



Rechnerarchitektur, Foliensatz 2

Zeitabläufe

G. Kemnitz

Institut für Informatik, Technische Universität Clausthal
22. Januar 2015



Inhalt des Foliensatzes

Pipeline-Verarbeitung

- 1.1 Befehlsbearbeitung
 - 1.2 Pipeline-Auslastung
 - 1.3 Lade/Speicher- Pipeline
 - 1.4 Sprung- Pipeline
 - 1.5 Der Beispielprozessor
 - 1.6 Aufgaben
- ### Speicher
- 2.1 Adressräume
 - 2.2 Lade- und Speicherbefehle
 - 2.3 Konstanteninitialisierung

2.4 Aufgaben

Kontrollfluss

- 3.1 Sprungbefehle
- 3.2 Fallunterscheidungen
- 3.3 Schleifen
- 3.4 Aufgaben

Unterprogramme

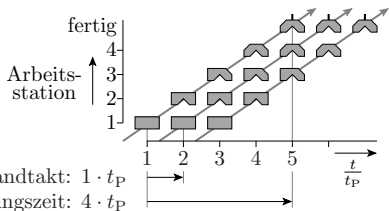
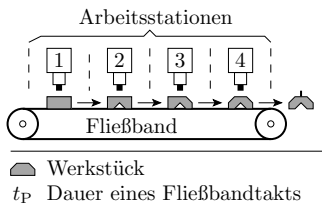
- 4.1 Lokale Variablen
- 4.2 Parameterübergabe
- 4.3 Rekursion
- 4.4 Aufgaben



Pipeline-Verarbeitung

Pipeline-Verarbeitung (dt. Fließband-Verarbeitung)

Pipeline-Verarbeitung ist die effektivste Form der Parallelverarbeitung, auch für Rechner:

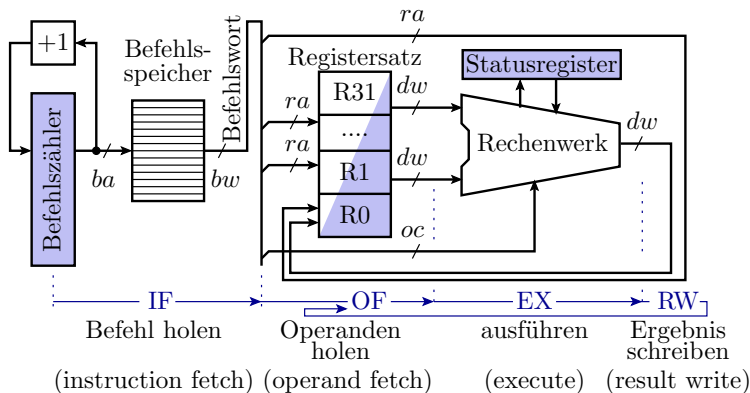


- Aufteilung der Arbeit in Schritte, die an einem Fließband (engl. Pipeline) abgearbeitet werden.
- Die Bearbeitungszeit je Werkstück ist N Schritte und es wird gleichzeitig an N Werkstücken gearbeitet.
- In jedem Schritt beginnt die Arbeit an einem Werkstück und ein Werkstück wird fertig.



Befehlsabarbeitung

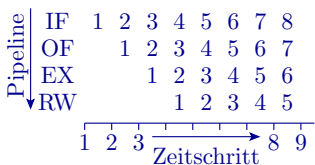
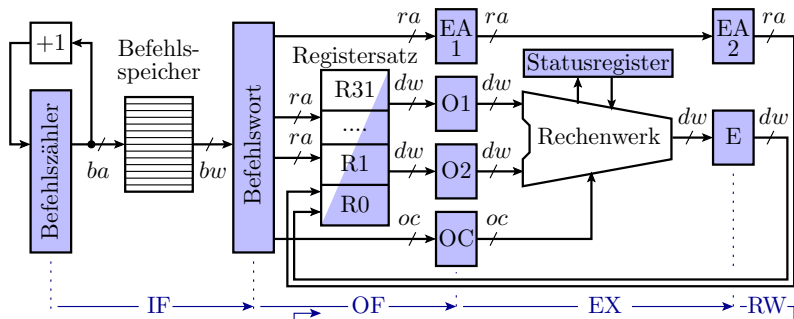
Kern eines RISC-Prozessors ohne Pipeline



- Quellregister: Befehlszähler, Status- und Operandenregister.
- Zielregister: Befehlszähler, Status- und Ergebnisregister.
- Verarbeitungsschritte: Befehl holen, Operanden holen, ...



Aufteilung in Pipeline-Phasen



weitergereichte Daten

ba Befehlsadresse

bw Befehlswort

ra Registeradresse

dw Datenwort

oc Operationscode

Register für

EA Ergebnisadresse

O1 Operand 1

O2 Operand 2

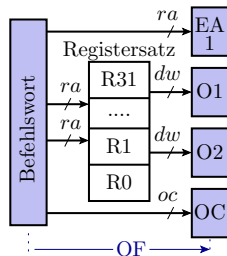
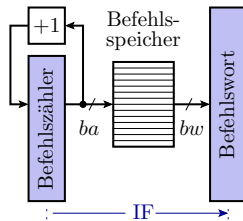
OC Operationscode

E Ergebnis



Die Aufteilung der Verarbeitungsschritte in Pipeline-Phasen erfolgt durch Einbau getakteter Register für die Zwischenergebnisse. Pipeline-Phasen im Beispielmodell:

- IF (Instruction Fetch) Befehl holen. Adressierung des Befehlsspeichers vom Befehlszähler und Übernahme des Befehlswortes in das Befehlswortregister.
- OF (Operand Fetch) Operanden holen: Adressierung des Registersatzes mit den Operandenadressen und Übernahme der Registerinhalte in die Operandenregister. Weitergabe der Ergebnisadresse und des Operationscodes an die nächste Pipeline-Phase.

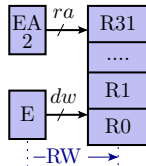
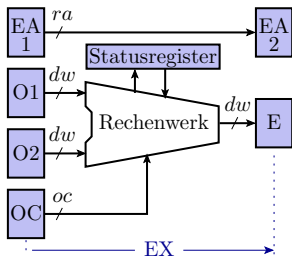




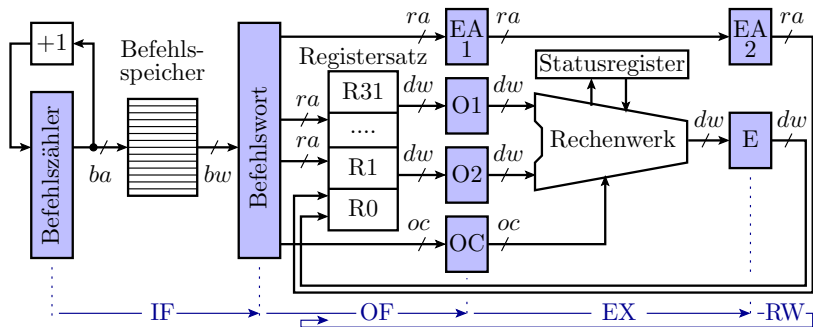
1. Pipeline-Verarbeitung

- EX (Execute): Befehle Ausführen:
Aus den Operanden und dem Operationscode bildet das Rechenwerk das Ergebnis und speichert es. Weitergabe der Ergebnisadresse an die nächste Pipeline-Phase.
- RW (Result Write) Ergebnis schreiben:
Adressierung des Registersatzes mit der Ergebnisadresse und Speichern des Ergebnisses.

