Rechnerarchitektur, Foliensatz 3
Kontrollfluss

G. Kemnitz

Institut für Informatik, TU Clausthal (RA-F3.pdf)

16. Januar 2020

Wiederholung

Kontrollfluss

2.1 MiPro

2.2 AVR

2.3 Warteschleife

Unterprogramme

3.1 MiPro

3.2 AVR UP-Aufruf, Stack, ...

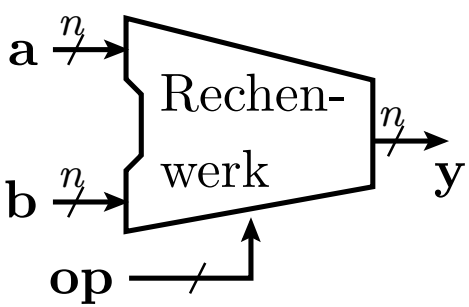
3.3 Lokale Variablen

3.4 Parameterübergabe

Aufgaben

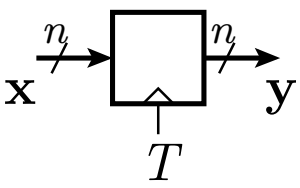
Wiederholung

Grundbausteine von Rechnern



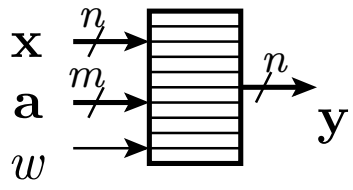
$y = op(a, b)$

Register



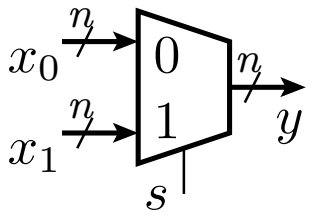
wenn $T = \_\!\!\_\!\!\_\!\!\_\!\!\_\!\!\_$
$y = x$
sonst speich.

adr. Speicher



wenn $w = 1$
$mem(a) = x$
$y = mem(a)$

Multiplexer



wenn $s = 0$
dann $y = x_0$
sonst $y = x_1$

...

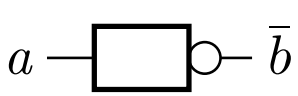
$\xrightarrow{n}$ n-Bit-Verbindung

a, **b**, **x**, **y** n-Bit-Datensignale

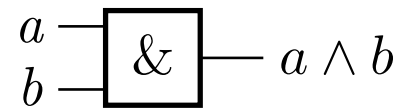
op Operationsauswahlsignal
a m-Bit-Adresssignal
T Taktsignal

- Rechenwerke, Register, ... bestehen aus Logikgattern:

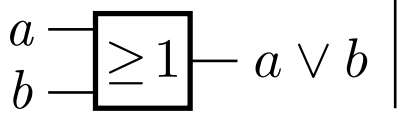
Inverter



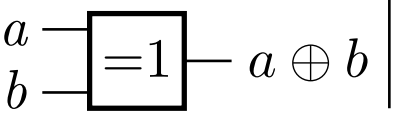
UND



ODER

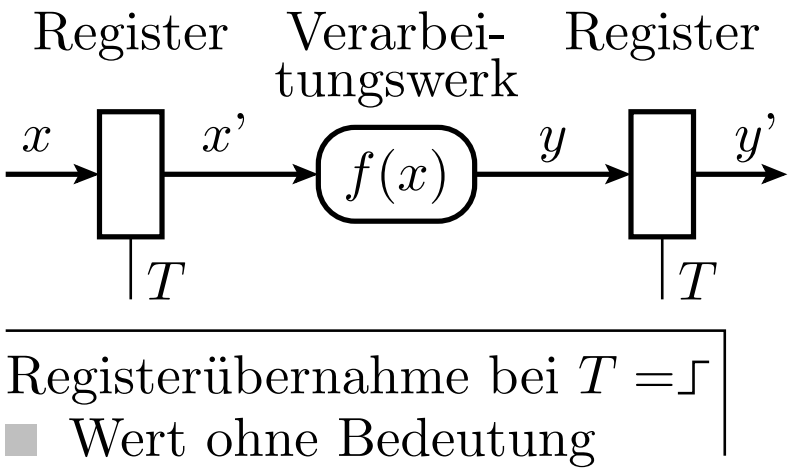


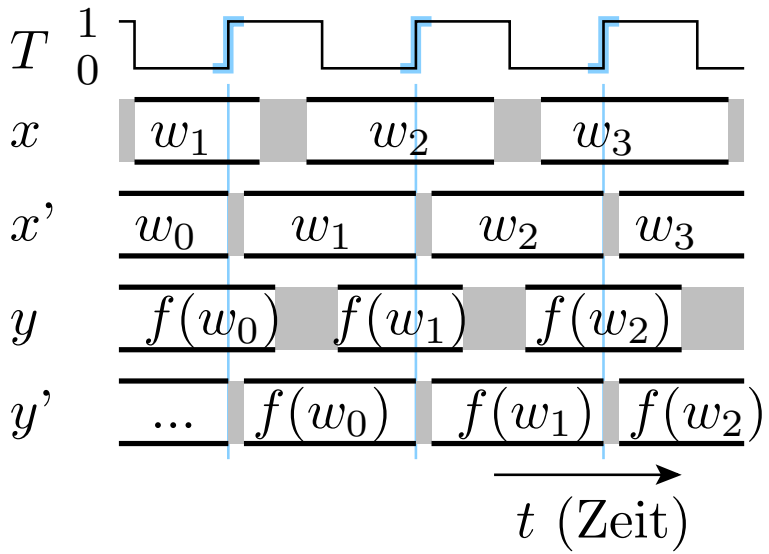
EXOR



...

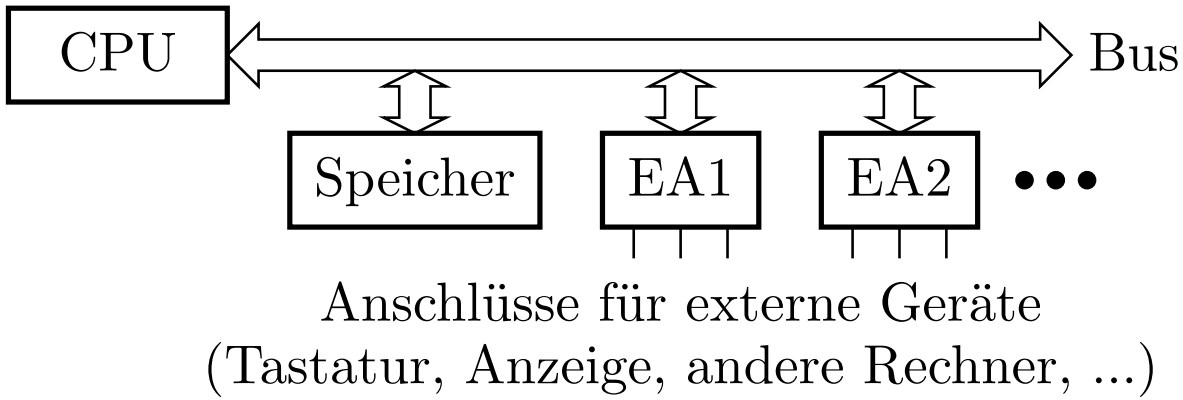
Ein Verarbeitungsschritt dauert einen Takt





- Zeitabläufe in Rechnern werden vom Takt gesteuert, einem periodisch zwischen 0 und 1 wechselndem Signal.
- Operanden, Adressen, ... werden immer mit der aktiven Taktflanke in Register übernommen und sonst gespeichert.
- Die Taktperiode muss mindestens so groß wie die maximale Verzögerung bei der Verarbeitung sein.

