (`0xfe`) kommt dabei an die kleinere Adresse (little Endian Format) Die Adresse dieser Speicherzelle bekommt den symbolischen Namen `MeinWort`

```
MeinDwort  DCD 0xaffedead
```

Mit dieser Direktive wird in der zuvor vereinbarten AREA Platz für vier Byte reserviert und dieser Speicher mit den Bytes `0xaf`, `0xfe` , `0xde` und `0xad` initialisiert. Das niederwertige Byte (`0xad`) kommt dabei an die kleinere Adresse (little Endian Format) Die Adresse dieser Speicherzelle bekommt den symbolischen Namen `MeinDwort`

```
Imem FILL 28,0xadde,4
```

Mit dieser Direktive wird in der zuvor vereinbarten AREA Platz für 28 Byte reserviert und dieser Speicher 7 mal mit den Bytes `0xde`, `0xad` , `0x0` und `0x0` initialisiert. Das niederwertige Byte (`0xde`) kommt dabei an die kleinere Adresse (little Endian Format) Die Adresse dieser Speicherzelle bekommt den symbolischen Namen `Imem`. De erste Parameter der `FILL` Direktive muß ein vielfaches des dritten Parameters sein.

Speicher reservieren und mit Nullen initialisieren Die Direktive `SPACE` sorgt dafür, dass Speicher reserviert und mit Nullen initialisiert wird. Beipiele:

```
MeinPlatz  SPACE 20
```

Mit dieser Direktive wird in der zuvor vereinbarten AREA Platz für 20 Byte reserviert und dieser Speicher mit Nullen initialisiert. Die Adresse dieser Speicherzellen bekommt den symbolischen Namen `MeinPlatz`.

Plazieren Mit der Direktive `ALIGN` kann gesteuert werden, ob die nachfolgenden Direktiven den Speicher auf Byte, Wort (16 Bit) oder Double Wort (32 Bit) legen soll.

```
ALIGN 2
```

Mit dieser Direktive wird in der im folgenden vereinbarten Speicher auf Wortgrenzen gelegt. Das Zusammenspiel dieser Direktiven ist in Abb. **??** dargestellt.

Informationen für den Linker und Debugger

Die Direktive `GLOBAL` sorgt dafür, dass eine Marke (hinter der sich der symbolische Name einer Adresse verbirgt) global bekannt gemacht wird. So kann der Linker externe Referenzen auf diese Marke mit der richtigen Adresse absättigen.
Die Direktiven `PROC` und `ENDP` sorgen dafür, dass die Aufrufstruktur von Unterprogrammen im Debugger ähnlich komfortabel analysiert werden kann, wie in Hochsprachen. Beipiel:

```
GLOBAL main
main PROC
forever b forever
   ENDP
```

Hier wird `main` exportiert und als Unterprogramm markiert.

2.2.3 Der Befehlssatz des CORTEX M3

Datentransferbefehle

Datentransferbefehle sind dazu da Daten aus dem Speicher in Register zu kopieren, um sie dann mithilfe der Datenverarbeitungsbefehle zu verabeiten, aber auch um die Registerinhalte, die die Ergebnisse der Datenverarbeitungsbefehle halten im Speicher abzulegen. Sie bestehen immer zum Einen aus einem Datenregister, dessen Inhalt abgespiechert werden soll, bzw in das die Kopie des Speichers abgelegt werden soll. Es wird im Folgenden mit `Rd` bezeichnet. Zum Anderen muss angegeben werden, an welcher Adresse die Kopie des Registers gelegt werden soll, bzw. von welcher Speicheradresse eine Kopie in das Register gelegt werden soll. Diese Speicheradresse wird immer relativ zu einer Adresse berechnet, die in einem anderen Register gespeichert ist, dem Adressregister, das im Folgenden mit `Rn` bezeichnet wird. Die Syntax dieses Befehls ist:

```
<LDR|STR>{cond}{B}{T} Rd,<Adresse>
```

Dabei haben die einzelnen Kürzel folgende Bedeutung:

LDR Datum aus dem Speicher in ein Register kopieren.

STR Datum aus dem Register in den Speicher kopieren.

{cond} Wenn das optionale aus zwei Buchstaben bestehende Mnemonic angefügt ist, wird das Statusregister `CPSR` ausgewertet, um zu entscheiden, ob der Befehl ausgeführt wird (Siehe Abschnitt. 2.2.3

{B} Wenn der optionale Buchstabe `B` angefügt ist, wird bei diesem Befehl ein Byte (8 Bit) kopiert, ansonsten ein Word (32 Bit)

{T} Wenn der optionale Buchstabe `T` angefügt ist, wird bei diesem Befehl der Offset erst nach der Adressierung des Speichers auf das Basis Register addiert.

Rd Register, aus dem (mit STR) in den Speicher kopiert wird bzw. in das (mit LDR) aus dem Speicher kopiert wird.

Rn und Rm Das sind Register, aus deren Inhalt sich die Speicheradresse berechnet. Wenn es sich bei `Rn` um `r15`, also um den Programmzähler handelt, subtrahiert der Assembler 8 vom Offset. Dadurch werden die 8 Byte, um die der Programmzähler wegen des Pipelining vorraus eilt, korrigiert. Write Back sollte in diesem Fall nicht benutzt werden.

<Adresse> Das ist ein Ausdruck, der folgende Formen haben kann:

- Ein Ausdruck, der eine Adresse generiert:

  `<Ausdruck>`

Der Assembler benutzt den Programmzähler als Basisadresse und addiert auf ihn den korrigierten Offset zu der Adresse, die sich hinter dem in `<Ausdruck>` angegebenen symbolischen Namen verbirgt. Wenn der Abstand zum aktuellen Befehl zu groß ist, gibt der Assembler einen Fehler aus.

- Eine preindizierte Adress Spezifikation:

  `[Rn]`

  Hier ist der Inhalt von `Rn` benutzt um den Speicher für den einfachen Datentransfer zu adressieren. Es wir kein Offset auf `Rn` addiert.

  `[Rn,<#Ausdruck>]{!}`

  Hier wird der Inhalt von `Rn` als Basisadresse benutzt, um den Speicher für den einfachen Datentransfer zu adressieren. Es wir aber hier ein Offset von `<Ausdruck>` Bytes auf `Rn` addiert, bevor der Speicher adressiert wird. Wenn im Befehl das optionale Ausrufezeichen angehängt ist, wird die so berechnete Adresse nach der Operation in `Rn` zurückgeschrieben.

  `[Rn,{-}Rm{,<shift>}]{!}`

  Hier wird der Inhalt von `Rn` als Basisadresse benutzt, um den Speicher für den einfachen Datentransfer zu adressieren. Es wir aber hier der Inhalt von `Rm` benutzt, um den Offset zu berechnen. Das Shiftregister kann den Inhalt von `Rm` zu modifizieren, bevor dieser Wert dann auf `Rn` addiert beziehungsweise subtrahiert wird, wenn das optionale Minuszeichen eingefügt ist. Das alles geschieht, bevor der Speicher adressiert wird. Wenn im Befehl das optionale Ausrufezeichen angehängt ist, wird die so berechnete Adresse nach der Operation in `Rn` zurückgeschrieben.

- Eine postindizierte Adressspezifikation:

  `[Rn],<#expression>`

  Hier wird der Inhalt von `Rn` als Basisadresse benutzt, um den Speicher für den einfachen Datentransfer zu adressieren. Es wir aber hier ein Offset von `<Ausdruck>` Bytes auf `Rn` addiert, nachdem der Speicher adressiert wird. Die so berechnete Adresse wird nach der Operation in `Rn` zurückgeschrieben.

Beispiele

```
str r1,[r2,r4]!
```

Hier wird die Summe aus `r2` und `r4` gebildet und an der so berechneten Adresse wird der Inhalt von `r1` gespeichert. Die berechnete Adresse wird in `r2` gespeichert. Die zwei anderen beteiligten Register werden nicht verändert.

```
ldr r1,[r2,#16]
```

Auf den Inhalt von `r2` wird die 16 addiert und der Inhalt des Speichers der so berechneten Adresse wird in das Register `r1` kopiert. `r2` wird dabei nicht modifiziert.

```
ldr r1,[r2,r3,lsl #2]
```

Auf den Inhalt von `r2` wird der mit 4 multiplizierte Inhalt von `r3` addiert. Der Inhalt des Speichers der so berechneten Adresse wird in das Register `r1` kopiert. `r2` wird dabei nicht modifiziert.

```
Cmd       ldr r1,MeinWort
            ......
ende       b ende
MeinWort   .word 4711
```

Hier berechnet der Assembler den Abstand der Speicheradressen zwischen dem Befehl mit dem symbolischen Namen `Cmd` und der Speicheradresse mit dem symbolischen Namen `MeinWort`, subtrahiert davon 8 (wg. Pipeline). Dies ist der Offset, der zur Laufzeit auf den Programmzähler addiert wird, um das Speicherelement mit dem symbolischen Namen `MeinWort` zu adressieren. Danch steht die Zahl 4711 in `r1`. Die Adressen von `Cmd` und `MeinWort` dürfen sich dabei nicht um mehr als 2047 unterscheiden. Man kann auf diese Art und Weise daher nicht das SRAM des CORTEX M3 adressieren. Folgendes Beispiel soll zeigen, wie man zwei Zahlen aus dem SRAM in die Register holt, addiert und im SRAM abspeichert:

```
          AREA Mydata, DATA, align=2
Smmnd1    DCD  1234
Smmnd2    DCD  2345
Ergbns    DCD  0
          AREA Mycode CODE, readonly, align = 2
ads1      DCD Smmnd1
ads2      DCD Smmnd2
ader      DCD Ergbns
main      ldr  r0,ads1 ;  Adresse des
                       ;  1. Summanden in r0
          ldr  r1,ads2 ;  Adresse des
                       ;  2. Summanden in r1
          ldr  r2,ader ;  Adresse des
                       ;  Ergebnisses in r2
          ldr  r3,[r0] ;  1. Summanden holen
          ldr  r4,[r1] ;  2. Summanden holen
          add  r5,r3,r4
          str  r5,[r2] ;  Summe speichern
ende      b ende
          end
```

Im folgenden Beispiel werden die ersten 5 Primzahlen aus dem Speicher in `r2` kopiert, in `r3` aufsummiert und das Ergebnis gespeichert. `r0` zeigt auf die Primzahlen, `r1` auf die Speicherzelle, in die das Ergebnis kopiert wird.

```
          AREA Mydata, DATA, align=2
Prims     DCD  2,3,5,7,11,13,17,19,23,29
Ergbns    DCD  0
          AREA Mycode, CODE,  align=2
adPr      DCD Prims
ader      DCD Ergbns
main      ldr  r0,adPr    ;  Adresse der
                          ;  1. Primzahl in r0
          ldr  r1,ader    ;  Adresse des
                          ;  Ergebnisses in r1
          mov  r3,#0
          ldr  r2,[r0],#4 ;  1. Primzahl in r2
                          ;   r0 zeigt auf 2.
          add  r3,r3,r2   ;  aufsummieren
          ldr  r2,[r0],#4 ;  2. Primzahl in r2
                          ;   r0 zeigt auf 3.
          add  r3,r3,r2   ;  aufsummieren
          ldr  r2,[r0],#4 ;  3. Primzahl in r2
                          ;   r0 zeigt auf 4.
          add  r3,r3,r2   ;  aufsummieren
          ldr  r2,[r0],#4 ;  4. Primzahl in r2
```

```
                        ;   r0 zeigt auf 5.
         add  r3,r3,r2   ;  aufsummieren
         ldr  r2,[r0]    ;  5. Primzahl in r2
         str  r3,[r1]    ;  Summe speichern
ende    b ende
          end
```

Beispiel aus der Vorlesung

```
;*****************************************
; Data section, aligned on 4-byte boundery
;*****************************************

AREA MyData, DATA, align = 2
CWort DCD 0x12345678

;*****************************************
; Code section, aligned on 4-byte boundery
;*****************************************

AREA MyCode, CODE, readonly, align = 2

;-----------------------------------------
; main subroutine
;-----------------------------------------
Zeiger  DCD CWort
GLOBAL main
main PROC
ldr  r3,Zeiger ;Adresse mit dem Symbolischen Namen
                    ;CWort in Register kopieren
ldr  r2,[r3]   ;Die 32 Bit Zahl 0x12345678 in R2
ldrb r1,[r3]   ;Die 8 Bit Zahl 0x78 in R1
ldrb r6,[r3,#1]! ;Die 8 Bit Zahl 0x56 in R6
                      ; und r3 um 1 erhöhen
ldrb r7,[r3,#1]! ;Die 8 Bit Zahl 0x34 in R7
                      ; und r3 um 1 erhöhen

forever b forever ; nowhere to retun if main ends
ENDP

END
```

Programmierung des Barrelshifters

Das Verschieben von Bits ist in vielen Anwendung eine häufig benutzte Operation. In den meisten CPUs wird diese Operation in der ALU durchgeführt. Im ARM7 ist zu diesem Zweck ein „Barrelshifter" vor der ALU angebracht, so dass die Bits im 2. Operanden vor der Verarbeitung in der ALU um eine bestimmte Anzahl von Stellen verschoben werden können. Vier Arten von Verschieben werden unterschieden:

Logisch nach rechts: Die einzelnen Bits der Register werden um die Anzahl von Stellen zu den niederwertigen Stellen hin verschoben. Die niederwertigsten Bits werden überschrieben. Die höchstwertigen Bits, die verschoben wurden, werden mit Nullen überschrieben. Eine logische Verschiebung nach rechts von Hex 3f09 um 3 sieht wie folgt aus:

```
0011 1111 0000 1001 0
```

```
0001 1111 1000 0100 1
0000 1111 1100 0010 2
0000 0111 1110 0001 3
```

Das Ergebnis ist also Hex 07e1. Die Mnemotechnische Abkürzung dafür ist LSR. Der Code dafür im Befehlswort ist 00.

Logisch nach links: Die einzelnen Bits der Register werden um die Anzahl von Stellen zu den höherwertigen Stellen hin verschoben. Die höherwertigsten Bits werden überschrieben. Die niederwertigen Bits, die verschoben wurden, werden mit Nullen überschrieben. Eine logische Verschiebung nach links von Hex 3f09 um 3 sieht wie folgt aus:

```
0011 1111 0000 1001 0
0111 1110 0001 0010 1
1111 1100 0010 0100 2
1111 1000 0100 1000 3
```

Das Ergebnis ist also Hex f848. Die Mnemotechnische Abkürzung dafür ist LSL. Der Code dafür im Befehlswort ist 01.

Arithmetisch nach rechts: Arithmetisch nach rechts wirkt genauso wie logisch nach rechts, nur dass die höherwertigen Bits mit höchstwertigen Bit von der Operation überschrieben werden. Eine arithmetische Verschiebung nach rechts von Hex 3f09 um 1 sieht wie folgt aus:

```
0011 1111 0000 1001 0
0001 1111 1000 0100 1
```

Das Ergebnis ist also Hex 1f84. Eine arithmetische Verschiebung nach rechts von Hex bf09 um 1 sieht wie folgt aus:

```
1011 1111 0000 1001 0
1101 1111 1000 0100 1
```

Das Ergebnis ist also Hex df84. Die Mnemotechnische Abkürzung dafür ist ASR. Der Code dafür im Befehlswort ist 10.

Rotation nach rechts: Bei der Rotation nach rechts wird das niederwertigste Bit jeweils in das höchstwertigste Bit kopiert. Ansonsten entspicht die Rotation nach rechts den Verschiebungen nach rechts. Eine Rotation nach rechts von Hex 3f09 um vier sieht wie folgt aus:

```
0011 1111 0000 1001 0
1001 1111 1000 0100 1
0100 1111 1100 0010 2
0010 0111 1110 0001 3
1001 0011 1111 0000 4
```

Das Ergebnis ist also Hex 93f0. Die Mnemotechnische Abkürzung dafür ist ROR. Der Code dafür im Befehlswort ist 11.

Datenverarbeitung

Die Datenverabeitungsbefehle haben grundsätzlich die Syntax :

```
1.  MOV,MVN
    <opcode>{cond}{S} Rd,<Op2>

2.  CMP,CMN,TEQ,TST
    <opcode>{cond} Rn,<Op2>

3.  AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC
    <opcode>{cond}{S} Rd,Rn,<Op2>

<Op2> ::= Rm,{<shift>}|<#Ausdruck>
<shift>::=<shiftname><register>|<shiftname><#expression>
```

`<shiftname>` sind die Mnemonics aus Abschnitt 2.2.3

Beispiele

```
mov r0,#0xa1       ; r0 = 000000a1
add r0,r0,r0,lsl #8  ; r0 = 0000a1a1
add r0,r0,r0,lsl #16 ; r0 = a1a1a1a1
```

Hier wurde ein Register mit einem 32 Bit Wert gefüllt ohne den Speicher zu benutzen.

Beispiele

```
mov r1,#0            ; Produkt
mov r0,#20           ; Faktor
add r1,r1,r0         ; r1 = 20
add r1,r1,r0,lsl #1  ; r1 = 20*3=20+20*2
add r1,r1,r0,lsl #2  ; r1 = 20*7=20*3+20*4
add r1,r1,r0,lsl #3  ; r1 = 20*15=20*7+20*8
```

Hier wurde eine Multiplikation auf Verschiebungen und Additionen zurückgeführt.

Die Condition Codes

In Abschnitt 1.2.3 wurde das „Current Program Status Register" eingeführt, das Informationen zum Zustand der CPU speichert. Es hat folgendes Format:

`nzcvuuuuuuuuuuuuuuuuuuuuIFTmmmmm`

Die `mmmmm` Bits zeigen an in welchem Modus sich die CPU befindet, `I` und`F` sind für die Interrupt Steuerung und `T` zeigt an welchen Befehlssatz die CPU gerade versteht (Thumb oder ARM). Die `u` Bits werden nicht benutzt, `nzcv` zeigen den Zustand der ALU nach Datenverarbeitungsbefehlen an, wenn bei ihnen das `s` Flag gesetzt was.

Das Zero Flag Das `z` Bit im CPSR zeigt an, dass bei einem Datenverarbeitungsbefehl ein Ergebnis = 0 entstanden ist. Bei der Subtraktion und bei dem Comparebefehl wird dadurch angezeigt, dass Subtrahend und Minuend gleich waren.

Das Negative Flag Das `n` Bit im CPSR ist einfach ein Kopie des MSB des Ergebnisregisters. Es zeigt an, dass das Ergebnis einer Operation negativ war. (Siehe 5.1.4). Bei der Verarbeitung von vorzeichenlosen Zahlen kann es ignoriert werden.

Das Überlauf Flag Das `v` Bit im CPSR zeigt an, dass bei einer Operation ein 2-er Komplement Überlauf aufgetreten ist.(Siehe 5.1.4). Bei der Verarbeitung von vorzeichenlosen Zahlen kann es ignoriert werden.

Das Carry Flag Das `c` Bit im CPSR zeigt an, dass bei einer Addition ein Ergebnis $> 2^{32} - 1$ entstanden ist. Bei der Subtraktion wird dieses Flag auch gesetzt, und zwar wenn der Minuend größer oder gleich dem Subtrahend war. (Wenn etwas Kleineres oder gleiches von etwas Größerem abgezogen wird) Bei Intel und Motorola CPUs ist das genau umgekehrt.

Die Flags bei den Datenverabeitungsbefehlen

Bei den Datenverarbeitungsbefehlen werden die Flags gesetzt, wenn im Befehlswert das `S` Bit gesetzt ist. Der Assembler setzt dieses Bit, wenn in der Instruktion hinter den Befehl und den optionalen Bedingungssuffix der Suffix `S` angehängt wird. Bei den Befehlen `CMP`,`CMN`,`TEQ` und `TST` ist dieser Suffix nicht nötig, da diese Befehle implementiert sind um die Flags zu setzten ohne das Ergebnis zu speichern.

