Rechnerarchitektur, Foliensatz 4 Ergänzungen

G. Kemnitz

January 16, 2020

Contents

1 Konstanten **1**

2 Multiplikation **3**

3 Division **6**

4 Gleitkommazahlen **8**

5 Rekursion **10**

6 Aufgaben **11**

1 Konstanten

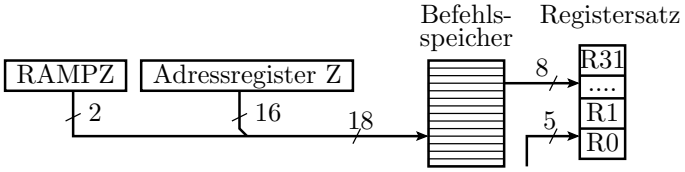
Speichern von Konstanten

Konstanten, die auch nach Neueinschalten des Prozessors noch vorhanden sein sollen, z.B. der Text »Hallo Welt« in

```
uint8_t a[]="Hallo Welt";
int main(){...}
```

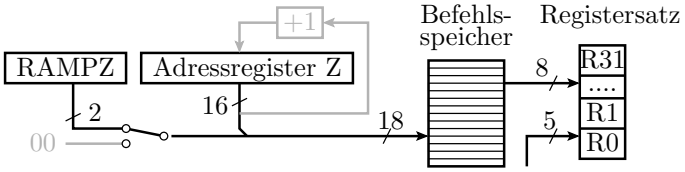
werden im Programmspeicher abgelegt und beim Programmstart in den Datenspeicher kopiert.

Die Adressierung des 256 kByte-Befehlsspeichers erfolgt indirekt mit einer 18-Bit Adresse. Die niederwertigen 16 Adressbits werden aus Register Z und die oberste 2 Bit aus EA-Register RAMPZ (Adresse 0x5B/0x3B) genommen.



Befehlsvariationen:

- höchste Adressbits 00 statt der Bits RAMPZ(1:0)
- mit Post-Inkrement



Operation	TZ	Op.-Code	Assembler
`Rd := p(Z)`	3	1001 000d dddd 0100	`lpm Rd, Z`
`Rd := p(Z); Z := Z+1`	3	1001 000d dddd 0101	`lpm Rd, Z+`
`Rd := p(RAMPZ:Z)`	3	1001 000d dddd 0110	`elpm Rd, Z`
`Rd := p(RAMPZ:Z); Z:=Z+1`	3	1001 000d dddd 0111	`elpm Rd, Z+`

(p(..) – Programmspeicherinhalt von ..)

Beim Übersetzen des Programms rechts schreibt der Compiler die Zeichenkettenkonstante »Hallo Welt« hinter die Endlosschleife des Startup-Codes ab Adresse 0xA1:

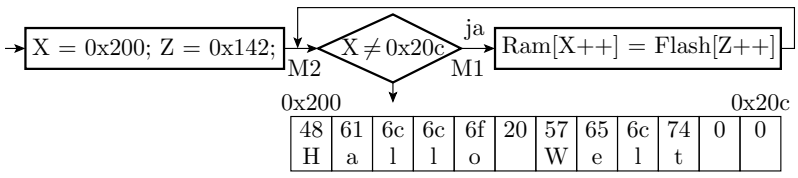
```
11  uint8_t a[] = "Hallo Welt";
12  uint8_t b[10];
13  int main(void){
14    uint8_t *p1=a;
15    uint8_t *p2=b;
16    while (*p1){
17      *(p2++) = *(p1++);
    }
}
```

Zeichenkettenkonstante	dissassembliert	als Ascii-Text
000000A1 48.61	ORI R20,0x18	Ha
000000A2 6c.6c	ORI R22,0xCC	ll
000000A3 6f.20	AND R6,R15	o␣
000000A4 57.65	ORI R21,0x57	We
000000A5 6c.74	ANDI R22,0x4C	lt
000000A6 00.00	NOP	\0

Der Disassembler kann Zeichenketten nicht von Programmcode unterscheiden.