Funktionsweise eines Universalrechners



- Befehle und Daten stehen in einem Speicher.
- Der Prozessor (CPU **C**entral **P**rocessing **U**nit) führt für jeden Befehl eine Folge von Aktionen aus:
  - Befehlswort lesen (IF **I**nstruction **F**etch)
  - Operanden Laden (OF **O**perand **F**etch)
  - Operation ausführen (EX **Ex**ecute)
  - Ergebnis schreiben (RW **R**esult **W**rite).

RISC-Prozessoren

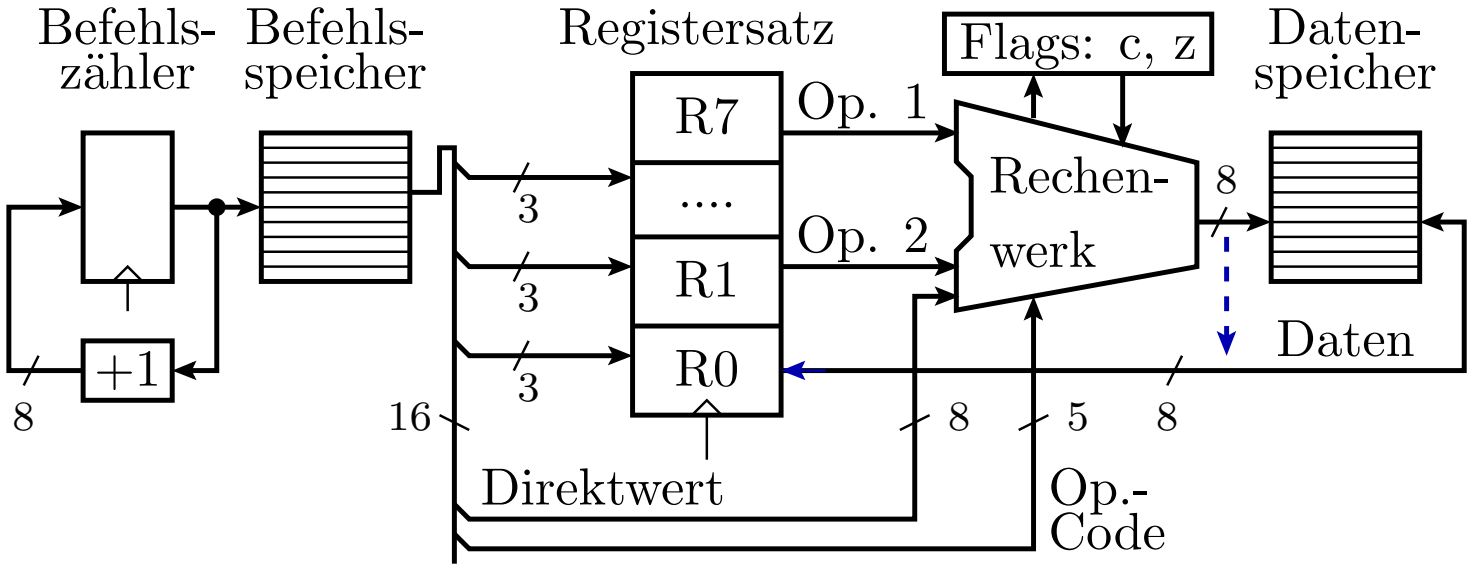
Nur Befehle, die in ein Befehlswort passen und in einem Schritt[1] abarbeitbar sind. Befehlssatz des Minimalprozessors:

Teilbitvektor	15 ... 11	10 9 8	7 6 5	4 3 2	1 0	cnr
nop:	00000					0
jump imm,cond	00001	cond	imm			1
cmd rd,imm	cnr	rd	imm			2 bis 14
cmd rd	cnr	rd				15
cmd rd,ra	cnr	rd	ra			16 bis 23
cmd rd,ra,rb	cnr	rd	ra	rb		24 bis 30

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

[1] Einer Pipeline-Zeitscheibe.

Verarbeitungsfluss eines RISC-Prozessors (MiPro)



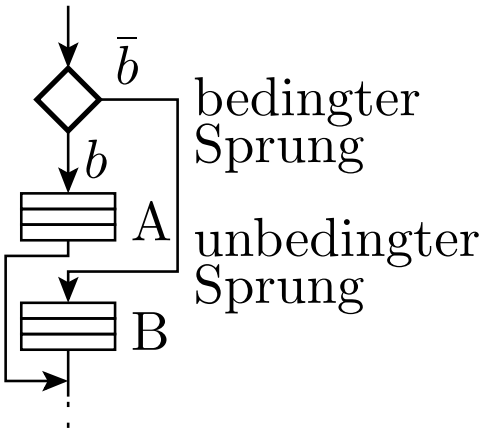
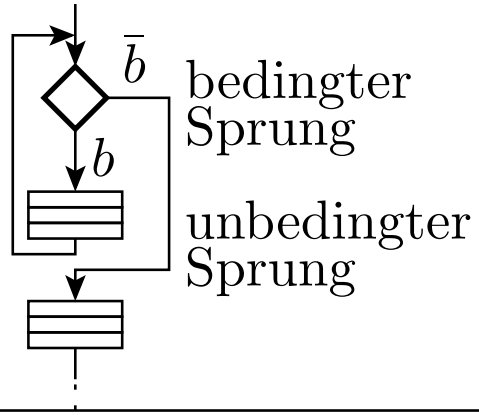
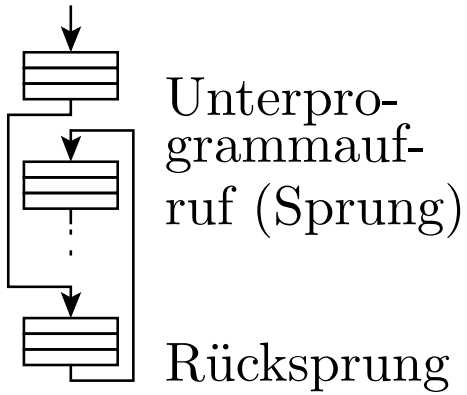
Die Ausführung von einem Befehl pro Schritt erfordert:

- getrennten Befehls- und Datenspeicher,
- 3-Port-Registersatz hier 8, typ. 32 Speicherplätze.
- Load/Store-Architektur (getrennte Lade- und Speicherbefehle).
- Datenspeicheradresse: Register, Konstante oder berechnet.

