Rechnerarchitektur, Foliensatz 1

G. Kemnitz

12. Januar 2017

Contents

1 Einführung

Lernziel und Strukturierung der Lehrveranstaltung

- Funktionsweise von und Befehlsabarbeitung in Rechnern.
- Nachbildung typischer Programmierelemente durch Maschinenbefehle.
- Programmieren, disassemblieren und hardware-nahes debuggen unter einer IDE.
- Hardware-Funktionen und ihre programmtechnische Nutzung.

Beispielrechner werden sein:

- Simulationsmodell eines Minimalprozessors zur Demonstration der internen Befehlsabarbeitung.
- 8-Bit-AVR-Prozessor auf einer Versuchsbaugruppe. Wesentlich komplexere Funktion, Programmierung in C und Assembler. Praktische Programmieraufgaben und Experimente.

Die Übungen finden im Labor statt.

Organisation, Leistungsnachweis

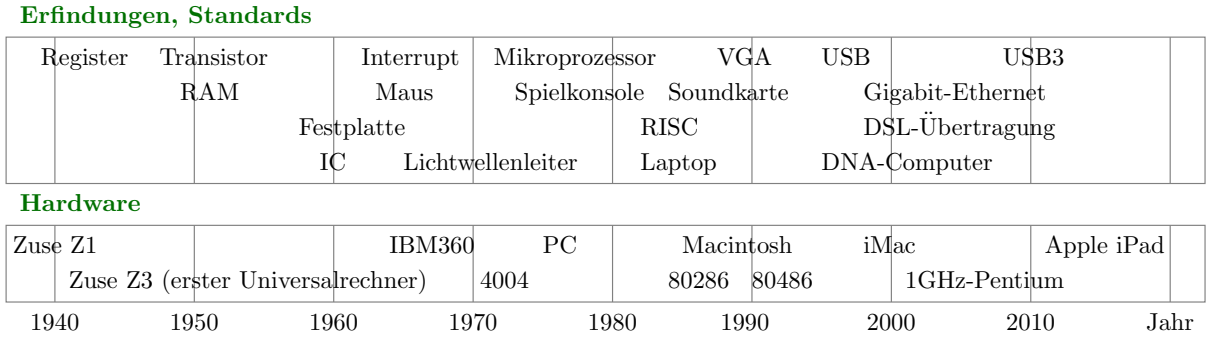
- Ab 15.12.2016 (8. Vorlesungswoche) jede Woche Vorlesung.
- Jede Woche Hausübungen (Abgabe zur nächsten Vorlesung).
- Ab der 2. Vorlesung (22.12.2016) Test am Vorlesungsende über den Inhalt der abgegebenen Hausübung.
- Ab 9. Vorlesungswoche (19./21.12.2016) jede Woche Übung im Labor in zwei Gruppen.

Leistungsnachweis, Erwerb:

- In allen bis auf einem Kurztests mindestens 40% und insgesamt mindesten 50% der Punkte und
- in allen bis auf einer Laborübung mindestens 60% der Punkte.

Alternativ[1] mündliche Kenntnisprüfung über den gesamten Übungs- und Vorlesungsstoff[2].

Entwicklungsgeschichte der Rechentechnik[3]



- Informatik und Rechner gibt es erst seit 60 Jahren.
- Aktuelle Prozessoren haben bis zu Miliarden Transistoren und können viele Milliarden Operationen pro Sekunde ausführen.
- Nach dem Moore'schen-Gesetz[4] weitere exponentielle Zunahme.

Die interne Befehlsabarbeitung eines Rechners lässt sich am einfachsten an einem Simulationsmodell, dessen Zustand auf eine Textzeile passt, veranschaulichen. Reale Rechner mit ihren vielen Zusatzfunktionen haben hunderte Seiten Benutzerdokumentationen, sind nur mit Entwicklungswerkzeugen (Compiler, Debugger, ...) nutzbar, ... Die Vorlesung wird deshalb 2 Rechner parallel behandeln:

[1] Auch bei entschuldigter Abwesenheit z.B. wegen Krankheit.
[2] Termine beim Dozent erfragen. Prüfungstermine bis max. Ende Februar.
[3] [http://de.wikipedia.org/wiki/Computer#Entwicklung_des_ modernen_turingm.C3.A4chtigen_Computers]

[4] Gordon Moore, Mitbegründer von Intel, prognostizierte in den sechziger Jahren eine jährliche Verdopplung der Transistoranzahl von integrierten Schaltungen. Bisher hat sich diese Prognose erfüllt.

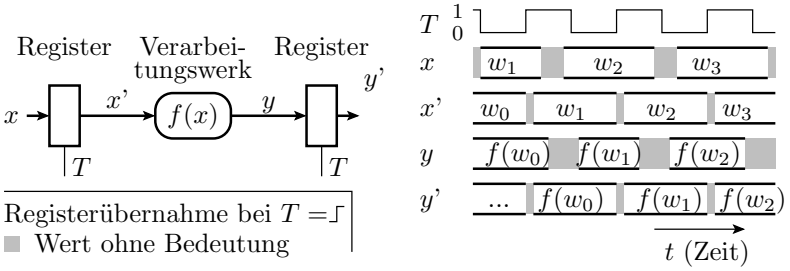
1. MiPro: Simulationsmodell eines Minimalrechners zur Untersuchung der prinzipiellen Funktionsweise eines Rechners.
   - Bitniveau, Register-Transfer-Ebene,
   - Abarbeitung von Maschinenbefehlen und
   - Nachbildung elementarer Berechnungen mit Maschinenbefehlen.
2. ATmega 2560: Realer 8-Bit-Mikrorechner auf einer Versuchsbaugruppe mit Entwicklungsumgebung:
   - Größere Adressbereiche. Ein- und Ausgabeeinheiten. ...
   - Programmieren in einer Hochsprache (C).
   - Disassemblieren und hardware-nahes Debuggen.
   - Programmierung von EA-Schnittstellen.

2 Rechner

2.1 Befehlsabarbeitung

Register und Verarbeitungswerke

Ein Rechner ist eine digitale Schaltung. Er besteht aus Registern und Verarbeitungswerken.



- Register übernehmen, wenn freigegebe, die Bitvektoren an ihren Eingängen mit der aktiven Taktflanke (im Bild der steigenden) und speichern sonst.
- Verarbeitungswerke bilden aus Eingaben verzögert Ergebnisse, z.B. die Summe von zwei Bitvektoren.

Register und Verarbeitungswerke bestehen wiederum aus Gattern mit den logischen Operationen:

- $a \wedge b$: UND, Ergebnis 1 wenn beide Operanden 1 sind,
- $a \vee b$: ODER, Ergebnis 1 wenn mindestens ein Operand 1 ist,
- $a \oplus b$: EXOR, Ergebnis 1 wenn genau ein Operand 1 ist,
- $\bar{a}$: Negation, Ergebnis 1, wenn der Operand 0 ist,

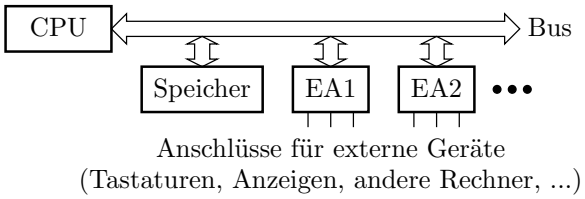
und Ausdrücken dieser Operationen, z.B. ein Umschalter:

$$y = (x_0 \wedge \bar{s}) \vee (x_1 \wedge s)$$

Gatter sind aus Transistoren aufgebaut, die als schnelle gesteuerte Schalter arbeiten[5] (Schaltzeiten Nanosekunden und weniger, sie).

Rechner führen einfache Operationen aus (Zählen, Addieren, bitweise Logikoperationen, ...), davon aber sehr viele pro Zeit.

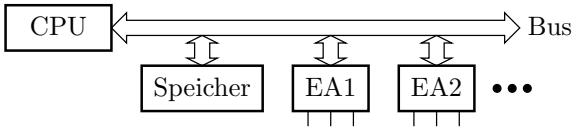
Von-Neumann-Architektur[6]



Ein Universalrechner besteht aus

- Prozessor (CPU **C**entral **P**rocessing **U**nit),
- einem adressierbaren Speicher mit den Befehlen und Daten und
- Ein- / Ausgabegeräten, auf die der Rechner wie auf einen Speicher zugreift, deren Speicherinhalte jedoch zusätzlich von externen Geräten gelesen und verändert werden.
- Bussen für den Datenaustausch.

Befehlsabarbeitung



Schritte der Befehlsabarbeitung:

- IF (**I**nstruction **F**etch) Befehl holen: Befehlsadresse senden, Befehlswort aus dem Befehlsspeicher holen, Adresse erhöhen.
- OF (**O**perand **F**etch) Operanden holen: Für alle Operanden Operandenadresse senden, Daten aus dem Datenspeicher holen.
- EX (**Ex**ecute) Operationsausführung: Operanden an das Rechenwerk anlegen. Ergebnis zwischenspeichern.
- RW (**R**esult **W**rite) Ergebnis schreiben: Ergebnisdaten und -adresse an den Datenspeicher schicken.

[5] Wird im 2. Semester in der Veranstaltung »Entwurf digitaler Schaltungen« behandelt.
[6] John von Neumann: First Draft of a Report on the EDVAC. 1945

RISC-Prozessoren

Die in den letzten 30 Jahren neu entwickelten Prozessoren haben eine RISC[7]-Architektur. Die Befehlssätze sind (überwiegend) auf Befehle reduziert, die in einem Schritt[8] abarbeitbar sind:

- Verlangt u.a. Parallelzugriff auf einen Befehls- und drei Datenspeicherplätze (Lesen vom Befehlswort und zwei Operanden, Schreiben eines Ergebnisses).
- Op.-Code und alle Adressen müssen in ein Befehlswort passen.
- ...

Komplexe Operationen, die mehrere Schritte für die Ausführung oder größere Befehlsworte benötigen, setzt der Compiler aus mehreren Befehlen zusammen.

Ein Prozessor ist nicht besser als sein Compiler.

2.2 Befehlsformate

Befehlsformate des Minimalprozessors

Der Minimalprozessor hat 16-Bit Befehlsworte, 8-Bit-Datenworte und 8-Bit-Adressen, für Befehle und Daten getrennt.

Teilbitvektor	15 ... 11	10 9 8	7 6 5	4 3 2	1 0	cnr
nop:	00000					0
jump imm,cond	00001	cond	imm			1
cmd rd,imm	cnr	rd	imm			2 bis 14
cmd rd	cnr	rd				15
cmd rd,ra	cnr	rd	ra			16 bis 21
cmd rd,ra,rb	cnr	rd	ra	rb		22 bis 28

- cnr: Befehlsnummer zur Unterscheidung der Befehle, 5 Bit.
- rd, ra, rb: Registeradressen, je 3 Bit
- imm (**Imm**ediate) Direktwert: Konstante, 8 Bit.
- cond (**Cond**ition): Sprungbedingung, 3 Bit.

Die restlichen 2 bis 11 Bits sind ungenutzt.

Beispielaufgabe

1. Wie viele Befehle können maximal mit einer 5-Bit-Befehlsnummer unterschieden werden?
2. Wie viele Register sind mit 3 Bit adressierbar?
3. Wie viele Speicherplätze kann der Befehls- bzw. Datenspeicher bei 8-Bit-Adressen maximal haben?
4. Kann ein Programm auf einem 8-Bit-Prozessor auch Datenojekte aus mehreren Bytes verarbeiten, z.B. Zeichenketten variabler Länge?

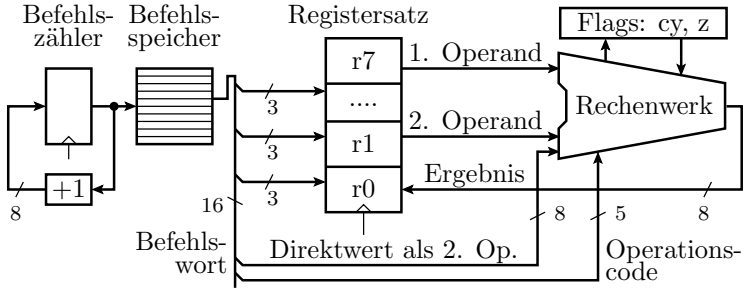
[7] RISC ist ein Akronym für **R**educed **I**nstruction **S**et **C**omputer.
[8] Genauer in einer Pipeline-Zeitscheibe (siehe später Foliensatz RA_F2).

Lösung Aufgabenteil 1

1. Mit einer 5-Bit-Befehlsnummer lassen sich max. 32 Befehle unterscheiden.
2. Mit 3 Bit lassen sich max. 8 Register adressieren.
3. Bei 8-Bit-Adressen ist die Anzahl der adressierbaren Befehle und Datenbytes auf 256 beschränkt.
4. Die Verarbeitung von Datenobjekten aus mehreren Bytes ist möglich, und zwar nacheinander mit mehreren Befehlen.

2.3 Minimalprozessor MiPro

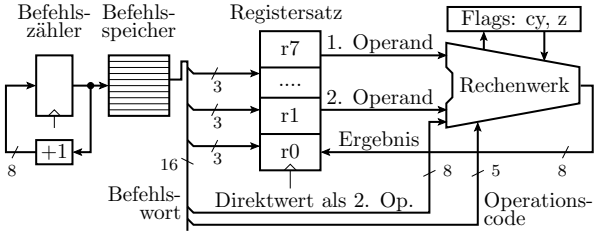
Die Hardware für Verarbeitungsbefehle



Die Hardware für Verarbeitungsbefehle besteht aus

- Befehlszähler, der nach jedem Takt um eins weiterzählt,
- Befehlsspeicher, der das Befehlswort zur Befehlsadresse liefert,
- 8-Bit-Registersatz, der für bis zu 2 Operanden Daten liefert und zum Abschluss der Befehlsausführung das Ergebnis übernimmt,
- Rechenwerk (Ergänzungen für weitere benötigte Befehle folgen).

Beispiel für Verarbeitungsbefehle



Die bisherige Hardware bildet mit einem Befehl ein Ergebnis, z.B.:

```
Befehl          |Operation
andr rd,ra,rb   |rd := ra and rb   (bitweises UND)
andi rd,imm     |rd := rd and imm
ld_i rd,imm     |rd := imm        (Lade Konstante)
move rd,ra      |rd := ra         (Kopiere Register)
```

Sie kann keine Daten aus dem Speicher Lesen, ...

2.4 AVR-Befehlsformate

AVR-Befehlsformate

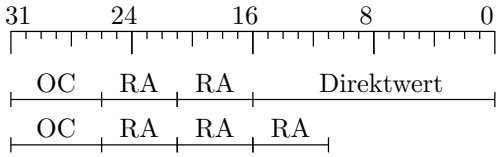
Gleichfalls 16-Bit-Befehls- und 8-Bit-Datenworte.

Operation	Op.-Code	Assembler
Rd←Rd∧Rr	0010 00rd dddd rrrr	and Rd, Rr
Rd←Rd∧k	0111 kkkk dddd kkkk	andi Rd, k
Rd←I/O(A) (Eingabe)	1011 0AAd dddd AAAA	in Rd, A
Rd←dmem(k)	1001 000d dddd 0000 kkkk kkkk kkkk kkkk	lds Rd, k
I/O(A) ←Rr (Ausgabe)	1011 1AAr rrrr AAAA	out A, Rd

- max zwei 5-Bit-Registeradressen[9],
- 16-Bit-Datenspeicheradressen, 17/18-Bit-Befehlsspeicheradr.,
- Unterschiedliche Bitanzahl für die Befehlsunterscheidung, ...

Die Befehlssatzbeschreibung allein ist 160 A4-Seiten lang, die Applikationsbeschreibung des Prozessors einige hundert Seiten, die Beschreibung der internen Funktion ist nicht veröffentlicht, ...

32- und 64-Bit-Architekturen



Das ideale Befehlsformat für RISC-Prozessoren ist 32-Bit:

- 6 Bit Operationscode für die Befehlsauswahl,
- bis zu drei 5-Bit-Registeradressen,
- 16-Bit-Direktwert und 32-Bit-Befehls- und Datenadressen.

Datenadressierung byteweise. Befehle dürfen nur auf durch 4 teilbaren Adressen stehen.

64-Bit-Prozessoren haben auch 32-Bit-Befehlsworte, aber 64-Bit-Adressen. Laden, Speichern Verarbeitung bis 8 Byte zeitgleich.

Beispielaufgaben

1. Wie viele Register sind mit 5-Bit-Adressen auswählbar?
2. Wie viele Befehls- und wie viele Datenspeicherplätze hat ein AVR-Prozessor mit 17-Bit-Befehls- und 16-Bit-Datenadressen?
3. Bis zu wie viele Datenbytes und bis zu wie viele Befehlsworte kann einen 32-Bit-Prozessor direkt adressieren?
4. Bis zu wie viele Datenbytes und bis zu wie viele Befehlsworte kann einen 64-Bit-Prozessor direkt adressieren?