Im Startup-Code vor »main():

```
 //0x007A  LDI R17,0x02  ; r17 := 0x02
 //0x007B  LDI R26,0x00  ;
 //0x007C  LDI R27,0x02  ; r27:r26(X) := 0x0200
 //0x007D  LDI R30,0x42  ;
 //0x007E  LDI R31,0x01  ; r31:r30(Z) := 0x0142
 //0x007F  LDI R16,0x00  ;
 //0x0080  OUT 0x3B,R16  ; RAMPZ := 0x00
 //0x0081  RJMP PC+0x03  ; springe zu M2
M1://0x82  ELPM R0,Z+    ; r0 := p(Z); Z := Z+1
 //0x0083  ST X+,r0      ; *(X) := r0; X := X+1
M2://0x84  CPI R26,0x0C  ;
 //0x0085  CPC R27,R17   ; ? r27:r26-0x020C
 //0x0086  BRNE PC-0x04  ; wenn ≠0, springe zu M1
```



Das mit

```
uint8_t b[10];
```

vereinbarte Feld, das der Compiler ab Adresse 0x20C platziert hat, wird mit Nullen initialisiert:

```
 //0x0087  LDI R18,0x02  ; r18 := 2
 //0x0088  LDI R26,0x0C  ; r27:r26(X) := 0x020C
 //0x0089  LDI R27,0x02  ;
 //0x008A  RJMP PC+0x02  ; spring zu M2
M1://0x8B  ST X+,R1      ; *(X) := 0; X := X+1
M2://0x8C  CPI R26,0x16  ;
 //0x008D  CPC R27,R18   ; ?: r27:r26-0x0216
 //0x008E  BRNE PC-0x03  ; wenn ≠0, springe zu M1
 //0x008F  RCALL PC+0x03 ; Aufruf von main()
 //0x0090  RJMP PC+0x09  ; Sprung hinter main()
 //0x0091  RJMP PC-0x91  ; Neustart
int main(void){
 //0x0092  LDS R24,0x0200;
```

2 Multiplikation

Multiplikation vorzeichenfreier Binärzahlen

$$
\begin{array}{r}
(a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0) \cdot (b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0) = \\
\hline
a_3 b_0 \cdot 2^3 + a_2 b_0 \cdot 2^2 + a_1 b_0 \cdot 2^1 + a_0 b_0 \cdot 2^0 \\
a_3 b_1 \cdot 2^4 + a_2 b_1 \cdot 2^3 + a_1 b_1 \cdot 2^2 + a_0 b_1 \cdot 2^1 \qquad\qquad \\
a_3 b_2 \cdot 2^5 + a_2 b_2 \cdot 2^4 + a_1 b_2 \cdot 2^3 + a_0 b_2 \cdot 2^2 \qquad\qquad\qquad\qquad \\
a_3 b_3 \cdot 2^6 + a_2 b_3 \cdot 2^5 + a_1 b_3 \cdot 2^4 + a_0 b_3 \cdot 2^3 \qquad\qquad\qquad\qquad\qquad\qquad \\
\hline
p_7 \quad p_6 \qquad\quad p_5 \qquad\quad p_4 \qquad\quad p_3 \qquad\quad p_2 \qquad\quad p_1 \qquad\quad p_0
\end{array}
$$

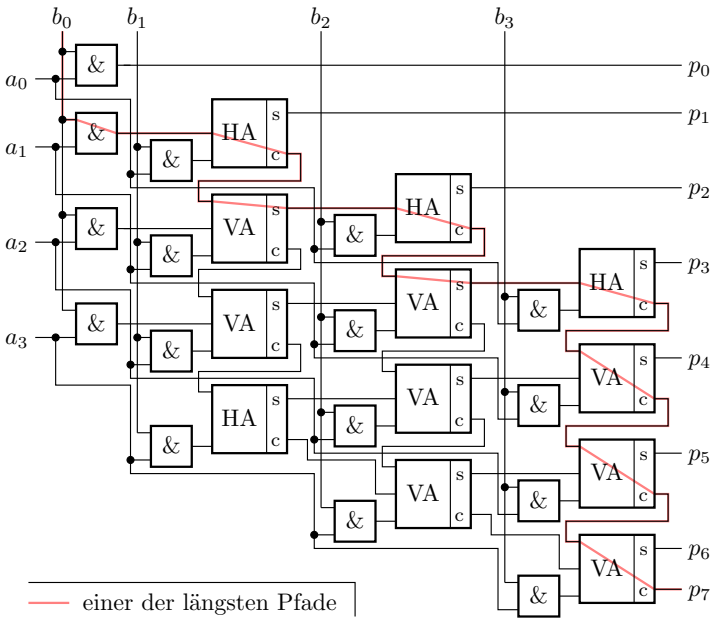
Eine $n \times n$-Bit-Multiplikation ist nachbildbar aus:

- $n \times n$ 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

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

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[1].

