Rechnerarchitektur, Foliensatz 2 Speicher und Rechenwerk

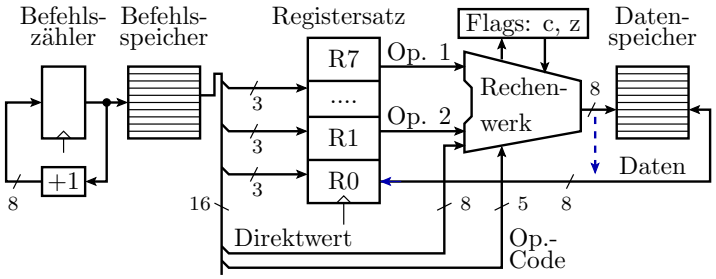
G. Kemnitz

18. Oktober 2021

Inhaltsverzeichnis

1 Speicher

RISC-Speicherarchitektur (MiPro)



256 × 8 Bit Befehls- und Datenspeicher, 8 × 8 Bit Registersatz.

Die Ausführung von einem Befehl pro Schritt erfordert:

- getrennten Befehls- und Datenspeicher,
- 3-Port-Registersatz hier 8, typ. 32 Speicherplätze.
- Datenspeicheradresse: Register, Konstante oder berechnet.

1.1 Lade- / Speicherbefehle MiPro

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 := *(imm)		3
stor rd,imm	*(imm) := rd	unver-	4
ld_r rd,ra	rd := *(ra)	ändert	17
st_r rd,ra	*(ra) := rd		18

Unterstütze Adressierungsarten:

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

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

Beispielprogramm

- Welche Werte werden in die Register, Flags 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|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
00|ld_i r0,48,..:2848|.. .. .. .. .. .. .. ..|.|.|
01|stor r0,05,..:2005|.. .. .. .. .. .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
02|ld_i r1,06,..:2906|.. .. .. .. .. .. .. ..|.|.|
03|ld_i r2,31,..:2a31|.. .. .. .. .. .. .. ..|.|.|
04|st_r r2,r1,..:9220|.. .. .. .. .. .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
05|load r3,05,..:1b05|.. .. .. .. .. .. .. ..|.|.|
06|ld_r r4,r1,..:8c20|.. .. .. .. .. .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
```

Zur Kontrolle

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,48,..:2848|48 .. .. .. .. .. .. ..|.|.|
01|stor r0,05,..:2005|** .. .. .. .. .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. 48 .. ..]
02|ld_i r1,06,..:2906|** 06 .. .. .. .. .. ..|.|.|
03|ld_i r2,31,..:2a31|** ** 31 .. .. .. .. ..|.|.|
04|st_r r2,r1,..:9220|** ** ** .. .. .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. ** 31 ..]
05|load r3,05,..:1b05|** ** ** 48 .. .. .. ..|.|.|
06|ld_r r4,r1,..:8c20|** ** ** ** 31 .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. ** ** ..]
```

1.2 ATmega2560

AVR-Speicherarchitektur

Befehlsspeicher (Flash): 128k×2 Byte. Adressierungsmöglichkeiten:

- 17 Bit-Befehlszähler.
- 17 Bit-Konstante[1] (direkte Sprünge).
- 16 Bit-Register erweitert um führende Null / SFR-Bit[2] für Befehlsadressen.

[1] Verlangt in Abweichung zum RISC-Konzept ein Doppelbefehlswort,
[2] SFR: **S**pecial **F**unction **R**egister: Port-Register, Stack-Pointer ...

- 16 Bit-Register erweitert um 2 führende Nullen / 2 SFR-Bits für Byte-Konstanten im Programmspeicher.

Datenspeicher (RAM): 8k×1 Byte + viele Spezialregister + ...:

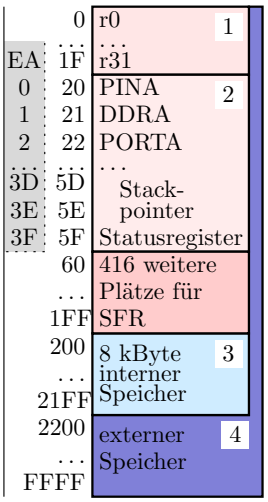
- direkte Adressierung mit 16 Bit-Konstanten,
- indirekte Adressierung mit Registerpaaren (2 Byte), ...

Nicht flüchtiger Datenspeicher (EEPROM): 2k×1 Byte :

- Lesen und Beschreiben über Spezielregister.