Beispiele

```
Vorher:  r0:fffffffe  r1:00000001  n:0 z:0 c:0
Befehl:  adds r0,r0,r1 ;r0=r0+r1, Flags
Nachher: r0:ffffffff  r1:00000001  n:1 z:0 c:0

Vorher:  r0:0000001b  r1:00000023  n:0 z:0 c:1
Befehl:  cmp r0,r1     ;wom=r0-r1, Flags
Nachher: r0:0000001b  r1:00000023  n:1 z:0 c:0

Vorher: r0:0000001b  r1:00000023  n:1 z:0 c:0
Befehl:  subs r0,r1,r0  ;r0=r1-r0, Flags
Nachher: r0:00000008  r1:00000023  n:0 z:0 c:1

Vorher:  r2:40001a98    n:0 z:0 c:1
Befehl:  movs r2,#0     ;r2=0, Flags
Nachher: r2:00000000    n:0 z:1 c:1

Vorher:  r2:000008f0  r3:40000004 r4:00000800 n:0 z:1 c:1
Befehl:  ands r3,r2,r4  ;r3=r2 AND r4, Flags
Nachher: r2:000008f0  r3:00000800 r4:00000800  n:0 z:0 c:1

Vorher:  r1:00000023  r4:00000800 n:0 z:0 c:1
Befehl:  tst r1,r4    ;wom=r1 AND r4, Flags
Nachher: r1:00000023  r4:00000800  n:0 z:1 c:1
```

Die Bedingungscodes im Befehl

Im Befehlssatz steuern die ersten 4 Bit unter welcher Bedingung die Befehle durchgeführt werden. Im Assembler werden jeder dieser Bitkombinationen ein Mnemonic Kürzel zugeordnet, das als Suffix an das Befehlswort angehängt werden kann:

0000 `xxxEQ` `xxx` wird ausgeführt wenn `Z==1`

0001 `xxxNE` `xxx` wird ausgeführt wenn `Z==0`

0010 `xxxCS` `xxx` wird ausgeführt wenn `C==1`

0011 `xxxCC` `xxx` wird ausgeführt wenn `C==0`

0100 `xxxMI` `xxx` wird ausgeführt wenn `N==1`

0101 `xxxPL` `xxx` wird ausgeführt wenn `N==0`

0110 `xxxVS` `xxx` wird ausgeführt wenn `V==1`

0111 `xxxVC xxx` wird ausgeführt wenn `V==0`

1000 `xxxHI xxx` wird ausgeführt wenn `(C==1) UND (Z==0)`

1001 `xxxLS xxx` wird ausgeführt wenn `(C==0) ODER (Z==1)`

1010 `xxxGE xxx` wird ausgeführt wenn `N==V`

1011 `xxxLT xxx` wird ausgeführt wenn `N== NICHT V`

1100 `xxxGT xxx` wird ausgeführt wenn `(Z==0) AND (N == V)`

1101 `xxxLE xxx` wird ausgeführt wenn `(Z==1) ODER (N == NICHT V)`

1110 `xxxAL xxx` wird unbedingt ausgeführt

Pseudobefehle

Der 2. Operand im `mov` Befehl lässt sich wegen des eingeschränkten Platzes im Befehlswort (12 Bit inclusive Barrelshifter Funktion) nur bedingt nutzen , um Konstanten in ein Register laden. Daher muss oft auf die oben besprochene Technik zurückgreifen, bei der diese Konstanten in der Nähe der Befehle im ROM abgelegt werden und relativ zum R15 Register (Program Counter) adressiert werden. Der Assembler unterstützt das durch sogenannte Pseudobefehle, die den Assembler veranlassen diesen Speicherbereich im ROM zu definieren und den Abstand des aktuellen Befehls zu diesen Speicherelement zu berechnen. Die Syntax ist:

```
ldr{cond} Rn,=Ausdruck
```

Dabei ist:
`cond` ein optionaler Bedingungscode
`Rn` das Register, in das die Konstante kopiert werden soll
`Ausdruck` eine numerische Konstante:
Der Assembler erzeugt einen `MOV` bzw. einen `MVN` Befehl, wenn die Konstante in den 2. Operanden passt.
Wenn nicht, legt der Assembler die Konsante ins ROM in der Nähe des Pseudo Befehls (Literal Pool) und generiert einen `R15` relativen `LDR` Befehl, der diese Konstante adressiert. Wenn es sich bei dem Ausdruck um eine Marke (Symbolischer Name einer Adresse) handelt, wird diese Adresse als Konstante benutzt.
Beispiel:

```
LDR     r3,=0xff0    ; loads 0xff0 into r3
                     ; =>  MOV r3,#0xff0
LDR     r1,=0xfff    ; loads 0xfff into r1
                     ; =>  LDR r1,[pc,offset_to_litpool]
                     ;     ...
                     ;     litpool DCD 0xfff
LDR     r2,=place    ; loads the address of
                     ; place into r2
                     ; =>  LDR r2,[pc,offset_to_litpool]
                     ;     ...
                     ;     litpool DCD place
```

Verzweigungsbefehle

Für Verzweigungen stehen zwei Befehle zur Verfügung:

Branch, Branch with Link Bei diesem Befehl wird das Programm an der Stelle fortgesetzt, die in einem nachfolgenden Ausdruck spezifiziert ist. Der Assembler errechnet den Abstand zum aktuellen Befehl und addiert diesen Offset auf den `PC` (unter Berücksichtigung der Pipeline. Optional kann der aktuelle Program Counter im Linkregister für Unterprogrammtechniken gespeichert.

Branch and Exchange Bei diesem Befehl wird das Programm an der Adresse fortgesetzt, die in einem Register steht.

Die Syntax des Branch Befehls und des Branch with Link Befehls ist:

```
B{L}{cond} <Ausdruck>
```

Der optionale Suffix `{L}` bewirkt, dass der Program Counter in `R14` gespeichert wird, bevor die in `Ausdruck` angegebene Adresse in `R15` (Dem Program Counter) geschrieben wird. Dies führt dazu, dass das Programm an dieser Adresse fortgesetzt wird.

Beispiele

```
Vorher:   r0:00000012  r1:00000031  n:0 z:0 c:0
Befehle:  cmp r0,r1     ; Flags fuer r0-r1 setzen
          bcs schongut  ; carry 1 wenn r0 groesser
          mov r3,r0     ; Sortieren
          mov r0,r1
          mov r1,r3
schongut                ;hier weiterrechnen
Nachher:  r0:00000031  r1:00000012  n:1 z:0 c:0
```

Der gleiche Algorithmus mit anderen Vorbedingungen:

```
Vorher:   r0:00000031  r1:00000012  n:0 z:0 c:0
Nachher:  r0:00000031  r1:00000012  n:0 z:0 c:1
```

Übung 64 Bit Addition: Gegeben seien zwei vorzeichenlose 64 Bit Zahlen, die in `r0` (low) und `r1` (high) bzw. in `r2` (low) und `r3` (high) gespeichert sind. Bitte schreiben Sie ein Programm, das die Summe dieser Zahlen bildet und diese Summe in `r4` (low) und `r5` (high) speichert.

Übung Multiplizieren durch Addieren und Schieben: Bitte schreiben Sie ein Programm, das zwei 16 Bit Zahlen (aus `r1` und `r2`) durch sukzessives Aufsummieren und Schieben multipliziert und das Ergebnis in r0 speichert.

2.2.4 Einfache Kontrollstrukturen und Nassi Shneyderman Diagramme

Mit den Condition Codes und den Verzweigungsbefehlen hat man nun die Möglichkeit unter bestimmten Bedingungen den sequenziellen Befehlsablauf zu unterbrechen und das Programm an einer anderen Adresse fortzusetzen. In den höheren Programmiersprachen wurde diese Möglichkeit auf das `GOTO` Statement abgebildet. Das hat in den 60er Jahren dazu geführt, dass viele Programme entwickelt wurden, in denen konzeptionslos hin und her gesprungen wurde. Der Begriff "Spaghetti-Code" wurde in dieser Zeit geprägt. In der theoretischen Informatik wurde dann gezeigt, dass alle sinnvollen Algorithmen sich durch Schleifen und Alternativen implementieren lassen. In dieser Zeit haben (1972/73) haben Isaac Nassi und Ben Shneiderman die sogenannten "Struktogramme" entwickelt, die in der DIN 66261 genormt sind.

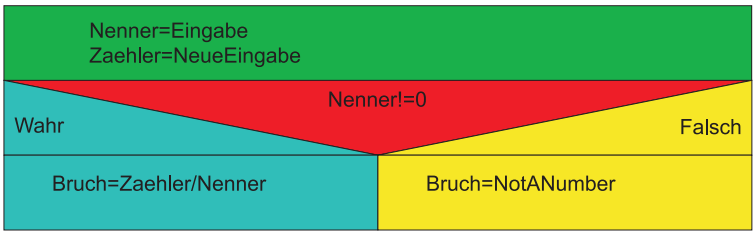


Abbildung 2.4: Nasi Shneidermann Diagramm für einen If-Then-Else Zweig

2.2.5 Alternativen

Oft muss unter einer Bedingung eine bestimmte Sequenz abgearbeitet werden, sollte diese Bedingung nicht erfüllt sein eine Andere. In Nasi Shneidermann Diagrammen wird diese Verzweigung durch ein Rechteck dargestellt, das drei Dreiecke enthält. Die alternativen Sequenzen werden nebeneinander unter dem Dreieck dargestellt, in dem eingetragen ist, ob die Bedingung erfüllt ist oder nicht. Im oberen Dreieck wird die Bedingung eingetragen. In Abbildung 2.4 ist ein Nasi Shneidermann Diagramm für eine Verzweigung dargestellt, in der im Fall eines Nenners, der größer als Null ist, der Bruch durch Division gebildet wird. Ansonsten wird in die Variable die Codierung für $NotANumber$ eingetragen.

If-Then-Else

Das folgende Programmfragment zeigt die Realisierung dieser Kontrollstruktur in C.

```
if (Nenner!=0) then
{
    Bruch = Zaehler / Nenner;
}
else
{
    Bruch = NotANumber;
}
```

In Assembler konnte das Programm wie folgt aussehen:

```
;r0 ist der Nenner
;r1 ist der Zaehler
   cmp r0,#0    ;r0-0 rechnen und flags setzen
   beq NennZ    ;in den Else Zweig wenn z flag gesetzt
   bl  divi     ;Rumpf vom Then Zweig
                ;Methode zum dividieren rufen
   b   EndIf1   ;Alternative beenden
NennZ
   bl  NAN      ;Rumpf vom Else Zweig
                ;Methode zum Fehler behandelnrufen
EndIf1         ;Ende des If Then Else
```

If-Then-Elseif-else

Hat man im else Zweig selbst noch eine Bedingung mit zwei alternativen Sequenzen, wird im else Zweig des Nasi-Shneidermann Diagramms eine Alternative eingetragen (siehe Abb. 2.5).

```
if (Nenner!=0) then
{
```

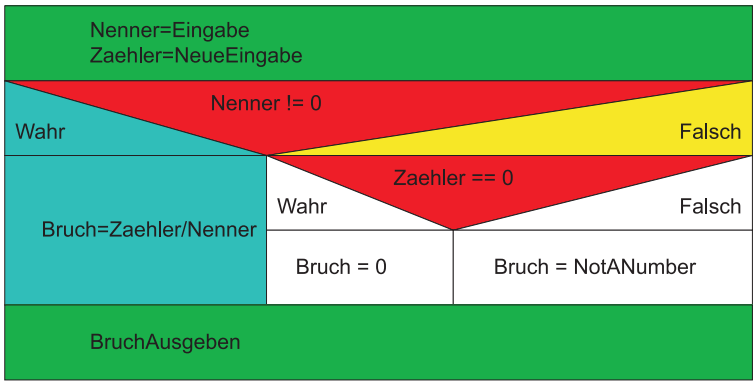


Abbildung 2.5: Nasi Shneidermann Diagramm für einen If-Then-Elseif Zweig

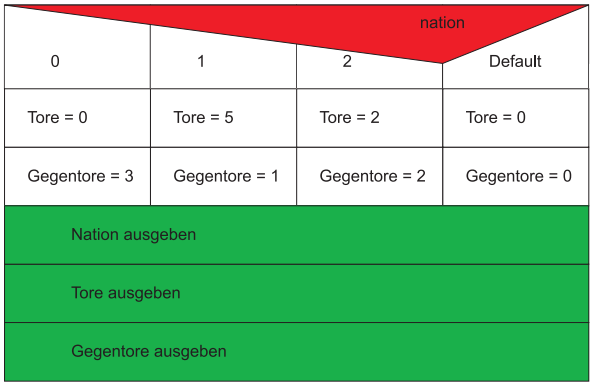


Abbildung 2.6: Nasi Shneidermann Diagramm für eine mehrfach Alternative

```
    Bruch = Zaehler / Nenner;
}
else if(Zaehler == 0)
{
    Bruch = 0;
}
else
{
    Bruch = NotANumber;
}
```

Die Implementation dieser Alternative ist eine Übung. Die Musterlösung steht im PUB.

Mehrfach Alternative Switch-Case

Müssen, abhängig vom Wert einer Variablen, unterschiedliche Sequenzen abgearbeitet werden, spricht man von einer Mehrfach Selektion. Die Variable muss von Typ Integer sein. Beispiel mit Nasi Shneidermann Diagramm und C Code.

```
#define Deutsch 0
#define Francais 1
#define British 2
int nation;
.....
```

```
switch (nation)
{
    case Deutsch:
        Tore = 0;
        Gegentore = 3;
        break;
    case Francais:
        Tore = 5;
        Gegentore = 1;
    break;
    case British:
        Tore = 2;
        Gegentore = 2;
    break;
    default:
        Tore = 0;
        Gegentore = 0;
}
```

In Assembler könnte das so aussehen:

```
; r0: Nation, r1: Tore r2: Gegentore
   cmp r0,#3
   ldrmi pc,[pc,r0,lsl #2]
   b Default
   DCD Deutsch
   DCD Francais
   DCD British
Default  mov r1,#0
          mov r2,#0
          b Weiter
Deutsch  mov r1,#0
          mov r2,#2
          b Weiter
Francais mov r1,#5
          mov r2,#1
          b Weiter
British  mov r1,#2
mov r2,#2
Weiter
```

Übung Bitte erklären Sie `ldrmi pc,[pc,r0,lsl #2]`

2.2.6 Schleifen

Wohl die wichtigste Kontrollstruktur ist die Schleife. Beim Entwurf von Programmen mit Schleifen muss man sich zunächst darüber im Klaren sein, unter welchen Bedingungen die Wiederholung der Sequenz innerhalb der Schleife beendet werden soll. Abhängig davon, wann die Bedingung für die Beendigung geprüft werden soll unterscheidet man drei Typen von Schleifen:

Kopfgesteuert

Wird die Abbruchbedingung vor der ersten Iteration geprüft, spricht man von einer kopfgesteuerten oder abweisenden Schleife. In Abbildung 2.7 ist das Nasi-Shneidermann Diagramm des Euklid'schen Algorithmus abgebildet. Hier folgt die C Implementation:

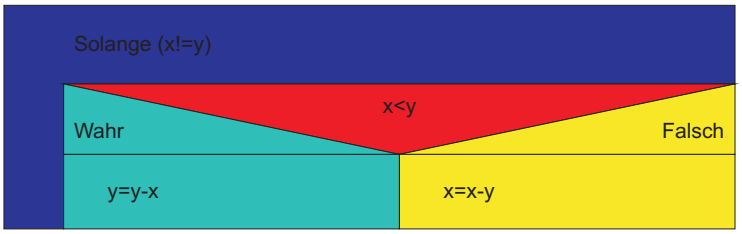


Abbildung 2.7: Nasi Shneidermann Diagramm für eine abweisende Schleife

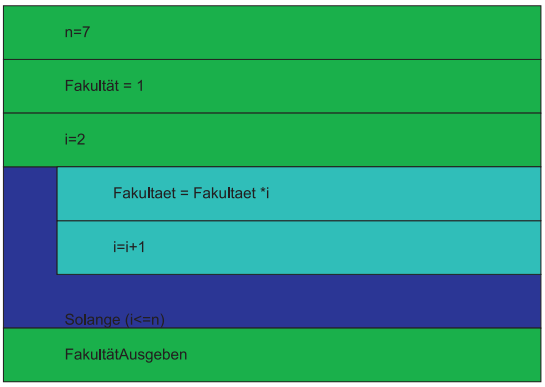


Abbildung 2.8: Nasi Shneidermann Diagramm für eine annehmende Schleife

```
while (x!=y)
{
    if (x<y)
    {
        y=y-x;
    }
    else
    {
        x=x-y;
    }
}
```

Übung In Assembler programmieren. Musterlösung ist im PUB.

Fußgesteuert

Wird die Abbruchbedingung nach jeder Iteration geprüft, spricht man von einer fußgesteuerten oder annehmenden Schleife. Häufig wird dieses Verfahren genutzt, um eine Eingabeaufforderung, das Einlesen der Eingabe sowie das Prüfen der Eingabe so lange zu wiederholen, bis eine vernünftige Eingabe erfolgt ist. Hier ist ein Beispiel für eine Implementation der Berechnung der Fakultät mit einer annehmenden Schleife. In Abb. 2.8 ist das Nasi- Shneidermann Diagramm, im Folgenden die Implementation in C:

```
n=7; Fakultaet =1; i = 2;
do
{
    Fakultaet = Fakultaet * i;
    i=i+1;
} while(i<=n)
```

Abbildung 2.9: Nasi Shneidermann Diagramm für eine Zählschleife

In Assembler programmieren. Musterlösung ist im PUB.

Zählschleife

Sehr häufig haben Schleifen eine fest definierte Anzahl von Durchläufen. Diese Art von Schleifen haben immer die gleiche Struktur: Erst wird die Zählvariable initialisiert (in OO Programmen sogar an dieser Stelle erzeugt), dann wird geprüft, ob die Zählvariable den Endwert erreicht hat (um dann die Schleife zu beenden). Nach jedem Schleifendurchlauf wird die Zählvariable dann um einen bestimmten Wert inkrementiert.

Eine Implementation dieser Kontrollstruktur in C sieht folgendermaßen aus:

```
for (Ausdruck1;Ausdruck2;Ausdruck3)
{
    Anweisung;
}
```

Beispiel: 1000 Byte im String löschen:

```
for (i=0;i<1000;i=i+1;)
{
    mem[i]=0;
}
```

Übung Bitte realisieren Sie dieses Programm in Assembler. Die Musterlösung finden Sie in meinem im PUB Verzeichnis.

2.2.7 Unterprogrammtechniken

Ein Unterprogramm ist ein Teil eines Programms, das von verschiedenen Stellen aus gerufen werden kann.

Verwendung der Befehle bl und bx

In Abschnitt 2.2.3 haben wir die Befehle `bl` und `bx` kennen gelernt. Bei dem Befehl `bl` verzweigt der Progammablauf in das Unterprogramm. Gleichzeitig wird die Adresse des Befehls nach dem `bl` - Befehl in `r14` (dem sogenannten "Link Register") gespeichert. Der Parameter des `bl` Befehls ist ein Ausdruck, es kann der symbolische Name der Startadresse des Unterprogramms sein. Der Assembler berechnet den Abstand dieser Adresse zu dem aktuellen Befehl und addressiert das Unterprogramm dann relativ zum Program Counter `r15`.

Beispiel Im Folgenden wird ein Programm betrachtet, das den Mittelwert aus zwei Zahlen berechnet:

```
main
   mov r1,#9   ; erster Parameter
   mov r2,#3   ; zweiter Parameter
   bl  domean  ; Aufruf des Unterprogramms
   mov r1,#13  ; erster Parameter
   mov r2,#11  ; zweiter Parameter
   bl  domean  ; Aufruf des Unterprogramms
; this ends our main, as we do not have to return
loop
   b loop
;   ----------------------
;   domean
;   berechnet den Mittelwert aus 2 Zahlen
;   Parameter : r1 erste Zahl
;               r2 zweite Zahl
;   Ergebnis: : r0
;   ----------------------

domean           ; Startadresse des Unterprogramms
   add r0,r1,r2      ; Summe bilden
   mov r0,r0,asr #1  ; 1 nach rechts schieben ist /2
   bx  lr            ; zurück ins Hauptprogramm
```

Bei den beiden `bl` Befehlen wird der jeweils aktuelle Befehlszähler (`pc` oder `r15`) in das sogenannte "Link Register" (`lr` oder `r14`) kopiert. In dem Unterprogrmm `domean` wird dann mit dem Befehl `bx lr` die im Link Register gespeicherte Adresse in den Befehlszähler kopiert, danach kann die CPU aus dem Programmspeicher den Befehl holen der hinter dem Unterprogrammaufruf steht. So hat die CPU zurück in das Hauptprogramm gefunden.