ATmega-Multiplikationsbefehl für vorzeichenfreie Zahlen:

Operation	Op.-Code	Assembler
R1:R0 := 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:

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

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

[1] Bei der Division wächst auch die Verzögerungszeit mit dem Quadrat der Bitanzahl. Deshalb sind HW-Dividierer unüblich.

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 (AL)
// 0x0098  LDS R21,0x0201 ; r21 := a.Byte1 (AH)
// 0x009A  LDS R18,0x0202 ; r18 := b.Byte0 (BL)
// 0x009C  LDS R19,0x0203 ; r19 := b.Byte1 (BH)
// <r27:r24 = r21:r20 * r19:r18>
// 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

```
// 0x009E  MUL R20,R18   ; r1:r0 = AL*BL
// 0x009F  MOVW R24,R0   ; r24 = L(AL*BL) (p.Byte0)
//                       ; r25 = H(AL*BL)
// 0x00A0  MUL R20,R19   ; r1:r0 = AL*BH
// 0x00A1  ADD R25,R0    ; r25 = H(AL*BL)+L(AL*BH)
// 0x00A2  MUL R21,R18   ; r1:r0 = AH*BL
// 0x00A3  ADD R25,R0    ; r25 = H(AL*BL)+L(AL*BH)
//                       ;     + L(AH*BL) (p.Byte1)
// 0x00A4  CLR R1        ; r1 = 0
// 0x00A5  MOVW R24,R24  ; kein erkannbarer Sinn
// 0x00A6  LDI R26,0x00  ; P.Byte3 = 0
// 0x00A7  LDI R27,0x00  ; P.Byte4 = 0
```

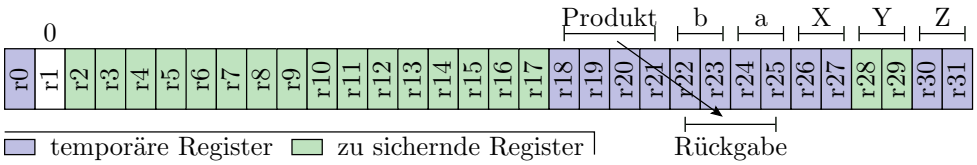
p.Byte0 und p.Byte1 werden richtig berechnet und p.Byte2 und p.Byte3 werden auf null gesetzt. Bytes auf 0.

Bug oder Feature? / Fehler oder spezifizierte Sollfunktion?

UP für 4 Byte-Produkt (wenige Befehle mehr)

```
uint32_t mult_u16_u32(uint16_t a, uint16_t b);

mult_u16_u32: ; Marke für Funktionsaufruf
  mul r24,r22 ; r1:r0   = AL*BL
  movw r18,r0 ; r19:r18 = AL*BL
  mul r25,r23 ; r1:r0   = AH*BH
  movw r20,r0 ; r21:r20 = AH*BH
  mul r24,r23 ; r1:r0   = AL*BH
  add r19,r0  ; r19     = H(AL*BL)+L(AL*BH)
  adc r20,r1  ; r20     = L(AH*BH)+H(AL*BH)+c
  clr r1      ; r1      = 0
  adc r21,r1  ; r21     = H(AH*BH)+c
  mul r25,r22 ; r1:r0   = AH*BL
  add r19,r0  ; r19 = H(AL*BL)+L(AL*BH)+L(AH*BL)
  ...         ;
  ret         ; Rücksprung
```