Datenspeicher ATmega2560

- 32 Arbeitsregister,
- Platz für 480 SFR,
- 8 kByte interner Datenspeicher,



adressierbar über LS-Befehle, mit 16-Bit-Direktwert »k«

```
lds Rd, K; sts K, Rd
```

oder 16-Bit-Adressregister X, Y und Z, ...

```
ld Rd, X; st X, Rd; ..
```

Die unteren 64 SFR haben zusätzlich eine 6-Bit I/O-Adresse »A« für den Zugriff mit:

```
in Rd, A ; Rd := *(A) (Eingabe)
out A, Rd; *(A) := Rr (Ausgabe)
```

1.3 AVR-Lade- und Speicherbefehle

Direkte Adresierung

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

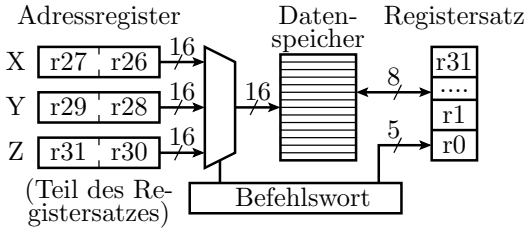
(TZ – Taktzyklen, Ausführungsschritte; *(K) – Inhalt von Speicherplatz K).

Beispiel:

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

Indirekte Adressierung mit Post-Increment, ...

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

Kopierschleife mit Pos-Increment

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

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 (DDRJ)
// 0x008A  LDI R25,0x01  ;
// 0x008F  SER R18       ; r18 := 0xFF
// 0x0090  MOVW R30,R24  ; r31:r30(Z) := r25:r24
// 0x0091  STD Z+0,R18   ; *(Z+0) := r18
 while(1) <Schalter lesen, LED-Ausgabe berechnen> ... }
```

Bei Handoptimierung würde man Port J direkte adressieren:

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

nur 2 statt 5 Befehle. Für höhere Optimierungsstufen als -O0 findet auch der Compiler bessere Code-Folgen.

1.4 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):

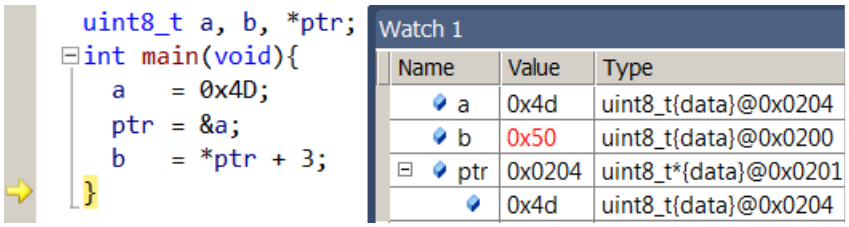
```
uint8_t dat = 45;
```

- 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	$[-2^{15}, 2^{15}-1]$	

Variablen und Zeiger

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



- Zeiger sind Variablen für Adressen, z.B.:

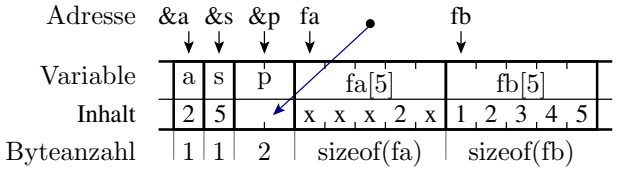
```
uint8_t a, *ptr; // Variable und Zeiger
a = *ptr;        // Zuweisung Inhalt *(ptr)
ptr = &a;        // Zuweisung Adresse von a
```

Felder

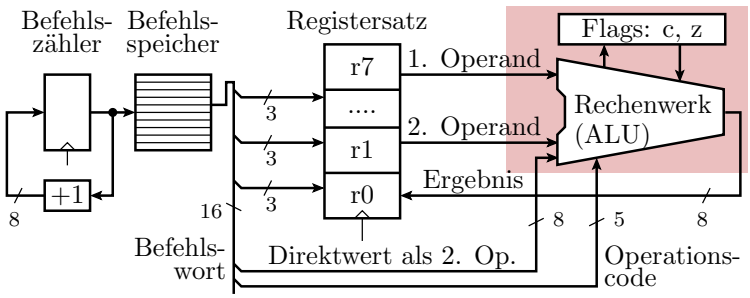
Reservierung von Datenspeicherplätzen, definiert durch:

- Zeigerkonstante für den Feldanfang und
- eine Konstante für die Anzahl der Elemente.