[9] Eine Adresse ist zum Teil gleichzeitig Operanden- und Zieladresse.

Lösung

1. 5-Bit-Registeradressen: $2^5 = 32$ Register.
2. 17-Bit-Befehlsadresse: $2^{17} = 128\,\text{k}$ Befehlsworte. 16-Bit-Datenadressen: $2^{16} = 64\,\text{k}$Bytes Daten.
3. 32-Bit-Prozessor: $2^{32} = 4\,\text{GByte}$ Daten und 2^{30} Befehlsworte.
4. 64-Bit-Prozessor: $2^{64} = 4 \cdot 2^{32}\,\text{GByte}$ Daten und 2^{62} Befehlsworte.

3 Bitverarbeitung

3.1 Logik- und Shift-Befehle

Bitweise Logikbefehle

a	b	$a \wedge b$	$a \vee b$	$a \oplus b$	$\bar{b}$
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

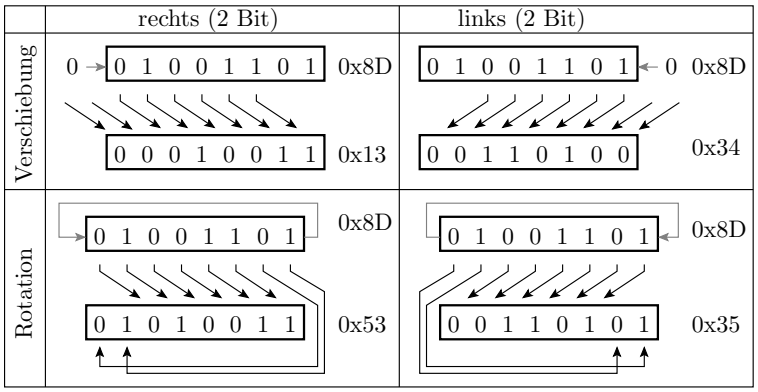
Beispiele:

$$\begin{array}{rl} a & 1011\ 0110\ \ (0\text{xB6}) \\ b & 1001\ 0011\ \ (0\text{x93}) \\ \hline a \wedge b & 1001\ 0010\ \ (0\text{x92}) \end{array} \qquad \begin{array}{rl} a & 1011\ 0110\ \ (0\text{xB6}) \\ b & 1001\ 0011\ \ (0\text{x93}) \\ \hline a \vee b & 1011\ 0111\ \ (0\text{xB7}) \end{array}$$

$$\begin{array}{rl} a & 1011\ 0110\ \ (0\text{xB6}) \\ b & 1001\ 0011\ \ (0\text{x93}) \\ \hline a \oplus b & 0010\ 0101\ \ (0\text{x25}) \end{array} \qquad \begin{array}{rl} b & 1001\ 0011\ \ (0\text{x93}) \\ \hline \bar{b} & 0110\ 1100\ \ (0\text{x6C}) \end{array}$$

- Einfachste mit Logikgattern realisierbare Funktionen.
- Nutzung innerhalb von Programmen z.B. zum Setzen, Löschen, Invertieren einzelner Bits in Datenobjekten.

Verschiebung und Rotation



- Verschiebung: Füllen freiwerdender Bits mit null.
- Rotation: Verschiebung im Kreis.

Bit- und Verschiebeoperationen in C

```
uint8_t a=0xB6, b=0x93, x = 0x4D, c;
...
c = a & b; // bitweises UND  (c: 0x92)
c = a | b; // bitweises ODER (c: 0xB7)
c = a ^ b; // bitweises EXOR (c: 0x25)
c =    ~b; // bitw. Negation (c: 0x6C)
c = x >> 2;// Rechtsverschiebung um 2 Bit (c: 0x13)
c = x << 2;// Linksverschiebung um 2 Bit  (c: 0x34)
c = (x>>2)|(x<<6);// Rechtsrot. um 2 Bit  (c: 0x53)
```

- Auch für Datenobjekte größer 8 Bit.
- Die Verschiebung kann auch eine Variable sein.

Logik- und Verschiebebefehle werden genutzt:

- Verarbeitung von Schaltereingaben,
- Verkettung und Trennen von Teilbitvektoren,
- Nachbildung der Multiplikation und Division, ...

3.2 Logikverarbeitung MiPro

Teilbitvektor	15 ... 11	10 9 8	7 6 5	4 3 2	1 0
cmd rd,ra,rb	cnr	rd	ra	rb	
cmd rd,ra	cnr	rd	ra		
cmd rd,imm	cnr	rd	imm		

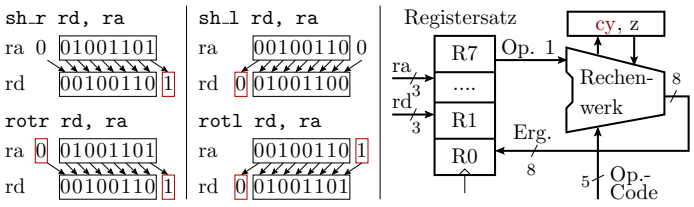
Logikbefehle des Minimalprozessors

Befehl	Operation	Flags	cnr
andr rd,ra,rb	rd := ra and rb	z	28
andi rd,imm	rd := rd and imm	z	12
or_r rd,ra,rb	rd := ra or rb	z	29
or_i rd,imm	rd := rd or imm	z	13
xorr rd,ra,rb	rd := ra xor rb	z	30
xori rd,imm	rd := rd xor imm	z	14
notr rd,ra	rd := not ra	z	23

Befehle zum Initialisieren und Kopieren von Registerinhalten:

ld_i rd,imm	rd := imm		5
move rd,ra	rd := ra		16

Verschiebe- und Rotationsbefehle



```
sh_r rd,ra     | rd := 0:ra  >> 1 | cy,z| 20
rotr rd,ra     | rd := cy:ra >> 1 | cy,z| 22
sh_l rd,ra     | rd := ra:0  << 1 | cy,z| 19
rotl rd,ra     | rd := ra:cy << 1 | cy,z| 21
```

- 0:ra – Verkettung von null und ra zu einem 9-Bit-Wert.
- 1-Bit-Verschiebung, rausgeschobenes Bit in cy.
- Verschiebebefehle übernehmen in das freiwerdende Bit null, Rotationsbefehle den bisherigen Wert von cy.

Beispielaufgabe

Welche Werte stehen nach Befehlsausführung in den Registern?

```
-----------------------------------------------
   Befehl     | Werte nach Befehlsausführung
..............|....r0...|....r1...|....r2...|c
ld_i  r0,45   |    .    |xxxx.xxxx|xxxx.xxxx|x
ld_i  r1,6E   |    .    |    .    |xxxx.xxxx|x
move  r2,r0   |    .    |    .    |    .    |
andr  r0,r0,r1|    .    |    .    |    .    |
notr  r0,r1   |    .    |    .    |    .    |
or_r  r2,r2,r1|    .    |    .    |    .    |
xorr  r0,r0,r2|    .    |    .    |    .    |
sh_l  r2,r0   |    .    |    .    |    .    |
rotl  r0,r1   |    .    |    .    |    .    |
-----------------------------------------------
```

(x – unbekannter Bitwert)

Lösung

```
-----------------------------------------------
   Befehl     | Werte nach Befehlsausführung
..............|....r0...|....r1...|....r2...|c
ld_i  r0,45   |0100.0101|xxxx.xxxx|xxxx.xxxx|x
ld_i  r1,6E   |0100.0101|0110.1110|xxxx.xxxx|x
move  r2,r0   |0100.0101|0110.1110|0100.0101|x
andr  r0,r0,r1|0100.0100|0110.1110|0100.0101|x
notr  r0,r1   |1001.0001|0110.1110|0100.0101|x
or_r  r2,r2,r1|1001.0001|0110.1110|0110.1111|x
xorr  r0,r0,r2|1111.1110|0110.1110|0110.1111|x
sh_l  r2,r0   |1111.1110|0110.1110|1111.1100|1
rotl  r0,r1   |1101.1101|0110.1110|1111.1100|0
-----------------------------------------------
```

3.3 AVR, 1. Programm

Programmentwicklung und Test

AVR-Programme werden in der Regel in C geschrieben und übersetzt. Zum Testen lassen sie sich schrittweise

- in C-Anweisungszeilen oder
- Assemblerzeilen (Maschinenbefehlen)

abarbeiten. Programmierung in Assembler möglich.

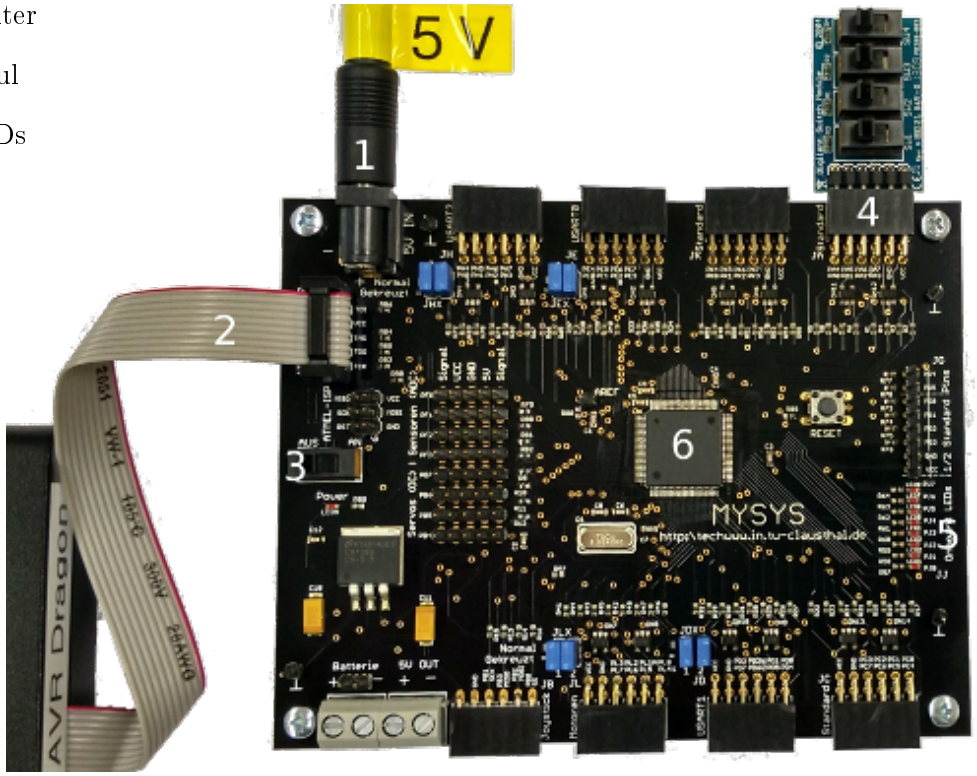
Programmentwicklung mit AVR-Studio:

- Projekt anlegen, Grundeinstellungen vornehmen.
- Programm eingeben.
- Übersetzen, Prozessor programmieren.
- Testen:
  - im Simulator
  - auf einer Versuchsbaugruppe

  jeweils im Schrittbetrieb, mit Haltepunkten oder freilaufend.

Versuchsboard

1. Anschluss 5V-Netzteil
2. Anschluss Programmer
3. Ein-Ausschalter
4. Schaltermodul
5. Ausgabe LEDs
6. Prozessor



Test mit Schaltereingabe und LED-Ausgabe

```
#include <avr/io.h>
uint8_t a, b, x, y;
int main(void){
  DDRA = 0x00;           // PORTA als Eingang
  DDRJ = 0xFF;           // PORTJ als Ausgang
  while(1){
    x = PINA;            // x(3:0) <= Schalter(3:0)
    a = 0b11 & x;        // a(1:0) <= Schalter(1:0)
    b = 0b11 & (x >> 2);// b(1:0) <= Schalter(3:2)
    y = (a & b) | (a|b)<<2 | (a^b)<<4 | ~a <<6;
    PORTJ = y;           // Ausgabe auf die 8 LEDs
  }
}
```

LED-Ausgabe (s_i – Wert von Schalter i):

LED8	LED7	LED6	LED5	LED4	LED3	LED2	LED1
$\bar{s}_1$	$\bar{s}_0$	$s_1 \oplus s_3$	$s_0 \oplus s_2$	$s_1 \vee s_3$	$s_0 \vee s_2$	$s_1 \wedge s_3$	$s_0 \wedge s_2$

AVR-Logikbefehle

AVR-Prozessoren haben wie der Minimalprozessor bitweise Logikbefehle, Verschiebe- und Rotationsbefehle um ein Bit und einige Spezialbitbefehle. Bitweise Logikbefehle:

Operation	Op.-Code	Assembler
Rd←Rd∧Rr	0010 00rd dddd rrrr	and Rd, Rr
Rd←Rd∧K	0111 kkkk dddd kkkk	andi Rd*, K
Rd←Rd∨Rr	0010 10rd dddd rrrr	or Rd, Rr
Rd←Rd∨K	0110 kkkk dddd kkkk	ori Rd*, K
Rd←Rd⊕Rr	0010 10rd dddd rrrr	eor Rd, Rr
RD ← $\overline{\text{Rd}}$	1001 010d dddd 0000	com Rd

(Rd – Zielregister und erster Operand; Rr – 2. Operand; K – 8-Bit-Konstante, * Nur anwendbar auf Register R16 bis R31).

ATmega-Verschiebebefehle

Operation	Operationscode	Assembler
C ← 0	1001 0100 1000 1000	clc
C ← 1	1001 0100 0000 1000	sec
C ← $b_7\,b_6\,b_5\,b_4\,b_3\,b_2\,b_1\,b_0$ ← 0	0000 11dd dddd dddd	lsl Rd
0 → $b_7\,b_6\,b_5\,b_4\,b_3\,b_2\,b_1\,b_0$ → C	1001 010d dddd 0110	lsr Rd
→ $b_7\,b_6\,b_5\,b_4\,b_3\,b_2\,b_1\,b_0$ → C	1001 010d dddd 0101	asr Rd
C ← $b_7\,b_6\,b_5\,b_4\,b_3\,b_2\,b_1\,b_0$ ← C	1001 11dd dddd dddd	rol Rd
C → $b_7\,b_6\,b_5\,b_4\,b_3\,b_2\,b_1\,b_0$ → C	1001 010d dddd 0111	ror Rd
$b_7\,b_6\,b_5\,b_4\,b_3\,b_2\,b_1\,b_0$	1001 010d dddd 0010	swap Rd

(clc – Clear Carry Flag; sec – Set Carry Flag; lsl – Logical Shift Left, identisch mit add Rd, Rd; rol – Rotate Left, identisch mit adc Rd, Rd)

Dissassemblierte Programmausschnitte

```
   uint8_t   a,       b,       x,       y; ...
//Adressen: &a=0x203 &b=0x200 &x=0x201 &y=0x202
   while(1){
    x = PINA;           // x(3:0) <= Schalter(3:0)
    a = 0b11 & x;
//  00098  LDS R24,0x0201   r24 = x     //&x=0x201
//  0009A  ANDI R24,0x03    r24 = r24 & 0b11
//  0009B  STS 0x0203,R24   a   = r24   //&a=0x203
    b = 0b11 & (x>>2);
//  0009D  LDS R24,0x0201   r24 = x     //&x=0x201
//  0009F  LSR R24          r24 = r24 >> 1
//  000A0  LSR R24          r24 = r24 >> 1
//  000A1  ANDI R24,0x03    r24 = r24 & 0b11
//  000A2  STS 0x0200,R24   b   = r24   //&b=0x200
    y = (a & b) | (a|b)<<2;
   }PORTJ = y;          // Ausgabe auf die 8 LEDs
```

```
// Variablen:  a(0x203), b(0x200), x(0x201), y(0x202)
  y = (a & b) | (a|b)<<2;
// 0x00A4  LDS R25,0x0203    r25 = a     //&a=0x203
// 0x00A6  LDS R24,0x0200    r24 = b     //&b=0x200
// 0x00A8  AND R24,R25       r24 = r24 & r25
// 0x00A9  MOV R18,R24       r18 = r24
// 0x00AA  LDS R25,0x0203    r25 = a     //&a=0x203
// 0x00AC  LDS R24,0x0200    r24 = b     //&b=0x200
// 0x00AE  OR R24,R25        r24 = r24 | r25
// 0x00AF  MOV R24,R24       r24 = r24 Sinnlos?
// 0x00B0  LDI R25,0x00      r25 = 0   Sinnlos?
// 0x00B1  LSL R24           r24.r25 >> 1
// 0x00B2  ROL R25
// 0x00B3  LSL R24           r24.r25 >> 1
// 0x00B4  ROL R25
// 0x00B5  OR R24,R18        r24 = r24 | r18
// 0x00B6  STS 0x0202,R24    y = r24     //&y=0x202
```