- Von rechts beginnen werden die ersten 18 Aufrufparameterbytes in den Registern r25:r8 übergeben.
- Die Rückgabe erfolgt in den Registern r25:r8.

```
mult_u16_u32:
  ...
  adc r20,r1    ; r20 = L(AH*BH)+H(AL*BH)+
                ;     + H(AH*BL)+c
  clr r1        ; r1  = 0
  adc r21,r1    ; r21 = H(AH*BH) + Überträge
  movw r22,r18  ; p (R25..r22) = (r21..r18)
  movw r24,r20  ;
  ret
```

Multiplikation vorzeichenbehafteter Zahlen

Im Gegensatz zur Addition und Subtraktion geänderter Algorithmus:

$$\begin{aligned}(A - a_{n-1}\cdot 2^n)\cdot(B - b_{n-1}\cdot 2^n) &= A\cdot B + \left(a_{n-1}\cdot b_{n-1}\cdot 2^{2n}\right)^* \\ &- (A\cdot b_{n-1} + B\cdot a_{n-1})\cdot 2^n\end{aligned}$$

(*– mit $2n$-Bits nicht darstellbar). Zusätzliche bedingte Subtraktion der um n Bit linksverschobenen Faktoren vom »vorzeichenfreien« Produkt, wenn das jeweils andere Vorzeichenbit eins ist.

AVR-Befehle zur Multiplikationen vorzeichenbehafteter Zahlen[2]:

Operation	Op.-Code	Assembler
R1:R0 := $\text{Rd}^{(s)} \cdot \text{Rr}^{(s)}$	1000 0010 dddd rrrr$^{(1)}$	`muls Rd, Rr`
R1:R0 := $\text{Rd}^{(s)} \cdot \text{Rr}^{(u)}$	1000 0011 0ddd 0rrr$^{(2)}$	`mulsu Rd, Rr`

$^{(u)}$ `vorzeichenfrei (unsigned);` $^{(s)}$ `vorzeichenbehaftet (signed);` $^{(1)}$ Rd, Rr nur R16 bis R31. $^{(2)}$ Rd, Rr nur R16 bis R23.

3 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
```

[2] Die nutzt der Compiler aber nicht unbedingt, siehe Aufgabe 4.2.

```
// 0x0096  LDS R22,0x0200 ; Aufrufparameter b
// 0x0098  LDS R23,0x0201
// 0x009A  RCALL PC+0x0008; Aufruf der Division
// 0x009B  STS 0x0205,R23 ; q ist Rückgabewert 2
// 0x009D  STS 0x0204,R22 ; was steht in r25:r24?
// 0x009F  LDI R24,0x00   ;
// 0x00A0  LDI R25,0x00   ; main() gibt null zurück
// 0x00A1  RET            ; Rücksprung von main
// 0x00A2  ...            ; Divisions-UP
```

Divisions-Unterprogramm

```
// 0x00A2  SUB R26,R26   ; r27:r26 := 0 (Rest)
// 0x00A3  SUB R27,R27   ;
// 0x00A4  LDI R21,0x11  ; r21:=0x11 (Schleifenz.)
// 0x00A5  RJMP PC+0x0008; springe zu M1
M2://0xA6  ROL R26       ; r27:r26 := 2·(r27:r26)+c
// 0x00A7  ROL R27       ;
// 0x00A8  CP R26,R22    ; ?: r27:r26-r23:r22
// 0x00A9  CPC R27,R23   ; wenn Differenz negativ,
// 0x00AA  BRCS PC+0x03  ; dann gehe zu M1
// 0x00AB  SUB R26,R22   ; r27:r26 -= r23:r22
// 0x00AC  SBC R27,R23   ;
M1://0xAD  ROL R24       ; r25:r24 := 2·(r25:r24)+c
// 0x00AE  ROL R25       ; (c ist neg. Ergebnisbit)
// 0x00AF  DEC R21       ; r21-- (Schleifenzähler)
// 0x00B0  BRNE PC-0x0A  ; wenn ≠0,springe zu M2
```

16 Schleifendurchläufe zu je etwa 10 Takten (ca. 20 µs).