```
uint8_t s, a, *p;// 8-Bit-Variablen, Zeiger auf ...
uint8_t fa[5]    // Feld, 5 Elemente, nicht inital.
uint8_t fb[]={1,2,3,4,5}; // Feld initialisiert
s = sizeof(fb);  // Anzahl der Feldelemente
p = fa+2;        // Adresse Element 2
a = *(fb+1);     // Inhalt Feldelement 1
fa[3] = a;       // Beschreibe Feldelement 3
```



2 Arithmetik



Das Rechenwerk eines Prozessors (ALU **A**rithmetic **L**ogical **U**nit) kann außer den bitweisen Logikoperationen und Verschiebeoperationen auch addieren und subtrahieren:

$$\begin{aligned} y &= a + b\,[+c] \\ y &= a - b\,[-c] \end{aligned}$$

(a,b,y – Bitvektoren der Prozessorverarbeitungsbreite, c – Carry Flag / Übertragsbit). Anwendung: Zählen, Adressrechnung, ...

2.1 Addition ganzer Zahlen

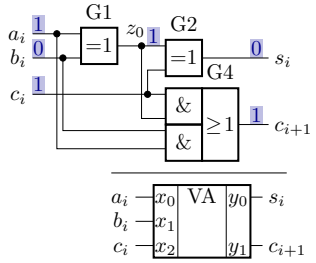
Addition von Binärzahlen

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

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

- 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.

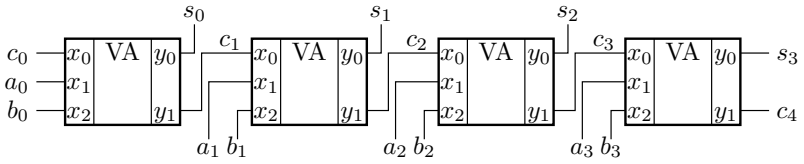
Eingabe b_i	a_i	c_i	Ist-Funktion z	c_{i+1}	s_i	Soll-Funktion $a_i+b_i+c_i$ dez	bin
0	0	0				0	00
0	0	1				1	01
0	1	0				1	
0	1	1				2	
1	0	0				1	
1	0	1	1	1	0	2	10
1	1	0				2	
1	1	1				3	11

Kontrolle durch Vervollständigung der Wertetabelle.

Addierer für Bitvektoren (Ripple-Addierer)

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

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.

2.2 Zweierkomplement und Subtraktion

Vorzeichenbehaftete Zahlen

Statt durch Vorzeichen und Betrag Darstellung durch »Stellenkomplement +1«. 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:

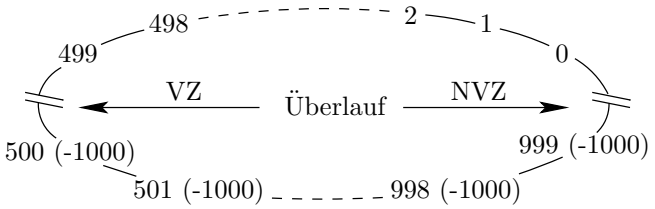
  $$Z + \bar{Z} + 1 = B^n$$

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

- Auflösung nach $-Z$:

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

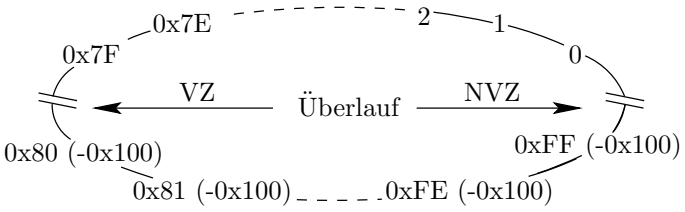
- Die Zählreihenfolge bleibt, nur der Darstellungsbereich verschiebt sich:



$$217 - 437 = 217 + \overline{437} + 1 = 217 + 562 + 1 = 780 - 1000 = -220$$

Zweierkomplement

- Basis: $B = 2$; 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

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

Subtraktion und Subtrahierer

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

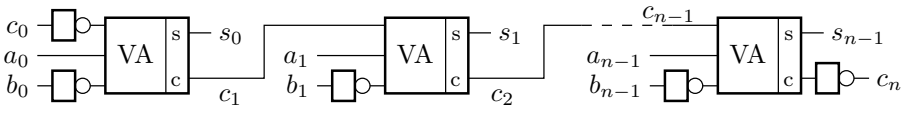
bitweise Subtraktion ($c_0 = 1$)

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
$(+c_i)$	(0)	(1)	(0)	(0)	(0)	(1)	(0)	(1)	$-c_0$ $\}\bar{c}_0$
		0	0	1	1	1	0	1	0

Daraus resultierende Schaltung des Subtrahierers:



2.3 Add- und Sub-Befehle MiPro

Additions- und Subtraktionsbefehle MiPro

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

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

Ergänzen der Registerinhalte, die der Prozessor verändert:

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

(.. – zu ergänzende / unbekannt Hex-Werte; ** – unverändert).

Lösung