Anmerkungen

- Variablen (globale) erhalten bei der Vereinbarung eine feste Adresse.

  ```
  uint8_t a, b, x, y; ...

  // a Adresse 0x203, b Adresse 0x200,
  // x Adresse 0x201, y Adresse 0x202
  ```

- Das Programm wurde mit Compiler-Optimierung »-O0« übersetzt. Bei dieser Optimierungsstufe wird jede C-Anweisung einzeln übersetzt, beginnend mit Laden der Variablen aus dem Speicher bis zum Zurückspeichern der Ergebnisse.

  ```
      a = 0b11 & x;
  // 0x0098  LDS R24,0x0201   r24 = x(0x201)
  // 0x009A  ANDI R24,0x03    r24 = r24 & 0b11
  // 0x009B  STS 0x0203,R24   a(0x203) = r24
  ```

Programmieren in Assembler

```
.global main
main:
 ldi r16, 0x18 ; r16 = 0b.... ....
 ldi r17, 0x24 ; r17 = 0b.... ....
 ori r16, 0x03 ; r16 = 0b.... ....
 andi r16, 0xFE; r16 = 0b.... ....
 eor r17, r16  ; r17 = 0b.... ....
 ret           ; Programmbeendigung
```

»main« ist die Marke für die Startadresse des Programms und muss als »global« vereinbart sein. Jede Assembler-Anweisung wird in einem Maschinenbefehl übersetzt:

```
Adresse   Befehl       |Adresse   Befehl
0x0007D  LDI R16,0x18  |0x00080  ANDI R16,0xFE
0x0007E  LDI R17,0x24  |0x00081  EOR R17,R16
0x0007F  ORI R16,0x03  |0x00082  RET
```

Spezielle Bitverarbeitungsoperationen

Viele Prozessoren haben Spezialbefehle, die ihnen in bestimmten Anwendungen Geschwindigkeitsvorteile verschaffen. ATmega-Prozessoren haben z.B. die Befehle:

```
sbi A, b ; setze Bit b in EA-Register A
cbi A, b ; lösche Bit b in EA-Register A
bld Rd, b; Bit LoaD, kopiere Bit b aus Register
         ; Rd in das Statusbit T
bst Rd, b; Bit STore, kopiere T nach Rd Bit b
```

Der Befehl

```
sbi 8, 3; setze Bit 3 in Port C
```

ersetzt z.B. die Befehlsfolge:

```
in r16, 8
ori r16, 0b00001000
out 8, r16
```

3.4 Aufgaben

Trace-Tabelle ausfüllen (MiPro)

Ergänzen Sie in der nachfolgenden Trace-Tabelle für die Programmabarbeitung auf dem Minimalprozessor die in die Register geschriebenen Werte als 2-stellige Hexadezimalzahlen:

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r1,4a,..:294a|00 .. 00 00 00 00 00 00|0|0|
01|ld_i r0,73,..:2873|.. .. 00 00 00 00 00 00|0|0|
02|move r2,r0,..:8200|.. .. .. 00 00 00 00 00|0|0|
03|move r3,r1,..:8320|.. .. .. .. 00 00 00 00|0|0|
04|andi r2,0f,..:620f|.. .. .. .. 00 00 00 00|0|0|
05|andi r3,f0,..:63f0|.. .. .. .. 00 00 00 00|0|0|
06|or_r r4,r2,r3:ec4c|.. .. .. .. .. 00 00 00|0|0|
07|xorr r5,r4,r4:f590|.. .. .. .. .. .. 00 00 0|1|
```

Lösung

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r1,4a,..:294a|00 4a 00 00 00 00 00 00|0|0|
01|ld_i r0,73,..:2873|73 4a 00 00 00 00 00 00|0|0|
02|move r2,r0,..:8200|73 4a 73 00 00 00 00 00|0|0|
03|move r3,r1,..:8320|73 4a 73 4a 00 00 00 00|0|0|
04|andi r2,0f,..:620f|73 4a 03 4a 00 00 00 00|0|0|
05|andi r3,f0,..:63f0|73 4a 03 40 00 00 00 00|0|0|
06|or_r r4,r2,r3:ec4c|73 4a 03 40 43 00 00 00|0|0|
07|xorr r5,r4,r4:f590|73 4a 03 40 43 00 00 00|0|1|
```

Befehlsfolge zur Logikverarbeitung (MiPro)

Schreiben Sie für den Minimalprozessor eine Befehlsfolge, die den Bitvektor **x**=0xB3 in Register r0 schreibt und daraus einen Bitvektor **y** in Register r1 nach folgender Vorschrift bildet:

y_7	y_6	y_5	...	y_2	y_1	y_0
$x_7 \oplus x_6$	$x_6 \oplus x_5$	$x_5 \oplus x_4$	...	$x_2 \oplus x_1$	$x_1 \oplus x_0$	$x_0 \oplus x_7$

1. Welchen Wert erhält y für **x**=0xB3?
2. Kontrollieren Sie durch Ausfüllen der nachfolgenden Tabelle, dass das Programm dieses Ergebnis liefert.

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,b3,..:28b3|b3 00 00 00 00 00 00 00|0|0|
01|.... ..,..,  :    |.. .. .. .. 00 00 00 00|0|0|
02|.... ..,..,  :    |.. .. .. .. 00 00 00 00|0|0|
03|.... ..,..,  :    |.. .. .. .. 00 00 00 00|0|0|
04|.... ..,..,  :    |.. .. .. .. 00 00 00 00|0|0|
```

Lösung

1. Sollwertberechnung für **y**:

$$\begin{array}{rcccccccc} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & \text{0xb3} \\ \oplus & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & \text{0x67} \\ \hline & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & \text{0xd4} \end{array}$$

2. Programm zur Bildung von **y** in r1 und Simulation:

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,b3,..:28b3|b3 00 00 00 00 00 00 00|0|0|
01|sh_l r2,r0,..:9a00|b3 00 66 00 00 00 00 00|1|0|
02|rotl r3,r0,..:ab00|b3 00 66 67 00 00 00 00|1|0|
03|xorr r1,r0,r3:f10c|b3 d4 66 67 00 00 00 00|1|0|
```

Abarbeitung von Logikbefehlen (AVR)

Welche Werte werden den Register r16 und r17 zugewiesen?

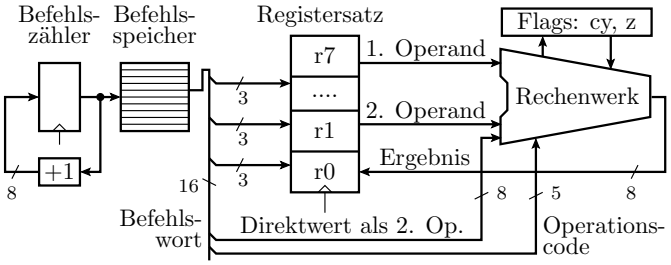
```
.global main
main:
 ldi r16, 0x18 ; r16 = 0b.... ....
 ldi r17, 0x24 ; r17 = 0b.... ....
 ori r16, 0x03 ; r16 = 0b.... ....
 andi r16, 0xFE; r16 = 0b.... ....
 eor r17, r16  ; r17 = 0b.... ....
 ret           ; Programmende
```

Lösung

```
.global main
main:
 ldi r16, 0x18 ; r16 = 0b0001 1000
 ldi r17, 0x24 ; r17 = 0b0010 0100
 ori r16, 0x03 ; r16 = 0b0001 1011
 andi r16, 0xFE; r16 = 0b0001 1010
 eor r17, r16  ; r17 = 0b0011 1110
 com r17       ; r17 = 0b1100 0001
 ret           ; Programmende
```

4 Lade- und Speicherbefehle

Wiederholung



Die bisher behandelte Prozessorschaltung kann aus

- einem oder zwei Registerinhalten oder
- einer Konstanten und einem Registerinhalt

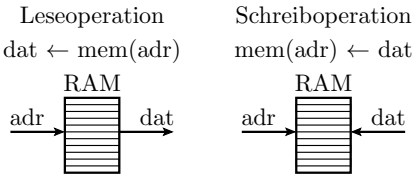
ein Ergebnis bilden und in einem Register speichern. Bisher behandelte Operationen:

- bitweise Logikoperationen, verschieben, rotieren,
- Konstante laden, Registerinhalt kopieren, ...

Lade- und Speicherbefehle

Speicher als Tabelle

Adresse	Daten
..000	0110 1000
..001	0101 1101
..010	1100 1000
...	...

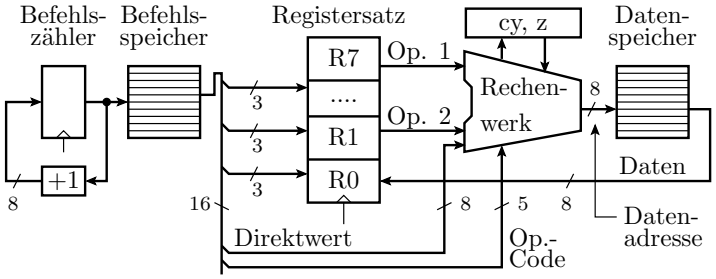


Der Datenspeicher ist als Tabelle organisiert. Unter jeder Adresse steht ein Datenbyte. Zur Verarbeitung müssen die Daten in Register geladen und die Ergebnisse zurückgespeichert werden. Damit in einem Schritt ausführbar, trennt ein RISC-Befehlssatz zwischen Lade-, Verarbeitungs- und Speicheroperationen. Adressierungsarten der Lade- und Speicheroperationen:

- direkt: Datenspeicheradresse steht im Befehlswort,
- indirekt: Speicheradresse wird vorher in ein Register geladen,
- mit Adressrechnung: Registerinhalt + Konstante, ...

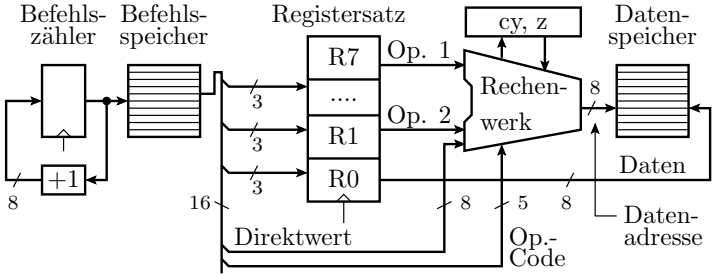
4.1 LS MiPro

Erweiterung um Ladeoperationen



- Aus max. zwei Registerinhalten oder einem Registerinhalt und einer Konstanten wird die Adresse berechnet.
- Das Zielregister übernimmt statt des Berechnungsergebnisses den aus dem Datenspeicher gelesenen Wert.

Erweiterung um Speicheroperationen



- Aus max. zwei Registerinhalten oder einem Registerinhalt und einer Konstanten wird die Adresse berechnet.
- Das »Zielregister« wird gelesen und sein Wert unter der berechneten Adresse im Datenspeicher abgelegt.

Lade- / Speicherbefehle des Minimalprozessors

Teilbitvektor	15 ... 11	10 9 8	7 6 5	4 3 2	1 0
cmd rd,ra	cnr	rd	ra		
cmd rd,imm	cnr	rd	imm		

Befehl	Operation	Flags	cnr
load rd,imm	rd := dmem(imm)		3
stor rd,imm	dmem(imm) := rd		4
ld_r rd,ra	rd := mem(ra)		17
st_r rd,ra	mem(ra) := rd		18

Unterstütze Adressierungsarten:

- direkt für die Adressierung von Variablen mit festen Adressen,
- indirekt für die Adressierung mit Zeigern.

Eine Adressrechnung, z.B. Registerinhalt + Konstante, in MiPro nicht implementiert, würde im Rechenwerk erfolgen.

Beispielprogramm

- Welche Werte werden in die Register und auf die Datenspeicherplätze mit den Adressen 5 und 6 geschrieben?

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,48,..:2848|.. 00 00 00 00 00 00 00|0|0|
01|stor r0,05,..:2005|.. .. .. .. .. .. .. ..|0|0|
              dmem = [00 00 00 00 00 .. .. 00]
02|ld_i r1,06,..:2906|.. .. .. .. .. .. .. ..|0|0|
03|ld_i r2,31,..:2a31|.. .. .. .. .. .. .. ..|0|0|
04|st_r r2,r1,..:9220|.. .. .. .. .. .. .. ..|0|0|
              dmem = [00 00 00 00 00 .. .. 00]
05|load r3,05,..:1b05|.. .. .. .. .. .. .. ..|0|0|
06|ld_r r4,r1,..:8c20|.. .. .. .. .. .. .. ..|0|0|
              dmem = [00 00 00 00 00 .. .. 00]
```

Zur Kontrolle

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,48,..:2848|48 00 00 00 00 00 00 00|0|0|
01|stor r0,05,..:2005|48 00 00 00 00 00 00 00|0|0|
              dmem = [00 00 00 00 00 48 00 00]
02|ld_i r1,06,..:2906|48 06 00 00 00 00 00 00|0|0|
03|ld_i r2,31,..:2a31|48 06 31 00 00 00 00 00|0|0|
04|st_r r2,r1,..:9220|48 06 31 00 00 00 00 00|0|0|
              dmem = [00 00 00 00 00 48 31 00]
05|load r3,05,..:1b05|48 06 31 48 00 00 00 00|0|0|
06|ld_r r4,r1,..:8c20|48 06 31 48 31 00 00 00|0|0|
              dmem = [00 00 00 00 00 48 31 00]
```

4.2 AVR-LS-Befehle

Lade- und Speicherbefehle des AVR-Prozessors

Direkte Adressierung mit einer 16-Bit Adresskonstanten (verlangt Doppelbefehlsworte und zwei Ausführungsschritte):

Operation	TZ	Op.-Code	Assembler
Rd←(k)	2	1001 000d dddd 0000 kkkk kkkk kkkk kkkk	`lds Rd, k`
(k)←Rd	2	1001 001d dddd 0000 kkkk kkkk kkkk kkkk	`sts k, Rd`

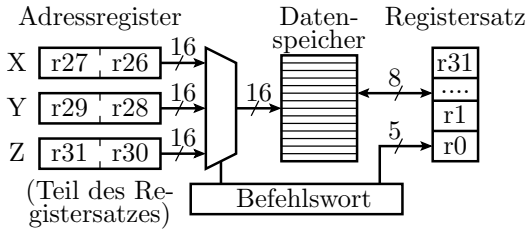
(TZ – Verarbeitungstaktzyklen; (k) – Datenspeicheradresse).

Beispiel:

```
ldi r1, 0x21  ; r1 = 0x21
lds r2, 0x200 ; r2 = mem(0x200)
add r2, r1    ; r2 = r2 + r1
sts r2, 0x200 ; mem(0x200) = r2
```

Indirekte Adressierung

Adressierung mit einem der 16-Bit-Adressregister X, Y oder Z, die je aus einem Paar der oberen Arbeitsregister gebildet werden.



Operation	TZ	Op.-Code	Assembler
Rd←(X)	2	1001 000d dddd 1100	ld Rd, X
Rd←(Y)	2	1000 000d dddd 1000	ld Rd, Y
Rd←(Z)	2	1000 000d dddd 0000	ld Rd, Z
(X)←Rr	2	1001 001r rrrr 1100	st X, Rr
...	...	...	...

Die indirekte Adressierung gibt es auch mit

- Post-Increment: Rd←(X); X←X+1; auch für Y und Z.
- Pre-Decrement: X←X-1; Rd←(X); auch für Y und Z.

Indirekte Adressierung mit Verschiebung

```
Rd←(Y+q) ; nur für Y und Z
```

Assemblernotationen:

```
ld  Rd, X+  ; Rd ←(X);  X ←X+1
ld  Rd, -X  ;  X ←X-1; Rd ←(X)
ldd Rd, Y+q ; Rd ←(Y+q)
st  X+,  Rr ; (X)←Rr;   X ←X+1
st  -X,  Rr ;  X ←X-1; (X)←Rr
std Y+q, Rr ; (Y+q)←Rr
```

Kopierschleife mit Pos-Increment:

```
X (r27:r26) = <Anfang zu kopierender Bytevektor>
Y (r29:r28) = <Anfang Kopierziel>
<wiederhole bis Ende des zu kopierenden Bytevektors>
  ld r1, X+; st X+,r1
```

Indizierte Adressierung im 1. Programmbeispiel

```
int main(void){
  DDRA = 0x00; // Schalter als Eingänge
  DDRJ = 0xFF; // LEDs als Ausgänge
// 0x008D  LDI R24,0x04  r25:r24 = 0x0104
// 0x008A  LDI R25,0x01  (Adresse von DDRJ)
// 0x008F  SER R18       r18 = 0xFF
// 0x0090  MOVW R30,R24  Z = r25:r24
// 0x0091  STD Z+0,R18   mem(Z+0) = r18
 while(1){<Schalter lesen, LED-Ausgabe berechnen>}
}
```

Kürzer wäre direkte Adressierung[10] von DDRJ (2 statt 5 Befehle):

```
ser r18         ;    r18 = 0xFF
sts 0x0104,r18  ;    mem(0x104) = r18
```

4.3 Variablen, Zeiger

Variablen

- Variablen sind in Hochsprachen Symbole für Adressen von Speicherplätzen, die beschrieben und gelesen werden können.
- Eine Variablenvereinbarung definiert Typ (z.B. uint8_t), Namen (z.B. dat) und optional einen Anfangswert (z.B. 45):

```
uint8_t dat = 45;
```

- Der Typ legt fest, wie viele Bytes zur Variablen gehören (z.B. 1 Byte) und was die Bytes darstellen (z.B. eine Zahl ohne Vorzeichen im Bereich von 0 bis 255).