Sichern der Arbeitsregister im Stack

Wenn ein aufrufendes Programm nach einem Unterprogammaufruf seine Register weiterverwenden möchte, müssen diese gesichert werden, wenn auch das Unterprogramm diese modifiziert. Hierfür wird ein eigens reservierter Speicherbereich mitsamt einem Adressregister benutzt. Dieses Adressregister zeigt auf das zuletzt geschriebene Datum in diesem Speicher. Diesen Speicherbereich nennt man Stack, und das zugehörige Adressregister ist der Stackpointer. Wird das Register später restauriert, kann der Speicher auch wieder freigegeben werden. Der Stackpointer kann dann auf das Speicherelement zeigen, in dem der Inhalt des zuletzt geretteten Registers steht. So ein Speicherbereich kann man zu steigenden oder zu fallenden Adressen hin wachsen lassen. In einem Projekt mit vielen Unterprogrammen muss vereinbart werden an welcher Stelle die Register, die im Unterprogramm modifiziert werden, die aber vom aufrufenden Programm benötigt werden, gesichert werden.

Aufrufendes Programm sichert Wenn es für das aufrufende Programm wichtig ist, dass das Register `r2` nach dem Unterprogrammaufruf seinen Wert noch hat, dann kann es selbst dieses Register retten und wieder restaurieren:

```
...
str r2,[r13,#-4]!
bl up
ldr r2,[r13],#4
...
```

```
 up  mov r2,#123
     ...
     bx lr
```

Dieses Verfahren hat den Nachteil, dass dieses Register auch gesichert und restauriert wird, wenn das Unterprogramm es gar nicht benutzt.

Aufgerufenes Programm sichert Die Alternative ist, dass das Unterprogramm alle Register sichert, die es modifiziert:

```
     ...
     bl up
     ...
 up  str r2,[r13,#-4]!
     mov r2,#123
     ...
     ldr r2,[r13],#4
     bx lr
```

Dieses Verfahren hat den Nachteil, dass dieses Register auch gesichert und restauriert wird, wenn das aufrufende Programm es nicht benötigt.

Parameter

Es gibt verschiedene Wege von einem Hauptprogramm aus Unterprogramme mit Parametern zu versorgen.

Register Die einfachste Methode einem Unterprogramm einen Wert zu übergeben ist es, diesen Wert in ein Register zu kopieren und das Unterprogramm aufzurufen. Dieses Verfahren wurde im obigen Beispiel mit dem Mittelwertprogramm benutzt. Die beiden Zahlen, aus denen der Mittelwert berechnet werden sollte, wurden in `r1` und `r2` geschrieben, das Unterprogramm hat diese Werte nicht modifiziert und das Ergebnis in `r0` geschrieben. Dieses Verfahren ist sehr schnell, da keine Speicherzugriffe nötig sind.

Stack Reicht die Anzahl der Register für die Parameterliste nicht aus, dann empfiehlt es sich die Parameter im Stack zu übergeben. Obiges Beispiel würde dann wie folgt aussehen:

```
   mov r1,#9   ; erster Parameter
   mov r2,#3   ; zweiter Parameter
   str r1,[sp,#-4]!
   str r2,[sp,#-4]!
   bl  domean  ; Aufruf des Unterprogramms
   add sp,sp,#8 ;Stackpointer restaurieren
   mov r1,#13  ; erster Parameter
   mov r2,#11  ; zweiter Parameter
   str r1,[sp,#-4]!
   str r2,[sp,#-4]!
   bl  domean  ; Aufruf des Unterprogramms
   add sp,sp,#8 ;Stackpointer restaurieren
loop   b loop
domean          ; Startadresse des Unterprogramms
   str r3,[sp,#-4]!  ; Arbeitsregister retten
   str r4,[sp,#-4]!  ;       "
   ldr r3,[sp,#8]    ; Parameter vom Stack holen
   ldr r4,[sp,#12]   ;       "
```

```
   add r0,r3,r4      ; Summe bilden
   mov r0,r0,asr #1  ; 1 nach rechts schieben ist /2
   ldr r4,[sp],#4    ; Arbeitsregister restaurieren
   ldr r3,[sp],#4    ;        "
   bx  lr            ; zurück ins Hauptprogramm
```

Hier die Bespielprogramme aus der Vorlesung: GGT und 64 Bit Addition, beides als Unterprogramm formuliert.

```
;  Wintersemester 2013/2014 RMPP Aufgabe 1


;*****************************************
; Data section, aligned on 4-byte boundery
;*****************************************
AREA MyData, DATA, align = 2
;*****************************************
; Code section, aligned on 4-byte boundery
;*****************************************

AREA MyCode, CODE, readonly, align = 2
;----------------------------------------
; main subroutine
;----------------------------------------
GLOBAL main
main PROC
ldr r0,=0x12345678
ldr r1,=0xfedcba98
ldr r2,=0xee111111
ldr r3,=0x01234567
bl  add64
mov r0,#88
mov r1,#24
bl  ggt
forever b forever ; nowhere to retun if main ends ENDP
;------------------------------------
; add64
; berechnet Summe aus 2 64 Bit Zahlen
; Parameter zwei integers
;                erster  in r0 und r1
;                zweiter in r2 und r3
;                r0 und r2 low
;                r1 und r3 high
; Ergebnis  in r4 (low) und r5
add64   PROC
   adds  r4,r2,r0
   add   r5,r3,r1
addscs r5,r5,#1
   bx    lr
   ENDP
;------------------------------------
; ggt
; berechnet groessten gemeinsamen Teiler
; Parameter zwei integers in r0 ist x und r1 ist y
```

```
; Ergebnis GGT in r0
ggt     PROC
next   cmp r0,r1 ; Anfang der While (r1!=r0) Schleife
       beq whileend
   ;if (x<y) Flags aus vorherigem cmp
   submi r1,r1,r0 ;r1=r1-r0
   ; else r0== r1 ist hier unmöglich wg beq
   subpl r0,r0,r1 ;r0=r0-r1
       b next  ; Ende der While (r1!=r0) Schleife
whileend
       bx lr
   ENDP
END
```

Gemeinsamer Speicher Bei größeren Datenmengen wie etwa Photos wäre eine Kopie für die Übergabe zu zeitaufwendig. Hier bietet es sich an in einem Register die Speicheradresse des Photos zu übergeben. So arbeiten Hauptprogramm und Unterprogramm dann im gemeinsamen Speicher.

Rückgabewerte

Die Ergebnisse von Unterprogrammen können in Register kopiert werden, aus denen sich das Hauptprogramm den Wert holt und speichert. Bei größeren Datenstrukturen bietet es sich an dem Unterprogramm in einem Register die Adresse des Speicherbereichs zu übergeben, in den das Ergebnis kopiert werden soll.

ARM Standard für Unterprogrammaufrufe

ARM und einige Hersteller von CPUs mit ARM Kernel haben den "Procedure Call Standard for the ARM® Architecture" herausgegeben. (Merkwürdige Lizenzbedingungen!) Hier wird vorgeschlagen, wie in einem Projekt die Register belegt werden, wie die Parameter übergeben werden und wie die Ergebnisse zurückgegeben werden.

Register Belegung Folgende Registerbelegung schlägt das Konsortium vor:

r15 Der Program Counter speichert die Adresse des nächsten Befehls.

r14 Das Linkregister speichert die Rücksprungadresse bei Unterprogrammaufrufen. Sollte innerhalb eines Unterprogramms ein weiteres Unterprogramm gerufen werden, ist es vor diesem weiteren Unterprogrammaufruf auf dem Stack zu retten und danach wieder zu restaurieren.

r13 Dieses Register sollte ausschließlich für die Stackverwaltung benutzt werden. Im Prinzip könnten auch andere Register benutzt werden um Stacks zu verwalten, aber für Parameterübergabe und Register sichern sollte dieser Stack benutzt werden.

r12 Wenn Unterprogramme aufgerufen werden, deren Adresse weiter als 24 Bit vom aufrufenden Befehl entfernt sind (beim LPC vom ROM ins RAM), dann reicht der Platz im `bl` Befehl nicht. Der Linker kann dann sogenannte "Furnier Routinen" einfügen, die dieses Problem umgehen. Diese Routinen verändern das `r12` Register. In diesem Fall müsste es vom aufrufenden Programm gerettet und restauriert werden falls nötig. Ansonsten kann es für Variablen genutzt werden.

r4-r11 Diese Register sollten in Unterprogammen für lokale Variablen benutzt werde. Verändert ein Unterprogramm diese Variablen sollte es sie am Anfang retten und vor dem Beenden restaurieren.

r2 und r3 Diese Register können für die Parameterübergabe benutzt werden. Sie können in einem Unterprogramm verändert werden, ohne dass sie restauriert werden müssen. Beim Unterprogrammaufruf geht man davon aus, dass sie nach dem Ablauf des Unterprogramms verändert sind.

r0 und r1 Auch diese Register können für die Parameterübergabe benutzt werden. Sie können in einem Unterprogramm verändert werden, ohne dass sie restauriert werden müssen. Beim Unterprogrammaufruf, geht man auch bei diesen Registern davon aus, dass sie nach dem Ablauf des Unterprogramms verändert sind. Darüber hinaus werden in diesen Registern die Ergebnisse des Unterprogramms an das Hauptprogramm übergeben.

Ergebnisrückgabe Die Art und Weise, wie die Ergebnisse zurückgegeben werden, hängt natürlich von dem Datentyp des Ergebnisses ab. Für die wichtigsten Datentypen schlägt das Konsortium vor:

Half- precision Floating Point Diese Datentypen werden in Single- precision Floting Point Daten gewandelt und in r0 zurückgegeben. (siehe Kapitel 5.3.2)

Kleine Datentypen Fundamentale Datentypen, die kleiner als 4 Bytes sind, werden mit Nullen oder dem Vorzeichen auf 32 Bit erweitert und in r0 zurückgegeben.

32 Bit Datentypen Fundamentale Datentypen in Wortgröße(insbesondere int, float) werden in r0 zurückgegeben.

64 Bit Datentypen Fundamentale Datentypen in doppelter Wortgröße(insbesondere long long, double) werden in r0 und r1 zurückgegeben.

128 Bit Datentypen Fundamentale Datentypen in vierfacher Wortgröße werden in r0 bis r3 zurückgegeben.

Zusammengesetzte 32 Bit Datentypen Zusammengesetzte Datentypen in Wortgröße werden in r0 zurückgegeben.

Größe zusammengesetzte Datentypen Zusammengesetzte Datentypen, die größer als 4 Byte sind, und deren Größe dynamisch ermittelt wird, werden in den Speicher kopiert, und dessen Adresse wird als extra Parameter beim Aufruf übergeben. Das Unterprogramm kopiert dann die Ergebnisse in diesen Speicherbereich.

Parameterübergabe Das Konsortium schlägt vor, die Parameter in den Registern `r0` bis `r3` zu übergeben. Sollten diese Register nicht ausreichen, wird der Stack für die Parameterübergabe benutzt. Weitere Einzelheiten, wie die Übergabe von zusammengesetzten Datentypen und Verwaltung von Systemen mit FPU sind im "Procedure Call Standard for the ARM® Architecture" nachzulesen.

Übung: Bitte schreiben Sie ein Unterprogramm, das den Funktionswert eines Polynoms 5. Grades berechnet:

$$f(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + a_3 \cdot x^3 + a_4 \cdot x^4 + a_5 \cdot x^5 \tag{2.1}$$

Die Parameter sollen sein:a_0, a_1, a_2, a_3, a_4, a_5 und$x$. Der Rückgabewert soll $f(x)$ sein. Musterlösung:

```
;******************** (C) COPYRIGHT HAW-Hamburg ********************************
;* File Name          : main.s
;* Author             : BAI2
;* Version            : V1.0
;* Date               : 25.10.2013
;* Description        : This is a simple main.
```

```
;
;*******************************************************************************

;*****************************************
; Code section, aligned on 8-byte boundery
;*****************************************

AREA mycode, CODE, READONLY, ALIGN = 3

;-----------------------------------------
; main subroutine
;-----------------------------------------
EXPORT main [CODE]

main PROC
        ; Parameter fuer poly 5,3,2 auf Stack a4,a5,x
mov r4,#0x71 ;fuer demozwecke
mov r5,#0x72 ;fuer demozwecke
mov r6,#0x73 ;fuer demozwecke
mov r7,#0x74 ;fuer demozwecke
mov r0,#5
str r0,[sp,#-4]!
mov r0,#3
str r0,[sp,#-4]!
mov r0,#2
str r0,[sp,#-4]!
        ; Parameter fuer poly 17,13,11,7 in r0-r3 a0,a1,a2,a3
mov r0,#17
mov r1,#13
mov r2,#11
mov r3,#7
bl poly
; Parameterbereich auf stack freigeben
add sp,sp,#12
forever b forever ; nowhere to retun if main ends
ENDP
poly    PROC
;Arbeitsregister retten
str r4,[sp,#-4]!
str r5,[sp,#-4]!
str r6,[sp,#-4]!
str r7,[sp,#-4]!
;Parameter a4-a5 und x vom Stack in Arbeitsregister
ldr r4,[sp,#24] ;a4
ldr r5,[sp,#20] ;a5
ldr r6,[sp,#16] ;x
; r7=a5*x
mul r7,r5,r6
; r7=r7+a4
add r7,r7,r4
; r7=x*r7
mul r7,r6,r7
; r7=r7+a3
add r7,r7,r3
```

```
; r7=x*r7
mul r7,r6,r7
; r7=r7+a2
add r7,r7,r2
; r7=x*r7
mul r7,r6,r7
; r7=r7+a1
add r7,r7,r1
; r7=x*r7
mul r7,r6,r7
; r7=r7+a0
add r7,r7,r0
;  Rueckgabewert in r0
mov r0,r7
;Arbeitsregister restaurieren
ldr r7,[sp],#4
ldr r6,[sp],#4
ldr r5,[sp],#4
ldr r4,[sp],#4
bx lr
ENDP
ALIGN

END
```

2.3 Fallstudien

2.3.1 Das Sieb des Erathostenes

```
AREA MyData, DATA, align = 2
prims SPACE 100
AREA |.text|, CODE, READONLY, ALIGN = 3
EXPORT main [CODE]
main   PROC
       LDR r2,=prims
       mov r1,#1
       mov r0,#0
       strb r0,[r2]
       strb r0,[r2,#1]

       ;Forschleife zur Initialisierung Primzahlenarray
       mov r3,#2
lini   cmp r3,#100
       beq liniend
    strb r1,[r2,r3]
add r3,r3,#1  ;i++
b lini
liniend ;ende Forschleife zur Initialisierung

       ;aeussere Forschleife zur Besetzung Primzahlenarray
       mov r3,#2
lallez cmp r3,#10
       bpl lallezend
```

```
          ;if untersuchte Zahl == Primzahl
          ldrb r4,[r2,r3]
          cmp r4,r1      ;if prim
          bne endifp
             ;Forschleife zum Loeschen von Vielfachen
             add r5, r3,r3
lallep       cmp r5,#100
             bpl lallepend
                strb r0,[r2,r5]
             add r5,r5,r3  ;j=j+i
             b lallep
lallepend    ;ende Forschleife zu Loeschen
endifp  ;ende if untersuchte Zahl == Primzahl
        add r3,r3,#1  ;i++
        b lallez
lallezend ; ende aeussere Forschleife zur Besetzung Primzahlenarray
forever b forever ; nowhere to retun if main ends
    ENDP
    ALIGN
    END
```

2.3.2 Groß Klein Schreibung, Strings

```
;******************** (C) COPYRIGHT HAW-Hamburg *********************************
;* File Name          : main.s
;* Author             : TI1
;* Version            : V1.0
;* Date               : 15.07.2012
;* Description        : This is a simple main.
;   : Testrahmen fuer 2 Unterprogramme.
;
;*******************************************************************************

;*****************************************
; Data section, aligned on 4-byte boundery
;*****************************************

AREA MyData, DATA, align = 2

text DCB "MaxMeyer",0
Wtext   SPACE 40
Ztext   SPACE 40

;*****************************************
; Code section, aligned on 8-byte boundery
;*****************************************

AREA MyCode, CODE, readonly, align = 2

;-----------------------------------------
; main subroutine
;-----------------------------------------
GLOBAL main
```

```
main PROC
        ldr r0,=text ;Pseudobefehl zu
             ;atext   DCD text
                     ;....
                     ; ldr r0,atext
        ldr r1,=Wtext ; Adresse des Speicherbereichs
              ; fuer konvertierten Text
bl WGT
        ldr r0,=text
        ldr r1,=Wtext
        ldr r2,=Ztext
        bl ANH
forever b forever ; nowhere to retun if main ends
ENDP
;--------------------------
; Tauscht Gross und Kleinbuchtaben
; Parameter; r0 Adresee des Strings, dessen Buchstaben
;               getauscht werden sollen
    ;            r1 Adresse des Speicherbereichs, in dem
        ;               die getauschten Buchstaben
;               abgelegt werden sollen
        ; Ergebnis : r0, Anzahl der Tauschvorgaenge
        ; Anmerkung; Funktioniert nur mit Buchstaben im String
WGT     PROC
        mov r3,#0  ;Anzahl der getauschten Buchstaben
strt    ldrb r2,[r0],#1
cmp  r2,#0
beq  fin
eor  r2,r2,#0x20
        strb r2,[r1],#1
add r3,r3,#1
        b strt
fin mov r0,r3
        bx lr
        ENDP
ANH     PROC
lp1     ldrb r3,[r0],#1
        cmp  r3,#0
beq  next
    strb  r3,[r2],#1
        b lp1

next    mov  r3,#0x20
    strb  r3,[r2],#1
lp2     ldrb r3,[r1],#1
        cmp  r3,#0
beq  ffin
    strb  r3,[r2],#1
        b  lp2
ffin    bx lr
ENDP
ALIGN
      END
```

2.3.3 Palindromerkennung

```
;* Author             : TI1
;* Date               : 26.05.2014
;* Description        : Palindromerkennung
;*******************************************************************************
;*****************************************
; Data section, aligned on 4-byte boundery
;*****************************************
AREA MyData, DATA, align = 2
EXPORT plndrm
plndrm DCB "nafreibierfan",0
kplndrm DCB "nofreibierfan",0
;*****************************************
; Code section, aligned on 8-byte boundery
;*****************************************
AREA MyCode, CODE, readonly, align = 2
;-----------------------------------------
; main subroutine
;-----------------------------------------
GLOBAL main
aplndrm DCD plndrm    ;Adressen als Konstanten im ROM
akplndrm DCD kplndrm
main PROC
LDR r0,aplndrm ;vorderer Zeiger
        bl palin
LDR r0,akplndrm ;vorderer Zeiger
        bl palin
forever b forever ; nowhere to retun if main ends
ENDP

;-----------------------------------------
; Unterprogramm zur Palindromerkennung
; Parameter  r0 Adresse des Strings, funktioniert nur
; mit ungerader Anzahl von Buchstaben also mit <<kanak>> nicht mit <<otto>>
;  Ergebnis r0 1 fuer Palindrom, -1 fuer kein Palindrom
palin   PROC
        str r4,[sp,#-4]!
mov r2,#1 ;r2 zeigt an ob palindrom, default ja , -1 heisst nein
        mov r1,r0
whlen ldrb r4,[r1]  ;stringlaenge ermitteln,
cmp r4,#0
beq whlenend
add r1,r1,#1 ;hinterer Zeiger
        b whlen
whlenend
        sub r1,r1,#1
                ; r3 und r4 fuer die Buchstaben aus dem String
whla    cmp r1,r0     ; 1. Abbruchbedingung
        beq whle
cmp r2,#1     ; 2. abbruchbedingung
bne whle
ldrb r3,[r0]
ldrb r4,[r1]
```

```
add r0,r0,#1
sub r1,r1,#1
cmp r3,r4
movne r2,#-1 ; Buchstaben waren nicht gleich, daher -1
b    whla
whle    mov  r0,r2 ;Ergebnis in r0
        ldr r4,[sp],#4
bx lr
ENDP
```

2.3.4 Fakultaet rekursiv

```
package faku;
public class Faku {
    public static void main (String [] arges){
        int resu;
        resu = faku(7);
        System.out.println(resu);
    }
public static int faku (int n){
    int faku;
    if(n==1){
        return (1);
    }else{
        return (faku(n-1)*n);
    }
  }
}
```

Obiges Java Programm sieht im Cortex m3 Assembler wie folgt aus:

```
;* Author              : TI1
;* Date                : 2.06.2014
AREA MyCode, CODE, readonly, align = 2
;-------------------------------------------
; main subroutine
;-------------------------------------------
GLOBAL main

main PROC
        mov r0,#7 ; 7! berechnen
bl  faku  ;  Aufruf Fakultaetsprogramm
forever b forever ; nowhere to retun if main ends
ENDP
faku    PROC
;  ------------------------
;  berechnet n! rekrsiv
;  Parameter n in r0
;  Ergebnis auch in r0
;  r4 fur n-1
;  r5 fur n
;  ------------------------
;   r4 und lr retten
   str r4,[sp,#-4]!
   str r5,[sp,#-4]!
```

```
   str lr,[sp,#-4]!
   cmp r0,#1
   beq fini
   sub r4,r0,#1
   mov r5,r0
   mov r0,r4
   bl  faku
   mul r0,r5,r0
fini
   ldr lr,[sp],#4
   ldr r5,[sp],#4
   ldr r4,[sp],#4
   bx lr
   ENDP
```

Kapitel 3

Die Programmiersprache C

3.1 Einführung

3.1.1 Voraussetzungen

Um in diesem Modul erfolgreich zu sein, ist es erforderlich dass die Teilnehmer sowohl Java als auch Assembler programmieren können.