1. Befehlsabarbeitung



Die Befehlsausführung dauert vier Schritte (Taktperioden). Es wird gleichzeitig an vier Befehlen gearbeitet. Der maximal erzielbare Verarbeitungsdurchsatz ist ein Befehl pro Schritt.



- Bei einer Aufteilung des Verarbeitungsflusses in mehrere gleichlange Pipeline-Phasen kann der Rechner wesentlich schneller getaktet werden und trotzdem in jedem Takt eine neue Operation beginnen und eine fertigstellen.
- Die Fertigstellung der einzelnen Befehle dauert mindest genauso lange wie ohne Pipeline.



Pipeline-Auslastung



Pipeline-Auslastung

Im Modell kann ein Nachfolgebefehl nur Ergebnisse von mind. 2 Takten zuvor gestarteten Befehlen weiterverarbeiten. Beispiel:

Addition von vier Registerinhalten

$$1: r_0 \leftarrow r_0 + r_1$$

$$2: r_0 \leftarrow r_0 + r_2$$

$$3: r_0 \leftarrow r_0 + r_3$$

* Register lesen

* n lesen und mit n überschreiben

	IF	OF				RW	EX		
	PC	BR	r_0	r_1	r_2	r_3	O1	O2	E
1			7	2	11	17			
2		B1							
3		B2	*	*			7	2	
		B3	*		*		7	11	9
			*9			*	7	17	18
			18						24
			24						

In r_0 steht in Takt 5 $r_0 + r_1$, in Takt 6 $r_0 + r_2$ und in Takt 7 $r_0 + r_3$. Zur Berechnung von $r_0 + r_1 + r_2 + r_3$ muss die OF-Phase des Nachfolgebefehl auf den Abschluss der RW-Phase warten.

Einfügen von nop^1 -Befehlen

Addition von vier
Registerinhalten

1: $r_0 \leftarrow r_0 + r_1$

2: nop

3: nop

4: $r_0 \leftarrow r_0 + r_2$

5: nop

6: nop

7: $r_0 \leftarrow r_0 + r_3$

* Register lesen
* n lesen und mit n
überschreiben

	IF	OF				RW	EX	
PC	BR	r_0	r_1	r_2	r_3	O1	O2	E
1		7	2	11	17			
2	B1							
3	nop							
4	nop							
5	B4							
6	nop							
7	nop							
	B7							

■ Wie lange dauert jetzt die Ausführung der drei Befehle?

¹ nop – No Operation, für den Beispielprozessor Op-Code 0x0000.



Programmoptimierung

- 1: $r_0 \leftarrow r_0 + r_1$
- 2: $r_2 \leftarrow r_2 + r_3$
- 3: nop
- 4: nop
- 5: $r_0 \leftarrow r_0 + r_2$

* Register lesen
 * n lesen und mit n
 überschreiben

		OF				RW		EX
IF		r_0	r_1	r_2	r_3	O1	O2	E
PC	BR	r_0	r_1	r_2	r_3	O1	O2	E
1		7	2	11	17			
2	B1							
3	B2	*	*			7	2	
4	nop			*	*	11	17	9
5	nop	9						28
	B5			28				
		*		*		9	28	
								37
		37						

Die Additionen $r_0 + r_1$ und $r_2 + r_3$ können direkt nacheinander erfolgen. Berechnung braucht zwei Takte weniger.

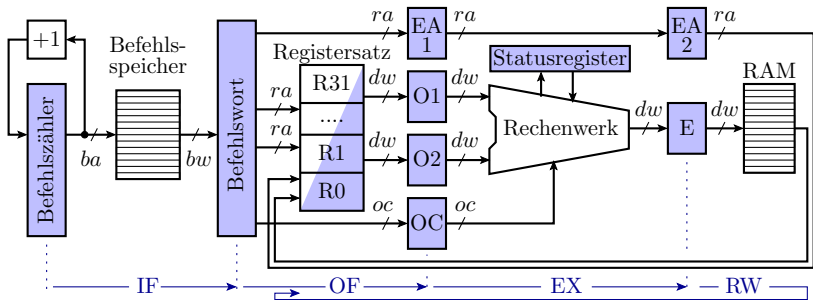
Fakt 1

Ein Prozessor ist nur so gut wie sein Compiler.



Lade/Speicher- Pipeline

Ladeoperation



- Erweiterung der RW-Phase um den Datenlesezugriff.
- Die aus dem Speicher gelesenen Daten sind bei dieser Pipeline-Struktur erst drei Befehle später nutzbar².
- Nicht optimierte Befehlsfolge für »Variable +1« ist »Lesen«, »2×nop«, Addieren, »2×nop« und »Schreiben« (10 Befehle).

²Es gibt HW-Lösungen, die dieses Pipeline-Problem umgehen.



Befehlsfolge für einen einfachen Increment:

```
uint8_t a; // die Adresse r28+4
...
a++;      // alter Wert von a sei 0x34
```

- 1: $r_0 \leftarrow \text{mem}(r_{28} + 4)$
- 2: nop
- 3: nop
- 4: $r_0 \leftarrow r_0 + 1$
- 5: nop
- 6: nop
- 7: $\text{mem}(r_{28} + 4) \leftarrow r_0$

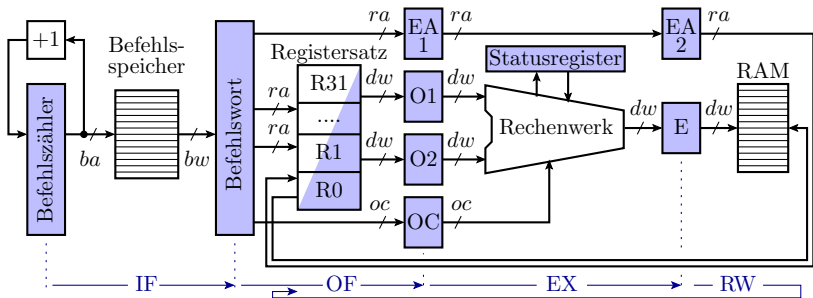
PC	BR	r_0	r_1	r_{28}	r_{29}	O1	O2	E
1		7	2	11	17			
2	B1							
3	nop							
4	nop							
5	B4							
6	nop							
7	nop							
	B7							

* Register lesen
 * n lesen und mit n überschreiben
 PC Befehlszähler
 BR Befehlsregister

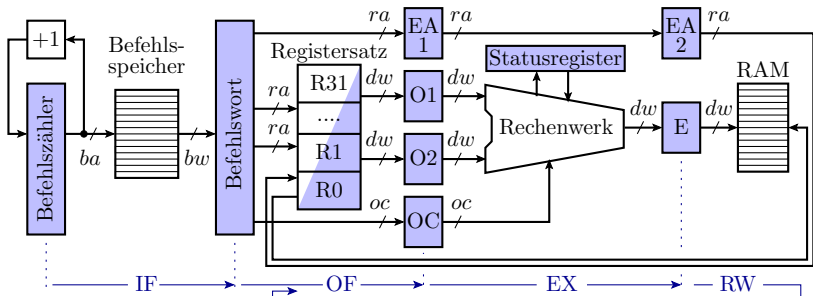




Speicheroperation



- Im Unterschied zur Ladepoperation ist in der Ex-Phase die Übertragungsrichtung vom Registersatz zum RAM.