	1 Byte		2 Byte		
ohne VZ	uint8_t	$[0, 255]$	uint16_t	$[0, 2^{16}-1]$	
mit VZ	int8_t	$[-128, 127]$	int16_t	$[-2^{15}, 2^{15}-1]$	

Wert und Adresse einer Variablen

- Der Compiler ordnet jeder Variablen eine Adresse oder ein Register zu. Adresse/Register im Debugger visualisierbar.

```
uint8_t a, b, *ptr;
int main(void){
  a   = 0x4D;
  ptr = &a;
  b   = *ptr + 3;
}
```

Watch 1

Name	Value	Type
a	0x4d	uint8_t{data}@0x0204
b	0x50	uint8_t{data}@0x0200
ptr	0x0204	uint8_t*{data}@0x0201
	0x4d	uint8_t{data}@0x0204

- C kennt auch Variablen für Adressen. Vereinbarung mit »<Typ> *<name>«, z.B.:

```
uint8_t *ptr;
```

[10] Ohne Compileroptimierung (-O0) Adressierung mit Adressregister Z, bei höheren Optimierungsstufen auch direkte Adressierung.

Zeiger

Ein Zeiger ist eine Variable für eine Adresse.

- Vereinbarung eines Zeigers auf Variablen eines bestimmten Typs (z.B. uint16_t):

```
uint16_t *ptr;
```

- Vereinbarung eines Zeigers für beliebige Adressen:

```
void *ptr;
```

- Die Adresse einer Variablen liefert der Operator »&«, z.B.:

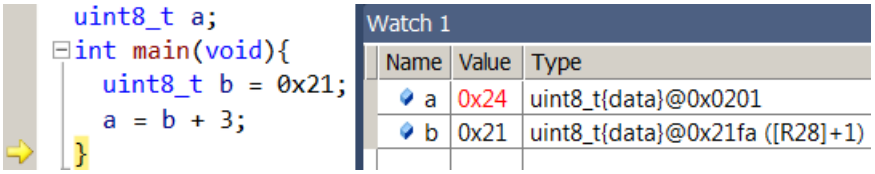
```
ptr = &a;
```

- Den Wert zu einer Adresse liefert der Operator »*«, z.B.:

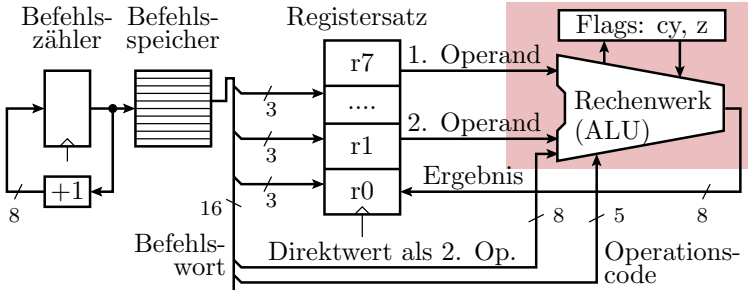
```
b = *ptr;
```

Globale und lokale Variablen

- Global: Außerhalb einer Funktion vereinbart. Feste Adressen im Datensegment. Existieren während der gesamten Programmlaufzeit.
- Lokal: Innerhalb einer Funktion vereinbart. Existieren nur während der Funktionsabarbeitung. Speicherplatz wird erst bei Funktionsaufruf auf dem sog. Stack reserviert.
- Lokale Variablen werden relativ zum Frame-Pointer [11] adressiert.



5 Arithmetik



[11] Der Compiler nimmt von unserem Prozessor als Frame-Pointer das 16-Bit-Register Y gebildet aus den 8-Bit-Registern r29 und r28.

Rechenwerk, ALU (**A**rithmetic **L**ogical **U**nit):

- bitweise logische Verknüpfungen (UND, ODER, EXOR, Negation).
- Verschiebe- und Rotationsbefehle.
- arithmetische Befehle (Add, Sub, Mult, ...)

Operationen, die das Rechenwerk nicht kann, werden mit Befehlsfolgen nachgebildet.

5.1 Addition

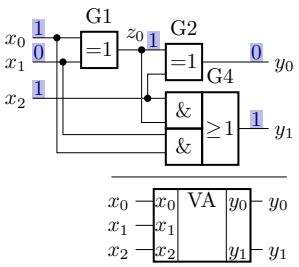
Addition von Binärzahlen

Die meisten arithmetischen Berechnungen (incl. Zählen, Subtraktion, Multiplikation, ...) basieren auf der Addition. Die binäre Addition erfolgt bitweise:

$$\begin{array}{rrrrrrrr} 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ +1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 & 1^{(1)} & 0^{(1)} & 0^{(1)} & 0 \end{array}$$

- Wiederhole für alle Bits beginnend mit dem niederwertigsten
  - Addition der Ziffern + Übertrag der vorherigen Stelle

Volladdierer



Gatterschaltung zur Aufsummierung von 3 Bits (zwei Summanden + Übertrag) zu einem Summen- und einem Übertragsbit.

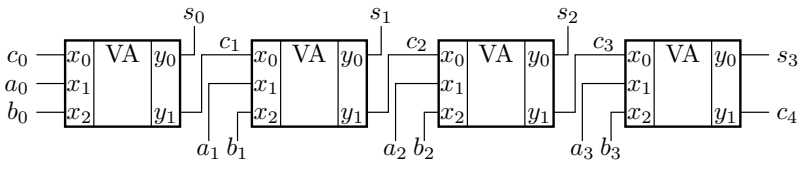
Eingabe x_2	x_1	x_0	Ist-Funktion z_0	y_1	y_0	Soll-Funktion $x_2+x_1+x_0$ dez	bin
0	0	0				0	00
0	0	1				1	01
0	1	0				1	
0	1	1				2	
1	0	0				2	
1	0	1	1	1	0	2	10
1	1	0				3	
1	1	1				3	11

Kontrolle durch Vervollständigung der Wertetabelle.

Addierer für Bitvektoren (Ripple-Addierer)

$$\begin{array}{rrrrrrrr} 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ +1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \hline 1 & 1 & 1 & 0 & 1^{(1)} & 0^{(1)} & 0^{(1)} & 0 \end{array}$$

Ein n-Bit-Addierer besteht aus einer Kette von n Volladdierern:



a, **b** – n-Bit-Summanden, c_0 – einlaufender Übertrag, **s** – n-Bit-Summe, c_n – Ergebnisübertrag.

5.2 Subtraktion

Vorzeichenbehaftete Summanden und Subtraktion

Statt durch Vorzeichen und Betrag werden vorzeichenbehaftete Zahlen in Rechnern durch »Stellenkomplement +1« dargestellt. Mathematische Grundlage:

- Das Stellenkomplement zu einer Ziffer b_i ist die Differenz zur größten darstellbaren Ziffer mit dem Wert $B-1$:
$$\bar{b}_i = B - 1 - b_i$$
(B – Basis des Zahlensystems, für Dezimalzahlen $B = 10$).

Beispiel: $\overline{437} = 562$

- Zahl plus Stellenkomplement gleich größte darstellbare Zahl.

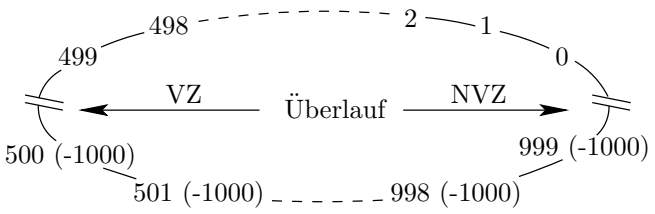
Beispiel: $437 + \overline{437} = 437 + 562 = 999$

- plus Eins gleich kleinste nicht darstellbare Zahl:
$$Z + \bar{Z} + 1 = B^n$$

$$Z + \bar{Z} + 1 = B^n$$

- Auflösung nach $-Z$:
$$-Z = \bar{Z} + 1 - \left[\underbrace{B^n}_{*}\right] \quad * \text{ nicht darstellbar}$$

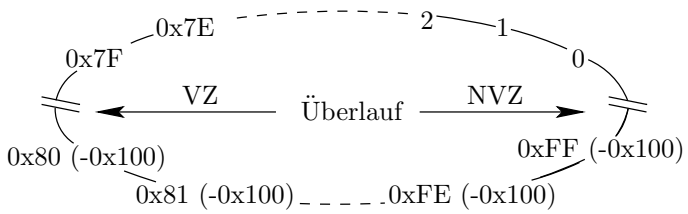
- Die Zählreihenfolge bleibt, nur der Darstellungsbereich verschiebt sich:



$$217 - 437 = 217 + \overline{437} + 1 = 217 + 562 + 1 = 778 - 1000 = -222$$

Zweierkomplement

- Basis: $B = 1$; Ziffern $\in \{0, 1\}$.
- Stellenkomplement: bitweise Negation.
- Das führende Bit ist das Vorzeichenbit.



	0	1	1	0	0	1	0	1	$(-0 \cdot 2^8)$ Erweiterung
$+$	1	0	0	0	0	0	1	1	$(-1 \cdot 2^8)$ zu vorzei-
									chenbehaf-
	1	1	1	0	$1^{(1)}$	$0^{(1)}$	$0^{(1)}$	0	$(-1 \cdot 2^8)$ tete Zahlen

$$0x65 + (0x83 - 0x100) = 0x65 - 0x7D = 0xE8 - 0x100 = -0x18$$

Beispielaufgaben

1. Wie berechnet sich der Wert und welchen Wertebereich haben vorzeichenfreie (NVZ) und vorzeichenbehaftete (VZ, Zweierkomplement-) Binärzahlen?
2. Zeigen Sie, dass für alle n-Bit Binärzahlen (NVZ und VZ) gilt:

$$\begin{aligned} \mathbf{b} - \mathbf{b} &= \mathbf{b} + \bar{\mathbf{b}} + 1 = 0 \text{ aus dem folgt:} \\ -\mathbf{b} &= \bar{\mathbf{b}} + 1 \end{aligned}$$

3. Durch welchen 8-Bit binären und durch welchen hexadezimalen Wert wird der dezimale Wert -15 im Zweierkomplement dargestellt?
4. Welche hexadezimale Darstellung hat der hexadezimale Wert -0x3A5F im Zweierkomplement?

Lösung

1.

	Wert	Minimum	Maximum
NVZ	$V_{\text{NVZ}} = \sum_{i=0}^{n-1} b_i \cdot 2^i$	0	$2^n - 1$
VZ	$V_{\text{VZ}} = V_{\text{NVZ}} - b_{n-1} \cdot 2^n$	-2^{n-1}	$2^{n-1} - 1$

2. Die Summe zueinander invertierter $n-$Bitvektoren:

$$b_{n-1} \ldots b_1 b_0 + \bar{b}_{n-1} \ldots \bar{b}_1 \bar{b}_0 = 1 \ldots 11$$

Dazu eins addiert ergibt null. Der Übertrag geht verloren.

3.

$$\begin{aligned} 15 &= 0b0000.1111 = 0x0F \\ -15 &= 0b1111.0000 + 1 = 0b1111.0001 = 0xF1 \end{aligned}$$

4. Der bitweise negierte Wert einer Hexadezimalzahl ist ihr Stellenkomplement, d.h. der Ersatz jeder Hex.-Stelle durch ihre Differenz zu 0xF. Dazu ist eins zu addieren:

$$-0x3A5F = 0xC5A0 + 1 = 0xC5A1$$

Subtraktion und Subtrahierer

Ersatz der Subtraktion durch Addition mit dem bitweise negierten Subrahenden und Übertrag:

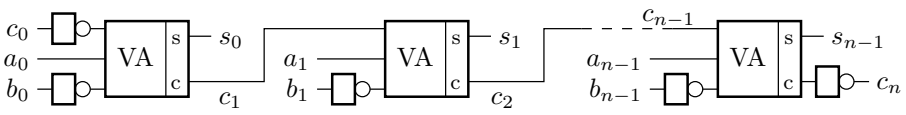
bitweise Subtraktion

a		1	1	1	0	0	1	0	1
$-b$	c_8	1	0	1	0	1	0	1	0
$(-c_i)$	(-1)	(0)	(-1)	(-1)	(-1)	(0)	(-1)	(0)	(-1) c_0
		0	0	1	1	1	0	1	0

Addition des Stellenkomplements +1

a		1	1	1	0	0	1	0	1
$+\bar{b}$		0	1	0	1	0	1	0	1
	$\bar{c}_8$								1 $\bar{c}_0$
$(+c_i)$	(0)	(1)	(0)	(0)	(0)	(1)	(0)	(1)	(-1)
		0	0	1	1	1	0	1	0

Daraus resultierende Schaltung des Subtrahierers:



5.3 Add- und Sub-Befehle MiPro

Additions- und Subtraktionsbefehle MiPro

Befehl	Operation	Flags	cnr
addr rd,ra,rb	rd := ra + rb	cy,z	24
addi rd,imm	rd := rd + imm	cy,z	8
adcr rd,ra,rb	rd := ra + rb + cy	cy,z	25
adci rd,imm	rd := rd + imm+ cy	cy,z	9
subr rd,ra,rb	rd := ra − rb	cy,z	26
subi rd,imm	rd := imm− rd	cy,z	10
sbcr rd,ra,rb	rd := rb − ra − cy	cy,z	27
sbci rd,imm	rd := imm− rd − cy	cy,z	11

- Bei Addition mehrerer Bytes werden die niederwertigen Bytes mit add und die höherwertigen mit addc addiert. Analog bei der Subtraktion.
- Der erste Operand kann eine Variable oder eine Konstante (Direktwert) sein.

Beispielaufgabe

Programmieraufgabe:

```
r0:r1 = 0x733A;          r2:r3 = 0x13E7;
r4:r5 = r0:r1 + r2:r3; r6:r7 = r0:r1 - r2:r3
```

Programm:

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r1,3a,..:293a|00 .. 00 00 00 00 00 00|0|0|
01|ld_i r0,73,..:2873|.. .. 00 00 00 00 00 00|0|0|
02|ld_i r3,e7,..:2be7|.. .. 00 .. 00 00 00 00|0|0|
03|ld_i r2,13,..:2a13|.. .. .. .. 00 00 00 00|0|0|
04|addr r5,r1,r3:c52c|.. .. .. .. .. .. 00 00|.|0|
05|adcr r4,r0,r2:cc08|.. .. .. .. .. .. 00 00|.|0|
06|subr r7,r1,r3:d72c|.. .. .. .. .. .. 00 ..|.|0|
07|sbcr r6,r0,r2:de08|.. .. .. .. .. .. .. ..|.|0|
```

- Sollwerte berechnen. Registerwerte ergänzen. Soll-/Ist-Vergl.

Lösung

Programmieraufgabe:

```
r0:r1 = 0x733A;         r2:r3 = 0x13E7;
r4:r5 = r0:r1 + r2:r3; Ergebnis: 0x8721
r6:r7 = r0:r1 - r2:r3; Ergebnis: 0x5F35
```

Programm mit ergänzten Registerinhalten:

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r1,3a,..:293a|00 3a 00 00 00 00 00 00|0|0|
01|ld_i r0,73,..:2873|73 3a 00 00 00 00 00 00|0|0|
02|ld_i r3,e7,..:2be7|73 3a 00 e7 00 00 00 00|0|0|
03|ld_i r2,13,..:2a13|73 3a 13 e7 00 00 00 00|0|0|
04|addr r5,r1,r3:c52c|73 3a 13 e7 00 21 00 00|1|0|
05|adcr r4,r0,r2:cc08|73 3a 13 e7 87 21 00 00|0|0|
06|subr r7,r1,r3:d72c|73 3a 13 e7 87 21 00 53|1|0|
07|sbcr r6,r0,r2:de08|73 3a 13 e7 87 21 5f 53|0|0|
```

5.4 Add- und Sub-Befehle AVR

AVR-Additions- und Subtraktionsbefehle

Operation	Op.-Code	Assembler
Rd← Rd+Rr	0000 11rd dddd rrrr	add Rd, Rr
Rd← Rd+Rr+C	0001 11rd dddd rrrr	addc Rd, Rr
Rd+1:Rd← Rd+1:Rd+K$^{(1)}$	1001 0110 KKdd KKKK	adiw Rd+1:Rd, K
Rd← Rd-Rr	0001 10rd dddd rrrr	sub Rd, Rr
Rd← Rd-K$^{(2)}$	0101 KKKK dddd KKKK	subi Rd, K
Rd← Rd-Rr-C	0000 10rd dddd rrrr	sbc Rd, Rr
Rd← Rd-K-C$^{(2)}$	0100 KKKK dddd KKKK	sbci Rd, K
Rd+1:Rd← Rd+1:Rd-K$^{(1)}$	1000 0111 KKdd KKKK	sbiw Rd+1:Rd, K

Die Addition mit einer Konstanten wird durch die Subtraktion mit der negierten Konstanten ersetzt. Verringerter WB für Registeradressen und Direktwerte: $^{(1)} d \in 24, 26, 28, 30; 0 \leq K \leq 63$; $^{(2)}$ $16 \leq d \leq 31$.