3.1.2 Eigenschaften

C gehört zu den imperativen Programmiersprachen und besitzt eine relativ kleine Menge an Schlüsselwörtern. Die Anzahl der Schlüsselwörter ist so gering, weil viele Aufgaben, welche in anderen Sprachen über eigene Schlüsselwörter realisiert werden, über einzubindende Bibliotheksroutinen (zum Beispiel die Ein- und Ausgabe auf der Konsole oder Dateien und die Verwaltung des dynamischen Speichers) oder über spezielle syntaktische Konstrukte (zum Beispiel Variablendeklarationen) realisiert werden.

C ermöglicht direkte Speicherzugriffe und sehr hardwarenahe Konstrukte. Es eignet sich daher gut zur Systemprogrammierung. Sollen Programme portierbar sein, sollte von diesen Möglichkeiten aber möglichst wenig Gebrauch gemacht werden. Da C direkte Speicherzugriffe kaum einschränkt, kann der Compiler (anders als zum Beispiel in Pascal) nur sehr eingeschränkt bei der Fehlersuche helfen.

Eine Modularisierung in C erfolgt auf Dateiebene. Eine Datei bildet eine Übersetzungseinheit. Intern benötigte Funktionen und Variablen können so vor anderen Dateien verborgen werden. Die Bekanntgabe der öffentlichen Funktionsschnittstellen erfolgt mit sogenannten Headerdateien. Damit verfügt C über ein schwach ausgeprägtes Modulkonzept.

Die Programmiersprache C wurde mit dem Ziel entwickelt, eine echte Sprachabstraktion zur Assemblersprache zu implementieren. Es sollte eine direkte Zuordnung zu wenigen Maschineninstruktionen geben, um die Abhängigkeit von einer Laufzeitumgebung zu minimieren. Als Resultat dieses Designs ist es möglich, C-Code auf einer sehr hardwarenahen Ebene zu schreiben, analog zu Assemblerbefehlen. Die Portierung eines C-Compilers auf eine neue Prozessorplattform ist, verglichen mit anderen Sprachen, wenig aufwändig. Beispielsweise ist der freie GNU-C-Compiler (gcc) für eine Vielzahl unterschiedlicher Prozessoren und Betriebssysteme verfügbar. Für den Entwickler bedeutet das, dass unabhängig von der Zielplattform fast immer auch ein C-Compiler existiert. C unterstützt damit wesentlich die Portierbarkeit von Programmen, sofern der Programmierer auf Assemblerteile im Quelltext und/oder hardwarespezifische C-Konstrukte verzichten kann. Bei der Mikrocontroller-Programmierung ist C die mit Abstand am häufigsten verwendete Hochsprache.

3.1.3 Vergleich zu anderen Sprachen

C gehört zu den imperativen Programmiersprachen, bei denen die Frage im Vordergrund steht, wie etwas berechnet werden soll . Im Gegensatz dazu stehen die deklarativen Sprachen, bei denen eher beschrieben wird, was berechnet werden soll. Bei den deklarativen Programmiersprachen unterscheidet man funktionale Sprachen, wie LISP, Logik basierte Sprachen, wie PROLOG und Abfragesprachen wie SQL.

Bei den imperativen Sprachen unterscheidet man zum Einen maschinenorientierte Sprachen, also die verschiedenen Assembler, zum Anderen prozedurale Sprachen wie PASCAL, ALGOL und eben C, und dann die objektorientierten Sprachen wie Smalltalk, EIFFEL und Java. C wurde auch zu einer objektorientierten Sprache weiterentwickelt. Das Ergebnis ist C++, was in dieser Veranstaltung aber keine Rolle spielen soll.

3.1.4 Das erste C Programm

Das erste C- Programm, das ich hier vorstellen möchte, ist ist ein Programm zu Berechnen des größten gemeinsamen Teilers nach dem Euklidschen Algorithmus.

```
/* Platz für Precompilerdirektiven und */
/* Vereinbarung globaler Daten         */
/*  .....                              */

/* Das Hauptprogramm                  */
int main()
{
/* Definition lokaler Daten            */
unsigned int x=144;
unsigned int y=1008;
unsigned int pgcd;
/* Befehle                            */
    while (x!=y)
    {
       if (x>y)
       {
          x=x-y;
       }
       else
       {
          y=y-x;
       }
   }
   pgcd=x;
   return(0);
}
/* ggf. weiter Unterprogramme          */
/* .....                              */
```

3.2 Einfache Datentypen

3.2.1 Zeichen

Der Datentyp `char` dient zur Speicherung alphanumerischer Daten. Er wird in C als Ganzzahl-Datentypen (mit besonderen Eigenschaften) behandelt. Außerdem repräsentiert ein char die kleinste adressierbare Einheit in C. Die Größe von Objekten und Typen wird stets als ganzzahliges

Vielfache von einem char angegeben. Wenn Hardwareregister byteweise programmiert werden, benutzt man diesen Datentyp. Auch für 8 Bit Arithmetik ist dieser Datentyp brauchbar, daher unterscheidet man auch `unsigned char` und `signed char`.

3.2.2 Numerische Datentypen

Der Datentyp `int` ist der Standarddatentyp für ganzzahlige Werte. Für eventuell größere oder kleinere Wertebereiche existieren die Typen `char`, `short int`, `long int` . Da der Sprachstandard die genaue Größe beziehungsweise den Wertebereich eines Typs nicht festschreibt, gilt nur folgende Relation:

`signed char <= short int <= int <= long int` . (`<=` bedeutet dabei, dass der rechts stehende Typ alle Werte des links stehenden Typs aufnehmen kann.) Zu all diesen Typen existieren noch vorzeichenlose Typen, die durch ein vorangestelltes `unsigned` notiert werden. Diese benötigen den gleichen Speicherplatz wie ihre entsprechenden vorzeichenbehafteten Typen. Optional können die vorzeichenbehafteten Typen auch durch ein vorangestelltes `signed` gekennzeichnet werden und das Schlüsselwort `int` kann bei den mehrteiligen Typnamen entfallen. Der Typ `char` ist ein eigener Datentyp, der jedoch, je nach Plattform, entweder zu `signed char` oder zu `unsigned char` ein identisches Bit-Layout und identische Rechenregeln besitzen muss. Für jeden Typ schreibt der Standard eine Mindestgröße vor. In einer Implementierung können die Werte auch größer sein. Die tatsächliche Größe eines Typs ist in der Headerdatei `<limits.h>` abgelegt. `INT_MAX` ersetzt der Präprozessor (siehe Kapitel 3.4) beispielsweise durch den Wert, den der Typ int maximal annehmen kann. `float`, `double` und `long double` sind die drei Datentypen für Gleitkommazahlen. Auf den meisten Architekturen entsprechen `float` und `double` den IEEE-Datentypen (siehe 5.3.2). Welchen Wertebereich ein Gleitkommazahltyp auf einer Implementierung einnimmt ist ebenfalls plattformabhängig, der Standard legt nur wieder fest, dass der Wertebereich von `float` nach `double` und von `double` nach `long double` jeweils entweder gleich bleibt oder zunimmt. Die genauen Eigenschaften und Wertebereiche auf der benutzten Architektur können über die Headerdatei `<float.h>` ermittelt werden.

3.2.3 Konstanten

Konstanten werden in C durch den Präprozessor verwaltet. Näheres dazu wird in Kapitel 3.4 diskutiert.

3.2.4 Variablen

Wenn wir mit Variablen arbeiten möchten, müssen wir uns erst überlegen welchen Typ diese Variablen haben (siehe oben), wann der Speicher für diese Variablen bereitgestellt werden soll und wann er wieder freigegeben werden kann. Darüber hinaus muss festgelegt werden, welchen Programmteilen der Zugriff auf diese Variablen erlaubt sein soll. Im obigen Programm wurde die Festlegung durch die drei Anweisungen

```
unsigned int x=144;
unsigned int y=1008;
unsigned int pgcd;
```

getroffen. Hier wird vereinbart, dass die Variablen `x,y,pgcd` positive ganze Zahlen sind, dadurch dass diese Vereinbarung hinter der ersten geschweiften Klammer nach `main()` getroffen wird legt man fest, dass der Speicher direkt nach dem Start von `main()` auf dem Systemstack reserviert wird und bei dem Befehl `return(0)` wieder freigegeben wird. Diese Variablen sind nur innerhalb von `main()` sichtbar. In der Literatur werden für diese Festlegungen die Begriffe Vereinbarung, Deklaration und Definition mit uneinheitlichen Bedeutungen benutzt. In dieser Veranstaltung werden diese Begriffe wie folgt benutzt:

Deklaration: Bei der Deklaration wird festgelegt welcher Datentyp mit einem Namen assoziiert wird.

Definition: Bei der Definition wird festgelegt welcher Datentyp mit einem Namen assoziiert wird und zusätzlich Speicher für diese Variable reserviert. Bei den drei obigen Anweisungen handelt es sich also um Definitionen.

Möchte man nur den einen Namen mit einem mit Datentyp assoziieren geschieht dies durch das Schlüsselwort `extern`. Beispiel:

```
extern unsigned int x;
```

Diese Anweisung sagt nur, dass eine Variable mit Namen `x` vom Typ vorzeichenlos ganzzahlig sein soll. Man betrachte folgende Anweisungen:

```
extern unsigned int x;
unsigned int x=144;
```

Hier würde der Compiler prüfen ob die Datentypen bei Deklaration und Definition übereinstimmen, was sie in diesem Beispiel tun. Im folgenden Beispiel aber nicht:

```
extern unsigned int x;
float x=144;
```

Der Compiler würden seinen Übersetztungslauf beenden mit der Fehlermeldung:
`error: conflicting types for 'x'`. Warum für die Deklaration das Schlüsselwort `extern` benutzt wird werden wir in Kapitel 3.3.2 diskutieren.

3.2.5 Typ Casting

Oft ist es nötig Daten zu verknüpfen, die von verschiedenen Typ sind. Man betrachte etwa eine Personengruppe mit eimen Alter zwischen 35 und 65 Jahren und möchte ihr Durchschnittsalter berechnen. n_i soll die Anzahl der Personen mit i Jahren sein. Das Durchschnittsalter $\bar{a}$ errechnet sich dann wie folgt:

$$\bar{a} = \frac{\sum_{i=35}^{65} n_i \cdot i}{\sum_{i=35}^{65} n_i} \tag{3.1}$$

Alle Zahlen auf der rechten Seite sind positive ganze Zahlen. Die linke Seite der Gleichung ist eine rationale Zahl. Nehmen wir an die Summe im Zähler ist ein einer Integervariablen `sumnii` und hat den Wert 6833. Die Summe im Nenner soll in der die Integervariablen `sumni` gespeichert sein und den Wert 134 haben. Für den Quotienten haben wir die Floatingpoint Variable `ma` im Speicher. Der C-Befehl `ma=sumnii/sumni` würde dafür sorgen, dass die Integerdivision `sumnii/sumni` durchgeführt wird (mit dem Ergebnis 50) und diese 50 in die Floating Point Zahl `ma` geschrieben wird. Möchte man die Nachkommastellen in der Variablen `ma` haben, müssen die Datentypen des Divisors und des Dividenden explizit in Floatingpoint Zahlen gewandelt werden. Der Befehl dazu ist:

```
ma=(float)sumnii/(float)sumni;
```

Hier wäre das Ergebnis nach der Operation in `ma` 50.99254. Dieses Typwandeln nennt man auch "Typ casting". (Geben Sie mal in einer Suchmaschine casting ein!)

3.3 Speicherverwaltung in C

3.3.1 Adressraum

Der Adressraum eines ablauffähigen Programms besteht aus vier Regionen, in denen die zur Ausführung benötigten Informationen gespeichert werden.

Code

Dieser Speicherbereich muss häufig gesondert behandelt werden, damit Programme in Harvard Rechnern und in Programmen im ROM gezielt in einen speziellen Adressbereich gelegt werden kann.

Daten

Hier werden die Daten abgelegt, von denen schon beim Start des Programms bekannt ist, dass für sie Speicher benötigt werden.

Stack

Es wurde ausführlich diskutiert, wie lokale Daten von Blöcken und Aufrufparameter von Unterprogrammen im Stack verwaltet werden. Beim Programmstart muss das System einen bestimmten Speicherbereich für den Stack reservieren. In vielen Systemen wird auch überprüft ob der Stack nicht über den zugesicherten Bereich hinaus wächst.

Heap

Diese Region steht dem Programm für die Verwaltung von dynamischen Daten zur Verfügung.Wenn etwa für in eine Liste Elemente eingefügt und wieder entfernt werden müssen, dann wird der Speicherbereich dafür hier alloziert und auch wieder freigegeben. (Vergleich Garbage Collector, Java). In C stehen hierfür die Bibliotheksfunktionen `malloc` und `free` zur Verfügung.

3.3.2 Mehrere Dateien

C ist eine Programmiersprache, die modulares Arbeiten unterstützt. Als Modul soll im folgenden der Quelltext einer einzelnen Datei, eines einzelnen Sourcefiles, angesehen werden. Das Programm seinerseits kann aus 1 bis endlich vielen Modulen bestehen, jedes Modul aus 0 bis endlich vielen Funktionen.

Beispiel: Ist der gesamte Quelltext eines Programms verteilt auf die Dateien `main.c`, `sub1.c` und `sub2.c`, so besitzt dieses Programm drei Module. Wird der Quellcode eines Programms in mehreren verschiedenen Dateien verwaltet, muss der Zugriff auf Code und Daten geregelt sein, deren Quellcode in anderen Dateien steht.

3.3.3 Globale Daten

Ein C-Programm besteht aus einer Reihe von externen (globalen) Objekten, das können Variablen oder Funktionen sein. (`main()` ist auf jeden Fall ein solches externes Objekt.) Dabei wird extern als Kontrast zu intern verwendet und bezeichnet alle Objekte, die außerhalb einer Funktion vereinbart werden. Die Variablendeklarationen innerhalb einer Funktion führen dementsprechend zu internen Variablen. Auch die Variablen(namen) in den Parameterlisten sind in diesem Sinne interne Größen. Funktionen sind stets extern.

Per Default haben externe Objekte die Eigenschaft, dass alle Verweise auf sie mit gleichem Namen auch das gleiche Objekt bezeichnen, sogar aus Funktionen heraus, die separat compiliert worden sind. Dies nennt man externe Bindung.

Gültig ist eine interne Variable stets nur in der Einheit, in der sie deklariert wurde. Eine externe Variable ist im gesamten Programm gültig, mit der Einschränkung des Überdecktwerdens durch gleichnamige lokale Variablen.

Funktionen, die in C immer auf der globalen Ebene stehen müssen, sind von sich aus extern, d.h. aus jedem Modul kann auf sie zugegriffen werden. Prototyping, siehe Kapitel 3.4, wird benutzt um die einzelnen Moduln konsistent zu halten. Dies kann durch explizites Hinzufügen des Schlüsselwortes extern vor den Rückgabetyp betont werden.

3.3.4 Speicher bei lokalen Variablen

Es gibt grundsätzlich zwei Speicherklassen in C: automatisch (`auto`) und statisch (`static`). Zusammen mit dem Kontext der Deklaration eines Objektes (z.B. einer Variablen) bestimmen verschiedene Schlüsselwörter die zu verwendende Speicherklasse.

Automatisch

Automatische Objekte existieren (nur) lokal in einem Block und werden bei Verlassen des Blockes zerstört. Deklarationen innerhalb eines Blockes kreieren automatische Objekte, wenn keine Speicherklasse explizit angegeben wird.

Statisch

Statische Objekte können lokal in einem Block, in einer Funktion oder auch außerhalb von allen Blöcken deklariert werden. Sie behalten ihre Speicherplätze und Werte aber in jedem Fall bei Verlassen von und beim Wiedereintritt in Blöcke und Funktionen bei! In einem Block (und in einer Funktion) werden Objekte mit dem Schlüsselwort `static` als statisch deklariert. Außerhalb von allen Blöcken sind Objekte stets statisch. Mit `static` können sie lokal für ein Modul (Quelltextfile) vereinbart werden, dadurch erhalten sie eine sogenannte interne Bindung. Für ein gesamtes Programm werden sie global bekannt, wenn keine Speicherklasse angegeben wird oder aber durch Verwendung des Schlüsselwortes `extern`, dadurch erhalten sie externe Bindung.

Register

Mit dem Schlüsselwort `register` deklarierte Objekte sind automatisch, werden jedoch nach Möglichkeit in Hardware-Registern verwaltet.

3.4 Preprozessor

In Abschitt 2.2.1 hatten wir zwischen Prozessor Instruktionen und Assembler Direktiven unterschieden. Bei den Prozessor Instruktionen hat der Assembler die Mnemonics in Maschienenbefehle übersetzt, die Assembler Direktiven waren eher Anweisungen an den Assembler selbst. Ähnlich sieht es bei C Programmen aus. Ein C Programm besteht (neben den Kommentaren) aus Befehlen, die in Maschinen Code übersetzt werden sollen und aus Anweisungen an den Compiler. Diese Anweisungen werden vor dem eigentlichen Compilerlauf abgearbeitet. Den Teil des Compilers, der diese Anweisungen abarbeitet, nennt man Preprozessor. Der Compiler erkennt diese Anweisungen an dem Doppelkreuz am Anfang einer Zeile. Es gibt Preprozessor Direktiven für folgende Aufgaben:

- Einfügen von Dateien
- Ersetzen von Text
- Makros
- bedingte Kompilierung
- ...

Die Preprozessor Direktiven haben folgende Syntax:

```
# Direktive Text
```

Typische Direktiven sind: `include`, `define`, `if`, `else`, `endif`, `ifdef`, `ifndef`, `error` Das Doppelkreuz gilt immer nur für eine Zeile, die nächste Zeile wird als ganz normale C Anweisung interpretiert. Fortsetzungszeilen werden durch einen Backslash markiert.