- Wie viele nop-Befehle müssen zwischen einem Speicherbefehl

$$\text{mem}(r_{28}+4) \leftarrow r_0$$

und einem Lesebefehl

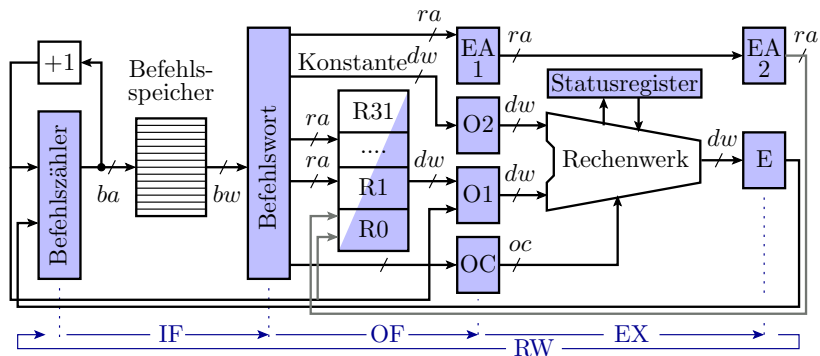
$$r_4 \leftarrow \text{mem}(r_{28}+4)$$

für dieselbe RAM-Adresse eingefügt werden, damit der geschriebene Wert gelesen wird?



Sprung- Pipeline

Sprung-Pipeline



Bei Ausführung eines Sprungs wird der Befehlszähler in der IF-, der OF- und der EX-Phase incrementiert. Erst nach der RW-Phase steht das Sprungziel im Befehlszähler³.

³Die Zeitscheiben nach einem Sprungbefehl, die noch linear abgearbeitet werden, heißen »Delay-Slots«.



Abarbeitung eines Sprungs in der Pipeline:

0x20: $PC \leftarrow PC + 0x10$ (B1)
 0x21: $r_0 \leftarrow r_0 + 3$ (B2)
 0x22: $r_1 \leftarrow r_1 \wedge 3$ (B3)
 0x23: $r_0 \leftarrow \text{mem}(r_{28} + 4)$ (B4)
 ...
 0x30: $r_0 \leftarrow r_0 + r_1$ (B5)

PC	BR	r_0	r_1	r_{28}	r_{29}	O1	O2	E
0x20		7	2	11	17			
0x21	B1							
0x22	B2							
0x23	B3							
0x30	B4							
				*				
						11	4	
								15
		34						

* Register lesen
 * n lesen und mit n
 überschreiben
 PC Befehlszähler
 BR Befehlsregister



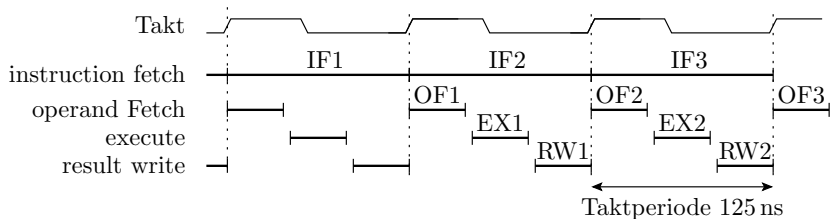
Der typische RISC-Prozessor hat nur null bis max. zwei Delay-Slots. Das wird durch überspringen von Pipeline-Phasen (Ergebnis-Forwarding) oder durch Anhalten der Pipeline erreicht.



Der Beispielprozessor

Unser Beispielprozessor ATmega 2560

- Befehl holen (IF) dauert einen Takt. Operanden holen (OF), Ausführen (EX) und Ergebnis schreiben (RW) dauern nur $\frac{1}{3}$ Takt.



- Operationsergebnisse für Folgeoperation verfügbar, d.h. lückenlose Abarbeitung ohne nop-Befehle möglich.
- Lade-, Speicher- und Sprungbefehle sowie die Multiplikation benötigen mindestens zwei Takte.



Aufgaben

Aufgabe 2.1: Addition von fünf Registerwerten

Die Werte der Register r_0 bis r_4 sollen in der Verarbeitungs-Pipeline auf Folie 7 addiert und die Ergebnisse in Register r_2 gespeichert werden. Entwickeln Sie dafür ein möglichst kurzes Programm und füllen Sie in Anlehnung an Folie 14 die nachfolgende Tabelle aus.

Bef.-Adr.	Operation	IF		OF					RW	EX	
		PC	BR	r_0	r_1	r_2	r_3	r_4	O1	O2	E
1		1		38	42	13	29	4			
2		2	B1								
3		:	B2								
4			:								
:											

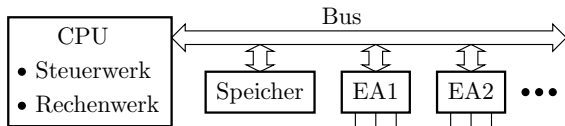
Ab dem wie vielten Befehl ist die Summe $38 + 42 + 13 + 29 + 4$ in Register r_0 verfügbar?



Speicher



Speicher



Ein Universalrechner besteht aus

- Prozessor (CPU – Central Processing Unit)
- Speicher(n) und
- EA- (Ein-/Ausgabe-, IO- (Input-/Output-) Registern.

Im Speicher hat jeder Befehl und jedes Datenobjekt eine Adresse. Die EA-Register sind wie ein Speicher organisiert. Befehle, Daten und EA-Register können sich einen Adressraum teilen, getrennte Adressräume besitzen oder in mehrere Adressräume auf unterschiedlichen Adressen eingebündelt sein. Bei mehreren Adressräumen erfolgt die Adressraumauswahl (Registeradresse, EA-Adresse, Befehlsadresse, ...) über den Befehl.



Adressräume



Adressaufteilung

Die Größe der Adressräume richten sich nach der Breite der Register und der Größe der Adresskonstanten in den Befehlsworten.

- 8-Bit Mikrorechner haben typ 16-Bit-Adressregister und 64-kByte Adressräume, u.U. auch mehrere über Zusatzregister oder die Befehlsart auswählbare.
- 32-Bit Prozessoren haben typ. einen gemeinsamen 4 GByte Adressraum für Befehle, Daten und EA-Register.

Beispielprozessor Atmega 2560:

- 17-Bit-Befehlszähler, 128 k×2 Byte großer Befehlsspeicher.
- 6-Bit IO-Adressen, 64 über IN-/OUT-Befehle ansprechbare EA-Register.
- 16-Bit Adressregister, 16-Bit-Datenadresskonstanten, 64 kByte Datenadressraum, in dem die Arbeits- und EA-Register mit eingeblendet sind.



Daten- und EA-Adressen des ATmega 2560

- 1 Arbeitsregister für Verarbeitungsbefehle.
- 2 Mit `in Rd, k ; k - 6-Bit Adresse`
`out k, Rd ;`

les- und schreibbare EA-Register (IO-Adresse k gleich RAM-Adresse minus $0x20$).

- 3 Der Compiler vergibt die Adressen $0x200$ aufsteigend für globale Variablen und nutzt die Adressen von $0x21FF$ absteigend als Stack.

- 4 Mit einer speziellen EA-Bit-Einstellung ist auch der Bereich von $0-0x21FF$ des externen Speichers les- und beschreibbar.