Dissassemblierte 16-Bit-Addition

```
#include <avr/io.h>
uint16_t a=0x2573, b=0x7FA6, s, d;
int main(){
   s = a + b + 0x13A5;
// 0x0096  LDS R18,0x0200;  r18 = a.Byte0
// 0x0098  LDS R19,0x0201;  r19 = a.Byte1
// 0x009A  LDS R24,0x0202;  r24 = b.Byte0
// 0x009C  LDS R25,0x0203;  r25 = b.Byte1
// 0x009E  ADD R24,R18   ;  r24 = r24 + r18
// 0x009F  ADC R25,R19   ;  r25 = r25 + r19 +c
// 0x00A0  SUBI R24,0x5B ;  r24 = r24 + 0xA5
// 0x00A1  SBCI R25,0xEC ;  r25 = r25 + 0x13+c
// 0x00A2  STS 0x0207,R25;  s.Byte1 = r25
// 0x00A4  STS 0x0206,R24;  s.Byte0 = r24

   d = a - b - 0x0163;   // siehe nächste Folie
}
```

Dissassemblierte 16-Bit-Subtraktion

```
   d = a - b - 0x0163;
// 0x00A6  LDS R18,0x0200;  r18 = a.Byte0
// 0x00A8  LDS R19,0x0201;  r19 = a.Byte1
// 0x00AA  LDS R24,0x0202;  r24 = b.Byte0
// 0x00AC  LDS R25,0x0203;  r25 = b.Byte1
// 0x00AE  MOVW R20,R18  ;  r21:r20 = r19:r18
// 0x00AF  SUB R20,R24   ;  r20 = r20 - r24
// 0x00B0  SBC R21,R25   ;  r21 = r21 - r25 -c
// 0x00B1  MOVW R24,R20  ;  r25:r24 = r21:r20
// 0x00B2  SUBI R24,0x63 ;  r24 = r24 - 0x63
// 0x00B3  SBCI R25,0x01 ;  r25 = r25 - 0x01-c
// 0x00B4  STS 0x0205,R25;  d.Byte1 = r25
// 0x00B6  STS 0x0204,R24;  d.Byte0 = r24
```

Es geht sicher auch ohne die beiden movw-Befehle.

5.5 Multiplikation

Multiplikation vorzeichenfreier Binärzahlen

$$\begin{array}{r}
(a_3\cdot 2^3+a_2\cdot 2^2+a_1\cdot 2^1+a_0\cdot 2^0)\cdot(b_3\cdot 2^3+b_2\cdot 2^2+b_1\cdot 2^1+b_0\cdot 2^0)= \\
\hline
a_3b_0\cdot 2^3+a_2b_0\cdot 2^2+a_1b_0\cdot 2^1+a_0b_0\cdot 2^0 \\
a_3b_1\cdot 2^4+a_2b_1\cdot 2^3+a_1b_1\cdot 2^2+a_0b_1\cdot 2^1 \\
a_3b_2\cdot 2^5+a_2b_2\cdot 2^4+a_1b_2\cdot 2^3+a_0b_2\cdot 2^2 \\
a_3b_3\cdot 2^6+a_2b_3\cdot 2^5+a_1b_3\cdot 2^4+a_0b_3\cdot 2^3 \\
\hline
p_7\quad p_6\qquad\quad p_5\qquad\quad p_4\qquad\quad p_3\qquad\quad p_2\qquad\quad p_1\qquad\quad p_0
\end{array}$$

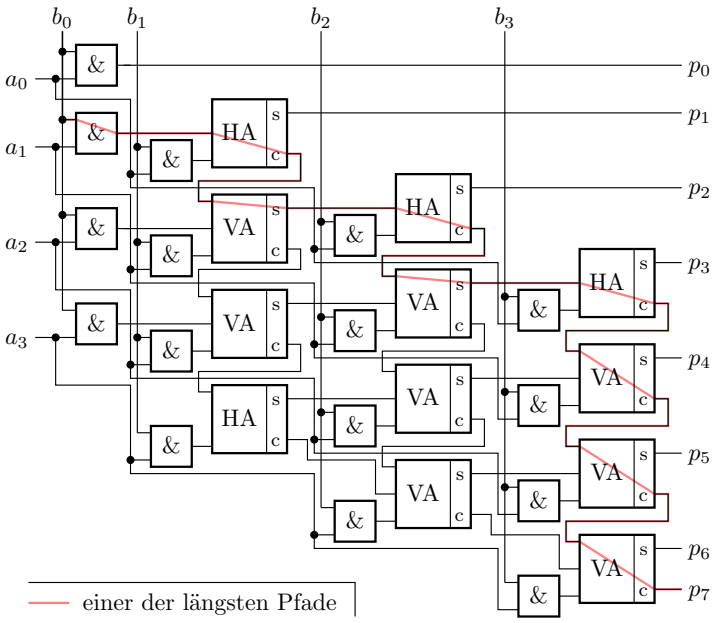
Eine $n\times n$-Bit-Multiplikation ist nachbildbar aus:

- $n\times n$ 1-Bit-Multiplikationen (UND-Verknüpfungen) und
- zeilen- und spaltenweisen Additionen mit Halb- und Volladdierern.

Das kleinste Produkt ist $0\cdot 0=0$. Das größte Produkt ist

$$\left(2^2-1\right)^2=2^{2n}-2\cdot 2^n+1$$

und benötigt $2n$-Bit Ergebnisregister (Doppelregister).



- Der Schaltungsaufwand eines $n \times n$-Bit Multiplizierers wächst mit n^2.
- Die Verzögerungszeit entlang des längsten Datenpfades, bis das Ergebnis garantiert fertig gebildet ist, wächst nur mit n.
- Ein $n \times n$-Bit-Matrixmultiplizierer hat etwa die 2...3-fache Verzögerung eines normalen n-Bit-Addierers.
- Prozessoren haben deshalb oft einen Multiplizierer, der eine Multiplikation in einem oder wenigen Takten ausführt[12].

ATmega-Multiplikationsbefehl für vorzeichenfreie Zahlen:

Operation	Op.-Code	Assembler
R[1:0]←Rd · Rr	`1001 11rd dddd rrrr`	`mul Rd, Rr`

Wählbare Operandenregister. Das höherwertige Ergebnisbyte wird immer in r1 und das niederwertige in r0 gespeichert.

Nachbildung 16-Bit- durch 8-Bit Multiplikationen

Zerlegung der Operanden und des Ergebnisses in Polynome von einzelnen Bytes a_i, b_i, ... :

$$a_3 \cdot 2^{24} + a_2 \cdot 2^{16} + a_1 \cdot 2^8 + a_0 = \left(b_1 \cdot 2^8 + b_0\right) \cdot \left(c_1 \cdot 2^8 + c_0\right)$$

Berechnung der Ergebnisbytes:

$$\begin{aligned} a_0 &= L\left(b_0 \cdot c_0\right) \\ a_2^* a_1 &= H\left(b_0 \cdot c_0\right) + L\left(b_1 \cdot c_0\right) + L\left(b_0 \cdot c_1\right) \\ a_3^* a_2 &= a_2 + H\left(b_1 \cdot c_0\right) + H\left(b_0 \cdot c_1\right) + L\left(b_1 \cdot c_1\right) \\ a_3 &= a_3 + H\left(b_1 \cdot c_1\right) \end{aligned}$$

[12] Bei der Division wachsen Schaltungsaufwand **und** Verzögerungszeit mit dem Quadrat der Bitanzahl. Deshalb sind HW-Dividierer unüblich.

- $H/L(\ldots)$ – höher-/niederwertiges Produktbyte.
- a_i^* : Zweites Byte zur Aufnahme der Überträge.

Dissassemblierte 16×16-Bit-Multiplikation

```
#include <avr/io.h>
uint16_t a=0x2573, b=0x7FA6;
uint32_t p;
int main(){
        p = a * b;
// 0x0096  LDS R20,0x0200  r20 = a.Byte0
// 0x0098  LDS R21,0x0201  r21 = a.Byte1
// 0x009A  LDS R18,0x0202  r18 = b.Byte0
// 0x009C  LDS R19,0x0203  r19 = b.Byte1
// <r[27:24] = r[21:20] *r[19:18]>
// 0x00A8  STS 0x0204,R24  p.Byte0 = r24
// 0x00AA  STS 0x0205,R25  p.Byte1 = r25
// 0x00AC  STS 0x0206,R26  p.Byte2 = r26
// 0x00AE  STS 0x0207,R27  p.Byte3 = r27
}
```

Übersetzung der eigentlichen Multiplikation

```
// <r[27:24] = r[21:20] *r[19:18]>
// 0x009E  MUL R20,R18     r1:r0 = r20 * r18
// 0x009F  MOVW R24,R0     r25:r24 = r1:r0
// 0x00A0  MUL R20,R19     r1:r0 = r20 * r19
// 0x00A1  ADD R25,R0      r25 = r25 + r0
// 0x00A2  MUL R21,R18     r1:r0 = r21 * r18
// 0x00A3  ADD R25,R0      r25 = r25 + r0
// 0x00A4  CLR R1          r1 = 0
// 0x00A5  MOVW R24,R24    kein erkannbarer Sinn
// 0x00A6  LDI R26,0x00    r26 = 0 (p.Byte2)
// 0x00A7  LDI R27,0x00    r27 = 0 (p.Byte3)
```

Das Programm berechnet nur die beiden niederwertigen Bytes des Produkts und setzt die höherwertigen Bytes auf 0.

Bug oder Feature? / Fehler oder spezifizierte Sollfunktion?

Berechnung alle 4 Produktbytes (2 Befehle mehr)

```
// <r[27:24] = r[21:20] *r[19:18]>
// 0x009E  MUL R20,R18     r1:r0 = a0*b0
// 0x009F  MOVW R24,R0     r25:r24 = r1:r0
// 0x009E  MUL R21,R19     r1:r0 = a1*b1
// 0x009F  MOVW R26,R0     r27:r26 = r1:r0
// 0x00A0  MUL R20,R19     r1:r0 = a0*b1
// 0x00A1  ADD R25,R0      r25 = r25 + r0
// 0x00A2  ADC R26,R1      r26 = r26 + r1 +c
// 0x00A3  CLR R1          r1 = 0
// 0x00A4  ADC R27,R1      r27 = r27 + r1 +c
// 0x00A5  MUL R21,R18     r1:r0 = a1*b0
// 0x00A6  ADD R25,R0      r25 = r25 + r0
// 0x00A7  ADC R26,R1      r26 = r26 + r1 +c
// 0x00A8  CLR R1          r1 = 0
// 0x00A9  ADC R27,R1      r27 = r27 + r1 +c
```

Multiplikation vorzeichenbehafteter Zahlen

- Multiplikation vorzeichenbehafteter Zahlen. Im Gegensatz zur Addition und Subtraktion geänderter Algorithmus:

$$\begin{aligned}(A - a_{n-1} \cdot 2^n) \cdot (B - b_{n-1} \cdot 2^n) &= A \cdot B \\ &- (A \cdot b_{n-1} + B \cdot a_{n-1}) \cdot 2^n \\ &+ \underbrace{a_{n-1} \cdot b_{n-1} \cdot 2^{2n}}_{\text{mit 2·n Bits nicht darstellbar}}\end{aligned}$$

  Zusätzliche bedingte Subtraktion der um n Bit linksverschobenen Faktoren vom »vorzeichenfreien« Produkt, wenn das jeweils andere Vorzeichenbit eins ist.

Division: Nachbildung als Schleife mit Verschiebebefehlen, Subtraktion und Fallunterscheidungen (siehe später Abschnitt 7.6).

5.6 Gleitkommazahlen

Festkomma- und Gleitkommazahlen

Zahlenwerte mit Nachkommastellen lassen sich auf dem Rechner als Festkomma- oder Gleitkommazahlen darstellen. Festkommazahlen haben eine gedachte Kommaposition und werden mit den arithmetischen Befehlen (Add, Sub, ...) für ganze Zahlen bearbeitet. Problematisch ist die Wahl der Anzahl der Vorkommastellen so, dass der Zahlenbereich nicht über- oder unterläuft und die Anzahl der Nachkommastellen, damit die Rundungsfehler vernachlässigbar bleiben.

Bei Gleitkommazahlen wird bei jeder arithmetischen Operation die Kommaposition des Ergebnisses mit berechnet und in der Zahldarstellung gespeichert.

Darstellung von Gleitkommazahlen

- Mantisse M: Wertebereich (normiert) $1 \leq Z(M) < 2$
- Charakteristik c: Kommaverschiebung, ganzzahlig; c_0 – Wert von c für Kommaverschiebung null
- Vorzeichenbit s

Wert für $0 < c < c_{\max}$ (normierte Darstellung[13]):

$$Z = (-1)^s \cdot (1, M_{-1} \ldots M_{-m}) \cdot 2^{c-c_0}$$

Wert für $c = 0$ (denormiert[14]):

$$Z = (-1)^s \cdot (M_0, M_{-1} \ldots M_{-m}) \cdot 2^{-c_0}$$

[13] In der normierten Darstellung ist die Mantisse M eine vorzeichenfreie Zahl mit einem Vorkommabit gleich eins, das nicht mit gespeichert wird.

[14] In der denormierten Darstellung kann das Vorkommabit auch null sein und wird mit gespeichert.

Sonderwerte $c = c_{\max}$:

$$Z = \begin{cases} \infty & \text{für } s = 0 \text{ und } m = 0 \\ -\infty & \text{für } s = 1 \text{ und } m = 0 \\ \text{nan} & \text{für } m \neq 0 \end{cases}$$

(nan, not a number – ungültig; $\pm\infty$ – positiver/negativer Wertebereichsüberlauf)

- 32-Bit-Format »IEEE-754 single«:

Bitvektor (31 … 24 | 23 … 16 | 15 … 8 | 7 … 0, Bitnummer)			Wert
s	c	M	
0	10000011	00100100 00000000 00000000	$+1.240000_{16} \cdot 2^{83_{16}-7f_{16}}$
+	$c = 83_{16}$	$M = 1,240000_{16}$	$= 18{,}25$
1	01111001	11001011 00111011 00001110	$-1.CB3A1C_{16} \cdot 2^{79_{16}-7f_{16}}$
−	$c = 79_{16}$	$M = 1,CB3A1C_{16}$	$\approx -2{,}8029 \cdot 10^{-2}$
0	00000000	00011001 01111010 01001100	$+0,32F464_{16} \cdot 2^{0-7f_{16}}$
	0/denorm.	$M = 0,32F464_{16}$	$\approx 1{,}170 \cdot 10^{-39}$

Nutzung von Gleitkommazahlen in C

```
11  float a, b, c;
12  int main(void){
13    a=b*c;
14  }
```

Name	Value	Type
a	-5588,22	float{data}@0x0208
b	14,74	float{data}@0x0200
c	-379,12	float{data}@0x0204

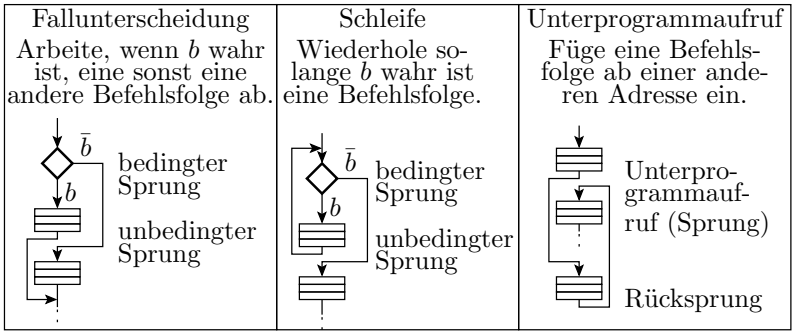
```
data 0x0200  0a d7 6b 41
data 0x0204  5c 8f bd c3
data 0x0208  d4 a1 ae c5
```

- Gleitkommazahlen werden unterstützt und ihre Werte sind im Debugger darstellbar.
- Die Byte-Darstellung ist im Speicher einsehbar.
- Die Unterprogramme für Gleitkommaoperationen sind hunderte von Befehlen lang und dauern hunderte von Maschinentakten (Zeitmessung siehe später Foliensatz 2, Abschn. 3.1).
- 32-Bit-Prozessoren haben oft Gleitkommarechenwerke, die Gleitkommaoperationen in wenigen Schritten ausführen.