Kontrollfluss

Steuerung des Kontrollflusses

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

Fallunterscheidung	Schleife	Unterprogrammaufruf
Wenn b wahr, arbeite Befehlsfolge A sonst Befehlsfolge B ab.	Wiederhole solange b wahr ist eine Befehlsfolge.	Füge eine Befehlsfolge ab einer anderen Adresse ein.
		

Sprünge, Unterprogrammaufrufe, ...

- Absoluter Sprung:

  ```
  PC := K
  ```

- relativer Sprung:

  ```
  PC := PC + 1 + K
  ```

- bedingter Sprung (in der Regel relativ)

  ```
  wenn b dann PC := PC + 1 + K
  sonst PC := PC + 1
  ```

- Unterprogrammaufruf:

  ```
  Rd := PC + 1; PC := K
  ```

- Rücksprung aus einem Unterprogramm:

  ```
  PC := Rr
  ```

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

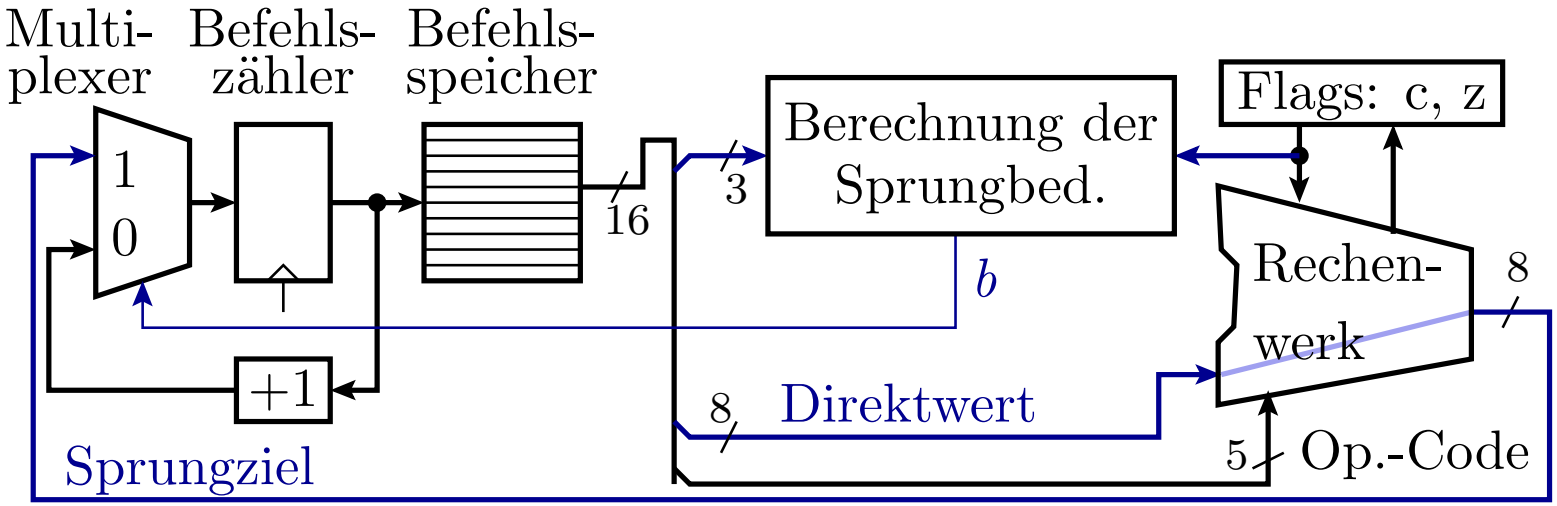
MiPro

Sprungbefehle des Minimalprozessors

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

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

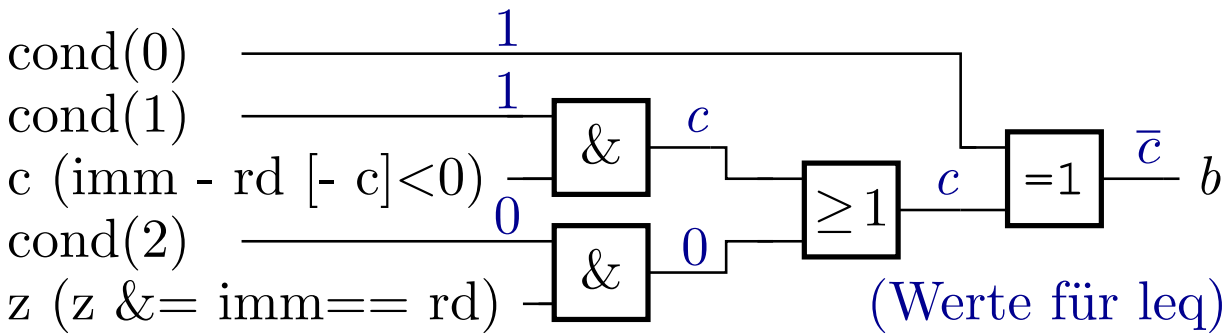
MiPro-Erweiterungen für Sprungbefehle




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

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

Berechnung der Sprungbedingung



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

Testbeispiel mit Fallunterscheidung

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

```
;=============Test1=============
0000: ld_i r0,07,..  ; r0 := 0x07
0001: comp r0,37,..  ; 0x37-r0: größer 0 (c:=0, z:=0)
0002: jump 05,leq..  ; wenn größer 0, springe zu 0x5
0003: ld_i r1,01,..  ; r1 := 0x01
0004: jump 06,alw..  ; springe zu 6
0005: ld_i r1,04,..  ; r1 := 0x04
0006: ld_i r2,1A,..  ; ... (immer)
;=============Test 2=============
0000: ld_i r0,48,..  ; r0 := 0x48
0001: comp r0,37,..  ; 0x37-r0: kleiner 0 (c:=1, z:=0)
0002: jump 05,leq..  ; wenn größer 0, springe zu 0x05
0003:  ...
```

Wie werden die beiden Testbeispiele abgearbeitet?

Lösung

;=========Test1=========

PC	Befehl assem.: hex	r0	r1	r2	r3	r4	r5	r6	r7	c	z
00	ld_i r0,07,..:2807	07	..	..	..	..	..	..	..	.	.
01	comp r0,37,..:3037	**	..	..	..	..	..	..	..	0	0
02	jump 05,leq..:0b05	**	..	..	..	..	..	..	..	*	*
05	ld_i r1,04,..:2904	**	04	..	..	..	..	..	..	*	*
06	ld_i r2,1A,..:2848	**	**	1A	..	..	..	..	..	*	*