	0	r0	1
EA	
	1F	r31	
0	20	PINA	2
1	21	DDRA	
2	22	PORTA	
	
3D	5D	Stack-pointer	
3E	5E		
3F	5F	Statusregister	
	60	416 weitere	
	...	Plätze für	
	1FF	EA-Register	
	200	8 kByte	3
	...	interner	
	21FF	Speicher	
	2200	externer	4
	...	Speicher	
	FFFF		



Lade- und Speicherbefehle



Adressierungsarten

Schnellzugriff auf kleine Teiladressbereiche:

- Registeradressen (5 Bit): Bis zu zwei Registeradressen je Verarbeitungsbefehl, eine je Lade-/Speicherbefehl, ...
- IO-Adressen (6 Bit): Zugriff auf eines von 64 EA-Registern.

Der Zugriff auf den kompletten 16-Bit Adressraum erfolgt mit Lade- und Speicherbefehlen mit unterschiedlichen

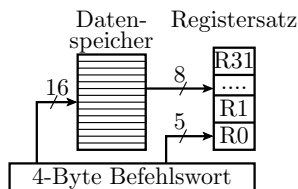
Adressierungsarten:

- direkt: Adressierung mit einer 16-Bit Konstante aus dem Befehlsword (erfordert 4-Byte Befehlsword).
- indirekt: Adressierung mit 16-Bit Register (X, Y oder Z)⁴.
- Indirekt mit Verschiebung: Adressierung mit einem 16-Bit-Register plus Konstante aus dem Befehlsword.

⁴Das sind Registerpaare: $X=(r27:r26)$, $Y=(r29:r28)$ und $Z=(r31,r30)$.

Direkte Adressierung

- Codierung der 16-Bit Speicheradresse im Befehlsword. Verlangt Doppelbefehlswords.



Operation	TZ	Op.-Code	Assembler
$Rd \leftarrow (k)$	2	1001 000d dddd 0000 kkkk kkkk kkkk kkkk	<code>lds Rd, k</code>
$(k) \leftarrow Rd$	2	1001 001d dddd 0000 kkkk kkkk kkkk kkkk	<code>sts k, Rd</code>

(TZ – Verarbeitungstaktzyklen; (k) – Datenspeicherinhalt Adresse k).

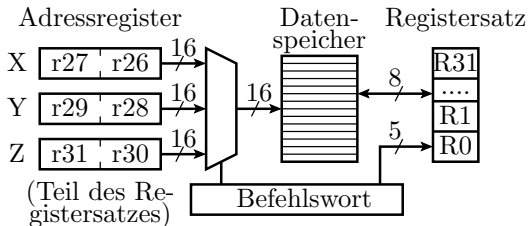
Beispiel:

```
ldi r1, 21 ; r1 ← 21
lds r2, 0x200; r2 ← (0x200)
add r2, r1 ; r2 ← r2 + r1
sts r2, 0x200
```



Indirekte Adressierung

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



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



Die indirekte Adressierung gibt es auch mit

- Post-Increment: $Rd \leftarrow (X); X \leftarrow X+1;$
- Pre-Decrement: $X \leftarrow X-1; Rd \leftarrow (X);$

(auch für Y und Z) sowie mit Verschiebung

$Rd \leftarrow (Y+q)$

(nur für Y und Z). Assemblernotationen:

ld Rd, X+ ; $Rd \leftarrow (X); X \leftarrow X+1$

ld Rd, -X ; $X \leftarrow X-1; Rd \leftarrow (X)$

ldd Rd, Y+q ; $Rd \leftarrow (Y+q)$

st X+, Rr ; $(X) \leftarrow Rr; X \leftarrow X+1$

st -X, Rr ; $X \leftarrow X-1; (X) \leftarrow Rr$

std Y+q, Rr ; $(Y+q) \leftarrow Rr$

Anwendungsbeispiel: Kopieren einer Zeichenkette:

	Zeichenkette a											Zeich. b		
Adresse - 0x200	0	1	3	4	5	6	7	8	9	A	B	C	D	...
Ascii-Zeichen	H	a	l	l	o		W	e	l	t	\0	frei		
Wert hexadezimal	48	61	6C	6C	6F	20	57	56	6C	74	00			



	Zeichenkette a											Zeich. b		
Adresse – 0x200	0	1	3	4	5	6	7	8	9	A	B	C	D	...
Ascii-Zeichen	H	a	l	l	o		W	e	l	t	\0	frei		
Wert hexadezimal	48	61	6C	6C	6F	20	57	56	6C	74	00			

Das Kopieren einer Zeichenkette lässt sich in C sehr kompakt mit zwei Zeigerregistern mit Post-Inkrement LS-Operationen beschreiben.

```

9  #include <avr/io.h>
10
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++);
18      }
19  }
```

Name	Value
☐ a	{uint8_t[11]{data}@0x0200}
[0]	0x48
[1]	0x61
[2]	0x6c
[3]	0x6c
[4]	0x6f
[5]	0x20
...	...
[9]	0x74
[10]	0x00
☐ b	{uint8_t[10]{data}@0x020c}



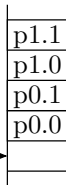
Übersetzung mit Compileroptimierung -O1

16		<code>while (*p1)</code>	
17		<code>*(p2++) = *(p1++);</code>	
<hr/>			
00092		<code>LDS R24,0x0200</code>	$r_{24} \leftarrow a[0]$
00094		<code>TST R24</code>	Flags C, Z, ... entsprechend r_{24} setzen
00095		<code>BREQ PC+0x09</code>	wenn $a[0]=0$, überspringe Schleife
00096		<code>LDI R26,0x01</code>	lade X mit 0x201
00097		<code>LDI R27,0x02</code>	(Adresse von $a[1]$)
00098		<code>LDI R30,0x0C</code>	lade Z mit 0x20C
00099		<code>LDI R31,0x02</code>	(Adresse von $b[0]$)
0009A		<code>ST Z+,R24</code>	$(Z) \leftarrow r_{24}; Z \leftarrow Z + 1$
0009B		<code>LD R24,X+</code>	$r_{24} \leftarrow (X); X \leftarrow X + 1$
0009C		<code>CPSE R24,R1</code>	überspringe Folgebefehl, wenn $r_{24} == r_1$ (r_1 is 0)
0009D		<code>RJMP PC-0x0003</code>	springe nach 0x9A



Übersetzung mit Compileroptimierung -O0

000000A1	LDD R24,Y+1	p1 laden und in Register Z kopieren
000000A2	LDD R25,Y+2	
000000A3	MOVW R30,R24	
000000A4	LDD R18,Z+0	$r18 \leftarrow *p1$
000000A5	LDD R24,Y+3	p2 laden und in Register Z kopieren
000000A6	LDD R25,Y+4	
000000A7	MOVW R30,R24	
000000A8	STD Z+0,R18	$*p1 \leftarrow r18$
000000A9	LDD R24,Y+3	p2 laden und um 1 erhöhen und zurückspeichern
000000AA	LDD R25,Y+4	
000000AB	ADIW R24,0x01	
000000AC	STD Y+4,R25	
000000AD	STD Y+3,R24	



Y: Framepointer für lokale Variablen (siehe später ab Folie 77).