3.4.1 Definition von Konstanten

Syntax:

```
# define Bezeichner Ersatztext
```

Präprozessor ersetzt vor dem Compilieren ab der Direktive den Bezeichner durch Ersatztext: Beispiel(Tafel, ggf mit Beamer und Vorführung): Symbolische Konstanten

```
#define HUND 1
#define KATZE 2
#define TIER KATZE
if (TIER==KATZE)
{
   printf("Das ist eine Katze");
}
else
{
   printf("Das ist keine Katze");
}
if (TIER==HUND)
{
   printf("Das ist ein Hund");
}
else
{
   printf("Das ist kein Hund");
}
```

Übung Wie sieht das Programm nach dem Lauf des Preprozessor aus? Welches Programm übersetzt der Compiler also dann? Gibt es C Anweisungen, die der Compiler weg optimieren kann? Wie würde das vom Compiler optimierte Programm dann aussehen?

Am Entwicklungssystem demonstrieren.

Eine wichtige Anwendung der `define` Direktive ist die Verwaltung von Arraygrößen (siehe Abschnitt 3.5.1). Dazu betrachten wir ein Programm, dass Speicher für 100 Zahle reserviert, und diese in einer Schleife mit den Zahlen von 0 bis 99 initialisiert:

```
int i; // Speicher fuer den Zaehlindex reservieren
int zahlen[100]; // Speicher fuer 100 Zahlen reservieren
for (i=0;i<100;i++)
{
   zahlen[i]=i; // Initialisierung
}
```

Solch ein Programmstück ist eine typische Fehlerquelle in C Programmen (siehe Abschnitt 3.5.1). Man sollte in solchen Situationen folgendes Programmstück vorziehen:

```
#define LEN 100
int i; // Speicher fuer den Zaehlindex reservieren
int zahlen[LEN]; // Speicher fuer 100 Zahlen reservieren
for (i=0;i<LEN;i++)
{
   zahlen[i]=i; // Initialisierung
}
```

Hier würde der Preprozessor überall im Programm den Bezeichner `LEN` durch den Ersatztext `100` ersetzen, bevor der eigentliche Compilerlauf durchgeführt wird. In beiden Fällen steht am Ende das identische Programm im Speicher.

Übung Was ist der Vorteil des zweiten Programms?

3.4.2 Mitübersetzen von anderen Dateien

Möchte man in einem Projekt mit mehreren Moduln (Dateien) an einer zentralen Stelle bestimmte Deklaration vornehmen und Konstanten definieren, so schrebit man diese in eine sogenannte Header Datei. Diese kann dan mit einer Preprozessor Direktive vor dem Überstzungslauf in die einzelnen Moduln kopiert werden. Die Syntax dieser Direktive ist:

```
#include "filename"  //fuer projektspezifische Dateien
```

Der Präprozessor entfernt die `#include-Zeile` und ersetzt sie durch den gesamten Quelltext der Include-Datei. Die Quelltextdatei selbst wird nicht physisch verändert, der Compiler erhält jedoch den modifizierten Text zur Übersetzung.Dateien werden in dem Verzeichnis gesucht, wo auch das C- Programm mit dieser Direktive steht. Zum Beispiel soll diese zentrale Datei `Tiere.h` sein:

```
#define HUND 1
#define KATZE 2
#define TIER KATZE
```

Diese Datei soll in die Datei `tiere.c` vor dem Übersetzungslauf eingefügt werden. Das geschieht mit Hilfe der `include` Direktive: tiere.c

```
#include "Tiere.h"
.....
.....
if (TIER==KATZE)
{
printf("Das ist eine Katze");
}
else
{
printf("Das ist keine Katze");
}
if (TIER==HUND)
{
printf("Das ist ein Hund");
}
else
{
printf("Das ist kein Hund");
}
```

Wenn in einem Projekt meherere Softwarevesionen verwaltet werden sollen, können die verschiedenen Versionen über die Header Datei `tiere.h` verwaltet werden, ohne `tiere.c` zu ändern. (Eine Version für Hunde, eine für Katzen)

Übung : Fügen Sie in diesem Projekt ein weiteres Tier in `tiere.h` hinzu und verwalten es in `tiere.c` mit `switch case`. Man unterscheidet projektspezifische Header Dateien und systemspezifische Header Dateien. Die systemspezifischen Header Dateien sucht der Compiler in dem Verzeichnis, wo auch die Header Dateien stehen. Bei dem `#include` Statement schreibt man dafür den Dateinamen in spitze Klammern. Die projektspezifischen Dateien stehen in dem Verzeichnis, in dem auch die C-Quell Files stehen. Bei dem `#include` Statement schreibt man dafür den Dateinamen in doppelte Hochkommata. Hier ein Beispiel für das Einbinden von system- und projektspezifische Header Dateien:

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

Eine sehr wichtig Anwendung von projektspezifischen Headerdateien das Sicherstellen der Konsistenz von Schnittstellen und Typen globaler Daten. Betrachten wir das Beispiel eines Projektes, das aus zwei Dateien besteht: Ein Hauptprogramm, das ein Unterprogramm aufruft (es soll `main.c` heißen) und ein Unterprogramm das eine Rechenoperation durchführt (der Einfachheit halber betrachten wir hier eine Summenbildung, später können wir auch eventuell die Anzahl der Primzahlen zwischen zwei Grenzen betrachten). `main.c`

```
extern int summand1;
extern int summand2;
extern int summe;
extern int summ2;
void summieren ();
int main()
{
   summand1=1;
   summand2=2;
   summieren();
   return (summe); //Betriebssystem erhält Ergebnis
}
```

Dieses Unterprogramm soll in einer anderen Datei stehen (nennen wir sie `sumup.c`). Aus Effizinzgründen werden die Parameter (hier die beiden Summanden) vom Hauptprogramm in einen Speicherbereich geschrieben, den das Unterprogramm zur Verfügung stellt: `sumup.c`

```
void summieren ();
extern int summand1;
extern int summand2;
extern int summe;
int summand1;
int summand2;
int summe;
void summieren()
{
   summe=summand1+summand2;
}
```

Übung Was passiert, wenn der Programmierer des Hautprogramms aus der Variablen `summe` eine Floating Point Variable macht?

Übung Wie kann man mit Hilfe der `include` Direktive und einer Datei `sumup.h` dazu beitragen, dass dieses Problem nicht auftritt.

3.4.3 Makros

Syntax:

```
#define Bezeichner (Parameter, Parameter,....) Ersatztext
```

Beispiel:

```
#define QUADRAT(x)  x*x
```

Diskussion: geht mit int und float

```
int z=5;
int erg;
erg=QUADRAT(z);
```

liefert 25 in erg aber:

```
erg=QUADRAT(z+2);
```

liefert 17 wg z+2*z+2 Übung

```
printf("Wert ist %d",z);
```

als Makro

3.4.4 Bedingte Kompilierung

Folgende Preprozessordirektiven stehen in C für die bedingte Compilierung zur Verfügung:

```
#if konstanter_Ausdruck
#elif konstanter_Ausdruck
#else
#endif
#ifdef Symbol
#ifndef Symbol
```

Mit Hilfe der bedingten Kompilierung kann gesteuert werden, welche Teile des Quellcodes übersetzt werden sollen. Aus verschiedenen Gründen (Effizienz, Geheimhaltung von Binärcode, ...)kann es wünschenswert sein, dass im Beispiel aus Kapitel 3.4.2 nur die Zeilen 6 und 18 in Binärcode zu übersetzen. Mit bedingter Compilierung würde das Programm folgendermaßen aussehen:

```
#include "Tiere.h"
.....
.....
#if (TIER==KATZE)
    printf("Das ist eine Katze");
#else
    printf("Das ist keine Katze");
#endif
#if (TIER==HUND)
    printf("Das ist ein Hund");
#else
    printf("Das ist kein Hund");
#endif
```

Wenn in der Datei Tiere.h der Symbolische Name TIER mit KATZE definiert ist wird der Preprozessor obige Datei vor dem Kompilierungsvorgang in folgendes Programm umwandeln:

```
    printf("Das ist eine Katze");
    printf("Das ist kein Hund");
```

Eine sehr beliebte Anwendung dieser bedingten Compilierung ist das Einfügen und Entfernen von Testausgaben.

Übung Bitte entwerfen Sie zwei Dateien, die durch bedingte Compilierung Testausgaben von Integerzahlen verwalten. Benutzen Sie für die Testausgaben folgendes Makro:

```
#define OUTI(n) printf("Mein Int hat den Wert %d\n",n);
```

In einer Datei (z.B debug.h) ist das Makro definiert und eine symbolische Konstante DBG. In einer anderen Datei wird 7 Fakultät berechnet. Alle Zwischenergebnisse werden ausgegeben, wenn DBG den Wert 1 hat, sonst sollen die Testausgaben nicht mitübersetzt werden. Es gibt zwei Lösungsmöglichkeiten. Bitte diskutieren Sie Vor- und Nachteile.

3.5 Zeiger und zusammengesetzte Datentypen

3.5.1 Arrays in C

In einem Array sind mehrere Elemente eines Datentyps gespeichert. Während in JAVA Arrays als Objekte mit Attributen wie `length` und Methoden wie `clone()` behandelt werden, ist in C ein Array lediglich eine Folge von Bytes. Die (maximale) Größe des Array muss vor dem Compilieren festgelegt werden. (Ausnahme dynamische Speicherverwaltung, siehe Abschnitt 3.6.3). Daher bietet es sich an die Arraygröße auch vom Präprozessor verarbeiten zu lassen. Hier ein Beispiel:

```
#define ARRL 7
#define SLEN 100
int main (void)
{
   unsigned int fibo[ARRL];
   char sout[SLEN];
   int i;
   fibo[0]=1;
   fibo[1]=1;
   for (i=2;i<ARRL;i++)
   {
      fibo[i]=fibo[i-1]+fibo[i-2];
   }
    while (1){}
}
```

Zur Laufzeit wird nicht geprüft, ob der Index nicht zu groß ist! Das ist eine häufige Fehlerursache. Folgendes Programm würde zur Laufzeit ein nicht definiertes Verhalten zeigen:

```
int main (void)
{
    char st1[5]="Hugo";
    char st2[5]="Egon";
    char st3[5]="Erna";
    st2[8]='W';
    while (1){ }
}
```

Wahrscheinlich würden die Variablen `st1` oder `st2` modifiziert.

Strings in C

Für Strings gibt es in C keinen eigenen Datentyp. Vielmehr ist in C ein String ein Array des Datentyps `char`, der um ein Byte länger ist, als für die Zeichenkette nötig. Am Ende der Zeichenkette muss dann die binäre Null stehen. Zur Verarbeitung von Strings stehen eine Reihe von Standardfunktionen zur Verfügung. (Siehe Abschnitt 3.6.4). Hier ist ein Bespielprogramm indem die Länge eines Strings berechnet wird. Das Ergebnis wird in der Variablen `len` gespeichert.

```
int main (void)
{
    char name[100]="Ahmed";
    int len=0;
    while ((name[len]!=0)&&(len<100))
    {
        len++;
    }
    while (1){}
}
```

3.5.2 Strukturen in C

Um in einem JAVA Programm Personen mit Namen und Personalnummer zu verwalten würde man etwa folgende Klasse definieren:

```
class Person {
    String name;
    int pnr;
    float gehalt;
}
```

In C benutzt man dafür Strukturen. Die Deklaration einer Struktur in C verdeutlicht das folgende Beispiel:

```
struct Person{
    char name[30];
    float gehalt;
    int pnr;
};
```

Sowohl in dem obigen JAVA Code, als auch in dem entsprechenden C Code Wird nur der Datentyp `Person` definiert, noch keine Variable definiert, also auch kein Speicher reserviert. In JAVA wird nun der Speicher reserviert, mit dem Befehl:

```
Person p=new Person();
```

`p` ist nun eine Referenz auf diesen Speicher. In C wird nun der Speicher reserviert, mit dem Befehl:

```
struct Person p;
```

Hier ist aber `p` keine Referenz sondern der Wert. Das wir durch die Wirkung der beiden folgenden Programmsequenzen deutlich: In C:

```
struct Person p1;   //C
struct Person p2;
p1.pnr=4720;
p2.pnr=5730;
p2=p1;
p1.pnr=1234;
```

Nach dieser Programmsequenz steht in `pnr` von `p2` die 4720, in `pnr` von `p1` die 1234 da bei dem Befehl in `p2=p1` die komplette Struktur als Wert in einen anderen Speicherbereich kopiert wird. Anders in JAVA:

```
Person p1=new Person(); //JAVA
Person p2=new Person();
p1.pnr=4720;
p2.pnr=5730;
p2=p1;
p1.pnr=1234;
```

Nach dieser Programmsequenz steht in `pnr` von`p2` die 1234, in `pnr` von`p1` auch die 1234 da bei dem Befehl in `p2=p1` dafür gesorgt wird dass `p2` auf den gleichen Speicherbereich zeigt wie `p1` .

Wertzuweisungen bei Strukturen in C

In C werden einzelnen Komponenten einer Struktur Werte mit Hilfe der Punktnotation zugewiesen, wobei der Name der Gesamtvariablen und der Komponente angegeben werden. Eine 3 prozentige Gehaltserhöhung für Soraya würde man im obigen Beispiel wie folgt programmieren:

```
struct Person Soraya;
Soraya.gehalt=Soraya.gehalt*1.03;
```

Man kann den Strukturkomponenten auch bei der Definition Initialwerte geben. Folgende Anweisung bewirkt dass die Variable `EinChef` mit dem Namen `Kemal`, dem Gehalt von 6543.21 und die Personalnummer mit 1213 bei der Definition initialisiert wird, die anderen entsprechend:

```
struct Person EinArbeiter = {"Qiang",5000.00,2131};
struct Person EinChef = {"Kemal",6543.21,1213};
struct Person EinDirektor ={"Obafemi",7800.32,765};
```

Wenn der Direktor in den Ruhestand geht, der Chef zum Direktor befördert wird und der Arbeiter zum Chef, wobei sie ihre Personalnummern behalten, müsste man das in C wie folgt realisieren:

```
EinDirektor = EinChef ;
EinChef =  EinArbeiter;
```

Die damit verbundene Gehaltserhöhung könnte man folgendermaßen in C realisieren:

```
EinDirektor.gehalt = EinDirektor.gehalt *1.3;
EinChef.gehalt= EinChef.gehalt*1.2;
```

Arrays von Strukturen in C

In Abschnitt 3.5.1 wurden Arrays von skalaren Datentypen behandelt. In C kann man auch Arrays von Strukturen nutzen. Möchte man etwa in einem C Programm mehrere Personen mit Namen, Personalnummer und Gehalt verwalten definiert man ein Array von Strukturen wie folgt:

```
struct Person p[5];
```

Man kann dieses Array von Strukturen auch bei der Definition initialisieren:

```
struct Person p[5]={{"Hugo",1500.12,4711},
                    {"Willi",500.12,4713},
                    {"Egon",2500.12,4719},
                    {"Fatima",700.12,4722},
                    {"Leyla",5500.12,4723}};
```

Hier ein Beispielprogramm, das obiges Array von Strukturen nach einer Person mit der Personalnummer 4719 durchsucht und deren Gehalt in der Variablen `GefundenesGehalt` speichert.

```
float GefundenesGehalt;
int schluessel=4720;
int i;
GefundenesGehalt=-1.0;
for (i=0;i<5;i++)
{
    if (p[i].pnr==schluessel)
    {
        GefundenesGehalt=p[i].gehalt;
    }
}
```

Sollte die Person mit der betreffenden Nummer nicht in der Liste sein (etwa wenn man nach einer Person mit der Personalnummer 4720 sucht) erkennt man an dem negativen Wert für die Variable `GefundenesGehalt`, dass eine Person mit dieser Nummer nicht in der Liste war.

3.5.3 Pointer und Referenzen

Folgendes Programm soll in Erinnerung rufen, wie in einem Assembler Programm zwei Zahlen, die im Speicher stehen, addiert werden und das Ergebnis abgespeichert wird:

```
Zahl1: .word 35
Zahl2: .word 17
ZahlS: .word 0
main:
    Ldr    r0,Zahl1 @Die Register r0,r1,r2
    Ldr    r1,Zahl2 @als Adressregister zeigen
    ldr    r2,ZahlS @auf Zahl1,Zahl2 und ZahlS
    ldr    r3,[r0]    @Daten aus dem Speicher holen
    ldr    r4,[r1]
    add    r5,r4,r3   @Addition durchführen
    str    r5,[r2]    @Ergebnis speichern
loop:
    b  loop
    .end
```

Zuerst werden die Adressen der Zahlen in drei Registern gespeichert. Danach werden diese Adressregister benutzt, um die zwei Summanden in Datenregister zu holen.

Folgendes C Programm leistet das Gleiche:

```
int x=144;
int y=1008;
int z;
int summand1;
int summand2;
int summe;
int *xp;int *yp;int *zp; // Zeigervariablen definieren
xp=&x;
yp=&y;
zp=&z; //Adressen der Variablen in Pointer speichern;
summand1=*xp;
summand2=*yp; // Summanden aus dem Speicher holen
summe =summand1+summand2; // Summe bilden
*zp=summe; //Ergebnis speichern;
```

Pointer und Arrays in C

Etwas verwirrend für jemanden, der C lernt ist die Tatsache, dass der Name eines Array gleichzeitig ein (konstanter) Pointer auf das erste Element des Arrays ist. Folgendes Programm ist also korrekt:

```
int x[10]; // Array fuer 10 integers
int *xp;   // Zeiger auf Integer
xp=x;      // dem Zeiger die Adresse des ersten Elements
           // zuweisen. xp=&x[0]; haette die gleiche Wirkung
```

Hier wird der Zeigervariablen `xp` die Adresse des ersten Elements von `x` zugewiesen. Zeigervariablen selbst kann man in C benutzen wie Arrays (so wie in Java die Referenzen auf Arrays). Man kann also die Adresse und Datentyp des Speicherbereichs kennt durch Indizierung aus diesen zugreifen. Beispiel:

```
1 char str[12]="----------";
2 int Zahlen[2]={-1,-1};
3 int *ip;
```

```
4 ip=Zahlen;
5 ip[0]=1;  // OK
6 ip[1]=3;  // OK
7 ip[3]=1970299243;  // Fatal
```

In den ersten drei Zeilen werden zwei Arrays und ein Zeiger definiert. In Zeile 4 bekommt der Zeiger `ip` die Adresse des Arrays `Zahlen`. In den Zeilen 5 und 6 werden die Elemente des Arrays `ip` modifiziert, in Zeile 7 wird mit Hilfe des Zeigers `ip` ein Speicherbereich verändert, der nicht für das Array `Zahlen`,reserviert wurde. Eventuell wird die Zeichenkette `str` verändert.

Zeiger auf Strukturen

In Abschnitt 3.5.2 haben wir gesehen, dass es in C eine Strukturvariable gibt, in Java gibt es nur Referenzen auf Objekte. In C sind die Referenzen auf Strukturen auch durch Pointer realisiert. Man kann die Adresse einzelner Elemente sowie die ganzer Strukturen speichern. Beispiel:

```
 1  struct Person p[5]={{"Zuebeyde","Hanin",1857,1881},
 2                  {"Helena","Glinskaya",1506,1530},
 3                  {"Barbara","Pierce",1925,1946},
 4                  {"Klara","Poelzl",1860,1889},
 5                  {"Letizia","Ramolino",1750,1769} };
 6   struct Person *ppa,*ppl;
 7   int *ip;
 8   char *cp;
 9   cp= &p[1].vorname[0];
10   ip= &p[2].geboren;
11   ppa=&p[0];
```

In Zeile 1-5 wird das array `p` von Strukturen vom Typ `Person` definiert und initialisiert. In Zeile 6 werden zwei Zeiger auf Strukturen von diesem Typ definiert, in Zeile 7 wird ein Zeiger auf eine Integervariable definiert, in Zeile 8 ein Zeiger auf eine Charactervariable. In Zeile 9 wird die Adresse des ersten Buchstaben des Strukturelements `vorname` des zweiten Elements des Arrays `p` in dem Pointer `cp` gespeichert. In Zeile 10 wird die Adresse des Strukturelements `geboren` des dritten Elements des Arrays `p` in den Pointer `ip` geschrieben. In Zeile 11 wird die Adresse der ersten Struktur des Arrays `p` in den Pointer `ppa` geschrieben.