```
//  0x00B1  COM R24       ; r25:r24 := not(r25:r24)
//  0x00B2  COM R25       ; (q := not(q))
//  0x00B3  MOVW R22,R24  ; r23:r22 := Quotient
//  0x00B4  MOVW R24,R26  ; r25:r24 := Rest
//  0x00B5  RET           ; Rücksprung
```

Das Unterprogramm berechnet für

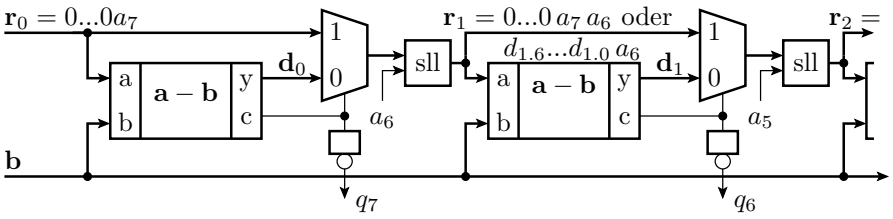
$$\frac{a}{b} = q + \frac{r}{b}$$

den Quotienten $q = a/b$ und den Divisionsrest $r = a\%b$. Algorithmus am Beispiel $a = 11$ und $b = 3$:

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

Extrahierter Berechnungsfluss

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



Laufzeitbetrachtung:

- Jede Subtraktion für q_{i-1} muss warten, bis q_i berechnet ist.
- Rechnenwerk mit einem statt n Sub/Mux ähnlich schnell.
- 1-Schritt-Dividierer im Gegensatz zu Multiplizierern unüblich.

4 Gleitkommazahlen

Festkomma- und Gleitkommazahlen

Zahlenwerte mit Nachkommastellen lassen sich auf dem Rechner als Festkomma- oder Gleitkommazahlen darstellen.

Festkommazahlen haben eine gedachte Skalierungsfaktor und werden mit den arithmetischen Befehlen (Add, Sub, ...) für ganze Zahlen bearbeitet. Probleme:

- Wahl der Anzahl der Vorkommastellen so, dass der Zahlenbereich nicht über- oder unterläuft und
- der Anzahl der Nachkommastellen so, das die Rundungsfehler vernachlässigbar bleiben.

Bei Gleitkommdarstellung wird der Wert durch Multiplikation mit einer Zweierpotenz so verschoben, dass eine 1 vor dem Komma steht, und die Verschiebung als »Charakteristik« in der Zahlendarstellung gespeichert.

$$v = \frac{s}{t}$$

s	t	v
1 mm	1 µs	1000 $\frac{\text{m}}{\text{s}}$
137 km	10 ms	$1{,}37 \cdot 10^{7}$ $\frac{\text{m}}{\text{s}}$
3,36 mm	1000 s	$1{,}36 \cdot 10^{-6}$ $\frac{\text{m}}{\text{s}}$

- Festkommadarstellung:

s in 10 µm	t in µs	v in $\frac{\mu\text{m}}{\text{s}}$
100	1	10^9 (>30 Bit)
$1{,}37{\cdot}10^{10}$ (>31 Bit)	10 000	$1{,}37{\cdot}10^{17}$ (>40 Bit)
336	10^9 (>30 Bit)	1 mit 36% Rundungsfehler

- Gleitkommadarstellung mit $1 \leq m < 2$ (m – normierter Wert):

s in m	t in s	v in $\frac{\text{m}}{\text{s}}$
$1{,}02..{\cdot}2^{-10}$	$1{,}04..{\cdot}2^{-20}$	$1{,}95..{\cdot}2^{9}$
$1{,}04..{\cdot}2^{17}$	$1{,}28..{\cdot}2^{-7}$	$1{,}33..{\cdot}2^{24}$
$1{,}72..{\cdot}2^{-10}$	$1{,}95..{\cdot}2^{9}$	$1{,}76..{\cdot}2^{-19}$

Darstellung von Gleitkommazahlen

Darstellung durch Vorzeichenbit s, Charakteristik c, Mantisse M:

- Normierte Darstellung ($0 < c < c_{\max}$):

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

  (c_0 – c-Wert für Kommaverschiebung null).

- Denormierte Darstellung für $c = 0$, Betrag $|Z| < 2^{-c_0}$:

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

  Echte Null: $c = 0$; $M = 0$

- 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; $\pm\infty$ – WB-Überlauf / -Unterlauf)

Umrechnung in die normierte Gleitkommadarstellung:

- Vorzeichenbit s und Betrag $|Z|$ bilden.
- Charakteristik c so festlegen, dass gilt:

  $$1 \le |Z| \cdot 2^{c_0-c} < 2$$

- Nachkommastellen von $|Z| \cdot 2^{c_0-c}$ als Mantisse M übernehmen.

32-Bit-Format »IEEE-754 single«:

Bitvektor (Bitnummer 31 … 0)			Wert
s (31)	c (30 … 23)	M (22 … 0)	
0 / +	1000001 1 (8 3); $c = 83_{16}$	0010010 00000000 00000000 0 (2 4 0 0 0 0); $M = 1{,}240000_{16}$	$+1.240000_{16} \cdot 2^{83_{16}-7f_{16}}$ $= 18{,}25$
1 / −	0111100 1 (7 9); $c = 79_{16}$	1100101 10011101 00001110 0 (C B 3 A 1 C); $M = 1{,}CB3A1C_{16}$	$-1.CB3A1C_{16} \cdot 2^{79_{16}-7f_{16}}$ $\approx -2{,}8029 \cdot 10^{-2}$
0	0000000 0; 0/denorm.	0001100 10111101 00011001 00 (, 3 2 F 4 6 4); $M = 0{,}32F464_{16}$	$+0{,}32F464_{16} \cdot 2^{0-7f_{16}}$ $\approx 1{,}170 \cdot 10^{-39}$

Nutzung von Gleitkommazahlen in C

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

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 RA-F6.pdf).
- 32-Bit-Prozessoren haben oft Gleitkommarechenwerke, die Gleitkommaoperationen in wenigen Schritten ausführen.

5 Rekursion

Rekursion

Ein rekursives Programm ruft sich so lange selbst auf, bis eine Abbruchbedingung erreicht ist. Beispiel für einen rekursiv beschreibaren 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: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 ja,
000082  BREQ PC+0x09    | Sprung zu M (Rückgabe r16:r19 unverändert)
```