000000AE	LDD R24,Y+1	p1 laden und um 1 erhöhen und zurückspeichern
000000AF	LDD R25,Y+2	
000000B0	ADIW R24,0x01	
000000B1	STD Y+2,R25	
000000B2	STD Y+1,R24	p1 laden und in Register Z kopieren r24 ← *p1 Wert testen, wenn nicht 0, Sprung zum Schleifenbeginn
000000B3	LDD R24,Y+1	
000000B4	LDD R25,Y+2	
000000B5	MOVW R30,R24	
000000B6	LDD R24,Z+0	
000000B7	TST R24	
000000B8	BRNE PC-0x17	

Mit -O0 übersetzte Programme:

- gliedern sich in Hochsprachenberechnungsschritte.
- In jedem Schritt werden die Daten aus dem Speicher geholt und die Ergebnisse in den Speicher geschrieben.
- Erlaubt Schrittbetrieb des C-Programms.



```

9  #include <avr/io.h>
10
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++);
18      }
19  }

```

Name	Value
[-] a	{uint8_t[11]{data}@0x0200}
[0]	0x48
[1]	0x61
[2]	0x6c
[3]	0x6c
[4]	0x6f
[5]	0x20
...	...
[9]	0x74
[10]	0x00
[+] b	{uint8_t[10]{data}@0x020c}

Das mit -O1 übersetzte Programm verwendet für die innere Schleife die minimale Befehlsfolge:

```

loop:          ; loop ist eine Marke der Programmadresse
  st Z+, r24
  ld r24, X+
  cpse r24, r1; Compare Skip Even
  rjmp loop   ; springe zurück zur Marke »loop«

```



Es gibt weitere Arten der Optimierung, z.B. die Vorgabe, für die Zeiger Register zu benutzen. Der Compiler nimmt dann aber nicht unbedingt die Register X, Y und Z, so dass das Programm nicht deutlich besser wird.

Selbst Probieren!

```
13 void main(void){
14     register uint8_t *p1=a;
15     register uint8_t *p2=b;
16     while (*p1){
17         *(p2++) = *(p1++);
    }
}
```

p1	0x0200	uint8_t*{registers}@ R17 R16
p2	0x0000	uint8_t*{registers}@ R15 R14



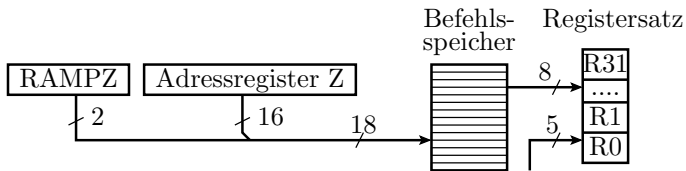
Konstanteninitialisierung



Speichern von Konstanten

Konstanten, die auch nach Neueinschalten des Prozessors noch vorhanden sein sollen, z.B. der Text »Hallo Welt« im Beispiel zuvor, müssen im Programmspeicher abgelegt und beim Programmstart in den Datenspeicher kopiert werden.

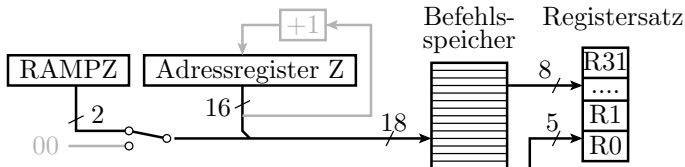
Die Adressierung des 256kByte-Befehlsspeichers erfolgt indirekt mit einer 18-Bit Adresse. Die niederwertigen 16 Adressbits werden aus Register Z und die oberste 2 Bit aus EA-Register RAMPZ (EA-Adresse 0x3B) genommen.





Befehlsvariationen:

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



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

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



Beim übersetzen des Programms 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.



Die Adressierung als Datenbytes erfolgt mit 18 Bit. Die beiden höchstwertigen Bits stehen im EA-Register RAMPZ (Adr. 0x5B).
Initialisierungsschleife für das Feld a:

00007A	LDI R17,0x02	r17 \leftarrow 2
00007B	LDI R26,0x00	X \leftarrow 0x200
00007C	LDI R27,0x02	
00007D	LDI R30,0x42	Z \leftarrow 0x142
00007E	LDI R31,0x01	
00007F	LDI R16,0x00	RAMPZ \leftarrow 0
000080	OUT 0x3B,R16	
000081	RJMP PC+0x0003	
000082	ELPM R0,Z+	r0 \leftarrow PMem(Z); Z \leftarrow Z+1
000083	ST X+,R0	DMem(X) \leftarrow r0; X \leftarrow X+1
000084	CPI R26,0x0C	Teste, ob X \neq 0x20C ist. Wenn ja, Spring zu Adresse 0x82
000085	CPC R27,R17	
000086	BRNE PC-0x04	



Das mit

```
uint8_t b[10]
```

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

00087	LDI R18,0x02	$r18 \leftarrow 2$
00088	LDI R26,0x0C	$X \leftarrow 0x20C$
00089	LDI R27,0x02	
0008A	RJMP PC+0x0002	
0008B	ST X+,R1	$D\text{Mem}(X) \leftarrow 0 (r0); X \leftarrow X+1$
0008C	CPI R26,0x16	Teste, ob X 0x216 ist. Wenn ja, Spring zu Adresse 0x8B
0008D	CPC R27,R18	
0008E	BRNE PC-0x03	



Aufgaben

Aufgabe 2.3: Zeichenketteninitialisierung

Die Befehlsfolge rechts initialisiert eine Zeichenkettenvariable.

- Von welchen Befehlen und mit welchen Werten werden die Zeigerregister X und Z vor der Schleife initialisiert?
- Wie lautet die Abbruchbedingung der Schleife?
- Welche Anfangs- und Endadresse hat die Zeichenkettenvariable und mit welchem Ascii-Text wird sie initialisiert?

Hinweis: Die Umrechnung in Ascii-Zeichen finden Sie mit Google unter dem Stichwort »Ascii-Tabelle«.

0007A	LDI R17,0x02
0007B	LDI R26,0x00
0007C	LDI R27,0x02
0007D	LDI R30,0x20
0007E	LDI R31,0x01
0007F	LDI R16,0x00
00080	OUT 0x3B,R16
00081	RJMP PC+0x0003
00082	LPM R0,Z+
00083	ST X+,R0
00084	CPI R26,0x0A
00085	CPC R27,R17
00086	BRNE PC-0x04



00090	48.69
00091	6c.66
00092	65.74
00093	65.78

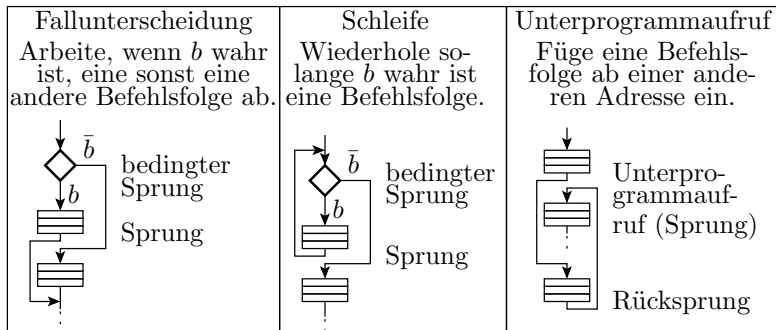


Kontrollfluss



Kontrollstrukturen

Der Kontrollfluss in Hochsprachen wird durch Fallunterscheidungen, Schleifen und Unterprogrammaufrufe beschrieben. Diese lassen sich durch unbedingte und bedingte Sprünge im Verarbeitungsfluss nachbilden.