;=========Test 2=========

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

. – unbekannt; * – keine Zuweisung

Testbeispiel mit Schleife

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

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

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

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

```
PC| Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,01,..:2801|01 .. .. .. .. .. .. ..|.|.|
01|ld_i r1,34,..:2934|** 34 .. .. .. .. .. ..|.|.|
02|st_r r1,r0,..:9100|** ** .. .. .. .. .. ..|.|.|
                ;dmem = [.. 34 .. .. .. .. .. ..]
03|subr r1,r1,r0:d120|** 33 .. .. .. .. .. ..|0|0|
04|addi r0,01,..:4001|02 ** .. .. .. .. .. ..|0|0|
05|comp r0,03,..:3003|** ** .. .. .. .. .. ..|0|0|
06|jump 02,leq..:0b02|** ** .. .. .. .. .. ..|*|*|
02|st_r r1,r0,..:9100|** ** .. .. .. .. .. ..|*|*|
                ;dmem = [.. ** 33 .. .. .. .. ..]
03|subr r1,r1,r0:d120|** 31 .. .. .. .. .. ..|0|0|
04|addi r0,01,..:4001|03 ** .. .. .. .. .. ..|0|0|
05|comp r0,03,..:3003|** ** .. .. .. .. .. ..|0|1|
06|jump 02,leq..:0b02|** ** .. .. .. .. .. ..|*|*|
02|st_r r1,r0,..:9100|** ** .. .. .. .. .. ..|*|*|
                ;dmem = [.. ** ** 31 .. .. .. ..]
03|subr r1,r1,r0:d120|** 2e .. .. .. .. .. ..|0|0|
04|addi r0,01,..:4001|04 ** .. .. .. .. .. ..|0|0|
05|comp r0,03,..:3003|** ** .. .. .. .. .. ..|1|0|
06|jump 02,leq..:0b02|** ** .. .. .. .. .. ..|*|*|
```

AVR

Unbedingte Sprünge

Es gibt drei Arten der Sprungzielvorgabe:

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

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

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, ungenutzte Bits 0); k – 12-Bit-Sprungdistanz, WB: -2048$\leq k \leq$2047.

Bedingte Sprünge

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

Identische Befehle mit bedeutungsorientierten Bezeichnern:

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

b		SREG(b) = 1	SREG(b) = 0
0	C	brcs (if **C**arry **S**et), brlo (if **Lo**wer$^{(u)}$)	brcc (if **C**arry **C**lear), brsh (if **S**ame or **H**igher$^{(u)}$)
1	Z	breq (if **Eq**ual)	brne (if **N**ot **E**qual)
2	N	brmi (if **Mi**nus)	brpl (if **Pl**us)
3	V	brvs (if O**v**erflow is **S**et$^{(s)}$)	brvc (if O**v**erflow **C**leared$^{(s)}$)
4	S	brge (**G**reater or **E**qual$^{(s)}$)	brlt (**L**ess **T**han$^{(s)}$)
5	H	brhs (if **H**alf Carry is **S**et)	brhc (if **H**alf Carry **C**leared)
6	T	brts (if **T** flag is **S**et)	brtc (if **T** flag is **C**leared)
7	I	brie (if **I**nterrupt **E**nabled)	brid (if **I**nterrupt **D**isabled)

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	1/2/3	1001 00rd dddd rrrr	cpse Rd,Rr
Bit b in Rr gesetzt	1/2/3	1111 111r rrrr 0bbb	sbrs Rr,b
Bit b, Rr gelöscht	1/2/3	1111 110r rrrr 0bbb	sbrc Rr, b
Bit b, IO-Reg. A eins	1/2/3	1001 1001 AAAA Abbb	sbis A,b
Bit b in IO-Reg. A null	1/2/3	1001 1011 AAAA Abbb	sbic A,b

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

Beispiel Betragsbildung

In einer Endlosschleife wird von Port A ein vorzeichenbehaftetes Byte gelesen, der Betrag gebildet und auf Port B ausgegeben:

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

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

Übersetzung mit »-O0«

```
int main(){
    while (1); a=PINA;          //Beginn Endlosschleife
M1://0x89   LDI R24,0x20        ; r25:r24 := 0x0020
// 0x008A   LDI R25,0x00        ; (0x20: Adresse PINA)
// 0x008B   MOVW R30,R24        ; r31:r30=(Z) := r25:r24
// 0x008C   LDD R24,Z+0         ; r24 := PINA
// 0x008D   STS 0x0200,R24      ; a := r24 (0x200: &a)
          if (a<0) a=-a;
// 0x008F   LDS R24,0x0200      ; r24 := a
// 0x0091   TST R24             ; Test r24
// 0x0092   BRGE PC+0x06        ; wenn ≥0 springe zu M2
// 0x0093   LDS R24,0x0200      ; r24 := a
// 0x0095   NEG R24             ; r24 := -r24
// 0x0096   STS 0x0200,R24      ; a := r24
M2:   PORTB = a; }              // mehrere Befehle
// 0x009E   RJMP PC-0x0015      ; springe zu M1
```

- Übersetzung mit »-O1«

```
int main(){
      while (1){
       a=PINA;
M1://0x85  IN R24,0x00     ; r24 := PINA
       if (a<0) a=-a;
// 0x0086  TST R24         ; Test r24
// 0x0087  BRLT PC+0x04    ; wenn <0, spring zu M2
       a=PINA;
// 0x0088  STS 0x0200,R24  ; a := r24
// 0x008A  RJMP PC+0x0004  ; springe zu M3
       if (a<0) a=-a;
M2://0x8B  NEG R24         ; r24 := -r24
// 0x008C  STS 0x0200,R24  ; PORTB := r24
       PORTB = a;
M3://0x8E  LDS R24,0x0200  ; r24 := a
// 0x0090  OUT 0x05,R24    ; PORTB := r24
      }
// 0x0091  RJMP PC-0x000C  ; springe zu M1
```

- Übersetzung mit »-O2«

```
int main(){
        while (1){
         a=PINA;
// 0x0085  IN R24,0x00     ; r24 := PINA
         if (a<0) a=-a; PORTB = a;
// 0x0086  SBRC R24,7      ; skip, wenn r24.7=0 (≥0)
// 0x0087  RJMP PC+0x0007  ; springe zu M2
M1://0x88  STS 0x0200,R24  ; a := r24
// 0x008A  OUT 0x05,R24    ; PORTB := r24
// 0x008B  IN R24,0x00     ; r24 = PINA
// 0x008C  SBRS R24,7      ; skip, wenn r24.7=1 (<0)
// 0x008D  RJMP PC-0x0005  ; springe zu M1
M2://0x8E  NEG R24         ; r24 := -r24
// 0x008F  RJMP PC-0x0007  ; springe zu M1
```

Je höher die Optimierung, desto schneller und kürzer das Programm.
Optimierte Programme nicht im Schrittbetrieb testbar.