6 Kontrollfluss

Steuerung des Kontrollflusses

Wenn ein Rechner nur Befehle nacheinander abarbeiten könnte, wäre jedes Programm nach wenigen Sekunden zu Ende. Die mehrfache Abarbeitung von Befehlsfolgen verlangt Fallunterscheidungen, Schleifen und Unterprogrammaufrufe, nachbildbar durch unbedingte und bedingte Sprünge im Verarbeitungsfluss.



Sprünge, Unterprogrammaufrufe, ...

- Absoluter Sprung:

  `PC ← K`

- relativer Sprung:

  `PC ← PC + 1 + K`

- bedingter Sprung

  ```
  wenn b dann PC ← PC + 1 + K
  sonst PC ← PC + 1
  ```

- Unterprogrammaufruf:

  `Rd← PC + 1; PC ← K`

- Rücksprung aus einem Unterprogramm:

  `PC ← Rr`

PC – Befehlszähler; K – Konstante, Rd – Register für die Speicherung der Rücksprungadresse; Rr – Register mit der Rücksprungadresse.

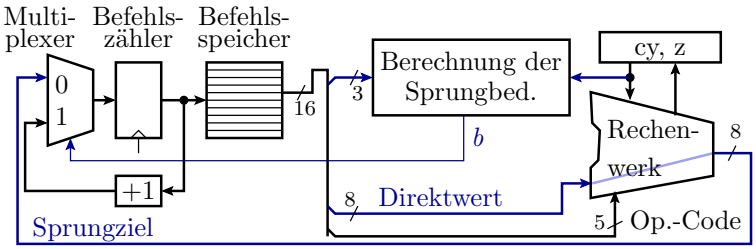
Sprungbefehle des Minimalprozessors

Befehl	Operation	Flags	cnr
jump imm,cond	if b: pc := imm		1
comp rd,imm	imm − rd	cy,z	6
cmpc rd,imm	imm − rd − cy	cy,z	7
call rd,imm	rd:=pc+1, pc:=imm		2
retu rd	pc := rd		15

- Nur absolute Sprünge zu einer 8-Bit-Adresskonstanten »imm«.
- Die 3-Bit-Sprungbedingung »cond« definiert Bedingungen in Abhängigkeit vom cy- und z-Flag, u.a. auch cond=001 für »springe immer« (unbedingter Sprung).
- comp und compc sind Subtraktionen, die nur die Flags für nachfolgende Sprünge, aber nicht die Differenzen speichern.
- Unterprogrammaufruf »call« und Rücksprung »retu« werden erst im nächsten Abschnitt behandelt.

6.1 Sprünge mit MiPro

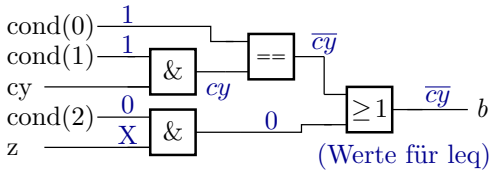
MiPro-Erweiterungen für Sprungbefehle




```
jump imm,cond; if (b) pc = imm; else pc++;
```

- Das Rechenwerk leiten die Konstante zu einem Multiplexer (Umschalter), der gesteuert vom berechneten Bedingungsbit b zwischen »nächster Befehl« und »Sprung« umschaltet.
- Die Berechnung der Sprungbedingung erfolgt mit einer Schaltung aus 4 Gattern (sie nächste Folie).

Berechnung der Sprungbedingung



cond	Wert	Bedeutung	Flag-Bedingung
nev	000	jump never	keine
alw	001	jump allways	keine
gth	010	jump greater than	cy=1
leq	011	jump if less or equal	cy=0
equ	100	jump if equal	z =1
neq	101	jump if not equal	z =0
geq	110	jump if greater or equal	cy=1 or z=1
lth	111	jump if less than	cy=0 and z=0

Testbeispiel mit Fallunterscheidung

```
wenn r0<0x37 dann r1 = 1; sonst r1 = 4;
```

```
;=====Test1=============================================
0000: ld_i r0,07,.. ; r0 = 0x07
0001: comp r0,37,.. ; r0-0x37
0002: jump 05,leq.. ; wenn ≤0, springe zu 0x5
0003: ld_i r1,01,.. ; r1 = 0x01
0004: jump 06,alw.. ; springe zu 6
0005: ld_i r1,04,.. ; r1 = 0x04
0006: ld_i r0,48,.. ; ...
=====Test 2=============================================
0000: ld_i r0,48,.. ; r0 = 0x48
0001: comp r0,37,.. ; r0-0x37
0002: jump 05,leq.. ; wenn ≤0, springe zu 0x05
0003: ...
```

Wie werden die beiden Testbeispiele abgearbeitet?

Lösung

```
=====Test1=============================
PC| Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,07,..:2807|07 00 00 00 00 00 00 00|0|0|
01|comp r0,37,..:3037|07 00 00 00 00 00 00 00|0|0|
02|jump 05,leq..:0b05|07 00 00 00 00 00 00 00|0|0|
05|ld_i r1,04,..:2904|07 04 00 00 00 00 00 00|0|0|
06|ld_i r0,48,..:2848|48 04 00 00 00 00 00 00|0|0|

=====Test 2=============================
PC| Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,48,..:2848|48 00 00 00 00 00 00 00|0|0|
01|comp r0,37,..:3037|48 00 00 00 00 00 00 00|1|0|
02|jump 05,leq..:0b05|48 00 00 00 00 00 00 00|1|0|
03|ld_i r1,01,..:2901|48 01 00 00 00 00 00 00|1|0|
04|jump 06,alw..:0906|48 01 00 00 00 00 00 00|1|0|
06|ld_i r0,48,..:2848|48 01 00 00 00 00 00 00|1|0|
```

Testbeispiel mit Schleife

```
    r0 = 1; r1 = 34;
M:  dmem(r0) = r1;
    r1 = r1 - r0; r0 = r0 +1;
    wenn r0 ≤ 3 springe zu M
```

Sprungbedingung für r0=2 und 3 erfüllt. 3 Schleifendurchläufe.

```
0000: ld_i r0,01,..
0001: ld_i r1,34,..
0002: st_r r1,r0,..
0003: subr r1,r1,r0
0004: addi r0,01,..
0005: comp r0,03,..
0006: jump 02,leq..
0007: noop ..,..,..
```

- In welcher Reihenfolge werden die Anweisungen abgearbeitet?
- Was wird in die Register und in den Speicher geschrieben?

```
PC| Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,01,..:2801|01 00 00 00 00 00 00 00|0|0|
01|ld_i r1,34,..:2934|01 34 00 00 00 00 00 00|0|0|
02|st_r r1,r0,..:9100|01 34 00 00 00 00 00 00|0|0|
                dmem = [00 34 00 00 00 00 00 00]
03|subr r1,r1,r0:d120|01 33 00 00 00 00 00 00|0|0|
04|addi r0,01,..:4001|02 33 00 00 00 00 00 00|0|0|
05|comp r0,03,..:3003|02 33 00 00 00 00 00 00|0|0|
06|jump 02,leq..:0b02|02 33 00 00 00 00 00 00|0|0|
02|st_r r1,r0,..:9100|02 33 00 00 00 00 00 00|0|0|
                dmem = [00 34 33 00 00 00 00 00]
03|subr r1,r1,r0:d120|02 31 00 00 00 00 00 00|0|0|
04|addi r0,01,..:4001|03 31 00 00 00 00 00 00|0|0|
05|comp r0,03,..:3003|03 31 00 00 00 00 00 00|0|1|
06|jump 02,leq..:0b02|03 31 00 00 00 00 00 00|0|1|
02|st_r r1,r0,..:9100|03 31 00 00 00 00 00 00|0|1|
```

```
                dmem = [00 34 33 31 00 00 00 00]
03|subr r1,r1,r0:d120|03 2e 00 00 00 00 00 00|0|0|
04|addi r0,01,...:4001|04 2e 00 00 00 00 00 00|0|0|
05|comp r0,03,...:3003|04 2e 00 00 00 00 00 00|1|0|
06|jump 02,leq..:0b02|04 2e 00 00 00 00 00 00|1|0|
```

6.2 AVR-Sprungbefehle

Unbedingte Sprünge

Es gibt drei Arten der Sprungzielvorgabe:

- direkt: Sprungziel ist eine Konstante im Befehlswort.
- indirekt: Sprungziel wird aus Registern gelesen.
- relativ: Sprungdistanz ist eine Konstante im Befehlswort.

Operation	TZ	Op.-Code	Assembler
PC←k	3	1001 0100 0000 110k kkkk kkkk kkkk kkkk	jmp k
PC←0:Z	2	1001 0100 0000 1001	ijmp
PC←EIND:Z	2	1001 0100 0001 1001	eijmp
PC←PC+1+k	2	1100 kkkk kkkk kkkk	rjmp

PC – Befehlszähler (Program Counter); Z – 16-Bit Adressregister aus r31 und r30; EIND – Verlängerungsregister für Z auf 17 Bit für indirekte Sprünge (EA-Adresse 0x3C); k – 12-Bit-Sprungdistanz, WB: $-2048 \leq k \leq 2047$.

Bedingte Sprünge

```
brbs b, k; Sprung, wenn Bit b in SREG eins ist.
brbc b, k; Sprung, wenn Bit b in SREG null ist.
```

Identische Befehle mit bedeutungsorientierten Bezeichnern:

```
br<Bed> k; Sprung, wenn <Bed> erfüllt ist
```

b		SREG(b) = 1	SREG(b) = 0
0	C	brcs (if Carry Set), brlo (if Lower$^{(u)}$)	brcc (if Carry Clear), brsh (if Same or Higher$^{(u)}$)
1	Z	breq (if Equal)	brne (if Not Equal)
2	N	brmi (if Minus)	brpl (if Plus)
3	V	brvs (if Overflow is Set)	brvc (if Overflow Cleared)
4	S	brge (Greater or Equal$^{(s)}$)	brlt (Less Than$^{(s)}$)
5	H	brhs (if Half Carry is Set)	brhc (if Half Carry Cleared)
6	T	brts (if T flag is Set)	brtc (if T flag is Cleared)
7	I	brhs (if Interrupt Enabled)	brhc (if Interrupt Disabled)

Skip-Befehle

Skip-Befehle überspringen bei erfüllter Bedingung den Nachfolgebefehl, der zwei oder vier Byte lang sein kann.

Skip-Bedingung	TZ	Op.-Code	Assembler
Rd=Rr	*	1001 00rd dddd rrrr	cpse Rd,Rr
Bit b in Rr gesetzt	*	1111 111r rrrr 0bbb	sbrs Rr,b
Bit b, Rr gelöscht	*	1111 110r rrrr 0bbb	sbrc Rr, b
Bit b, IO-Reg. A eins	*	1001 1001 AAAA Abbb	sbis A,b
Bit b in IO-Reg. A null	*	1001 1011 AAAA Abbb	sbic A,b

(* 1 Takt bei nicht erfüllter Bedingung, 2 Takte, wenn ein 2-Byte-, und 3 Takte, wenn ein 4-Byte-Befehl übersprungen wird. A – IO-Register 0 bis 31).

Beispiel Betragsbildung

In einer Endlosschleife wird von PortA ein vorzeichenbehaftetes Byte gelesen, der Betrag gebildet und auf PORTB ausgegeben:

```
#include <avr/io.h>
int8_t a;
int main(){
  while (1){
    a=PINA;        // Lesen von PORTA
    if (a<0) a=-a;
    PORTB = a;     // Ausgabe an PORTB
  }
}
```

- Die Endlosschleife wird mit einem unbedingten Sprung am Schleifenende und
- die Fallunterscheidung mit einem bedingten Sprung oder einer Skip-Anweisung realisisiert.

Übersetzung mit »-O0«

```
int main(){
  while (1); a=PINA;
// 0x0089  LDI R24,0x20    ; r25,r24 = 0x0020
// 0x008A  LDI R25,0x00    ; (Adresse von PINA)
// 0x008B  MOVW R30,R24    ; Z = Adresse(PINA)
// 0x008C  LDD R24,Z+0     ; r24 = PINA
// 0x008D  STS 0x0200,R24  ; a = r24 (adr(a)=0x200))
         if (a<0) a=-a;
// 0x008F  LDS R24,0x0200  ; r24 = a
// 0x0091  TST R24         ; Test, ob ≥0
// 0x0092  BRGE PC+0x06    ; spring, wenn ≥ nach 0x98
// 0x0093  LDS R24,0x0200  ; r24 = a
// 0x0095  NEG R24         ; r24 = - r24
// 0x0096  STS 0x0200,R24  ; a = r24
    PORTB = a; }          // mehrere Befehle
// 0x009E  RJMP PC-0x0015  ; Schleifenende
```

- Übersetzung mit »-O1«

```
int main(){
        while (1){
         a=PINA;
// 0x0085  IN R24,0x00     ; r24 = PINA
         if (a<0) a=-a;
// 0x0086  TST R24         ; Test, ob null oder negativ
// 0x0087  BRLT PC+0x04    ; spring, wenn < nach 0x8B
         a=PINA;
// 0x0088  STS 0x0200,R24 ; a = r24
// 0x008A  RJMP PC+0x0004 ; springe nach 0x8E
         if (a<0) a=-a;
// 0x008B  NEG R24         ; r24 = - r24
// 0x008C  STS 0x0200,R24 ; PORTB = r24
         PORTB = a;
// 0x008E  LDS R24,0x0200 ; r24 = a
// 0x0090  OUT 0x05,R24    ; PORTB = r24
        }
// 0x0091  RJMP PC-0x000C ; Schleifenende
```

- Übersetzung mit »-O2«

```
int main(){
        while (1){
         a=PINA;
// 0x0085  IN R24,0x00     ; r24 = PINA
         if (a<0) a=-a;
// 0x0086  SBRC R24,7      ; überspringe, wenn r24.7=1
// 0x0087  RJMP PC+0x0007 ; spring nach 0x8E
// 0x0088  STS 0x0200,R24 ; a = r24
// 0x008A  OUT 0x05,R24    ; PORTB = r24
// 0x008B  IN R24,0x00     ; r24 = a
// 0x008C  SBRS R24,7      ; überspringe, wenn r24.7=1
// 0x008D  RJMP PC-0x0005 ; spring nach 0x88
// 0x008E  NEG R24         ; r24 = - r24
// 0x008F  RJMP PC-0x0007 ; spring nach 0x88
```

Je höher die Optimierung, desto schneller und kürzer das Programm.

Optimierte Programme arbeiten aber nicht unbedingt eine C-Anweisung nach der anderen ab. Dann Debuggen von C-Programe im Schrittbetrieb nur noch eingeschränkt möglich.

6.3 Warteschleife

Warteschleife

Ziel sei ein kleines Programm, das PORTJ so langsam hochzählt, dass das Zählen mit Leuchtdioden beobachtbar ist.

- Bei 8 Millionen Takten pro Sekunde soll der Prozessor zyklisch ca. 4 Millionen Takte nichts tun und dann den Ausgabewert um eins erhöhen.
- Lösungsansatz: Warteschleife, die $N \approx 5 \cdot 10^5$ mal die Abarbeitungszeit von 8 Befehlen verbraucht[15]:

[15] Nachträglich kontrollieren, wie viele Befehle in der innersten Schleife abgearbeitet werden und N korrigieren.

```
int main(){
  register uint32_t a;
  while (1) {
    for (a=0; a<500000; a++);
    PORTJ ++;
  }
}
```

Optimierung mit -O0

```
int main(){
for (a=0; a<1000000; a++){
// 0x0085  MOV R14, R1     ; R14 bis R17 löschen
// 0x0086  MOV R15, R1     ; Befehl 2
// 0x0087  MOVW R16, R14   ; Befehl 3
// 0x0088  RJMP PC+0x0006 ; spring nach 0x88
// 0x0089  SER R24         ; r[17:14] += 1
// 0x008A  SUB R14,R24     ; 2. Befehl
// 0x008B  SBC R15,R24     ; 3. Befehl
// 0x008C  SBCI R16,0xFF   ; 4. Befehl
// 0x008D  SBCI R17,0xFF   ; 5. Befehl
// 0x008E  ...             ; Vergl. r[17:14]
// 0x008F  ...             ; mit 0x000F4240
```

```
// 0x008E  LDI R30,0x40    ; Vergl. r[17:14]
// 0x008F  CP R14,R30      ; mit 0x000F4240 (Subtr.
// 0x0090  LDI R30,0x42    ; ohne Summe speichern)
// 0x0091  CPC R15,R30     ; 4. Befehl
// 0x0092  LDI R30,0x0F    ; 5. Befehl
// 0x0093  CPC R16,R30     ; 6. Befehl
// 0x0094  CPC R17,R1      ; 7. Befehl
// 0x0095  BRCS PC-0x0C    ; wenn Diff. negativ,
//                         ; springe zu 0c89
// 0x0096  LDI R24,0x05    ; Z = Adr(PORTJ) = 0x105
// 0x0097  LDI R25,0x01    ; 2. Befehl
// 0x0098  MOVW R30,R24    ; 3. Befehl
// 0x0099  LDD R18,Z+0     ; r18 = PORTJ
// 0x009A  SUBI R18,0xFF   ; r18 = r18 +1
// 0x009B  MOVW R30,R24    ; ohne Funktion
// 0x009C  STD Z+0,R18     ; PORTJ = r18
// 0x009D  RJMP PC-0x0018 ; spring nach 0x85
```

0xC+1=13 Befehle in der inneren Schleife. $N \approx 4 \cdot 10^6/13 \approx 3 \cdot 10^5$.