Zeigerarithmetik

Beispiel: Kopieren eines Integerarrays mit Zeigerarithmetik

```
void up3(int *dst,int *src,int anz)
{
   int i;
   for (i=0;i<anz;i++)
   {
      *dst++=*src++;
   }
}
```

3.5.4 Unterprogramme

Definition Aufruf Rückkehr

Parameter

Per Wert Adressen Arrays Strukturen

Funktionspointer

Ein Funktionsaufruf ist Sprung an eine Adresse und Pointer speichern Adressen. Daher kann man einen Funktions-Aufruf mit Pointer realisieren. Die Deklaration eines Funktionspointers geht wie folgt:
`<return value> (*pointer)(<input>);` deklariert Pointer `pointer` für Funktionen mit Parametern `<input>` und Ergebnis vom Typ `<return value>` Bei Zuweisung müssen Pointer `pointer` und Funktion dieselbe Struktur haben, also gleicher Return-Value und gleiche Input-Parameter-Liste. Der Aufruf einer Funktion über Pointer wie bei normalem Funktionsaufruf! Beispiel:

```
1 #include <stdio.h>
2
3 void output1(char* string) {
4  printf("*%s*\n",string);
5 }
6
7 void output2(char* string) {
8  printf("#%s#\n",string);
9 }
10
11 main() {
12  char string[] = "Hello World";
13 void (*output)(char* string) = NULL;
14
15  output = output1;
16  output(string);
17
18  output = output2;
19  output(string);
20 }
```

Ausgabe dieses Programmes:

```
*Hello World*
#Hello World#
```

3.6 Ausgewählte Standardbibliotheksfunktionen

3.6.1 Mathematische Funktionen

3.6.2 Formatierte Ein - Ausgabe

Die `printf`-Funktionen ermöglichen Ausgabe-Umwandlungen unter Formatkontrolle.

`int fprintf(FILE *stream, const char *format, ...)` Diese Funktion wandelt Ausgaben um und schreibt sie in `stream` unter Kontrolle von `format`. Der Resultatwert ist die Anzahl der geschriebenen Zeichen, er ist negativ, wenn ein Fehler passiert ist. Die Format-Zeichenkette enthält zwei Arten von Objekten: gewöhnliche Zeichen, die in die Ausgabe kopiert werden, und Umwandlungsangaben, die jeweils die Umwandlung und Ausgabe des nächstfolgenden Arguments von `fprintf` veranlassen. Jede Umwandlungsangabe beginnt mit dem Zeichen % und endet mit einem Umwandlungszeichen. Zwischen % und dem Umwandlungszeichen kann, der Reihenfolge nach, folgendes angegeben werden:

Steuerzeichen (flags) (in beliebiger Reihenfolge), die die Umwandlung modifizieren:
- veranlaßt Ausrichtung des umgewandelten Arguments in seinem Feld nach links.
+ bestimmt, dass die Zahl immer mit Vorzeichen ausgegeben wird. Leerzeichen wenn das erste Zeichen kein Vorzeichen ist, wird ein Leerzeichen vorangestellt.

0 legt bei numerischen Umwandlungen fest, daß bis zur Feldbreite mit führenden Nullen aufgefüllt wird.

eine Zahl, die eine minimale Feldbreite festlegt. Das umgewandelte Argument wird in einem Feld ausgegeben, das mindestens so breit ist und bei Bedarf auch breiter. Hat das umgewandelte Argument weniger Zeichen als die Feldbreite verlangt, wird links (oder rechts, wenn Ausrichtung nach links verlangt wurde) auf die Feldbreite aufgefüllt. Normalerweise wird mit Leerzeichen aufgefüllt, aber auch mit Nullen, wenn das entsprechende Steuerzeichen angegeben wurde.

Ein Punkt, der die Feldbreite von der Genauigkeit (precision) trennt. Eine Zahl, die Genauigkeit, die die maximale Anzahl von Zeichen festlegt, die von einer Zeichenkette ausgegeben werden, oder die Anzahl Ziffern, die nach dem Dezimalpunkt bei e, E, oder f Umwandlungen ausgegeben werden, oder die Anzahl signifikanter Ziffern bei g oder G Umwandlung oder die minimale Anzahl von Ziffern, die bei einem ganzzahligen Wert ausgegeben werden sollen (führende Nullen werden dann bis zur gewünschten Breite hinzugefügt).

Die Umwandlungszeichen und ihre Bedeutung sind im Folgenden erklärt. Wenn das Zeichen nach % kein Umwandlungszeichen ist, ist der Verlauf undefiniert.

printf Umwandlungen

d,i Das Argument ist `int` dezimal mit Vorzeichen.

o Das Argument ist `int` oktal ohne Vorzeichen (ohne führende Null).

x,X Das Argument ist `int` hexadezimal ohne Vorzeichen (ohne führendes 0x oder 0X) mit abcdef bei 0x oder ABCDEF bei 0X.

u Das Argument ist `int` dezimal ohne Vorzeichen.

c Das Argument ist `int` int einzelnes Zeichen, nach Umwandlung in unsigned char.

s Das Argument ist `char*` aus einer Zeichenkette werden Zeichen ausgegeben bis vor `'\0'` oder so viele Zeichen, wie die Genauigkeit verlangt.

f Das Argument ist `double` dezimal als `[-]mmm.ddd`, wobei die Genauigkeit die Anzahl der d festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.

e,E Das Argument ist `double` dezimal wobei die Genauigkeit die Anzahl der d festlegt. Voreinstellung ist 6; bei 0 entfällt der Dezimalpunkt.

g,G Das Argument ist `double`

p Das Argument ist `void*` als Zeiger (Darstellung hängt von Implementierung ab).

Folgende Escapesequenzen können im Formatstring benutzt werden:

- \n (newline)
- \t (tab)
- \v (vertical tab)
- \f (new page)
- \b (backspace)
- \r (carriage return)

`int printf(const char *format, ...)` ist äquivalent zu `fprintf(stdout,...)`.

`sprintf` funktioniert wie `printf`, nur wird die Ausgabe in den Zeichenvektor `s` geschrieben und mit `'\0'` abgeschlossen. `s` muß groß genug für das Resultat sein. Im Resultatwert wird `'\0'` nicht mitgezählt.

Beispiele:

```
sprintf(ostr,"Hallo TI-Labor\n TI-Labor\r TI-Labor");
TFT_puts(ostr);
```

Folgendes Programm:

```
int main(void)
{
 char name[30]="Berta";
int alter =90;
int num;
char ostr[150];
int i;
// Demo Sprintf
sprintf(ostr,"Liebe Oma %s \n\r Herzlichen Glueckwunsch zum %dten \n\r Dein Enkel",
name,alter);
  Init_TI_Board();
TFT_cls();
  TFT_puts("Hallo TI-Labor");
  TFT_carriage_return();
  TFT_newline();
  TFT_puts(ostr);
  while(1)
  {

  }
  return 0;

}
```

macht folgende Ausgabe am TFT Bildschirm:

```
Hallo TI-Labor
Liebe Oma Berta
 Herzlichen Glueckwunsch zum 90ten
 Dein Enkel
```

3.6.3 Speicherverwaltung

malloc() free()

3.6.4 Stringverabeitung

In der Definitionsdatei `<string.h>` werden Funktionen für Zeichenketten vereinbart.

In der folgenden Tabelle sind die Variablen `s` und `t` vom Typ `char *`, die Parameter `cs` und `ct` haben den Typ `const char *`, der Parameter `n` hat den Typ `size_t` und `c` ist ein `int`-Wert, der in `char` umgewandelt wird. Die Vergleichsfunktionen behandeln ihre Argumente als `unsigned char` Vektoren.

`char *strcpy(s,ct)` Zeichenkette `ct` in Vektor `s` kopieren, inklusive `'\0'`, liefert `s`.

`char *strncpy(s,ct,n)` höchstens `n` Zeichen aus `ct` in `s` kopieren, liefert `s`. Mit `'\0'` auffüllen, wenn `ct` weniger als `n` Zeichen hat.

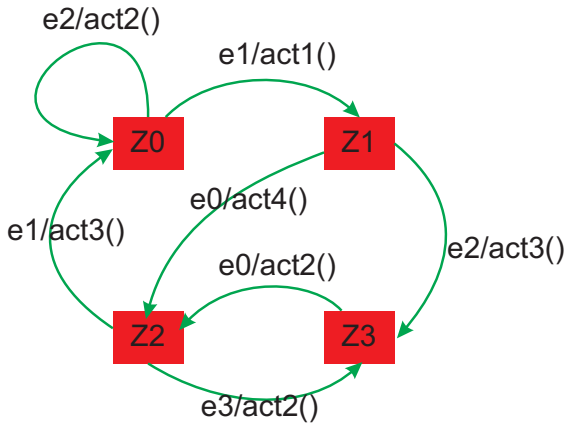


Abbildung 3.1: Zustandsautomat

`char *strcat(s,ct)` Zeichenkette `ct` hinten an die Zeichenkette `s` anfügen, liefert `s`.
`char *strncat(s,ct,n)` höchstens `n` Zeichen von `ct` hinten an die Zeichenkette `s` anfügen und `s` mit `'\0'` abschließen, liefert `s`.
`int strcmp(cs,ct)` Zeichenketten `cs` und `ct` vergleichen, liefert `<0` wenn `cs<ct`, 0 wenn `cs==ct`, oder `>0`, wenn `cs>ct`.
`int strncmp(cs,ct,n)` höchstens `n` Zeichen von `cs` mit der Zeichenkette `ct` vergleichen, liefert `<0` wenn `cs<ct`, 0 wenn `cs==ct`, oder `>0` wenn `cs>ct`.
`size_t strlen(cs)` liefert Länge von `cs` (ohne `'\0'`).

3.7 Fallstudien

3.7.1 Programmieren von Automaten mit Funktionspointern und Strukturen

Man betrachte einen Automaten mit 4 Zuständen Z0-Z3 und 4 Ereignissen e0-e3. Auf diese Ereignisse soll mit den Methoden `act1()`, `act2()`, `act3()` und `act4()` reagiert werden. Die Reaktion ist im Zustandsautomat in Abb. 3.1 spezifiziert. Wenn im Hauptprogramm in einer Endlosschleife geprüft wird ob ein Ereignis aufgetreten ist, dann könnte der Zustandsautomat nach folgendem Muster realisiert werden:

```
void fsm(int); void noact(); // Prototypen
void act1();void act2();void act3();void act4();
struct EvReact {int NextState;void (*DoIt)(void);};
struct EvReact ERTab[4][4]=
     {{{0,noact},{1,act1} ,{0,act2} ,{0,noact}},
      {{2,act4} ,{1,noact},{3,act3} ,{1,noact}},
      {{2,noact},{0,act3} ,{2,noact},{3,act2}} ,
      {{2,act2} ,{3,noact},{3,noact},{3,noact}} };
int main()
{
   unsigned int event;
   while(1)
   {
      event=checkevent();
      fsm(event);
   }
}
```

```
void fsm(int ev){
    static int state=0;
    ERTab[state][ev].DoIt();
    state=ERTab[state][ev].NextState;
}
void act1(){ // code fuer act1 }
void act2(){ // code fuer act1 }
void act3(){ // code fuer act1 }
void act4(){ // code fuer act1 }           */
```

Im zweidimensionalen Array von Strukturen `ERTab` sind die Zeilennummern die Zustände, die Spaltennummern die Ereignisse.

3.7.2 Sieb des Erathostenes

```
#include <math.h>
#define ANZ 100
int main(void){
   static char prims[ANZ]; //1 ist Primzahl, 0 nicht
   int i, j,endi; // indizes
   prims[0]=0;
   prims[1]=0;
   for (i=2;i<ANZ;i++){
      prims[i]=1;
   }
   endi=(int) sqrt((double)ANZ);
   for (i=0;i<endi;i++){
      if(prims[i]==1){
         for (j=i+i;j<ANZ;j=j+i){
            prims[j]=0;
         }
      }
  }
  while(1){};
}
```

3.7.3 Mischen von C und Assembler

```
/**
Programm zu addieren von vier Zahlen mit Assembler Unterprogramm
up. up ist in Assembler und ruft addit. addit ist C Programm
*/
int up(int,int,int,int,int (*)(int,int));
int addit(int ,int );
int main(void)
{

  int i;
  i=up(7,8,9,9,addit);
i=up(i,i,i,i,addit);
  return 0;
}
int addit(int a,int b){
int c=a+b;
```

```
return(c);
}
/**
Programm zu addieren von vier Zahlen mit Assembler Unterprogramm
up. up ist in Assembler und ruft addit. addit ist C Programm
*/
int up(int,int,int,int,int (*)(int,int));
int addit(int ,int );
int main(void)
{

  int i;
  i=up(7,8,9,9,addit);
i=up(i,i,i,i,addit);
  return 0;
}
int addit(int a,int b){
int c=a+b;
return(c);
}


      AREA mycode,CODE
  GLOBAL up
up    PROC  ;addiert 4 Zahlen mit Hife von addit
      str r4,[sp,#-4]! ;Arbeitsregister retten
      str r5,[sp,#-4]!
      str r6,[sp,#-4]!
      str r7,[sp,#-4]!
      str r8,[sp,#-4]!
      str r9,[sp,#-4]!
      str lr,[sp,#-4]!
  mov  r4,r0  ;  1. Summand in Arbeitsregister
  mov  r5,r1  ;  2. Summand in Arbeitsregister
  mov  r6,r2  ;  3. Summand in Arbeitsregister
  mov  r7,r3  ;  4. Summand in Arbeitsregister
  ldr  r8,[sp,#28] ; Adresse von Addit in Arbeitsregister
  mov  r9,#0   ;Akkumulator fuer Summe
  mov  r0,r4  ; 1. Summand an addit uebergeben
  mov  r1,r5   ; 2. Summand an addit uebergeben
  blx  r8    ;addit rufen
  mov  r9,r0  ;summe akkumulieren
  mov  r0,r6    ; 3. Summand an addit uebergeben
  mov  r1,r7  ; 4. Summand an addit uebergeben
  blx  r8     ;addit rufen
  add  r9,r9,r0 ;summe akkumulieren
  mov  r0,r9    ; Ergebnis an main zurueckgeben
      ldr lr,[sp],#4 ;Arbeitsregister restaurieren
      ldr r9,[sp],#4
      ldr r8,[sp],#4
      ldr r7,[sp],#4
      ldr r6,[sp],#4
      ldr r5,[sp],#4
      ldr r4,[sp],#4
```

```
  bx lr           ;zurueck nach main
  ENDP
  END
```

Kapitel 4

Ein-Ausgabe

4.1 Einführung

4.1.1 Konzepte

Um Ein-Ausgabehardware anzusprechen muss man sie zunächst ardessieren. Man unterscheidet zunächst zwei Konzepte der Adressierung von IO Hardware: Die IO Hardware liegt im physikalischen Adressbereich des Speichers (Memory mapped IO) oder aber die IO Hardware hat einen eigenen Speicherbereich. Im ersten Fall muss sich die IO Hardware verhalten wie ein Speicher. Vorteilhaft ist, dass man keine eigenen Befehle für die Adressierung der Hardware braucht. Die Nachteile sind Probleme mit Cache und virtuellen Adressen. Im zweiten Fall muss die CPU ein Signal zur Verfügung stellen, das der Hardware anzeigt, ob die am Adressbus anliegende Adresse für die Adressierung von Speicher oder IO bestimmt ist. Beim IA32 ist dies der M/IO pin, der durch eine 1 anzeigt dass ein Speicherbaustein angesprochen wird. Eine 0 zeigt an, dass es sich um eine IO Adresse handelt.

Beim Memory mapped IO können der mov und alle Befehle mit Speicheradressierung benutzt werden. Problem: Eingabe sollten nur gelesen werden, auf Ausgabegeräte sollte nur geschrieben werden. Entsprechend können in einer Hochsprache Wertzuweisungen an eine (bzw. von einer) bestimmten Adresse erfolgen. Bei einem eigenen Adressraum für IO sind Befehle implementiert, die dafür sorgen, dass neben dem Datentransfer auch der M/IO pin auf 0 gelegt wird. Diese Befehle sind im IA32 out outs outsb outsw outsd für die Ausgabe und in ins insb insw insd für die Eingabe. Für diese Befehle gibt es in den meisten Hochsprechen (außer z. B. in PEARL) keine entsprechenden Elemente. Daher stellen häufig Betriebssysteme Unterprogramme bereit, die das ermöglichen.

4.1.2 Digital E-A im STM32

Im STM32 sind viele IO Ports eingebaut die über spezielle Register angsteuert werden, um sie als Eingänge zu lesen oder als Augänge zu schreiben. Diese Register liegen im Adressraum der CPU. Die Adressen sind in der Datei `stm32f4xx.h` definiert. Mit der Funktion `Init_TI_Board();` wird das TI Board so konfiguriert, dass bei Port G die 8 niederwertigen Bits als Ausgabe arbeiten und bei Port F die Bits 1-7 als Eingabe genutzt werden können. So kann mit dem Befehl `i=GPIOE->IDR;` das Eingaberegister in die Variable `i` eingelesen werden. Nun können wir alle gesetzten Bits von `i` in den Ausgabebits des Ports setzen: `GPIOG->BSRRL=i`

Wir können auch Ausgabebits des Ports löschen und setzen:

```
GPIOG->BSRRL=0x77; // 6 Lampen einschalten
GPIOG->BSRRH=0x11; // 2 Lampen ausschalten
GPIOG->ODR=0x73;   // Bitmuster 0x73 anlegen
i=GPIOE->IDR;
```

```
sprintf(ostr," Eingabe war %d ",((~i)&0xff));
```

In Assembler

```
PERIPH_BASE     equ      0x40000000
AHB1PERIPH_BASE equ      (PERIPH_BASE + 0x00020000)
;blaue LEDs
GPIOG_BASE      equ      (AHB1PERIPH_BASE + 0x1800)
GPIO_G_SET equ      GPIOG_BASE + 0x18
GPIO_G_CLR equ      GPIOG_BASE + 0x1A
GPIO_G_PIN equ      GPIOG_BASE + 0x10

;rote LEDs / Taster
GPIOE_BASE      equ      (AHB1PERIPH_BASE + 0x1000)
GPIO_E_PIN      equ      GPIOE_BASE + 0x10

ldr R3, =GPIO_E_PIN   ;Tasten lesen
ldr R4, [R3]
  and R4, #0x01         ;S0 testen
      cmp R4, #0x01
```

Ein Beispielprogramm:

```
#include "main.h"
#include <stdio.h>
#include <string.h>
#include "TI_Lib.h"
#include "tft.h"
#include "stm32f4xx.h"
#include "stm32f4xx_gpio.h"
#include "stm32f4xx_rcc.h"

int main(void)
{
 char name[30]="Berta";
int alter =90;
int num;
char ostr[150];
int i;
// Demo Sprintf
sprintf(ostr,"Liebe Oma %s \n\r Herzlichen Glueckwunsch zum %dten \n\r Dein Enkel",
name,alter);
  Init_TI_Board();
TFT_cls();
  TFT_puts("Hallo TI-Labor");
  TFT_carriage_return();
  TFT_newline();
  TFT_puts(ostr);

  // Demo TI Board
while(1)
{
GPIOG->BSRRL=0x77; // 6 Lampen einschalten
GPIOG->BSRRH=0x11; // 2 Lampen ausschalten
```

```
i=GPIOE->IDR;
    if (!(i&0x1)) num=1;
   if (!(i&0x2)) num=2;
   if (!(i&0x4)) num=3;
   if (!(i&0x8)) num=4;
   if (!(i&0x10)) num=5;
   if (!(i&0x20)) num=6;
   if (!(i&0x40)) num=7;
   if (!(i&0x80)) num=8;
  GPIOG->ODR=num; // Bitmuster 0x73 anlegen

  }
  return 0;

}
```

Bitte Beachten Sie auch die in der Vorlesung diskutierten Programme im pub verzeichnis unter ?????