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
$PC \leftarrow k$	3	1001 0100 0000 110k kkkk kkkk kkkk kkkk	jmp k
$PC \leftarrow 0:Z$	2	1001 0100 0000 1001	ijmp
$PC \leftarrow EIND:Z$	2	1001 0100 0001 1001	eijmp
$PC \leftarrow PC + 1 + k$	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: $-2048 \leq k \leq 2047$).



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).



Betragsbildung mit Skip-Befehl

```
11  int8_t a;
12  void main(void){
13      a = PINA;
14      if (a<0) a=-a;
15      PORTC = a;
16  }
```

```
00008A  IN  R24,0x00
00008B  SBRC R24,7
00008C  NEG  R24
00008D  STS  0x0200,R24
00008F  LDS  R24,0x0200
000091  OUT  0x08,R24
000092  RET
```

- Der in r24 eingelesene Wert ist negativ, wenn das führende Bit (7) eins ist.
- Der Skip-Befehl überspringt die nachfolgende Negation, wenn r24, Bit 7 null ist.

(Das Programm wurde mit Compiler-Optimierung -O2 übersetzt, weil auch bei -O1 noch einen bedingten Sprung verwendet wird.)



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



Die bedingten Sprünge werten ein Bit aus dem Statusregister (EA-Adresse 0x3F) aus und führen bei erfüllter Bedingung relative Sprünge $PC \leftarrow PC + 1 + k$ mit $-64 \leq k \leq 63$ aus.

Beispiel: »Betragbildung mit -O1«

<pre> 11 int8_t a=-9; 12 void main(void){ 13 a = PINA; 14 if (a<0) a=-a; 15 PORTC = a; 16 }</pre>	<pre> 0008A IN R24,0x00 0008B TST R24 0008C BRLT PC+0x04 0008D STS 0x0200,R24 0008F RJMP PC+0x0004 00090 NEG R24 00091 STS 0x0200,R24 00093 LDS R24,0x0200 00095 OUT 0x08,R24 00096 RET</pre>
--	---

- »tst r24« (TeST for zero or minus) testet den in r24 eingelesenen Wert und setzt die Flags Z, N und S.
- »brlt PC+4« springt, bei $r24 < 0$ zu Adresse 0x91.
- ... (10 Befehle, statt 7 bei Optimierung mit -O2)



Fallunterscheidungen



Fallunterscheidung mit if, else if und else

12	<code>void main(void){</code>	0007D	<code>LDI R25,0x2A</code>	Schreiben der 3 Ausgabewerte in Register
13	<code> register uint8_t a;</code>	0007E	<code>LDI R19,0x85</code>	
14	<code> while (1){</code>	0007F	<code>LDI R18,0x1D</code>	
15	<code> a = PINA;</code>	00080	<code>IN R24,0x00</code>	a ← PINA
16	<code> if (a<3)</code>	00081	<code>CPI R24,0x03</code>	wenn a < 3?
17	<code> PORTC = 0x2A;</code>	00082	<code>BRCC PC+0x03</code>	springe nach else-if
18	<code> else if (a<9)</code>	00083	<code>OUT 0x08,R25</code>	PORTC ← 0x2A
19	<code> PORTC = 0x1D;</code>	00084	<code>RJMP PC-0x0004</code>	
20	<code> else</code>	00085	<code>CPI R24,0x09</code>	wenn a < 9?
21	<code> PORTA = 0x85;</code>	00086	<code>BRCC PC+0x03</code>	springe nach else ...
22	<code> }</code>	00087	<code>OUT 0x08,R18</code>	PORTC ← 0x1D
23	<code>}</code>	00088	<code>RJMP PC-0x0008</code>	
		00089	<code>OUT 0x02,R19</code>	PORTA ← 0x85
		0008A	<code>RJMP PC-0x000A</code>	

(Übersetzt mit -O1).



Fallunterscheidung mit Case

<pre> 11 void main(void){ 12 uint8_t a = PINA; 13 switch (a){ 14 case 1: 15 PORTB = 0x43; 16 break; 17 case 2: 18 case 3: 19 PORTB = 0xD1; 20 break; 21 case 4: 22 PORTC = 0x42; 23 case 5: 24 PORTB = 0x4A; 25 break; 26 default: 27 PORTB = 0; } </pre>	<pre> 0007D IN R24,0x00 a ← PINA 0007E CPI R24,0x04 wenn a>4 0007F BRCC PC+0x06 gehe zu 0x85 00080 CPI R24,0x02 sonst wenn a>2 00081 BRCC PC+0x0C gehe zu 0x8D 00082 CPI R24,0x01 wenn a≠ 1 00083 BRNE PC+0x12 gehe zu 0x93 00084 RJMP PC+0x06 gehe zu 0x8A 00085 CPI R24,0x04 wenn a=4 00086 BREQ PC+0x0A gehe zu 0x90 00087 CPI R24,0x05 wenn a=5 00088 BRNE PC+0x0D gehe zu 0x95 00089 RJMP PC+0x09 gehe zu 0xA2 </pre>																								
	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Fall</th> <th>1</th> <th>2,3</th> <th>4</th> <th>5</th> <th>sonst</th> </tr> </thead> <tbody> <tr> <td>Adresse</td> <td>0x8A</td> <td>0x8D</td> <td>0x90</td> <td>0x92</td> <td>0x95</td> </tr> <tr> <td>PORTB</td> <td>0x43</td> <td>0xD1</td> <td>0x4A</td> <td>0x4A</td> <td>0</td> </tr> <tr> <td>PORTC</td> <td></td> <td></td> <td>0x42</td> <td></td> <td></td> </tr> </tbody> </table>	Fall	1	2,3	4	5	sonst	Adresse	0x8A	0x8D	0x90	0x92	0x95	PORTB	0x43	0xD1	0x4A	0x4A	0	PORTC			0x42		
Fall	1	2,3	4	5	sonst																				
Adresse	0x8A	0x8D	0x90	0x92	0x95																				
PORTB	0x43	0xD1	0x4A	0x4A	0																				
PORTC			0x42																						



Fall 1	0008A	LDI R24,0x43		PORTB ← 0x43			
	0008B	OUT 0x05,R24					
	0008C	RET			Rücksprung		
Fall 2,3	0008D	LDI R24,0xD1		PORTB ← 0xD1			
	0008E	OUT 0x05,R24					
	0008F	RET			Rücksprung		
Fall 5	Fall 4	00090	LDI R24,0x42		PORTC ← 0x42		
		00091	OUT 0x08,R24				
		00092	LDI R24,0x4A				PORTB ← 0x4A
		00093	OUT 0x05,R24				
00094	RET	Rücksprung					
sonst	00095	OUT 0x05,R1		PORTB ← 0			
	00096	RET			Rücksprung		



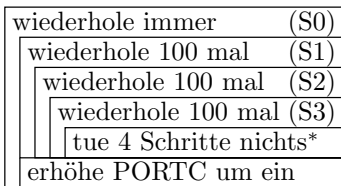
Schleifen



Warteschleife

Ziel sei ein kleines Programm, das PORTC so langsam hochzählt, dass es 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: Endlosschleife mit drei verschachtelten Wiederholschleifen



* 4 Schritte insgesamt mit Zähl- und Sprungoperation

```

12 void main(void){
13     register uint8_t a, b, c;
14     while (1){
15         for (a=0; a<100; a++){
16             for (b=0; b<100; b++){
17                 for (c=0; c<100; c++){
20                     PORTC += 1;
                }
            }
        }
    }
}