Optimierung mit -O1

```
int main(){
// Initialisierung von Registern mit Konstanten
// 0x007D  LDI R21,0x40 ; r[18:r21] = 500.000
// 0x007E  LDI R20,0x42 ; 2. Befehl
// 0x007F  LDI R19,0x0F ; 3. Befehl
// 0x0080  LDI R18,0x00 ; 4. Befehl
// 0x0081  LDI R30,0x05 ; Z = Adr(PORTJ) = 0x105
// 0x0082  LDI R31,0x01 ; 2. Befehl
        for (a=0; a<1000000; a++){
// 0x0083  MOV R24,R21  ; r[27:24] = r[18:21] = 500.000
// 0x0084  MOV R25,R20  ; 2. Befehl
// 0x0085  MOV R26,R19  ; 3. Befehl
// 0x0086  MOV R27,R18  ; 4. Befehl
// innere Schleife
// 0x0087  SBIW R24,0x01; r[27:r24] += 1
// 0x0088  SBC R26,R1   ; 2. Befehl
// 0x0089  SBC R27,R1   ; 3. Befehl
// 0x008A  BRNE PC-0x03 ; wenn ≠, spring zu 0x0087
```

```
        PORTJ ++;
// 0x008B  LDD R24,Z+0  ; r24 = PORTJ
// 0x008C  SUBI R24,0xFF; r24 = r24 + 1
// 0x008D  STD Z+0,R24  ; PORTJ = r24
  }
// 0x008E  RJMP PC-0x000B; springe nach 0x83
```

- Zählrichtung auf abwärts geändert.
- Nur 4 Befehle in der innersten Schleife.
- Erhöhung der Iterationsanzahl auf:

$$N = \frac{4 \cdot 10^6}{4} = 10^6$$

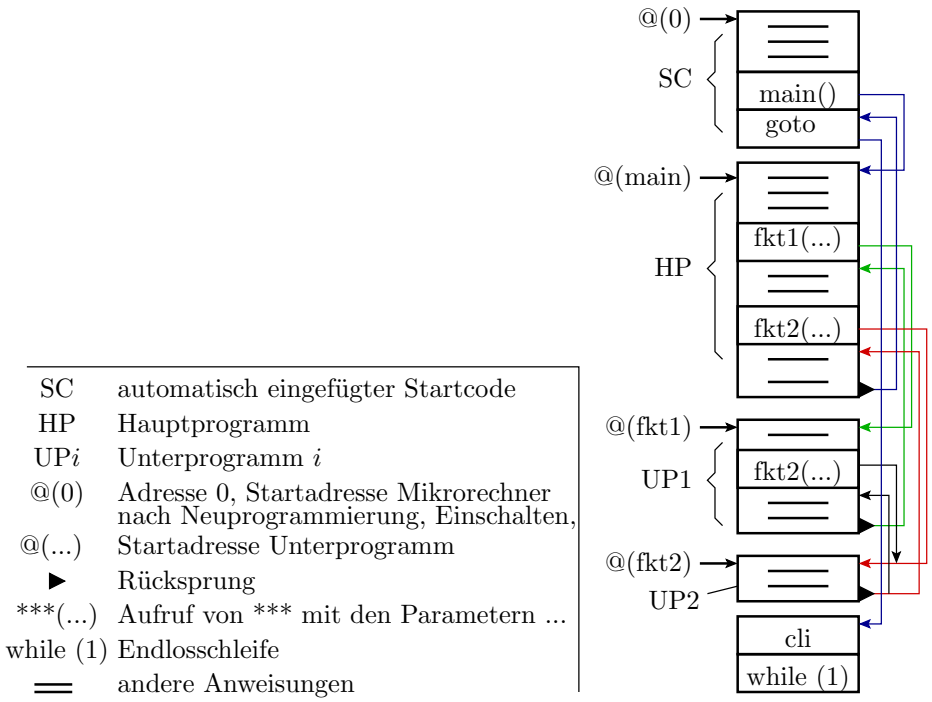
Wegen der Abhängigkeit vom Prozessor, dessen Takt, der Compiler-Optimierung, ... Wartezeiten besser mit Timer erzeugen (siehe später Foliensatz RA_F2).

7 Unterprogramme

Unterprogramme

Unterprogramme sind Programmbausteine,

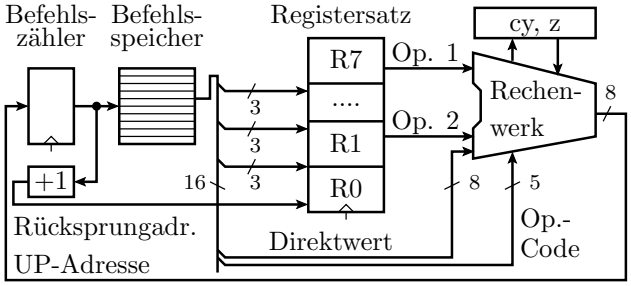
- die nur einmal im Befehlsspeicher stehen und
- durch Aufruf ihrer Adresse mehrfach in den Programmfluss eingefügt werden.



7.1 Hardware-Erweiterung MiPro

Hardware-Erweiterung für Call- und Return-Befehl

Befehl	Operation	Flags	cnr
call rd,imm	rd:=pc+1, pc:=imm		2
retu rd	pc := rd		15



Unterprogrammaufrufe auf MiPro

Das nachfolgende Unterprogramm bekommt in dmem(1) einen Wert und in r1 eine Adresse übergeben und schreibt den übergebenen Wert + 0x13 in den Datenspeicher auf die Übergabeadresse:

```
0000: ld_i r0,35,..        Unterprogramm:
0001: stor r0,01,..        0010: load r3,01,..
0002: ld_i r1,02,..        0011: addi r3,13,..
0003: call r5,10,..        0012: st_r r3,r1,..
0004: ld_i r0,46,..        0013: retu r5,..,..
0005: stor r0,01,..
0006: ld_i r1,04,..
0007: call r5,10,..
0008: jump 08,alw.. ; Endlosschleife
```

Testbeispiele:

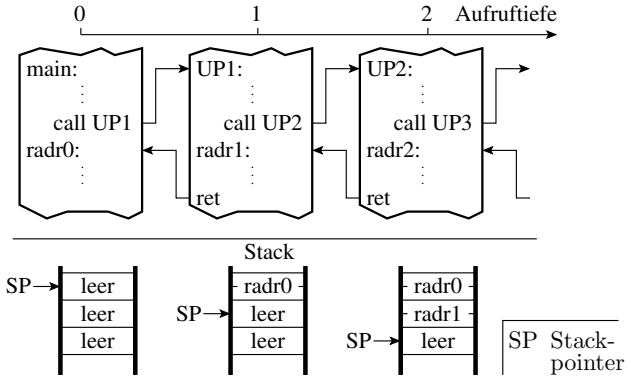
- Aufruf mit dmem(1)=0x35 und r1=2, Ergebnis dmem(2)=0x48
- Aufruf mit dmem(1)=0x46 und r1=4, Ergebnis dmem(4)=0x59

Beispielprogrammablauf mit Unterprogramm

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,35,..:2835|35 00 00 00 00 00 00 00|0|0|
01|stor r0,01,..:2001|35 00 00 00 00 00 00 00|0|0|
               dmem = [00 35 00 00 00 00 00 00]
02|ld_i r1,02,..:2902|35 02 00 00 00 00 00 00|0|0|
03|call r5,10,..:1510|35 02 00 00 00 04 00 00|0|0|
10|load r3,01,..:1b01|35 02 00 35 00 04 00 00|0|0|
11|addi r3,13,..:4313|35 02 00 48 00 04 00 00|0|0|
12|st_r r3,r1,..:9320|35 02 00 48 00 04 00 00|0|0|
               dmem = [00 35 48 00 00 00 00 00]
13|retu r5,..,..:7d00|35 02 00 48 00 04 00 00|0|0|
04|ld_i r0,46,..:2846|46 02 00 48 00 04 00 00|0|0|
05|stor r0,01,..:2001|46 02 00 48 00 04 00 00|0|0|
               dmem = [00 46 48 00 00 00 00 00]
  Fortsetzung nächste Folie =>
```

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
=> Fortsetzung
06|ld_i r1,04,..:2904|46 04 00 48 00 04 00 00|0|0|
07|call r5,10,..:1510|46 04 00 48 00 08 00 00|0|0|
10|load r3,01,..:1b01|46 04 00 46 00 08 00 00|0|0|
11|addi r3,13,..:4313|46 04 00 59 00 08 00 00|0|0|
12|st_r r3,r1,..:9320|46 04 00 59 00 08 00 00|0|0|
               dmem = [00 46 48 00 59 00 00 00]
13|retu r5,..,..:7d00|46 04 00 59 00 08 00 00|0|0|
08|jump 08,alw..:0908|46 04 00 59 00 08 00 00|0|0|
08|jump 08,alw..:0908|46 04 00 59 00 08 00 00|0|0|
               dmem = [00 46 48 00 59 00 00 00]
```

Stapelverwaltung der Rücksprungadressen



Damit Unterprogramme selbst Unterprogramme (inc. sich selbst) aufrufen können, werden Rückkehradressen auf einem Stapelspeicher (Stack) abgelegt und beim Rücksprung nach dem Prinzip »Last In First Out« wieder entnommen.

7.2 AVR UP-Aufruf, Stack, ...

Befehle für die Arbeit mit Unterprogrammen

Operation	T	Op.-Code	Assembler
PC←PC+k+1, STACK←PC+1, SP←SP-3	3	1101 kkkk kkkk kkkk	rcall k
PC←0b00:Z, STACK←PC+1, SP←SP-3	4	1001 010 0000 1001	icall k
PC←EIND:Z, STACK←PC+1, SP←SP-3	4	1001 010 0001 1001	eicall
PC←k, STACK←PC+1, SP←SP-3	5	1001 0100 0000 111k kkkk kkkk kkkk kkkk	call k
PC←STACK, SP←SP+3	5	1001 010 0000 1000	ret
STACK←Rr, SP←SP-1	2	1001 001d dddd 1111	push Rd
Rd←STACK, SP←SP+1	2	1001 000d dddd 1111	pop Rr

- k – 12 Bit (±2k) Sprungdistanz bzw. 17-Bit-Sprungziel.
- push und pop: Zwischenablage Registerinhalte auf Stack.

Stack einrichten

Der Stack ist ein Bereich des Datenspeichers, der vom Stackpointer adressiert wird. Der Stackpointer besteht aus den EA-Registern SPL und SPH auf den Adressen 0x3D und 0x3E. Auf dem Stack werden gespeichert:

- die Rücksprungadressen,
- die mit push gesicherten Registerinhalte und
- die lokalen Variablen.

Der Stack muss vor dem ersten Unterprogrammaufruf, d.h. vor Aufruf von main() initialisiert werden. Unser Compiler initialisiert den Stack im Startup-Code mit der höchsten Adresse des internen RAMs 0x21FF:

```
00074  SER R28
00075  LDI R29,0x21
00076  OUT 0x3E,R29
00077  OUT 0x3D,R28
```

7.3 Lokale Variablen

Globale und lokale Variablen

Globale Variablen

- werden außerhalb der Unterprogramme vereinbart und
- haben feste Adressen.

Lokale Variablen

- werden innerhalb der Unterprogramme vereinbart und

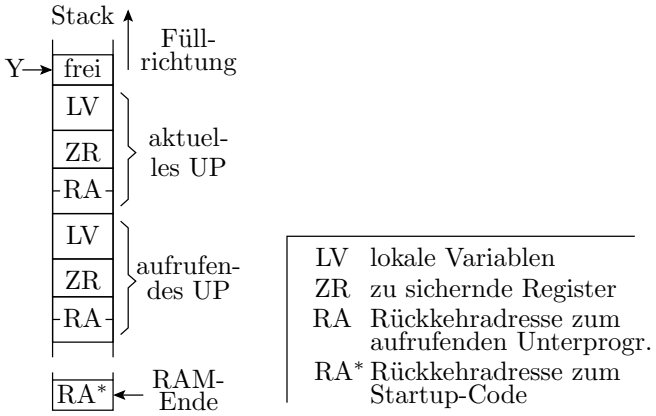
- erhalten Adressen auf dem Stack[16] relativ zum Frame-Pointer.

```
11  uint8_t g1, g2;
12  void main(void){
13    uint8_t l1 = 0x83;
14    uint8_t l2 = 0x45;
15    uint8_t l3 = 0x7A;
16    g1 = l1 + l2;
17    g2 = g1 + l3;
    }
```

Name	Value	Type (Locals)
l1	0x83	uint8_t{data}@0x21f8 ([R28]+1)
l2	0x45	uint8_t{data}@0x21f9 ([R28]+2)
l3	0x7a	uint8_t{data}@0x21fa ([R28]+3)

Name	Value	Type (Watch 1)
g1	0xc8	uint8_t{data}@0x0200
g2	0x42	uint8_t{data}@0x0201

Hier ist der Frame-Pointer Y gemeint.



- Im Beispielprozessor werden die Adressen für globale Variablen ab 0x200 aufsteigend vergeben. Der Stack beginnt am Speicherende und wird absteigend gefüllt. Die lokalen Variablen werden relativ zum Framepointer (Register Y) adressiert.
- Beim Unterprogrammaufruf werden Rücksprungadresse und zu sichernde Register (ZR) auf den Stack gelegt. Dann wird für die lokalen Variablen Platz geschaffen und dem Framepointer der Wert des Stackpointers zugewiesen.
- Beim Rücksprung zum aufrufenden Programm wird der Stack in umgekehrter Reihenfolge abgeräumt. Die lokalen Variablen sind danach ungültig.

Beispielprogramm mit -O0

```
11  uint8_t g1, g2;
12  void main(void){
13    uint8_t l1 = 0x83;
14    uint8_t l2 = 0x45;
15    uint8_t l3 = 0x7A;
16    g1 = l1 + l2;
17  } g2 = g1 + l3;
```

```
00085  PUSH R28          | 1
00086  PUSH R29          |
00087  RCALL PC+0x0001   | 2
00088  IN R28,0x3D       | 3
00089  IN R29,0x3E       |
0008A  LDI R24,0x83      | 13
0008B  STD Y+1,R24       |
0008C  LDI R24,0x45      | 14
0008D  STD Y+2,R24       |
0008E  LDI R24,0x7A      | 15
0008F  STD Y+3,R24       |
00090  LDD R25,Y+1       | 16
00091  LDD R24,Y+2       |
00092  ADD R24,R25       |
00093  STS 0x0200,R24    |
00095  LDS R25,0x0200    | 17
00097  LDD R24,Y+3       |
00098  ADD R24,R25       |
00099  STS 0x0201,R24    |
0009B  POP R0            | 4
0009C  POP R0            |
0009D  POP R0            |
0009E  POP R29           |
0009F  POP R28           |
000A0  RET               |
```

[16] Ab -O1 erhalten Variablen, wenn Platz ist, Registeradressen.

```
00085  PUSH R28          | 1
00086  PUSH R29          |
00087  RCALL PC+0x0001   | 2
00088  IN R28,0x3D       | 3
00089  IN R29,0x3E       |

0009B  POP R0            | 4
0009C  POP R0            |
0009D  POP R0            |
0009E  POP R29           |
0009F  POP R28           |
000A0  RET               |
```

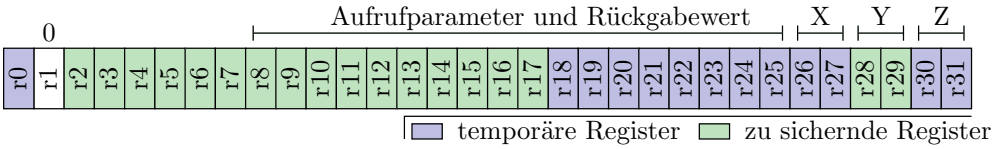
1. Sichern des Framepointers des aufrufenden Programms.
2. Der rcall-Befehl verringert den Stackpointer um 3. Das er dabei die Rückkehradresse 0x000088 auf den Stack schreibt, stört nicht, weil dieser Wert nie gelesen wird.
3. Zuweisen des neuen Stackpointer-Wertes an den Framepointer. Danach haben die lokalen Variablen die Adressen:

Variable	l1	ls2	l3
Adresse	Y+1	Y+2	Y+3

4. 3×pop r0 erhöht den Stackpointer um 3. Dann wird der alte Framepointer-Wert zurückgeholt und zurückgesprungen.