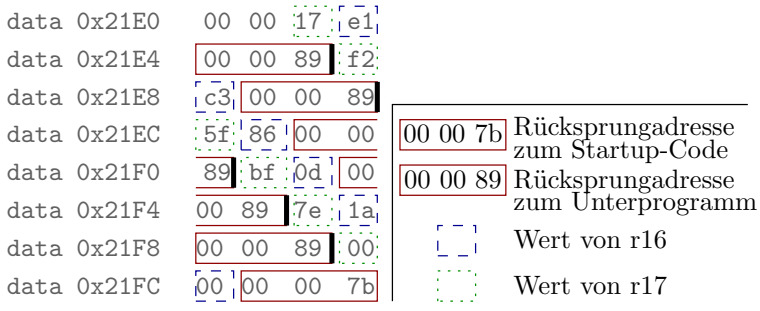
```
00083  LSR R25          |
00084  ROR R24          | Rechstverschiebung der 4
00085  ROR R23          | Bytes der Variablen a.
00086  ROR R22          |
00087  SUBI R20,0x01    | b ← b-1
00088  RCALL PC-0x000B  | Erneuter Aufruf von sich selbst.
00089  MOVW R16,R22     | Rückgabewert in r22 bis r25 in
0008A  MOVW R18,R24     | r16 bis r19 kopieren.
M:008B MOV R22,R16      |
0008C  MOV R23,R17      | Rückgabe des verschobenen Wertes
0008D  MOV R24,R18      | aus r16:r19 in r22:r25
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.



6 Aufgaben

Aufgabe 4.1: Multiplizierunterprogramm

Die nächste Folie zeigt ein Unterprogramm, das aus 16-Bit Faktoren ein 32-Bit-Produkt bildet, ein Hauptprogramm, das dieses mit Beispielzahlen aufruft und das mit -O1 übersetzte disassemblierte Programm. Das Programm verhält sich nicht wie erwartet.

1. In welchen Registern werden die Faktoren übergeben und in welchen Registern erwartet das Hauptprogramm das Ergebnis?
2. Bestimmen Sie auf der nächsten Folie die Werte, die nach jeder Anweisung in den Registern stehen und füllen Sie die Tabelle aus.
3. Was ist an der Berechnung falsch?