Kapitel 5

Daten im Computer

5.1 Negative Zahlen

5.1.1 Betrag mit Vorzeichen

Wir sind im täglichem Leben gewohnt den Betrag einer Zahl im Dezimalsystem anzugeben. Wenn es sich um eine negative Zahl handelt wird dies durch ein zusätzliche Stelle markiert, die + oder - sein kann, das + kann man auch weglassen. Im Dualsystem bietet es sich an festzulegen, dass ein Bit für das Vorzeichen reserviert ist. Bei einer 8 stelligen Dualzahl Z mit den Ziffern $z_0, z_1, ...z_7$(8 Bit bzw. 1 Byte) würde das für den Wert W bedeuten:

$$W = (-1)^{z_7} \sum_{i=0}^{6} z_i \cdot 2^i \tag{5.1}$$

Diese Art der Darstellung negativer Zahlen hat mehrere Eigenschaften:

- Die kleinste darstellbare Zahl ist $1111111 = -127_{10}$, die größte ist $0111111 = +127_{10}$
- die Null kommt doppelt vor: $00000000 = +0_{10} = -0_{10} = 10000000$ Bei allen arithmetischen Operationen muss das Vorzeichenbit gesondert behandelt werden.
- Der Betrag mit Vorzeichen wird trotz seiner Unhandlichkeit für die Darstellung der Mantisse in Floating Point Zahlen benutzt. (siehe 5.3.2)

Übung Was ist die 8-Bit Betrag mit Vorzeichen Darstellung der -7?

Übung Was ist die 8-Bit Betrag mit Vorzeichen Darstellung der -17?

5.1.2 Excess Darstellung

Bei der Excess Darstellung wird von der binärkodierten Zahl eine konstante Zahl B (der„Bias") subtrahiert. So ist der Wert W einer n-Bit Excess B Zahl:

$$W = (\sum_{i=0}^{n} z_i \cdot 2^i) - B \tag{5.2}$$

Übung Was ist die 8-Bit Excess 127 Darstellung der -7?

Übung Was ist die 8-Bit Excess 127 Darstellung der -17?

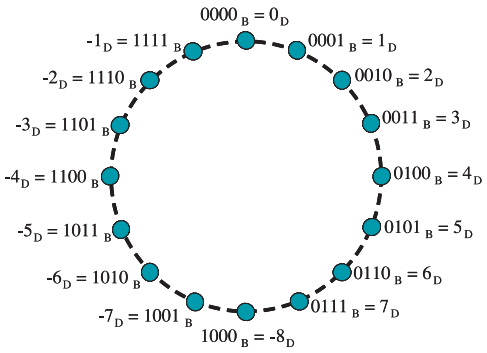


Abbildung 5.1: Beispiel: Zahlendarstellung im 4-Bit 2er Komplement

Anwendung

Die Vorzeichen Excess Darstellung wird für die Darstellung des Exponenten in Floating Point Zahlen benutzt. (siehe 5.3.2)

5.1.3 1 er Komplement

Im 1 er Komplement wird eine Zahl dadurch negiert, dass alle einzelne Bits invertiert werden.

Übung Was ist die 8-Bit 1er Komplement Darstellung der -7?

Übung Was ist die dezimale Darstellung des Ergebnisses, wenn man die (8 Bit) 1er Komplement Darstellungen des Ergebnisses der Bitweisen Addition von a) 1 und -1 b)2 -1 c) c)-1 +4 ins Dezimalsystem wandelt?

Übung Was ist Zahlenbereich des 4 8-Bit 1er Komplements ?

5.1.4 2 er Komplement

Das Zweierkomplement ist die vorherrschende Art, mit der negative ganze Zahlen im Computer dargestellt werden. Mit Hilfe des Rechenwerkes werden Rechenoperationen auf Zahlen in dieser Darstellung durchgeführt. Da bei binären Kodierungen von negativen Zahlen sowohl Vorzeichen als auch die eigentliche Zahl durch Bits dargestellt werden, ist es wichtig zu wissen, welches Bit wofür verwendet wird. Üblicherweise wird dies erreicht, indem sämtliche Zahlen eine konstante Stellenzahl haben und bei Bedarf mit führenden Nullen aufgefüllt werden. Die unten angeführten Beispiele verwenden je sieben Ziffern für die Kodierung der Zahl und eine Ziffer für die Kodierung des Vorzeichens (8 Bits, das heißt 1 Byte).

Codierung

Positive Zahlen werden im Zweierkomplement mit einer führenden 0 (Vorzeichenbit) versehen und ansonsten nicht verändert. Negative Zahlen werden wie folgt kodiert:

1. Den Betrag der Zahl ins Binärsystem umrechnen
2. Alle Bits Invertieren (wie im 1er Komplement)
3. Eins addieren

Den (dezimalen) Betrag einer negativen im 2er Komplement kodierten Zahl findet man folgendermaßen:

1. Alle Bits Invertieren (wie im 1er Komplement)
2. Eins addieren
3. Den Betrag der Zahl nach Formeln 5.4 ins Dezimalsystem umrechnen

Beispiele

-4_{10} **8 Bit**

Betrag $|-4_{10}| = 00000100_2$ (in Hexadezimal ist das 04_{16})

Invertieren $\overline{00000100_2} = 11111011_2$ (in Hexadezimal ist das FB_{16})

Eins addieren $11111011_2 + 1 = 11111100_2$ (in Hexadezimal ist das FC_{16})

$11111100 = -4_{10}$

11110000_2 **8 Bit 2er Komplement Zahl ins Dezimalsystem umrechnen** (in Hexadezimal ist das $F0_{16}$)

Invertieren $\overline{11110000_2} = 00001111_2$ (in Hexadezimal ist das $0F_{16}$)

Eins addieren $00001111_2 + 1 = 00010000_2$ (in Hexadezimal ist das 10_{16})

Betrag nach Formel 5.4 $W = 2^4 = 16_{10}$

$11110000_2 = -16_{10}$

Weitere Beispiele:

- $+4_{10} = 00000100_2$ (in Hexadezimal ist das 04_{16})
- $-1_{10} = 11111111_2$ (in Hexadezimal ist das FF_{16})
- $127_{10} = 01111111_2$ (in Hexadezimal ist das $7F_{16}$)
- $-128_{10} = 10000000_2$ (in Hexadezimal ist das 80_{16})

Übung Was ist die 8 Bit 2-er Komplement Darstellung der -111?

Übung Was ist die 8 Bit 2-er Komplement Darstellung der -4?

Übung Was ist die 8 Bit 2-er Komplement Darstellung der -33?

Übung Was ist die 8 Bit 2-er Komplement Darstellung der -1? Durch die im Zweierkomplement verwendete Kodierung wird erreicht, dass nur eine einzige Darstellung der Null existiert.

Null negieren im Zweierkomplement 8 Bit

Betrag $|-0_{10}| = 00000000_2$

Invertieren $\overline{00000000_2} = 11111111_2$

Eins addieren $11111111_2 + 1 = 00000000_2$

Zahlenbereich

Mit n Bits lassen sich Zahlen von -2^{n-1} bis $+2^{n-1}-1$ darstellen. Beispiele:

8 Bit -128_{10} bis $+127_{10}$

16 Bit -32768_{10} bis $+32767_{10}$

32 Bit -2147483648_{10} bis $+2147483647_{10}$

64 Bit $-9223372036854775808_{10}$ bis $+9223372036854775807_{10}$

Rechenoperationen

Addition und Subtraktion benötigen keine Fallunterscheidung. Die Subtraktion wird auf eine Addition zurückgeführt. Beispiel: -4 + 3 = -1 führt zu

		1	1	1	1	1	1	0	0	$\texttt{FC}_{16}$
+		0	0	0	0	0	0	1	1	$\texttt{03}_{16}$
=	0	1	1	1	1	1	1	1	1	$\texttt{FF}_{16}$

Die eingerahmte Null zeigt an, dass es bei der Addition der höchstwertigen Bits keinen Übertrag gegeben hat. Das ist, wenn man es als 2er Komplementzahl interpretiert -1_{10} Führt man obige Berechnung 252 + 3 für zwei vorzeichenlose 8 Bit Zahlen durch, erhält man ebenso FF_{16}. Diese Zahl ist, umgerechnet nach Formel 5.4 255. Das bedeutet: Wenn wir einen Computer zwei Zahlen addieren lassen, dann macht er es (so lange wir den Wertebereich nicht verlassen) richtig. Wenn wir ihm 2er Komplement Zahlen zum Addieren geben errechnet er eine 2er Komplement Summe. Wenn wir ihm vorzeichenlose Zahlen zum Addieren geben errechnet er (ohne es zu wissen) eine vorzeichenlose Zahl als Summe. Problematisch ist nur, wenn wir als Programmierer diese Summe dann als 2er Komplementzahl interpretieren. Beispiel: -4 + 4 = 0 führt zu

		0	0	0	0	0	1	0	0	$\texttt{04}_{16}$
+		1	1	1	1	1	1	0	0	$\texttt{FC}_{16}$
=	1	0	0	0	0	0	0	0	0	$\texttt{00}_{16}$

Die eingerahmte Eins zeigt an, dass es bei der Addition der höchstwertigen Bits einen Übertrag gegeben hat. Im Ergebnis werden nur 8 Bit gespeichert. Das ist, wenn man es als 2er Komplementzahl interpretiert 0_{10}, korrekt. Führt man obige Berechnung 252 + 4 für zwei vorzeichenlose 8 Bit Zahlen durch, erhält man ebenso 00_{16}. Diese Zahl ist, umgerechnet nach Formel 5.4 0. Das kann nicht korrekt sein, da das richtige Ergebnis (256) nicht in einer 8 Bit Zahl gespeichert werden kann (die größte 8 Bit Zahl ist 255!) Beispiel: -4 +(- 3) = -7 führt zu

		1	1	1	1	1	1	0	0	$\texttt{FC}_{16}$
+		1	1	1	1	1	1	0	1	$\texttt{FD}_{16}$
=	1	1	1	1	1	1	0	0	1	$\texttt{F9}_{16}$

Die eingerahmte Eins zeigt an, dass es bei der Addition der höchstwertigen Bits einen Übertrag gegeben hat. Im Ergebnis werden nur 8 Bit gespeichert. Das ist, wenn man es als 2er Komplementzahl interpretiert -7_{10}, korrekt. Führt man obige Berechnung 252 + 253 für zwei vorzeichenlose 8 Bit Zahlen durch, erhält man ebenso $F9_{16}$. Diese Zahl ist, umgerechnet nach Formel 5.4 249. Das kann nicht korrekt sein, da das richtige Ergebnis (505) nicht in einer 8 Bit Zahl gespeichert werden kann (die größte 8 Bit Zahl ist 255!)

Eine andere Vorgehensweise zur Errechnung des Wertes W einer Zweierkomplementzahl ist die folgende. Habe die Darstellung der Zahl im Zweierkomplement n Stellen, gegeben sind also n Bits $a_{n-1}, a_{n-2}, a_{n-3}, \cdots, a_1, a_0$:

$$x_{10} = -2^{n-1} \cdots a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i \tag{5.3}$$

5.1.5 Flags

Wie schon erwähnt kann es bei der binären bei einer begrenzten. Zahl von Stellen zu einer Überschreitung des Zahlenbereichs kommen. Je nach dem, ob es sich eine Addition von vorzeichenlosen

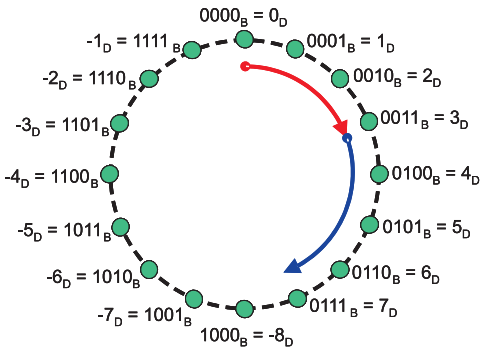


Abbildung 5.2: Beispiel: Veranschaulichung der Summe von 3+4 im 4-Bit 2er Komplement. Weder Carry noch Overflow sind gesetzt.

oder 2er Komplementzahlen handelt redet man von Übertrag oder Überlauf (Carry, Overflow) Das Carry Flag wird mit C abgekürzt, das Overflow Flag mit V. Beispiel: 4+(-4) im 4 Bit 2er Komplement:

V	C		a_3	a_2	a_1	a_0	Hex	unsigned	signed
			0	1	0	0	4_{16}	4_{10}	4_{10}
		+	1	1	0	0	C_{16}	12_{10}	-4_{10}
0	1	=	0	0	0	0	0_{16}	16_{10}	0_{10}

5.1.6 Veranschaulichung am Zahlenkreis

Man kann sich sich sowohl die Addition im 2er Komplement als auch die Addition von Vorzeichenlosen Zahlen als eine Winkeladdition im Zahlenkreis Verdeutlichen, wie dei folgenden Beispiele zeigen:

Beispiel: 3+4 im 4 Bit 2er Komplement

V	C		a_3	a_2	a_1	a_0	Hex	unsigned	signed
			0	0	1	1	3_{16}	3_{10}	3_{10}
		+	0	1	0	0	4_{16}	4_{10}	4_{10}
0	0	=	0	1	1	1	7_{16}	7_{10}	7_{10}

Das Ergebnis ist sowohl in seiner vorzeichenlosen, als auch in seiner vorzeichenbehafteten Interpretation korrekt. Weder Carry noch Overflow sind gesetzt. (Siehe Abb. 5.2)

Beispiel: 3+6 im 4 Bit 2er Komplement

V	C		a_3	a_2	a_1	a_0	Hex	unsigned	signed
			0	0	1	1	3_{16}	3_{10}	3_{10}
		+	0	1	1	0	6_{16}	6_{10}	6_{10}
1	0	=	1	0	0	1	9_{16}	9_{10}	-7_{10}

Das Ergebnis ist nur in seiner vorzeichenlosen Interpretation korrekt, in seiner vorzeichenbehafteten Interpretation nicht. Overflow ist gesetzt, Carry nicht. (Siehe Abb. 5.3)

Beispiel: -3+(-4) im 4 Bit 2er Komplement

V	C		a_3	a_2	a_1	a_0	Hex	unsigned	signed
			1	1	0	1	D_{16}	13_{10}	-3_{10}
		+	1	1	0	0	C_{16}	12_{10}	-4_{10}
0	1	=	1	0	0	1	9_{16}	9_{10}	-7_{10}

Das Ergebnis ist nur in seiner vorzeichenbehafteten Interpretation korrekt, in seiner vorzeichenlosen Interpretation nicht. Carry sind gesetzt, Overflow nicht. (Siehe Abb. 5.4)

Beispiel: -3+(-6) im 4 Bit 2er Komplement

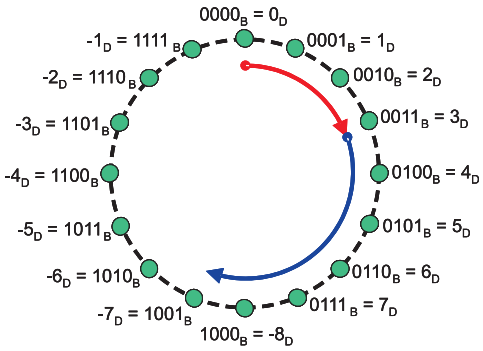


Abbildung 5.3: Beispiel: Veranschaulichung der Summe von 3+6 im 4-Bit 2er Komplement. Carry ist gelöscht, Overflow gesetzt.

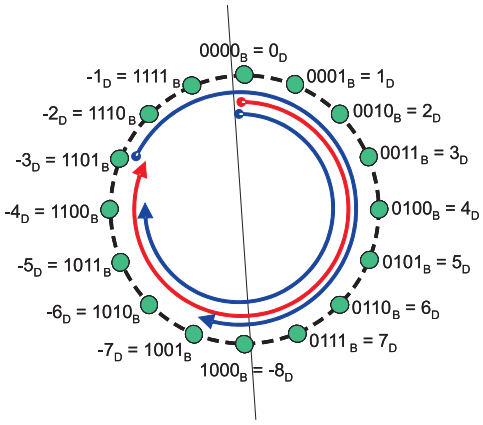


Abbildung 5.4: Beispiel: Veranschaulichung der Summe von -3+(-4) im 4-Bit 2er Komplement. Carry ist gesetzt, Overflow gelöscht.

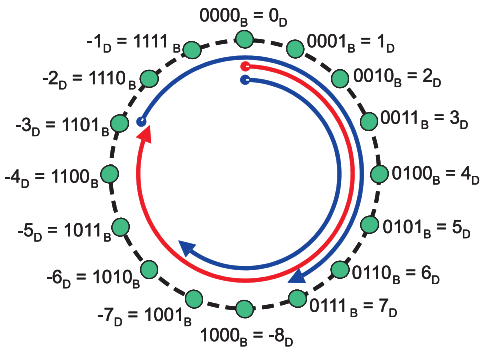


Abbildung 5.5: Beispiel: Veranschaulichung der Summe von (-3)+(-6) im 4-Bit 2er Komplement. Carry und Overflow sind gesetzt.

V	C		a_3	a_2	a_1	a_0	Hex	unsigned	signed
			1	1	0	1	D_{16}	13_{10}	-3_{10}
		+	1	0	1	0	A_{16}	10_{10}	-6_{10}
1	1	=	0	1	1	1	7_{16}	7_{10}	77_{10}

Das Ergebnis ist weder in seiner vorzeichenbehafteten Interpretation korrekt, noch in seiner vorzeichenlosen Interpretation. Carry und Overflow sind gesetzt. (Siehe Abb. 5.5)

5.2 Weitere Übungen

Wie viele vorzeichenlose Zahlen lassen sich mit 5 bit darstellen?

Wie viele vorzeichenlose Zahlen lassen sich mit 23 bit darstellen?

Wie viele Zahlen im Zweierkomplement lassen sich mit 5 bit darstellen?

Wie viele Zahlen im Zweierkomplement lassen sich mit 23 bit darstellen?

Wie viele Binärstellen werden benötigt, um eine 5-stellige Dezimalzahl darzustellen?

Wie viele Binärstellen werden benötigt, um eine10-stellige Dezimalzahl darzustellen?

5.3 Gebrochene Zahlen

Mit der Beschränkung des niedrigsten Exponenten auf 0 kann man nur Ganze Zahlen darstellen. Lässt man auch negative Exponenten zu, kann man auch rationale Zahlen in einem Stellenwertsystem schreiben, wobei der Übergang vom nichtnegativen zum negativen Exponenten durch ein Trennzeichen markiert wird, beispielsweise ein Komma: $1234,56 = 1 \cdot 10^3 + 2 \cdot 10^2 + 3 \cdot 10^1 + 4 \cdot 10^0 + 5 \cdot 10^{-1} + 6 \cdot 10^{-2}$ Die Ziffern einer rationalen Zahl p/q erhält man durch das Verfahren der schriftlichen Division. Im 10er-System spricht man auch von Dezimalbruch-Entwicklung. Hat q zur Basis b teilerfremde Primfaktoren, bricht die schriftliche Division nicht ab, sondern liefert eine sich wiederholende Folge von Ziffern. Diese wird Periode genannt und durch Überstreichen gekennzeichnet, z. B. Die Basis b muss nicht notwendigerweise eine natürliche Zahl sein. Es wurde nachgewiesen, dass sämtliche komplexen Zahlen mit Betrag größer 1 als Basis eines Stellenwertsystems verwendet werden können. Ebenso sind Zahlensysteme mit gemischten Basen möglich. Beispiele hierfür findet man in Knuth, The Art of Computer Programming. Eine andere Darstellung für rationale und irrationale Zahlen ist der Kettenbruch, welcher bessere Approximationen liefert als die Stellenwertsysteme.