Gesicherte und zu sichernde Register

Außer dem Framepointer r29 und r28 müssen auch die anderen vom aufrufenden Programm genutzten Register vor Änderung durch das aufgerufene Programm auf den Stack gesichert werden.



Für den gcc in AVR-Studio gilt für die Registernutzung:

- In r1 wird bei UP-Aufruf der Wert null erwartet.
- r0, r18 bis r27 (incl. X), r30 und r31 (Z): Temporäre Register, die das aufgerufene Unterprogramm verändern darf. (Sicherung vor Aufruf.)
- r2 bis r17, r28:29 (Y): Zu sichernde Register. Vor Veränderung auf den Stack sichern und vor dem Rücksprung wiederherstellen.

- Bei Übersetzung mit -O0 erhalten nicht mit »register« vereinbarte Variablen Speicherplätze.
- Ab -O1 werden Variablen auch so freie Register zugeodnet.

Das folgende mit -O1 übersetzte Hauptprogramm legt die Variablen a, b und c in Registern an und optimiert die Variablen d und e weg.

```
10   uint8_t g;
11  void main(void){
12    uint8_t a = PINA;
13    uint8_t b = PINB;
14    uint8_t c = PINC;
15    uint8_t d = a + b;
16    uint8_t e = a - c;
17    g = d | e;
    }
```

Watch 1

Name	Value	Type
a	0x00	uint8_t{registers}@R24
b	0x02	uint8_t{registers}@R18
c	0x00	uint8_t{registers}@R25
d	Optimized away	Error
e	Optimized away	Error
g	0x00	uint8_t{data}@0x0200

Die genutzten Register r24, r18 und r25 sind temporäre Register und müssen nicht gesichert werden.

Die Mehrheit der C-Anweisung werden in dem Beispiel direkt in einen Maschinenbefehl übersetzt.

```
10   uint8_t g;                Adresse: 0x200
11  void main(void){
12    uint8_t a = PINA;  | 00085  IN R24,0x00
13    uint8_t b = PINB;  | 00086  IN R18,0x03
14    uint8_t c = PINC;  | 00087  IN R25,0x06
                         | 00088  MOV R19,R24
16    uint8_t e = a - c; | 00089  SUB R19,R25
15    uint8_t d = a + b; | 0008A  ADD R24,R18
17    g = d | e;         | 0008B  OR R24,R19
                         | 0008C  STS 0x0200,R24
    }                    | 0008E  RET
```

Mit dem zusätzlichen Aufruf eines Unterprogramms, das auch Register für seine lokalen Variablen verwendet, nimmt der Compiler statt der temporären Register r18, r24 und r25 die zu sichernden Register r17, r28 und r29:

```
18  void main(void){
19    uint8_t a = PINA;
20    uint8_t b = PINB;
21    uint8_t c = PINC;
22    UP();
23    uint8_t d = a + b;
24    uint8_t e = a - c;
25    g = d | e;
26  }
```

Watch 1

Name	Value	Type
a	0xff	uint8_t{registers}@R28
b	0x21	uint8_t{registers}@R29
c	0x00	uint8_t{registers}@R17
d	Optimized away	Error
e	Optimized away	Error
g	0x00	uint8_t{data}@0x0200
h	0x00	uint8_t{data}@0x0201

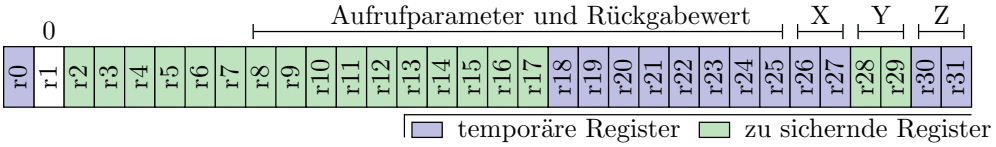
Diese werden am Anfang von main() auf den Stack gesichert und am Ende von main() wieder von Stack geholt.

```
                                    18 void main(void){
0008B  PUSH R17
0008C  PUSH R28                        1
0008D  PUSH R29
0008E  IN R28,0x00                  19     uint8_t a = PINA;
0008F  IN R29,0x03                  20     uint8_t b = PINB;
00090  IN R17,0x06                  21     uint8_t c = PINC;
00091  RCALL PC-0x000C              22     UP();
00092  MOV R24,R28
00093  SUB R24,R17                  24     uint8_t e = a - c;
00094  ADD R28,R29                  23     uint8_t d = a + b;
00095  OR R28,R24                   25     g = d | e;
00096  STS 0x0200,R28
00098  POP R29                         2
00099  POP R28
0009A  POP R17                      26 }
```

1. Register r17, r28 und r29 auf den Stack ablegen.
2. Register r17, r28 und r29 vom Stack zurückladen.

7.4 Parameterübergabe

Registernutzung und Parameterübergabe



- Von rechts beginnen werden die ersten 18 Aufrufparameterbytes in den Registern r25:8 und alle weiteren auf dem Stack übergeben. 1-Byte Parameter nutzen nur jedes zweite Byte.
- Die Rückgabe erfolgt in den Registern r25:8.

```
12  uint16_t UP(uint16_t a,    // Übergabe in r25:r24
13              uint16_t b){   // Übergabe in r23:r22
14    uint16_t c = a << 1;
15    uint16_t d = c | b;
16    return d;                // Rückgabewert in r25:r24
17  }
```

```
12  uint16_t UP(uint16_t a,
13              uint16_t b){
14    uint16_t c = a << 1;
15    uint16_t d = c | b;
16    return d;
17  }
```

Name	Value	Type
a	0x034a	uint16_t{registers}@ R25 R24
b	0x0127	uint16_t{registers}@ R23 R22
c	Unknown locati	Error
d	Unknown locati	Error

- Registerzuordnung der Übergabeparameter wie vorhergesagt.

```
12  uint16_t UP(uint16_t a,
13                uint16_t b){
14    uint16_t c = a << 1;
15    uint16_t d = c | b;
16    return d;
17  }
```

Name	Value	Type
a	Unknown locat	Error
b	Unknown locat	Error
c	0x0694	uint16_t{registers}@ R25 R24
d	0x07b7	uint16_t{registers}@ R23 R22

- Wenn a und b nicht mehr gebraucht werden, Neuvergabe der Register, im Beispiel an die Variablen c und d.
- Vor dem Rücksprung muss der Wert der Variablen d (r23:r22) in das Registerpaar r25:r24 kopiert werden.

```
0007D  LSL R24
0007E  ROL R25
0007F  OR R22,R24
00080  OR R23,R25
00081  MOV R24,R22
00082  MOV R25,R23
00083  RET
```

```
12  uint16_t UP(uint16_t a,
13                uint16_t b){
14    uint16_t c = a << 1;
15    uint16_t d = c | b;
16    return d;
17  }
```

d (r23:r22) auf den Rückgabeplatz (r25:r24) kopieren

```
00084  LDI R22,0x27
00085  LDI R23,0x01
00086  LDI R24,0x4A
00087  LDI R25,0x03
```

Übergabewerte für a und b schreiben

```
19  int main(){
20    uint16_t e=UP(0x34a, 0x127);
21    return e + 4;
22  }
```

```
00088  RCALL PC-0x000B
00089  MOVW R18,R24
0008A  SUBI R18,0xFC
0008B  SBCI R19,0xFF
```

Subtraktion 0xFE=-4

```
0008C  MOV R24,R18
0008D  MOV R25,R19
0008E  RET
```

e (r19:r18) auf den Rückgabeplatz (r25:r24) kopieren

7.5 Rekursion

Rekursion

Ein rekursives Programm ruft sich so lange selbst auf, bis eine Abbruchbedingung erreicht ist. Beispiel für einen rekursiv beschreibarer Algorithmus ist die Berechnung der Fakultät:

$$n! = \begin{cases} n \cdot (n-1)! & n > 1 \\ 1 & \text{sonst} \end{cases}$$

Ein rekursives Programm speichert bei jedem Aufruf von sich selbst die Rücksprungadresse und die zu sichernden Registerinhalte auf den Stack und reserviert Platz für die lokalen Variablen. Gute Demonstration einer tiefen Unterprogrammverschachtelung und einer intensiven Stack-Nutzung.

Rekursive Rechtsverschiebung

```
11  uint32_t rshift(uint32_t a, uint8_t n){
12    if (n==0) return a;
13    else return rshift(a>>1, n-1);
    }
16  void main(void){
17    uint32_t b=rshift(0x5F317E1A, 5);
    }
```

Die 4-Byte Variable a (r25, r24, r23, r22) wird bei jedem Aufruf halbiert und die Variable n (r20) um eins verringert:

	Name	Value	Type
1. Aufruf	a	0x5f317e1a	uint32_t{registers}@ R25 R24 R23 R22
	n	0x05	uint8_t{registers}@R20
2. Aufruf	a	0x2f98bf0d	uint32_t{registers}@ R25 R24 R23 R22
	n	0x04	uint8_t{registers}@R20
...	...		
5. Aufruf	a	0x05f317e1	uint32_t{registers}@ R25 R24 R23 R22
	n	0x01	uint8_t{registers}@R20

Das Hauptprogramm

```
00092  LDI R20,0x05      Variable B mit 5 initialisieren
00093  LDI R22,0x1A
00094  LDI R23,0x7E      Variable a mit
00095  LDI R24,0x31      0x5F317E1A
00096  LDI R25,0x5F      initialisieren
00097  RJMP PC-0x001A    Sprung zum Unterprogramm.
```

- Durch den Ansprung des Unterprogramms ist der Rücksprung vom Unterprogramm gleich der Rücksprung von main() zum Startup-Code (Adresse 0xB7).

Das Unterprogramm beginnt ab Adresse 0x7D:

```
00007D  PUSH R16        Registerinhalte von r16 und r17
00007E  PUSH R17        auf den Stack ablegen.
00007F  MOVW R16,R22    Die Werte der Aufrufvariablen a
000080  MOVW R18,R24    in die Register r16 bis r19 kopieren.
000081  TST R20         Test, ob Aufrufparameter b null ist. Wenn
000082  BREQ PC+0x09    ja, springe zum Ende des Unterprogramms.
```

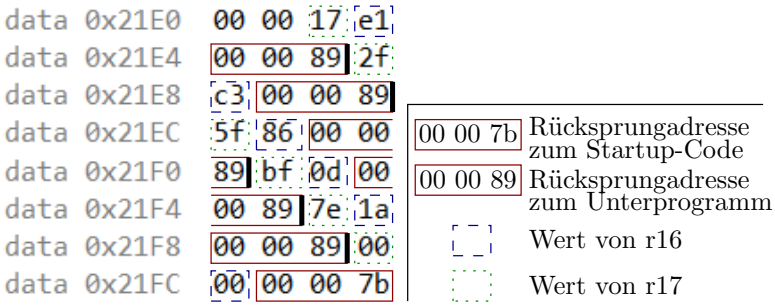
```
00083  LSR R25
00084  ROR R24          Rechstverschiebung der 4
00085  ROR R23          Bytes der Variablen a.
00086  ROR R22
00087  SUBI R20,0x01    b ← b-1
00088  RCALL PC-0x000B  Erneuter Aufruf von sich selbst.
00089  MOVW R16,R22     Rückgabewert in r22 bis r25 in
0008A  MOVW R18,R24     r16 bis r19 kopieren.
0008B  MOV R22,R16
0008C  MOV R23,R17      r16 bis r19 zurück in r22 bis
0008D  MOV R24,R18      r25 kopieren (offenbar ohne Sinn).
0008E  MOV R25,R19
0008F  POP R17          Die auf den Stack abgelegten Inhalte
00090  POP R16          zurück in r16 und r17 kopieren.
00091  RET              Rücksprung
```

Bei jedem Aufruf werden auf den Stack abgelegt:

- die Rücksprungadresse (17 Bit, 3 Bytes)
- die mit push abgelegten Registerinhalte von r16 und r17.

Beim 5. Stopp am Unterbrechungspunkt liegen auf dem Stack:

- 21FD bis 21FF: Rücksprungadresse zum Startup-Code,
- 5×die Rücksprungadresse zu sich selbst und
- 6× die Registerinhalte von r16 und r17.



7.6 Re-Engineering Division

Division (Compileroptimierung -O1)

```
#include <avr/io.h>
uint16_t a=0x6F, b=0x11, q;
int main(){
  q = a/b;
}
```

```
// 0x0092  LDS R24,0x0202 ; Aufrufparameter a
// 0x0094  LDS R25,0x0203
// 0x0096  LDS R22,0x0200 ; Aufrufparameter b
// 0x0098  LDS R23,0x0201
// 0x009A  RCALL PC+0x0008; Aufruf der Division
// 0x009B  STS 0x0205,R23 ; q = Rückgabewert_2
// 0x009D  STS 0x0204,R22
// 0x009F  LDI R24,0x00   ; Rückgabewert von main()=0
// 0x00A0  LDI R25,0x00
// 0x00A1  RET            ; Rücksprung von main()
// 0x00A2  ...            ; Divisions-UP
```

Divisions-Unterprogramm

```
//  0x00A2  SUB R26,R26   ; r27:r26 = 0 (Rest)
//  0x00A3  SUB R27,R27   ; 2. Befehl
//  0x00A4  LDI R21,0x11  ; r21=0x11 (Schleifenz.)
//  0x00A5  RJMP PC+0x0008; springe nach M1 (0xAD)
// M2:0xA6  ROL R26       ; r27:r26 << 1 + cy
//  0x00A7  ROL R27       ; (Schiebe Übertrag in r)
//  0x00A8  CP R26,R22  ; Teste r27:r26-r23:r22
//  0x00A9  CPC R27,R23   ; (Teste r-b)
//  0x00AA  BRCS PC+0x03  ; wenn r-b<0, gehe zu M1
//  0x00AB  SUB R26,R22   ; sonst r27:r26 -= r23:r22
//  0x00AC  SBC R27,R23   ; (r = r-b)
// M1:0xAD  ROL R24        ; r25:r24 << 1 + cy
//  0x00AE  ROL R25       ; (Schiebe cy in qn)
//  0x00AF  DEC R21       ; r21-- (Schleifenzähler)
//  0x00B0  BRNE PC-0x0A  ; springe nach M2 (0xB6)
```

16 Schleifendurchläufe zu je etwa 10 Takten (ca. 50 ms).

```
//  0x00B1  COM R24         ; r25:r24 = not(r25:r24)
//  0x00B2  COM R25         ; (q = not(pn))
//  0x00B3  MOVW R22,R24    ; Rückgabe q als 2. Parameter
//  0x00B4  MOVW R24,R26    ; Rückgabe r als 1. Parameter
//  0x00B5  RET             ; Rücksprung
```

$$\frac{\mathbf{a}}{\mathbf{b}} = \mathbf{q} + \frac{\mathbf{r}}{\mathbf{b}}$$

Berechnung des Quotienenten $\mathbf{q}$ und des Divisionsrest $\mathbf{r}$ für $a = 11$ und $b = 3$:

Bitnummer	3	2	1	0	Ergebnis
Rest	1011 →	1011 →	1011 →	0101 →	$\mathbf{r} = 0010$ (2)
Subtrahend	− 0011	− 0011	− 0011	− 0011	
Differenz	negativ	negativ	0101	0010	
Quotient	$q_3 = 0$	$q_2 = 0$	$q_1 = 1$	$q_0 = 1$ →	$\mathbf{q} = 0011$ (3)

Extrahierter Divisionsalgorithmus

Bitnummer	3	2	1	0	Ergebnis
Rest	1011 →	1011 →	1011 →	0101 →	$\mathbf{r} = 0010$ (2)
Subtrahend	− 0011	− 0011	− 0011	− 0011	
Differenz	negativ	negativ	0101	0010	
Quotient	$q_3 = 0$	$q_2 = 0$	$q_1 = 1$	$q_0 = 1$ →	$\mathbf{q} = 0011$ (3)