```
10  uint32_t mult(uint16_t a,
11                uint16_t b){
12    return a*b;
    }
15  void main(){
16    uint32_t p=mult(0x3412, 0xF123);
    }
```

Hauptprogramm:

```
0008A  LDI R22,0x23
0008B  LDI R23,0xF1
0008C  LDI R24,0x12
0008D  LDI R25,0x34
0008E  RCALL PC-0x001
0008F  RET
```

Unterprogramm:

```
0007D  MOVW R18,R24
0007E  MUL R22,R18
0007F  MOVW R24,R0
00080  MUL R22,R19
00081  ADD R25,R0
00082  MUL R23,R18
00083  ADD R25,R0
00084  CLR R1
00085  MOV R22,R24
00086  MOV R23,R25
00087  LDI R24,0x00
00088  LDI R25,0x00
00089  RET
```

C	r25	r24	r23	r22	r19	r18	r1	r0

Lösung

1. Übergabe a in r25:r24 und b in r23:r22. Rückgabe in r25:r22.
2. Registerwerte:

	C	r25	r24	r23	r22	r19	r18	r1	r0
0007D MOVW R18,R24		0x34	0x12	0xF1	0x23	0x34	0x12		
0007E MUL R22,R18								0x02	0x76
0007F MOVW R24,R0		0x02	0x76						
00080 MUL R22,R19								0x07	0x1C
00081 ADD R25,R0	0	0x1E							
00082 MUL R23,R18								0x10	0xF2
00083 ADD R25,R0	1	0x10							
00084 CLR R1								0x00	
00085 MOV R22,R24					0x76				
00086 MOV R23,R25				0x10					
00087 LDI R24,0x00			0x00						
00088 LDI R25,0x00		0x00							
00089 RET									

3. In r25:r24 wird 0 statt der Produktbytes zurückgegeben.

Aufgabe 4.2: Multiplikation mit Vorzeichen

```
11   int8_t b, c;
12   int16_t a;
13   void main(void){
14     a = b*c;
     }
```

Optimierung mit O1

```
00085  LDS R25,0x0200
00087  LDS R24,0x0201
00089  MULS R25,R24
0008A  MOVW R24,R0
0008B  CLR R1
0008C  STS 0x0203,R25
0008E  STS 0x0202,R24
00090  RET
```

Optimierung mit O0

```
0000008C  CLR R21
0000008D  SBRC R20,7
0000008E  COM R21
00000092  CLR R19
00000093  SBRC R18,7
00000094  COM R19
00000095  MUL R20,R18
00000096  MOVW R24,R0
00000097  MUL R20,R19
00000098  ADD R25,R0
00000099  MUL R21,R18
0000009A  ADD R25,R0
```

Untersuchung des generierten Codes.

Lösung für -O0

```
                                r20 := b; r18 := c
0000008C  CLR R21       }       vorzeichenbehaftete
0000008D  SBRC R20,7    }       Erweiterung von b
0000008E  COM R21       }       auf 16 Bit
00000092  CLR R19       }       vorzeichenbehaftete
00000093  SBRC R18,7    }       Erweiterung von c
00000094  COM R19       }       auf 16 Bit
00000095  MUL R20,R18     ; r1:r0 := b0 · c0
00000096  MOVW R24,R0     ; r25:r24 := b0 · c0
00000097  MUL R20,R19     ; r1:r0 := b0 · c1
00000098  ADD R25,R0      ; r25 := r25 +L(b0 · c1)
00000099  MUL R21,R18     ; r1:r0 := b1 · c0
0000009A  ADD R25,R0      ; r25 := r25 +L(b1 · c0)
```

- Vorzeichenerweiterung der Operanden auf 16 Bit.
- Die niederwertigen Produktbytes sind bei vorzeichenbehafteter und vorzeichenfreier Multiplikation gleich.

Lösung für -O1