```
r0:r1 := 0x733A;
r2:r3 := 0x13E7;
r4:r5 := r0:r1 + r2:r3; Ergebnis: 0x8721
r6:r7 := r0:r1 - r2:r3; Ergebnis: 0x5F53
```

Programm mit ergänzten Registerinhalten:

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

2.4 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
Rdd := Rdd + K$^{(6)}$	1001 0110 kkdd kkkk	adiw Rd+1:Rd, K
Rd := Rd - Rr	0001 10rd dddd rrrr	sub Rd, Rr
Rd := Rd$^{(2)}$- K$^{(6)}$	0101 kkkk dddd kkkk	subi Rd, K
Rd := Rd - Rr - c	0000 10rd dddd rrrr	sbc Rd, Rr
Rd := Rd$^{(2)}$- K$^{(8)}$ - c	0100 kkkk dddd kkkk	sbci Rd, K
Rdd := Rdd - K$^{(6)}$	1000 0111 kkdd kkkk	sbiw Rd+1:Rd, K

(Rdd – Doppelregister R25:R24, R27:R26 (X), R29:R28 (Y) oder R31:R30 (Z); K$^{(6)}$ – 6-Bit-Konstante, WB: 0..63; $^{(2)}$ nur R16 bis R31; K$^{(8)}$ – 8-Bit-Konstante; Additionen von 8-Bit-Konstanten erfolgen durch Subtraktion des 2er-Komplements.

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 - 0x5B(1)
// 0x00A1  SBCI R25,0xEC ; r25 := r25 - 0xEC-c(2)
// 0x00A2  STS 0x0207,R25;  s.Byte1 := r25
// 0x00A4  STS 0x0206,R24;  s.Byte0 := r24
```

(1) $0x5B = \overline{0xA5} + 1$; (2) $\underline{0xEC} - c = \overline{0x13} + 1 + c$

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 auch ohne die beiden movw-Befehle.

3 Aufgaben

Aufgabe 2.1: Load/Store MiPro

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
00|ld_i r0,3e,..:283e|.. .. .. .. .. .. .. ..|.|.|
01|ld_i r1,2a,..:292a|.. .. .. .. .. .. .. ..|.|.|
02|ld_i r2,af,..:2aaf|.. .. .. .. .. .. .. ..|.|.|
03|ld_i r3,01,..:2b01|.. .. .. .. .. .. .. ..|.|.|
04|stor r0,00,..:2000|.. .. .. .. .. .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
05|st_r r1,r3,..:9160|.. .. .. .. .. .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
06|addi r3,01,..:4301|.. .. .. .. .. .. .. ..|.|.|
07|st_r r2,r3,..:9260|.. .. .. .. .. .. .. ..|.|.|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
08|load r4,00,..:1c00|.. .. .. .. .. .. .. ..|.|.|
09|addr r5,r3,r4:c570|.. .. .. .. .. .. .. ..|.|.|
0a|noop ..,..,..:0000|.. .. .. .. .. .. .. ..|.|.|
```

Lösung

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
        ;dmem(0:7) = [.. .. .. .. .. .. .. ..]
00|ld_i r0,3e,..:283e|3e .. .. .. .. .. .. ..|.|.|
01|ld_i r1,2a,..:292a|** 2a .. .. .. .. .. ..|.|.|
02|ld_i r2,af,..:2aaf|** ** af .. .. .. .. ..|.|.|
03|ld_i r3,01,..:2b01|** ** ** 01 .. .. .. ..|.|.|
04|stor r0,00,..:2000|** ** ** ** .. .. .. ..|.|.|
        ;dmem(0:7) = [3e .. .. .. .. .. .. ..]
05|st_r r1,r3,..:9160|** ** ** 01 .. .. .. ..|.|.|
        ;dmem(0:7) = [** 2a .. .. .. .. .. ..]
06|addi r3,01,..:4301|** ** ** 02 .. .. .. ..|0|0|
07|st_r r2,r3,..:9260|** ** ** ** .. .. .. ..|*|*|
        ;dmem(0:7) = [** ** af .. .. .. .. ..]
08|load r4,00,..:1c00|** ** ** ** 3e .. .. ..|*|*|
09|addr r5,r3,r4:c570|** ** ** ** ** 40 .. ..|0|0|
0a|noop ..,..,..:0000|** ** ** ** ** ** .. ..|*|*|
```