```



- Mit -O1 erzeugtes Maschinenprogramm:

0007D	LDI R21,0x64	initialisiere r18, r20 und
0007E	MOV R20,R21	r21 mit 100 (Anzahl
0007F	MOV R18,R21	der Schleifendurchläufe)
00080	MOV R19,R21	initialisiere S1: r19=100
00081	RJMP PC+0x0009	
00082	SUBI R24,0x01	zähle S3: r24 --
00083	BRNE PC-0x01	wiederhole, solange r24≠0
00084	SUBI R25,0x01	zähle S2: r25 --
00085	BREQ PC+0x03	wenn null, springe zu S1
00086	MOV R24,R18	initialisiere S3: r24=100
00087	RJMP PC-0x0005	springe zum Anfang von S3
00088	SUBI R19,0x01	zähle S1: r19 --
00089	BREQ PC+0x03	wenn null, springe zu S0
0008A	MOV R25,R20	initialisiere S2: r25=100
0008B	RJMP PC-0x0005	springe zurück in S2
0008C	IN R24,0x08	
0008D	SUBI R24,0xFF	PORTC ++
0008E	OUT 0x08,R24	
0008F	RJMP PC-0x000F	springe zum Anfang von S0



Aufgaben



Aufgabe 2.4: Sprungbedingung

Welches Statusbit werten die nachfolgenden bedingten Sprünge aus und bei welchem Bitwert wird der Sprung ausgeführt?

Bezieht sich der Vergleich auf vorzeichenfreie oder vorzeichenbehaftete Zahlen?

- 1 brlt (Branch if Less Than)
- 2 brpl (Branch if Plus)
- 3 brlo (Branch if Lower)

Das Statusregister des ATmega2560:

Bitnummer:	7	6	5	4	3	2	1	0
Bitname:	I	T	H	S	V	N	Z	C

C – Carry Flag, Z – Zero Flag, N – Negative Flag, V – Überlauf Zweierkomplement, S – Vorzeichen Zweierkomplement, H – Half Carry, T – Zwischenspeicher Bitkopieren, I – globale Interrupt-Freigabe.



Aufgabe 2.5: Reengineering If-Anweisung

Die nachfolgenden Assemblerprogrammausschnitte stammen von einem C-Programm:

```
(u)int8_t a; // Adresse 0x201
(u)int8_t b; // Adresse 0x200
...
if (a ?? b) PORTA = 0x3;
```

Bestimmen Sie aus der verwendeten Sprunganweisung, ob a und b als vorzeichenbehaftet oder vorzeichenfrei vereinbart waren und welche Vergleichsoperation für ?? im C-Programm stand?

```
00085 LDS R25,0x0201
00087 LDS R24,0x0200
00089 CP R24,R25
0008A BRCC PC+0x03
```

a)

```
00085 LDS R25,0x0201
00087 LDS R24,0x0200
00089 CP R24,R25
0008A BRLT PC+0x03
```

b)



Aufgabe 2.6: Reengineering Switch-Anweisung

Ergänzen Sie in dem nachfolgenden C-Programm die fehlenden Konstanten K1 bis K6 anhand des zugehörigen Assemblerprogramms, in das der Compiler die dargestellte Switch-Anweisung übersetzt hat.

```

11 void main(){
12     switch (PINA) {
13         case K1:
14         case K2: PORTC = K4; break;
15         case K3: PORTC = K5; break;
16         default: PORTC = K6;
17     }
18 }

```

PINA hat Adresse 0
PORTC hat Adresse 8

```

0007D IN R24,0x00
0007E CPI R24,0x15
0007F BREQ PC+0x05
00080 CPI R24,0x38
00081 BREQ PC+0x06
00082 CPI R24,0x12
00083 BRNE PC+0x07
00084 LDI R24,0x27
00085 OUT 0x08,R24
00086 RET
00087 LDI R24,0x44
00088 OUT 0x08,R24
00089 RET
0008A LDI R24,0x22
0008B OUT 0x08,R24
0008C RET

```



Unterprogramme

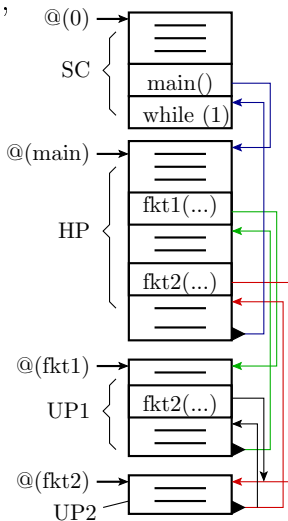


Unterprogramme

Unterprogramme sind Programmbausteine,

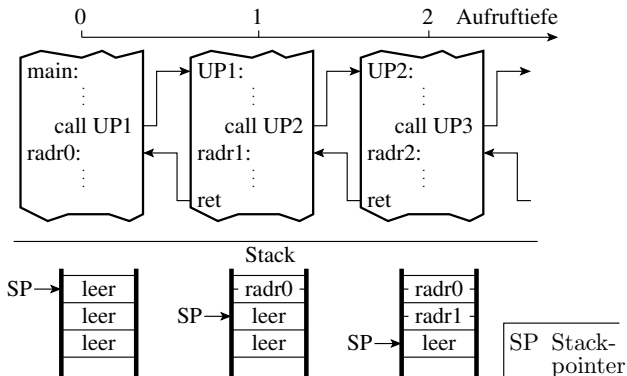
- die als Befehlsfolge vom Programmiergerät in den Befehlsspeicher (Flash) geschrieben und
- durch Aufruf ihrer Adresse in den Programmfluss anderer Programme eingefügt werden.

SC	automatisch eingefügter Startcode
HP	Hauptprogramm
UP _{<i>i</i>}	Unterprogramm <i>i</i>
@(0)	Adresse 0, Startadresse Mikrorechner nach Neuprogrammierung, Einschalten,
@(...)	Startadresse Unterprogramm
▶	Rücksprung
***(...)	Aufruf von *** mit den Parametern ...
while (1)	Endlosschleife
==	andere Anweisungen





Verwaltung der Rücksprungadressen



Die Rückkehradressen werden auf einem Stapelspeicher (Stack) nach dem Prinzip »First In Last Out« gespeichert und beim Rücksprung wieder entnommen. Nach diesem Prinzip können Unterprogramme selbst Unterprogramme verwenden.