Warteschleife

Warteschleife

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

- Bei 8 Millionen Takten pro Sekunde soll der Prozessor zyklisch ca. 4 Millionen Takte nichts tun und dann den Ausgabewert um eins erhöhen.
- Lösungsansatz: Warteschleife, die $N = 10^6$ mal n Befehle in der innersten Schleife abarbeitet. Wenn n bekannt ist, N anpassen:

```
int main(){
  register uint32_t a;
  while (1) {
    for (a=0; a<1000000; a++);
    PORTJ ++;
  }
}
```

Optimierung mit -O0

```
while (1){
 for (a=0;a<0xF4240;a++); // 0xF42F0 = 1000000
M1: //0x85  MOV R14, R1     ; r14 = 0          } r17:r14 := 0
// 0x0086  MOV R15, R1     ; r15 = 0          }
// 0x0087  MOVW R16, R14   ; r17:r16 := r15:r14
// 0x0088  RJMP PC+0x0006  ; springe zu M3
M2: //0x89  SER R24        ; r24 = 0xFF
//  0x008A  SUB R14,R24    ;
//  0x008B  SBC R15,R24    ;                  } r17:r14 ++
//  0x008C  SBCI R16,0xFF  ;                  }
//  0x008D  SBCI R17,0xFF  ;                  }
M3: //0x8E  ...            ; Vergl. r17:r14
//  0x008F  ...            ; mit 0x000F4240
 ⋮  innere Schleife: 13 Befehle
//  0x0095  BRCS PC-0x0C   ; springe zu M2
```

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

```
M1: //0x85 ...                ; r17:r14 := 0
M2: //0x89 ...                ; r17:r14 := r17:r14 - 1
M3: //0x8E  LDI R30,0x40      ;
// 0x008F   CP R14,R30        ; ?: r14-0x40
// 0x0090   LDI R30,0x42      ;
// 0x0091   CPC R15,R30       ; ?: r15-0x42-c
// 0x0092   LDI R30,0x0F      ;
// 0x0093   CPC R16,R30       ; ?: r15-0x0f-c
// 0x0094   CPC R17,R1        ; ?: r17-0x00-c
// 0x0095   BRCS PC-0x0C      ; wenn neg., springe zu M2
   PORTJ++;
// 0x0096   LDI R24,0x05      ; r24 := 0x05
// 0x0097   LDI R25,0x01      ; r25 := 0x01
// 0x0098   MOVW R30,R24      ; r31:r30(Z) := r25:r24
// 0x0099   LDD R18,Z+0       ; r18 := PORTJ
// 0x009A   SUBI R18,0xFF     ; r18 := r18 +1
// 0x009B   MOVW R30,R24      ; ohne Funktion
// 0x009C   STD Z+0,R18       ; PORTJ := r18
// 0x009D   RJMP PC-0x0018    ; spring zu M1
```

*

Z = 0x105
Adresse Port J

* Vergleich r17:r14 mit 0x000F4248 (Subtraktion ohne Ergebnisspeicherung)

Optimierung mit -O1

```
int main(){
// Initialisierung von Registern mit Konstanten
// 0x007D   LDI R21,0x40 ;
// 0x007E   LDI R20,0x42 ;
// 0x007F   LDI R19,0x0F ;
// 0x0080   LDI R18,0x00 ;
// 0x0081   LDI R30,0x05 ;
// 0x0082   LDI R31,0x01 ;
 while(1){for (a=0; a<1000000; a++); ...
M1: //0x83   MOV R24,R21   ;
// 0x0084   MOV R25,R20   ;
// 0x0085   MOV R26,R19   ;
// 0x0086   MOV R27,R18   ;
// innere Schleife
M2: //0x87   SBIW R24,0x01; r27:r24 --
// 0x0088   SBC R26,R1    ; r26 -= 0 -c
// 0x0089   SBC R27,R1    ; r27 -= 0 - c
// 0x008A   BRNE PC-0x03  ; wenn r27:r24 nicht 0, springe zu M2
```

r18:r21 := 1.000.000

r31:r30(Z) := 0x105
(0x105: Adresse von Port J)

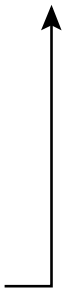
Anfang While-Schleife

r27:r24 := r18:r21

r27:r24 -= 1

n =4 Befehle in der inneren Schleife.

```
 PORTJ ++;
// 0x008B  LDD R24,Z+0  ; r24 := PORTJ
// 0x008C  SUBI R24,0xFF; r24 := r24 + 1
// 0x008D  STD Z+0,R24  ; PORTJ := r24
   }
// 0x008E  RJMP PC-0x000B; springe zu M1
```



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

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

Wegen der Abhängigkeit vom Prozessor, dessen Takt, der Compiler-Optimierung, ... Wartezeiten besser mit Timer erzeugen (siehe später Foliensatz RA-F6.pdf).