Aufgabe 2.2: Variablen und Zeiger

Für den nachfolgenden Ausschnitt aus einem C-Programm sind die Werte und für den Zeiger die unter der Adresse gespeicherten Werte als Hex-Zahlen zu ergänzen:

```
//    Variablen     |Adresse|  Wert  |Inhalt
uint8_t a=0x21; //  |&a:0203|a: 21   |
uint8_t b=0x42; //  |&b:0204|b: 42..|
uint8_t c=0x00; //  |&c:0205|c: 00..|
uint8_t *p = &a;//  |&p:0206|p: 0203|*p: ..
c += *p;        //  |&c:....|c: ....|
p += 1;         //  |&p:....|p: ....|*p: ..
c += *p + a;    //  |&c:....|c: ....|
p = &a;         //  |&p:....|p: ....|*p: ..
*p= a + *(&a+1);//  |&p:....|p: ....|*p: ..
```

Welcher Wert steht am Ende in der Variablen a?

Lösung

```
//    Variablen     |Adresse|  Wert  |Inhalt
uint8_t a=0x21; //  |&a:0203|a: 21..|
uint8_t b=0x42; //  |&b:0204|b: 42..|
uint8_t c=0x00; //  |&c:0205|c: 00..|
uint8_t *p = &a;//  |&p:0206|p: 0203|*p: 21
c += *p;        //  |&c:0205|c: 21..|
p += 1;         //  |&p:0206|p: 0204|*p: 42
c += *p + a;    //  |&c:0205|c: 63..|
p = &a;         //  |&p:0206|p: 0203|*p: 21
*p= a + *(&a+1);//  |&p:0206|p: 0203|*p: 63
```

Endwert in Variable a: 0x63

Aufgabe 2.3: Zahlendarstellung, Zweierkomplement

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 welche 8-Bit binär und durch welche 8-Bit Hex.-Zahl wird der dezimale Wert -15 im Zweierkomplement dargestellt?

4. Durch welche 4-stellige Hex.-Zahl wird der hexadezimale Wert -0x3A5F im Zweierkomplement dargestellt?

Lösung

1.

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

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

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

   Dazu eins addiert ergibt null. Der Übertrag geht verloren.

3. $$\begin{aligned} 15 &= 0\text{b}0000.1111 = 0\text{x}0\text{F} \\ -15 &= 0\text{b}1111.0000 + 1 = 0\text{b}1111.0001 = 0\text{xF}1 \end{aligned}$$

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:

$$-0\text{x}3\text{A}5\text{F} = 0\text{xC}5\text{A}0 + 1 = 0\text{xC}5\text{A}1$$

Aufgabe 2.4: Arithmetische Rechtsverschiebung

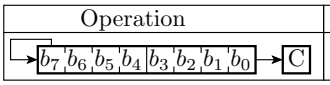
Die folgende (arithmetische) Rechtsverschiebung ist eine Division durch 8 unter Erhalt des Vorzeichens:

```
int16_t a, b;
b = a >> 3;    // b := a/8
```

1. Bestimmen Sie die Sollwerte von b für a=0xF3A2 und 0x41D3.
2. Schreiben Sie ein Assemblercodefolge mit folgender Registerzuordnung:

r18	r19	r20	r21
b_0	b_1	a_0	a_1

Lösungshinweis: Sie benötigen den Befehl für die arithmetische Rechtsverschiebung:

Operation	Operationscode	Assembler
	1001 010d dddd 0101	asr Rd

Lösung

```
        ;      Aufgabe a      |      Aufgabe b
        ;       r21:r20       |       r21:r20
movw r20,r18;-------------------------------------
        ;1111.0011.1010.0010|0100.0001.1101.0011

asr r21 ;1111.1001.1010.0010|0010.0000.1101.0011
ror r20 ;1111.1001.1101.0001|0010.0000.1110.1001

asr r21 ;1111.1100.1101.0001|0001.0000.1110.1001
ror r20 ;1111.1100.1110.1000|0001.0000.0111.0100

asr r21 ;1111.1110.1110.1000|0000.1000.0111.0100
ror r20 ;1111.1110.0111.0100|0000.1000.0011.1010
```

Kontrollrechnung:

a	dez.	b=a/8	hex. Betrag	2er-Kompl	Ist-Wert
0xF3A2	-3165	-396	-0x018C	0xFE73	0xFE74
0x41D3	16.851	2.106	0x083A		0x083A