5.3.1 Fixed Point

Alle Zahlensysteme, die wir bis jetzt diskutiert haben, haben das Komma rechts von der niederwertigsten Ziffer. Daher liegt eine 16 Bit vorzeichenlose Zahl zwischen und 65535. So ein Zahlen-

system ist geeignet Objekte zu zählen, wenn man aber den Sinus von verschiedenen Winkeln in einer Tabelle ablegen möchte ist es passender das Komma rechts von der höchstwertigen Ziffer zu platzieren. Der Wert einer n- stelligen Dualzahl wäre dann:

$$W = \sum_{i=0}^{n-1} 2^{-1 \cdot i} \cdot s_i \tag{5.4}$$

Beispiele für 16 stellige Fixed Point Zahlen mit dem Komma rechts von der höchstwertigen Ziffer:

$$\begin{aligned}
0.000000000000000 &= 0_{10} \\
1.100000000000000 &= 1.5_{10} &&= \frac{1}{1} + \frac{1}{2} \\
0.010000000000000 &= 0.25_{10} &&= \frac{1}{4} \\
1.000000000000000 &= 1.0_{10} &&= \frac{1}{1} \\
1.111111111111111 &= 1.9999969482421875_{10} &&= \frac{1}{1} + \frac{1}{2} + \frac{1}{4} \cdots \frac{1}{32768} \\
0.000000000000001 &= 0.000030517578125_{10} &&= \frac{1}{32768}
\end{aligned} \tag{5.5}$$

Das Komma kann natürlich auch an jeder anderen Stelle festgelegt werden. Letztendlich handelt es sich hier um ganz normale Dualzahlen, die im Computer gespeichert sind, deren Wertebereich durch einen konstanten Faktor angepasst wird. Im obigen Beispiel ist dieser Faktor 2^{-15}. Bei der Addition können wegen des Distributivgesetzes diese Fixed Point Zahlen behandelt werden wie normale Dualzahlen. Bei der Multiplikation muss der konstante Faktor noch einmal dazu multipliziert werden. Ohne Festkommaarithmetik wären viele Kaufmännischen Anwendungen undenkbar. Unabhängig davon wie groß der Skalierungsfaktor ist, das Verhältnis der größten darstellbaren Zahl zur kleinsten darstellbaren Zahl ist immer $2^n - 1$

5.3.2 Floating Point

Um dieses Verhältnis zu vergrößern bedient man sich der Gleitkommazahlen. Hier wird mit jeder Zahl explizit der Skalierungsfaktor gespeichert. Das Prinzip entspricht etwa der wissenschaftlichen Notation von Dezimalzahlen: Für die Masse eines Elektrons sagt man sie beträgt $9,1 \cdot 10^{-28}$ g, die Masse der Sonne $1,99 \cdot 10^{33}$ g beträgt. In dieser Notation legt man in einer Zahl (in dem Exponenten der 10er Potenz) die Größenordnung der Zahl fest. In der anderen Zahl (in der Mantisse) gibt man den „genauen" Wert an. Die Genauigkeit wird durch die Nachkommastellen festgelegt.

IEEE Standard 754-1985

Nach dem in den 70 Jahren jeder Computerhersteller seine eigene Fließkomma Arithmetik implementierte (DEC arbeitete mit Mantissen zwischen 0.5 und 1.0, IBM benutzte Potenzen zur Basis 16 für den Skalierungsfaktor) und somit binäre Daten praktisch nicht austauschbar waren, beschäftigte sich die IEEE (Institute of Electrical and Electronic Engineers) fast 3000 Tage und Nächte (von 1977-1985) mit einer Vereinheitlichung. Das Ergebnis hatte folgende Eckdaten:

Wortgröße Die Floating Point Zahlen sind für 32, 64 und 96 Bit definiert. Im Folgenden beschränken wir uns auf 32 Bit.

Betrag Mantisse Der Betrag der Mantisse wird in 24 Bit gespeichert. Er liegt zwischen 1 und $2 - 2^{-23}$. Das ist ähnlich, wie in der wissenschaftlichen Schreibweise im Dezimalsystem: Die Mantisse ist immer größer gleich 1,0 und kleiner als 10. Wenn die Mantisse nicht in diesem Bereich liegt, wird sie mit der Zehnerpotenz multipliziert, die sie in diesen Bereich bringt. Diese Zehnerpotenz wird entsprechend im Exponenten verrechnet. Es handelt sich bei der

Mantisse also um eine Festkommazahl, deren Komma rechts von der höchstwertigen Ziffer liegt. Im IEEE Format ist es entsprechend eine Festkommazahl, deren Komma rechts von der höchstwertigen Ziffer liegt. Wegen des Wertebereichs ist das höchstwertige Bit immer gesetzt. Ein Bit das immer gesetzt ist, braucht nicht gespeichert zu werden. Man kann es sich einfach dazu denken. So ist es auch im IEEE Format.

Vorzeichen Mantisse Das Vorzeichen der Mantisse ist in einem Bit gesondert gespeichert. Die Mantisse ist also als Betrag mit Vorzeichen gespeichert.

Exponent Der Exponent ist in der 8-Bit Excess 127 Darstellung gespeichert. Die Zahl 00000001_2 entspricht also der -126_{10}, die 11111110_2 entspricht dann der 127_{10} Die Zahlen 00000000 und 11111111 sind für andere Zwecke reserviert.

Die Mantisse ist in den niederwertigen 23 Bit $f_{22}, f_{21}, \cdots, f_0$ gespeichert, der Exponent in den Bits $f_{30}, f_{29}, \cdots, f_{23}$ und das Vorzeichen in dem Bit f_{31}. Der Wert errechnet sich dann folgendermaßen:

$$W = (-1)^{f_{31}} \cdot (1 + \sum_{i=0}^{22} f_i \cdot 2^{i-23}) \cdot 2^{(\sum_{i=23}^{30} f_i \cdot 2^{i-23})-127} \tag{5.6}$$

Beispiele

0	1000	0000	100	0000	0000	0000	0000	0000

$$W = (-1)^0 \cdot (1 + 1 \cdot 2^{-1}) \cdot 2^{(1 \cdot 2^7)-127} = 3_{10} \tag{5.7}$$

0	1000	0000	000	0000	0000	0000	0000	0000

$$W = (-1)^0 \cdot 1 \cdot 2^{(1 \cdot 2^7)-127} = 2_{10} \tag{5.8}$$

1	1000	0000	000	0000	0000	0000	0000	0000

$$W = (-1)^1 \cdot 1 \cdot 2^{(1 \cdot 2^7)-127} = -2_{10} \tag{5.9}$$

0	1000	0001	101	0000	0000	0000	0000	0000

$$\begin{aligned} W &= (-1)^0 \cdot (1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-3}) \cdot 2^{(1 \cdot 2^7 + 1 \cdot 2^0)-127} \\ &= 1,625 \cdot 2^{129-127} = 6,5_{10} \end{aligned} \tag{5.10}$$

1	0111	1110	000	0000	0000	0000	0000	0000

$$\begin{aligned} W &= (-1)^1 \cdot 1 \cdot 2^{(1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1)-127} \\ &= -1 \cdot 2^{126-127} = -0,5_{10} \end{aligned} \tag{5.11}$$

0	1000	0001	010	0000	0000	0000	0000	0000

$$\begin{aligned} W &= (-1)^0 \cdot (1 + 1 \cdot 2^{-2}) \cdot 2^{(1 \cdot 2^7 + 1 \cdot 2^0)-127} \\ &= 1,25 \cdot 2^{129-127} = 5_{10} \end{aligned} \tag{5.12}$$

0	0111	1111	101	0000	0000	0000	0000	0000

$$\begin{aligned} W &= (-1)^0 \cdot (1 + 1 \cdot 2^{-1} + 1 \cdot 2^{-3}) \\ &\cdot 2^{(1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)-127} \\ &= 1,625 \cdot 2^0 = 1,625_{10} \end{aligned} \tag{5.13}$$

Zahlenbereich Denn Zahlenbereich kann man sich verdeutlichen, wenn man die größte und die kleinste darstellbare Zahl betrachtet:

Größte normalisierte Zahl (Anmerkung: e darf nicht 0 oder 255 sein !) 0 11111110 11111111111111111111111 $= 2^{127} \cdot 1,999999881 = 3.40282347 \cdot 10^{38}$

Kleinste positive normalisierte Zahl 0 00000001 00000000000000000000000 $= 2^{-126} \cdot 1,0 = 1.17549435 \cdot 10^{-38}$

Wie man sieht kann man in diesem Format mit 32 Bit sowohl die Masse von 100000 Sonnen in Gramm darstellen, als auch die Masse der Bruchstücke eines Elektrons in Gramm, wenn man es in 10 Milliarden gleiche Teile zerlegt hat. Und das auf 0,00001 % genau.

Folgende Sonderfälle sind definiert:

$f_{30} \cdots f_{23} = 0, f_{22} \cdots f_0 = 0$ Zahl ist $\pm$ 0

$f_{30} \cdots f_{23} = 0, f_{22} \cdots f_0 \neq 0$ Zahl ist denormalisiert, das gedachte erste Bit der Mantisse ist 0, dadurch kann man noch kleinere Zahlen darstellen.

$f_{30} \cdots f_{23} = 255, f_{22} \cdots f_0 = 0$ Zahl ist $\pm\infty$.

$f_{30} \cdots f_{23} = 255, f_{22} \cdots f_0 \neq 0$ Zahl ist keine vernünftige Fließkommazahl. Wer versucht mit solchen Zahlen in einem Fließkommaprozessor zu rechnen handelt sich sofort eine Fehlermeldung ein („not a number" (NaN)) .

Beim Rechnen mit Fließkomma kann man leicht schlechte Ergebnisse erhalten, wenn man sehr große und sehr kleine Zahlen vergleicht. Kein vernünftiger Mensch kommt auf die Idee einmal die Sonne zu wiegen, dann die Sonne und ein Elektron zu wiegen, die Ergebnisse zu subtrahieren und zu hoffen dann die korrekte Elektronenmasse zu erhalten. So etwas sollte man als Programmierer auch vermeiden.

5.4 Andere Informationen

5.4.1 Texte-ASCII

ASCII (American Standard Code for Information Interchange) ist ein 1968 von Bob Bemer eingeführter standardisierter Zeichensatz zur Textdarstellung für Computer und andere Kommunikationseinrichtungen, welcher auf dem lateinischen Alphabet, wie es im modernen Englisch benutzt wird, basiert.

ASCII beschreibt als Code die Zuordnung von digital dargestellten Ganzzahlen zu den in der normalen Schriftsprache geschriebenen Zeichen. Mit Hilfe des Codes können digitale Geräte Textinhalte als Zahlenfolgen senden, empfangen und verarbeiten. Da diese Geräte intern nur Zahlen verarbeiten, ist ein solcher Code zur Nutzung von Textinhalten unbedingt erforderlich.

Geschichte

Eine der frühsten Formen der Digitalisierung war der Morsecode. Der Morsecode wurde mit der Einführung von Fernschreibern aus den Telegrafennetzen verdrängt und durch den Baudot-Code und Murray Code ersetzt. Vom Fünf-Bit-Murray-Code zum Sieben-Bit-ASCII war es dann nur noch ein kleiner Schritt - auch ASCII wurde zuerst für bestimmte amerikanische Fernschreiber-Modelle, wie den Teletype ASR33, eingesetzt. In den Anfängen des Computerzeitalters entwickelte sich ASCII zum Standard-Code für Schriftzeichen. Als Beispiel wurden viele Bildschirme (z. B. VT100) und Drucker nur mit ASCII angesteuert.

ASCII diente ursprünglich der Darstellung von Schriftzeichen der englischen Sprache. Die erste Version, noch ohne Kleinbuchstaben und mit kleinen Abweichungen vom heutigen ASCII, entstand im Jahr 1963. Im Jahr 1968 wurde dann der bis heute gültige ASCII-Code festgelegt. Um später auch Sonderzeichen anderer Sprachen darstellen zu können (z. B. deutsche Umlaute), wurde der

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	np	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!		#	$	%	&	
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	I	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

00_H - $1F_H$: Steuer - zeichen

$\$20_H$: Leer - zeichen

ab 30_H: Ziffern

ab 41_H: Gross - buchstaben

ab 61_H: Klein - buchstaben

Abbildung 5.6: Die ASCII Tabelle

Code von sieben auf acht Bit erweitert. Allerdings bot auch der Acht-Bit-Code zu wenig Platz, um alle Sonderzeichen gleichzeitig unterzubringen, wodurch mehrere verschiedene Erweiterungen (siehe unten) notwendig wurden. Keine dieser Acht-Bit-Erweiterungen sollte aber als ASCII bezeichnet werden, um Verwirrung zu vermeiden (ASCII bezeichnet nur den einheitlichen Sieben-Bit-Code). Auch war es nicht möglich z. B. die tausenden chinesischen Schriftzeichen mit einem Acht-Bit Code darzustellen. Das führte später zu Unicode, einem Zeichensatz, der inzwischen einen Großteil aller Schriftzeichen der Menschheit enthält. Unicode hat ASCII heute in vielen Bereichen abgelöst.

ASCII beschreibt einen Sieben-Bit-Code. Dieser Code verwendet binäre Ganzzahlen, die mit sieben binären Ziffern dargestellt werden (entspricht 0 bis 127), um Informationen darzustellen. Schon früh haben Computer mehr als 7 Bits, oft mindestens Acht-Bit-Zahlenworte, verwendet. Das achte Bit kann für Fehlerkorrekturzwecke (Paritätsbit) auf den Kommunikationsleitungen oder für andere Steuerungsaufgaben verwendet werden; heute wird es aber fast immer zur Erweiterung von ASCII auf einen der diversen Acht-Bit-Codes verwendet.

Fortschritte in der Technik und die internationale Verbreitung erzeugten eine Reihe von Variationen und Erweiterungen des Codes, die nicht alle untereinander kompatibel sind und nicht für alle Systeme gleichermaßen verwendet werden können.

Zusammensetzung

Die ersten 32 ASCII-Zeichencodes (von 00 bis 1F) sind für Steuerzeichen (control character) reserviert. Dies sind Zeichen, die keine Schriftzeichen darstellen, sondern die zur Steuerung von solchen Geräten dienen (oder dienten), die ASCII verwenden (etwa Drucker). Steuerzeichen sind beispielsweise der Wagenrücklauf für den Zeilenumbruch oder Bell (die Glocke); ihre Definition ist historisch begründet.

Code 0x20 (SP) ist das Leerzeichen (engl. space oder blank), welches in einem Text als Leer- und Trennzeichen zwischen Wörtern verwendet und auf der Tastatur durch die große breite Leertaste erzeugt wird.

Die Codes 0x21 bis 0x7E sind alle druckbaren Zeichen, die sowohl Buchstaben, Ziffern und Satzzeichen (siehe Abbildung 5.6) enthalten.

Code 0x7F (alle sieben Bits auf eins gesetzt) ist ein Sonderzeichen, welches auch als „Löschzeichen” bezeichnet wird (DEL). Dieser Code wurde früher wie ein Steuerzeichen verwendet, um auf Lochstreifen oder Lochkarten ein bereits gelochtes Zeichen nachträglich durch das Setzen aller

Bits, d. h. durch Auslochen aller sieben Markierungen, löschen zu können. Einmal vorhandene Löcher kann man schließlich nicht wieder rückgängig machen.

Erweiterungen

ASCII enthält keine diakritischen Zeichen, die in fast allen Sprachen auf der Basis des lateinischen Alphabets verwendet werden.

Der internationale Standard ISO 646 (1972) war der erste Versuch, dieses Problem anzugehen, was allerdings zu Kompatibilitätsproblemen führte. Er ist immer noch ein Sieben-Bit-Code und weil keine anderen Codes verfügbar waren wurden einige Codes in neuen Varianten verwendet.

So ist etwa die ASCII-Position 93 für die rechte eckige Klammer (]) in der deutschen Zeichensatz-Variante ISO 646-DE durch das große U mit Trema (Umlaut) (Ü) und in der dänischen Variante ISO 646-DK durch das große A mit Ring (Krouzek) (Å) ersetzt. Bei der Programmierung mussten die eckigen Klammern durch die entsprechenden nationalen Sonderzeichen ersetzt werden. Dies führte oft zu ungewollt komischen Ergebnissen, indem etwa die Einschaltmeldung des Apple II von „APPLE][" zu „APPLE ÜÄ" mutierte.

Verschiedene Hersteller entwickelten eigene Acht-Bit-Codes. Der Codepage 437 genannte Code war lange Zeit der am weitesten verbreitete, er kam auf dem IBM-PC unter MS-DOS, und heute noch in DOS- oder Eingabeaufforderungs-Fenstern von MS-Windows, zur Anwendung.

Auch bei späteren Standards wie ISO 8859 wurden acht Bits verwendet. Dabei existieren mehrere Varianten, zum Beispiel ISO 8859-1 für die westeuropäischen Sprachen, welches in MS-Windows (außer DOS-Fenster) Standard ist, daher sehen z. B. bei unter DOS erstellten Textdateien die deutschen Umlaute falsch aus, wenn man sie unter Windows ansieht. Viele ältere Programme, die das achte Bit für eigene Zwecke verwendeten, konnten damit nicht umgehen. Sie wurden im Laufe der Zeit oft den neuen Erfordernissen angepasst.

Um den verschiedenen Anforderungen der verschiedenen Sprachen gerecht zu werden, wurde der Unicode (in seinem Zeichenvorrat identisch mit ISO 10646) entwickelt. Er verwendet bis zu 32 Bit pro Zeichen und könnte somit über vier Milliarden verschiedene Zeichen unterscheiden. Dies wird jedoch auf etwa 1 Million erlaubte Code-Werte eingeschränkt. Damit können alle bislang von Menschen verwendeten Schriftzeichen dargestellt werden, sofern sie denn in den Unicode-Standard aufgenommen wurden. UTF-8 ist eine 8-Bit-Kodierung von Unicode, die mit ASCII abwärtskompatibel ist. Ein Zeichen kann dabei ein bis sechs 8-Bit-Wörter einnehmen. Sieben-Bit-Varianten müssen nicht mehr verwendet werden. Dennoch kann Unicode auch mit Hilfe von UTF-7 in 7 Bit kodiert werden. UTF-8 entwickelt sich zur Zeit (2005) zum einheitlichen Standard unter den meisten Betriebssystemen. So nutzen unter anderem einige Linux-Distributionen UTF-8 standardmäßig, und immer mehr Webseiten werden in UTF-8 ausgeliefert.

ASCII enthält nur wenige Zeichen, die allgemein verbindlich zur Formatierung oder Strukturierung von Text verwendet werden; diese gehen aus den Steuerbefehlen der Fernschreiber hervor. Hierzu zählen insbesondere der Zeilenvorschub (Linefeed), der Wagenrücklauf (Carriage Return), der horizontale Tabulator, der Seitenvorschub Form Feed) und der vertikale Tabulator. In typischen ASCII-Textdateien findet sich neben den druckbaren Zeichen meist nur noch der Wagenrücklauf oder der Zeilenvorschub, um das Zeilenende zu markieren, wobei in DOS- und Windows-Systemen üblicherweise beide nacheinander verwendet werden, bei Apple- und Commodore-Rechnern nur der Wagenrücklauf, auf Unix-artigen Systemen nur der Zeilenvorschub. Die Verwendung weiterer Zeichen zur Textformatierung ist bei verschiedenen Anwendungprogrammen zur Textverarbeitung unterschiedlich. Zur Formatierung von Text werden heute auch verstärkt Markup-Codes wie z. B. HTML verwendet.

5.4.2 Farben & Töne

Im Computer werden aber nicht nur Zahlen und Buchstaben behandelt, im zunehmenden Maße werden auch audiovisuelle Daten im Computer verarbeitet. Genau wie bei den Texten werden hier einzelnen Bildpunkten Zahlenwerte zugeordnet, die dann etwa der Helligkeit oder der Intensität

eines einzelnen Farbtons entsprechen. Auch bei Tönen werden einzelnen Noten Zahlen zugeordnet (MIDI) oder dem Luftdruck zu einem bestimmten Zeitpunkt.