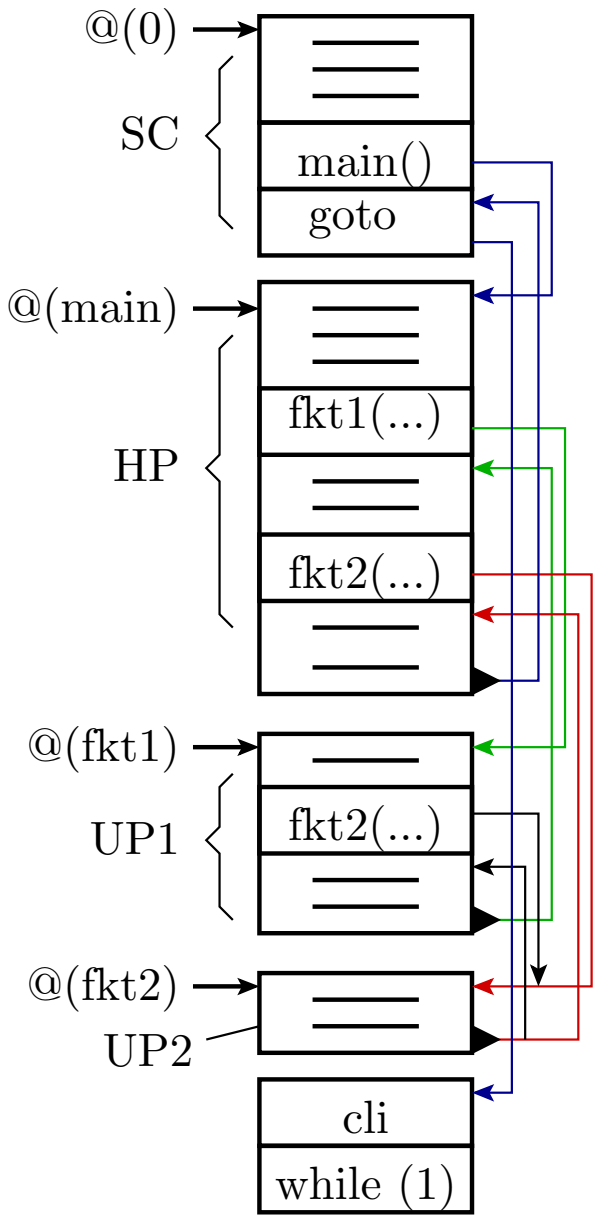
Unterprogramme

Unterprogramme

Unterprogramme sind Programmbausteine,

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

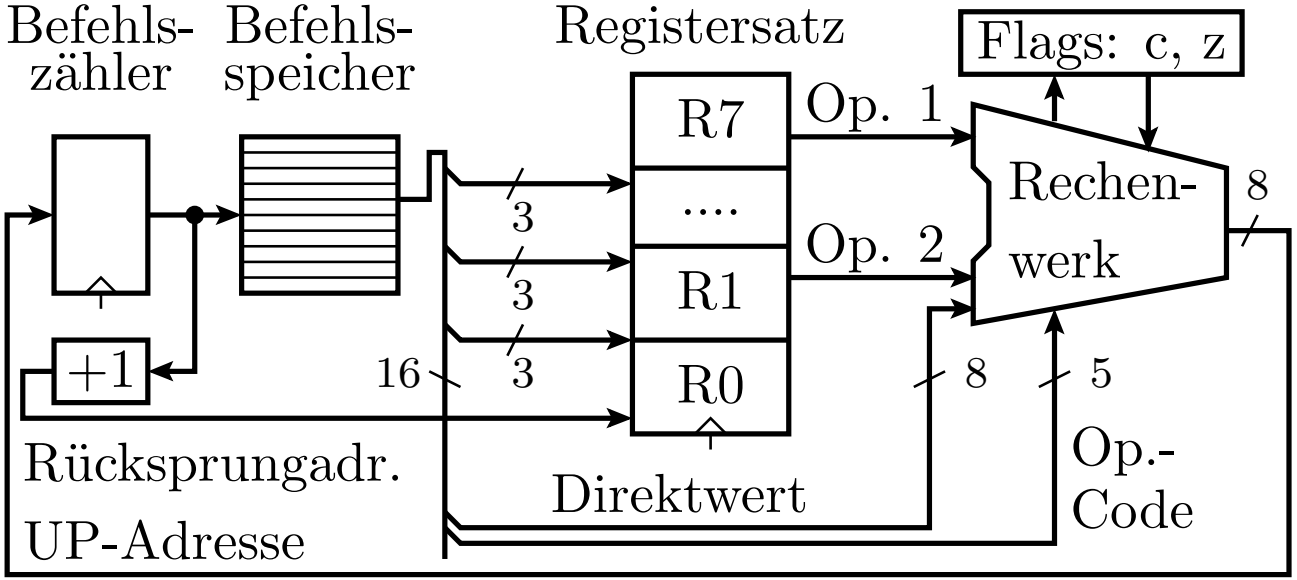
SC	automatisch eingefügter Startcode
HP	Hauptprogramm
UPi	Unterprogramm i
@(0)	Adresse 0, Startadresse Mikrorechner nach Neuprogrammierung, Einschalten,
@(...)	Startadresse Unterprogramm
▶	Rücksprung
═	andere Anweisungen



MiPro

Hardware-Erweiterung für Call- und Return-Befehl

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



Unterprogrammaufrufe auf MiPro

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

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

Testbeispiele:

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

Programmablauf

- Aufruf mit *(1)=0x35 und r1=2, Ergebnis *(2): 0x48

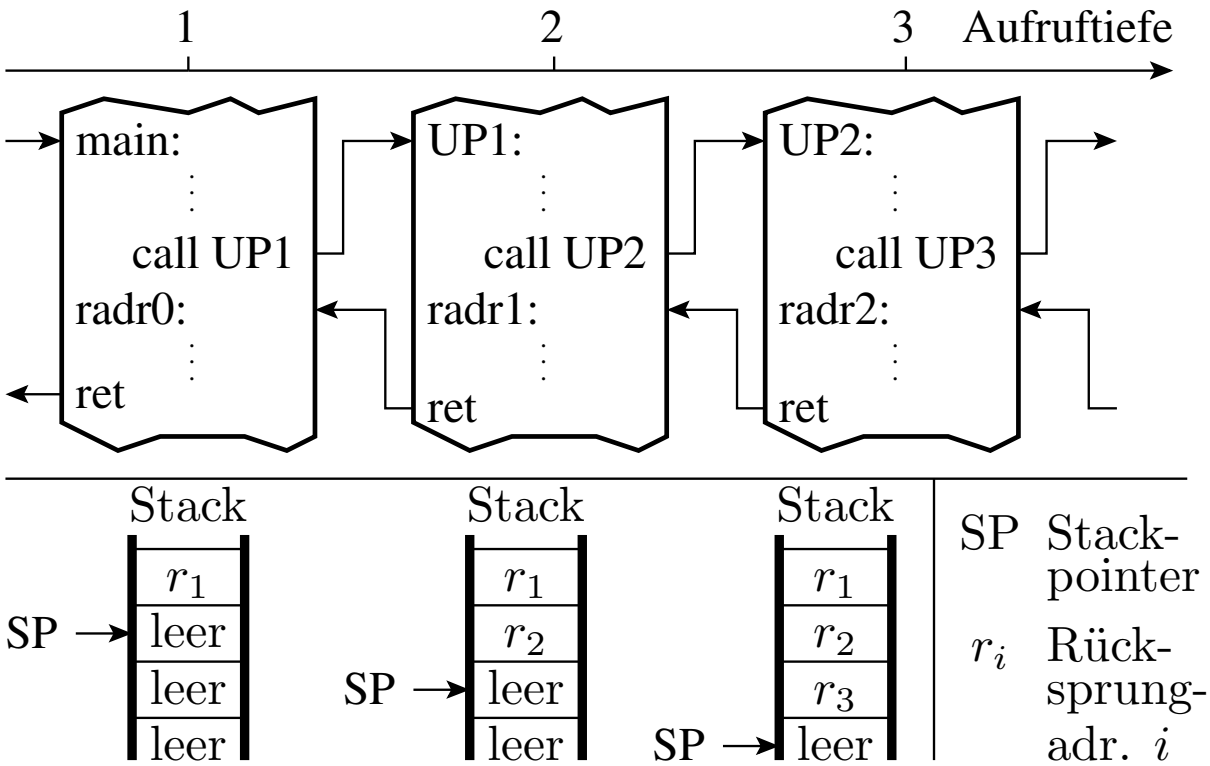
```
PC| Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|c|z|
00|ld_i r0,35,..:2835|35 .. .. .. .. .. .. ..|.|.|
01|stor r0,01,..:2001|** .. .. .. .. .. .. ..|.|.|
                ;dmem = [.. 35 .. .. .. .. .. ..]
02|ld_i r1,02,..:2902|** 02 .. .. .. .. .. ..|.|.|
03|call r5,10,..:1510|** ** .. .. .. 04 ..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

. – unbekannt; * – keine Zuweisung

- Aufruf mit *(1)=0x46 und r1=4, Ergebnis *(4)=0x59

```
PC| Befehl assem.: hex |r0 r1 r2 r3 r4 r5 r6 r7|c|z|
  ;=> Fortsetzung
04|ld_i r0,46,..:2846|46 ** .. ** .. ** .. ..|*|*|
05|stor r0,01,..:2001|** ** .. ** .. ** .. ..|*|*|
                ;dmem = [.. 46 ** .. .. .. .. ..]
06|ld_i r1,04,..:2904|** 04 .. ** .. ** .. ..|*|*|
07|call r5,10,..:1510|** ** .. ** .. 08 .. ..|*|*|

 ;Unterprogramm
10|load r3,01,..:1b01|** ** .. 46 .. ** .. ..|*|*|
11|addi r3,13,..:4313|** ** .. 59 .. ** .. ..|0|0|
12|st_r r3,r1,..:9320|** ** .. ** .. ** .. ..|*|*|
                ;dmem = [.. ** ** .. 59 .. .. ..]
13|retu r5,..,..:7d00|** ** .. ** .. ** .. ..|*|*|