Befehle für die Arbeit mit Unterprogrammen

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

- push und pop dienen zur Zwischenablage von Registerinhalten auf dem 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
```



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 Stack⁵ 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.

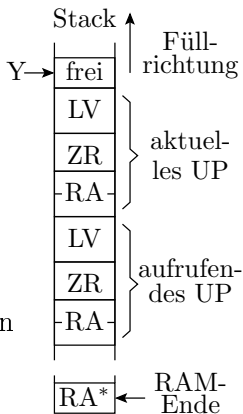
⁵Ab -O1 erhalten Variablen, wenn Platz ist, Registeradressen.



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.



LV lokale Variablen
 ZR zu sichernde Register
 RA Rückkehradresse zum aufrufenden Unterprogr.
 RA* Rückkehradresse zum Startup-Code



Beispielprogramm mit -O0

11	uint8_t g1, g2;	0008E	LDI R24,0x7A	15
12	void main(void){	0008F	STD Y+3,R24	
13	uint8_t l1 = 0x83;	00090	LDD R25,Y+1	16
14	uint8_t l2 = 0x45;	00091	LDD R24,Y+2	
15	uint8_t l3 = 0x7A;	00092	ADD R24,R25	
16	g1 = l1 + l2;	00093	STS 0x0200,R24	
17	g2 = g1 + l3;	00095	LDS R25,0x0200	
	}	00097	LDD R24,Y+3	17
00085	PUSH R28	00098	ADD R24,R25	
00086	PUSH R29	00099	STS 0x0201,R24	
00087	RCALL PC+0x0001	0009B	POP R0	
00088	IN R28,0x3D	0009C	POP R0	
00089	IN R29,0x3E	0009D	POP R0	
0008A	LDI R24,0x83	0009E	POP R29	
0008B	STD Y+1,R24	0009F	POP R28	
0008C	LDI R24,0x45	000A0	RET	
0008D	STD Y+2,R24			



```

00085  PUSH R28      |
00086  PUSH R29      | 1
00087  RCALL PC+0x0001 | 2
00088  IN R28,0x3D  | 3
00089  IN R29,0x3E  |
                                |
0009B  POP R0       |
0009C  POP R0       |
0009D  POP R0       | 4
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:

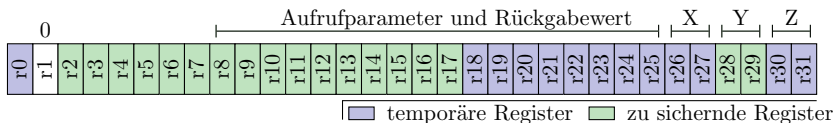
Variable	l1	ls2	l3
Adresse	Y+1	Y+2	Y+3

- 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 dem Stack gesichert werden.



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

- Von r1 wird erwartet, dass sein Wert immer null ist
- r0, r18:27 (incl. X) und r30:31(Z): Temporäre Register, die von aufgerufenen Unterprogrammen verändert werden dürfen. (Sicherung durch aufrufendes Programm.)
- r2:17, r28:29 (Y): zu sichernde Register. Sie muss das aufgerufene Programm, falls es sie nutzt, auf dem Stack sichern.



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){
    1
    10008B PUSH R17
    10008C PUSH R28
    10008D PUSH R29
    19 uint8_t a = PINA;
    20 uint8_t b = PINB;
    21 uint8_t c = PINC;
    22 UP();
    24 uint8_t e = a - c;
    23 uint8_t d = a + b;
    25 g = d | e;
    100096 STS 0x0200,R28
    100098 POP R29
    100099 POP R28
    10009A POP R17
    26 }
```

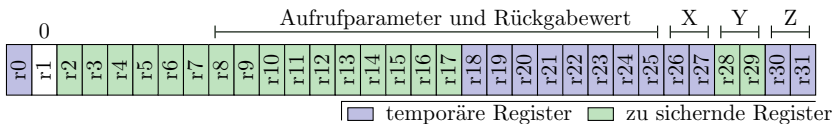
- 1 Register r17, r28 und r29 auf dem Stack sichern.
- 2 Register r17, r28 und r29 vom Stack holen.



Parameterübergabe



Registernutzung und Parameterübergabe



- Von links 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 locati	Error
b	Unknown locati	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

```

00088 RCALL PC-0x000B
00089 MOVW R18,R24
0008A SUBI R18,0xFC
0008B SBCI R19,0xFF
0008C MOV R24,R18
0008D MOV R25,R19
0008E RET

```

```

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

```

Subtraktion $0xFE = -4$

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



Rekursion



Rekursion

Ein rekursives Programm ruft sich so lange selbst auf, bis eine Abbruchbedingung erreicht ist. Beispiel für ein rekursiv beschreibbarer 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	Rechtverschiebung der 4 Bytes der Variablen a.
00085	ROR R23	
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 r16 bis r19 kopieren.
0008A	MOVW R18,R24	
0008B	MOV R22,R16	r16 bis r19 zurück in r22 bis r25 kopieren (offenbar ohne Sinn).
0008C	MOV R23,R17	
0008D	MOV R24,R18	
0008E	MOV R25,R19	
0008F	POP R17	Die auf den Stack abgelegten Inhalte zurück in r16 und r17 kopieren.
00090	POP R16	
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.

```

data 0x21E0  00 00 17 e1
data 0x21E4  00 00 89 2f
data 0x21E8  c3 00 00 89
data 0x21EC  5f 86 00 00
data 0x21F0  89 bf 0d 00
data 0x21F4  00 89 7e 1a
data 0x21F8  00 00 89 00
data 0x21FC  00 00 00 7b
  
```

00 00 7b	Rücksprungadresse zum Startup-Code
00 00 89	Rücksprungadresse zum Unterprogramm
	Wert von r16
	Wert von r17



Aufgaben



Aufgabe 2.7: Übergaberegister

Das nachfolgende in einer Header-Datei vereinbarte Unterprogramm:

```
uint16 UP(uint8_t a, uint16_t b, uint8_t c);
```

soll in Assembler geschrieben werden. In welchen Registern bekommt das Assemblerprogramm die Operanden übergebenen und in welchen Registern muss der Rückgabewert stehen?

Aufgabe 2.8: Multiplizierunterprogramm

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

- In welchen Registern werden die Faktoren übergeben und in welchen Registern erwartet das Hauptprogramm das Ergebnis?
- Bestimmen Sie auf der nächsten Folie die Werte, die nach jeder Anweisung in den Registern stehen und füllen Sie die Tabelle aus.
- Was ist bei der Berechnung falsch?
- Schreiben Sie ein Assemblerprogramm mit derselben Aufrufschnittstelle, das ein korrektes 4-Byte-Produkt berechnet und zurück gibt.



Aufgabe 2.9: Schleife mit Fehler

Das nachfolgende C-Programm enthält eine while-Schleife, in der die Variable a solange ihr Wert kleiner 256 ist um 1 erhöht wird. Dazu sind die disassemblierten mit O0 und mit O1 übersetzten Programme gezeigt.

C-Programm mit while-Schleife

```
11 void main(void){
12     uint8_t a;
13     while(a<256){
14         a++;
15     }
16 }
```

Mit O1 compiliertes Programm

```
0007D RJMP PC-0x0000
```

Mit O0 compiliertes Programm

```
0007D PUSH R28
0007E PUSH R29
0007F PUSH R1
00080 IN R28,0x3D
00081 IN R29,0x3E
00082 LDD R24,Y+1
00083 SUBI R24,0xFF
00084 STD Y+1,R24
00085 RJMP PC-0x0003
```



- Warum wird das Programm mit `-O1` in eine Endlosschleife übersetzt, die nichts tut?
- Verhält sich das mit `-O0` übersetzte Programm anders?
- Wie ist das C-Programm zu verändern, damit die Schleife abbricht, wenn `a` nicht mehr kleiner als 256 ist?
- Welchen Wert hat in dem mit `-O0` übersetzten Programm der Stackpointer am Eintrittspunkt von `main()` (Halt vor Adresse `0x7D`)?
- Welchen Wert hat in dem mit `-O0` übersetzten Programm der Framepointer nach Abarbeitung der Anweisung auf Adresse `0x81`?

Hinweis: Vor Aufruf von `main()` wird der Stackpointer mit `0x21FF` initialisiert und eine Rücksprungadresse beansprucht 3 Bytes auf dem Stack.