```
00085  LDS R25,0x0200    lade b
00087  LDS R24,0x0201    lade c
00089  MULS R25,R24      vorzeichenbehaftete Multiplikation
0008A  MOVW R24,R0      ; r25:r24 := r1:r0
0008B  CLR R1
0008C  STS 0x0203,R25   } Ergebnis speichern
0008E  STS 0x0202,R24   }
00090  RET
```

Während eine 16×16 Bit-Unsigned-Multiplikation die führenden 16 Produktbits auf null setzt (siehe Folie/Seite 5), berechnet die 8×8 Bit-Signed-Multiplikation ein 2 Byte-Produkt.

Feature oder Bug?

Aufgabe 4.3: Division

```
11  uint8_t a, b, r, q;
12 int main(void){
13    r = a%b;  // Berechnung des Divisionsrests
14    q = a/b;  // Berechnung des Quotienten
15 }
```

Quotient und Divisionsrest werden in C mit unterschiedlichen Operatoren, d.h. in getrennten Anweisungen beschrieben. Unser Compiler berechnet beide Werte mit demselben Unterprogramm:

- Rückgabe Rest in r25:r24 und
- Rückgabe Quotient in r23:r22.

Ruft das übersetzte Programm das Divisionsunterprogramm ein- oder zweimal auf?

Lösung

```
r = a%b;  // Berechnung des Divisionsrests
0089  LDS R24,0x0202    ; r24 := a
008B  LDS R25,0x0200    ; r25 := b (ohne Sinn)
008D  MOV R22,R25       ; r22 := b
008E  RCALL PC+0x0011   ; Unterprogrammaufruf
008F  MOV R24,R25       ; Divisionsrest in r speichern
0090  STS 0x0201,R24
q = a/b;  // Berechnung des Quotienten
0092  LDS R24,0x0202    ; r24 := a
0094  LDS R25,0x0200    ; r25 := b (ohne Sinn)
0096  MOV R22,R25       ; r22 := b
0097  RCALL PC+0x0008   ; Unterprogrammaufruf
0098  STS 0x0203,R24    ; Quotient in q speichern
009F  SUB R25,R25       ; Beginn Unterprogramm Division
          ...
```

- Das Unterprogramm wird zweimal aufgerufen.

Aufgabe 4.4: Festkommazahlen

Eine Festkommazahl mit n-Vorkomma- und m-Nachkommabits hat den Wert:

$$Z = \sum_{i=-m}^{n-2} b_i \cdot 2^i - b_{n-1} \cdot 2^{n-1} \tag{1}$$

Bestimmen Sie für folgende Werte den nächsten mit »int16_t«, 12 Vorkomma- und 4 gedachten Nachkommabits darstellbaren Wert:

1. $w_1 = 164{,}4297$
2. $w_2 = -418{,}295$

Lösung

Bei vier Nachkommabits und »vorzeichhenbehaftet« wird im Rechner der 2^4-fache gerundete Wert im Zweierkomplement dargestellt:

w_i	$w_i \cdot 2^4$	gerundet	hexadezimal	Zweierkomplement
164,4297	2630,9	2631	0x0a47	0x0a47
$-418,295$	-6692,7	-6693	-0x1a25	0xe5db

Aufgabe 4.5: Gleitkommazahlen

Nach Abarbeitung der Anweisung im nachfolgenden C-Programm haben die Variablen die im Debug-Fenster angezeigten Werte:

```
float a;
float b=14.74;
float c=-379.12;
int main(){
 a = b + c;
}
```

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

- Wie bestimmt man im Debugger die Hex-Darstellung dieser Werte und wie lautet diese?
- Wie lauten die Vorzeichenbits s, die Exponenten $c - c_0$ (dezimal), die Mantissen $M^{(6)}$ abgeschnitten nach 6 Nachkommabits und die sich daraus berechnenden Werte:

$$Z^{(6)} = (-1)^2 \cdot 2^{c-c_0} \cdot M^{(6)}$$

Lösung

- Die Hex-Darstellung findet man im Datenspeicher unter den angegebenen Adressen:

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

- Bestandteile und auf 6 Mantissenachkommastellen gerundete Werte:

	Adr.	Hex-Wert	s	$c - c_0$	$M^{(6)}$	$Z^{(6)}$
b	0x200	0x416bd70a	0	3	0b1,110101	14,63
c	0x204	0xc3bd8f5c	1	8	0b1,011110	-376
a	0x208	0xc5aea1d4	1	12	0b1,010111	-5568