 ;Endlosschleife
08|jump 08,alw..:0908|** ** .. ** .. ** .. ..|*|*|

---

 ;dmem Programmende: [.. 46 48 .. 59 .. .. ..]
```

Stapelverwaltung der Rücksprungadressen



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

AVR UP-Aufruf, Stack, ...

Befehle für die Arbeit mit Unterprogrammen

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

- 12 Bit-Sprungdistanz ($\pm$2k) bzw. 16- oder 17-Bit-Sprungziel.
- push und pop: Zwischenablage Registerinhalte auf Stack.

Stack einrichten

Der Stack ist ein Bereich des Datenspeichers, der vom Stackpointer adressiert wird. Der Stackpointer besteht aus den SFR (Special Function Registern) SPL und SPH mit 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:

```
0x0074   SER R28         ; r29:r28 := 0x21FF
0x0075   LDI R29,0x21    ;
0x0076   OUT 0x3E,R29    ; SP  := r29:r28
0x0077   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 Stack[2] 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)

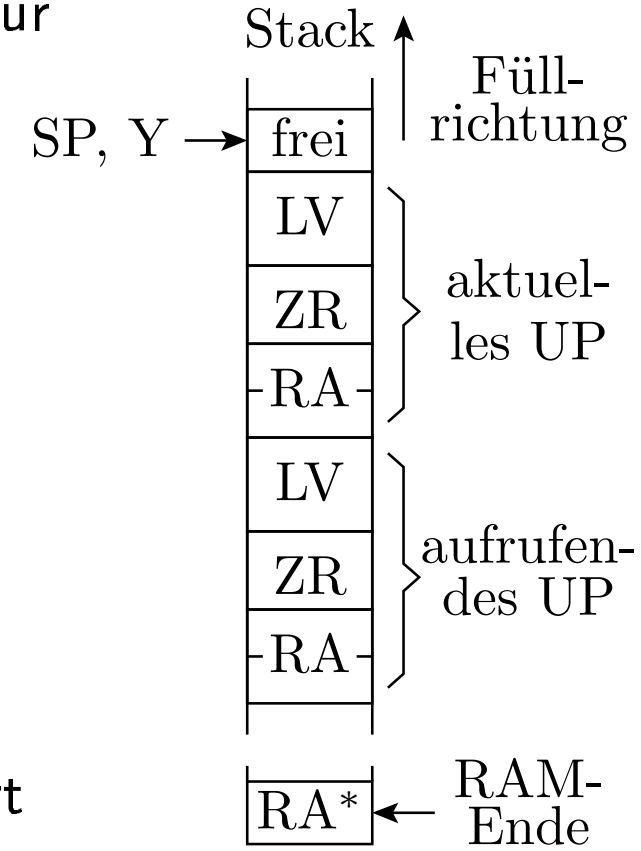
Name	Value	Type	Watch 1
g1	0xc8	uint8_t{data}@0x0200	
g2	0x42	uint8_t{data}@0x0201	

Hier ist der Frame-Pointer Y gemeint.

[2] Ab -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;
12   void main(void){
13     uint8_t l1 = 0x83;
14     uint8_t l2 = 0x45;
15     uint8_t l3 = 0x7A;
16     g1 = l1 + l2;
17   } g2 = g1 + l3;
```

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

```
00085  PUSH R28          | 1
00086  PUSH R29          |
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               |
```

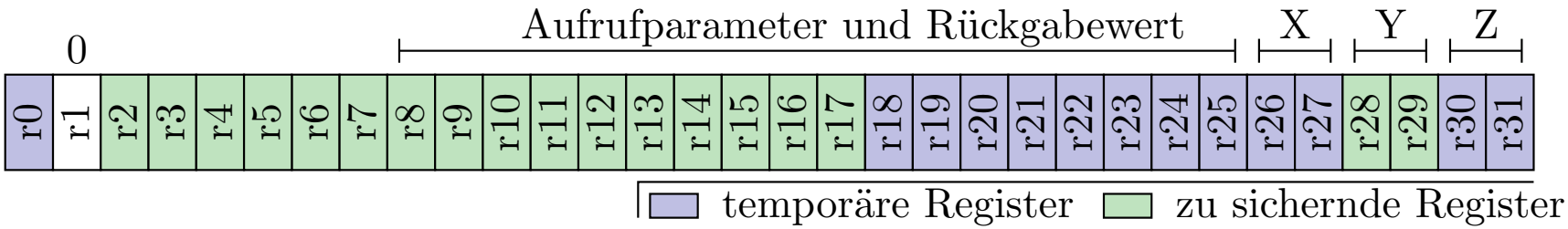
1. Sichern des Framepointers des aufrufenden Programms.
2. Der rcall-Befehl verringert den Stackpointer um 3. Das er dabei die Rückkehradresse 0x000088 auf den Stack schreibt, stört nicht, weil dieser Wert nie gelesen wird.
3. Zuweisen des neuen Stackpointer-Wertes an den Framepointer. Danach haben die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

Variable	l1	l2	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 Y (r29:r28) müssen auch die anderen vom aufrufenden Programm genutzten Register vor Änderung durch das aufgerufene Programm auf den Stack gesichert werden.



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

- In r1 wird bei UP-Aufruf der Wert null erwartet.
- r0, r18 bis r27, r30 und r31 (incl. X und Z): Temporäre Register, die das aufgerufene Unterprogramm verändern darf. (Sicherung vor Aufruf.)
- r2 bis r17, r28 und r29 (incl. Y): Vor Veränderung zu sichernde und vor Rücksprung wiederherstellende Register.

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

Das folgende mit -O1 übersetzte Hauptprogramm hält die sichtbaren Variablen, im Bild a bis c, in Registern. Der Variablen d wird erst nach Zeile 15 und e nach Zeile 16 ein Register zugeordnet.

```
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;
11 void main(void){
12     uint8_t a = PINA;
13     uint8_t b = PINB;
14     uint8_t c = PINC;

16     uint8_t e = a - c;
15     uint8_t d = a + b;
17     g = d | e;

   }
```

Adresse: 0x200

```
00085  IN R24,0x00      a
00086  IN R18,0x03      b
00087  IN R25,0x06      c
00088  MOV R19,R24
00089  SUB R19,R25      e
0008A  ADD R24,R18      d
0008B  OR R24,R19       d
0008C  STS 0x0200,R24
0008E  RET              g
```

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

```
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

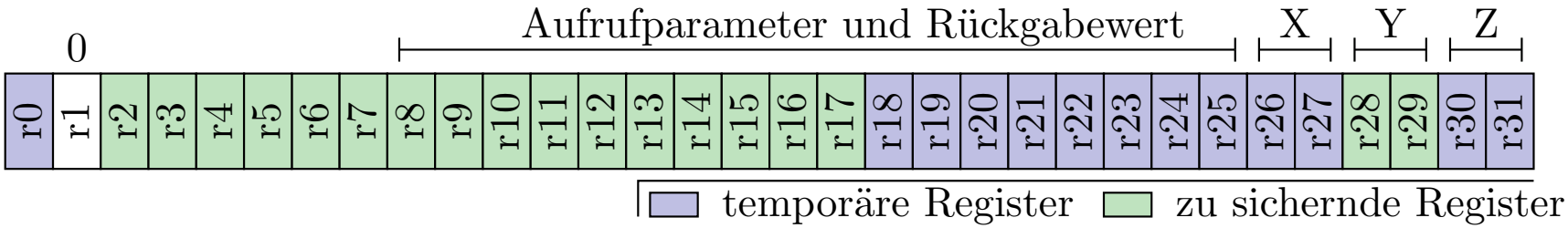
Die r17, r29 und r17 werden am Anfang von main() zusätzlich auf den Stack gesichert und am Ende von main() wieder vom Stack geholt.

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

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

Parameterübergabe

Registernutzung und Parameterübergabe



- Von rechts beginnen werden die ersten 18 Aufrufparameterbytes in den Registern r25:r8 und alle weiteren auf dem Stack übergeben. 1-Byte Parameter nutzen nur jedes zweite Byte.
- Die Rückgabe erfolgt in den Registern r25:r8.

```
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  }
```

Inhalt von d aus r23:r22 in die Rückgaberegister r25:r24 kopieren

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

Übergabewerte für a und b schreiben

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

Subtraktion 0xFFFC=-4

Inhalt von e aus r19:r18 in die Rückgaberegister r25:r24 kopieren

Aufgaben

Aufgabe 3.1: Sprungbedingung

Welche Statusbits 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 Then)
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.

Lösung

Die Lösung steht in der Tabelle auf Folie/Seite 22:

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

1. brlt : S=0, signed
2. brpl: N=0, signed
3. brlo: C=1, unsigned

Aufgabe 3.2: Reengineering If-Anweisung

Das Assemblerprogramm:

```
        LDS R25,0x201;
        LDS R24,0x200; Sprung-|Aufgabenteil
        CP R24,R25    ; befehl |   a)   |   b)
        xxxx PC+3     ;  xxxx: | BRCC | BRLT
        LDI R23,0x03 ;
        OUT 0, R23
    M1:...
```

wurde aus folgender C-Anweisungsfolge generiert:

```
(u)int8_t a; // Adresse 0x201, Typ gesucht
(u)int8_t b; // Adresse 0x200, Typ gesucht
if (a ?? b) PORTA = 0x3;// ?? Vergleichoperator
```

Bestimmen Sie jeweils den Typ der Operanden a und b und den Vergleichsoperator »??«, den das Assemblerprogramm nachbildet.

Lösung

1. »brcc« (Branch if Carry Cleared, uint): $b - a \geq 0 \Rightarrow a \leq b$:

```
    LDS R25,0x201; uint8_t a; r25 := a
    LDS R24,0x200; uint8_t b; r24 := b
    CP  R24,R25   ; teste r24-r25 (b-a)
    BRCC PC+x03   ; springe, wenn b-a≥0
    LDI R23,0x03  ; Ausführung wenn:
    OUT 0, R23    ; if (a<=b) PORTA = 0x3;
M1:...
```

2. »brlt« (Branch if Less Then, int): $b - a < 0 \Rightarrow a > b$:

```
    LDS R25,0x201; int8_t a; r25 := a
    LDS R24,0x200; int8_t b; r24 := b
    CP  R24,R25   ; teste r24-r25 (b-a)
    BRLT PC+x03   ; springe, wenn b-a<0
    LDI R23,0x03  ; Ausführung wenn:
    OUT 0, R23    ; if (a>b) PORTA = 0x3;
M1:...
```

Aufgabe 3.3: Reengineering Switch-Anweisung

Ergänzen Sie in dem nachfolgenden C-Programm die fehlenden Konstanten K1 bis K6 anhand des zugehörigen Assembler-Programms, 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
```

Lösung

```
     in R24,0x00   ; r24 := PINA
     cpi r24,0x15  ; ?: r24-0x15
     breq PC+0x05  ; wenn 0, springe zu M1
     cpi r24,0x38  ; ?: r24-0x38
     breq PC+0x06  ; wenn 0, springe zu M2
     cpi r24,0x12  ; ?: r24-0x12
     brne PC+0x07  ; wenn nicht 0, springe zu M3
M1:  ldi r24,0x27  ; für PINA=0x15 oder PINA=0x12
     out 0x08,r24  ; PORTC := 0x27
     ret
M2:  ldi r24,0x44  ; für PINA=0x38
     out 0x08,r24  ; PORTC := 0x44
     ret
M3:  ldi r24,0x22  ; sonst
     out 0x08,r24  ; PORTC := 0x22
     ret
```

- für PINA=0x15 oder PINA=0x12: PORTC:=0x27
- für PINA=0x38: PORTC:=0x44
- sonst: PORTC:=0x22

```
11 void main(){
12    switch (PINA) {
13      case 0x12:
14      case 0x15: PORTC = 0x27; break;
15      case 0x38: PORTC = 0x44; break;
16      default:   PORTC = 0x22;
17    }
18 }
```

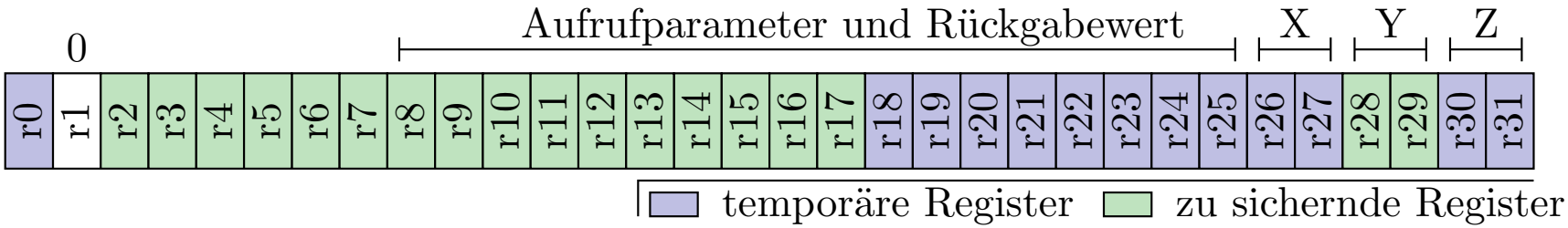
Aufgabe 3.4: Übergaberegister

Das nachfolgende in einer Header-Datei vereinbarte Unterprogramm:

```
uint16_t 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?

Lösung



- Von rechts beginnen werden die ersten 18 Aufrufparameterbytes in den Registern r25:r8 übergeben.
- 1-Byte Parameter nutzen nur jedes zweite Byte.
- Die Rückgabe erfolgt in den Registern r25:r8.

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

- Übergabe: a: r24, b: r23:r22, c: r20
- Rückgabe: r25:r24

Aufgabe 3.5: Schleife mit Fehler

Das nachfolgende C-Programm enthält eine while-Schleife, in der die Variable a solange um 1 erhöht wird, wie ihr Wert kleiner 256 ist. 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
```

4. Aufgaben

1. Warum wird das Programm mit -O1 in eine Endlosschleife übersetzt, die nichts tut?
2. Verhält sich das mit -O0 übersetzte Programm anders?
3. Wie ist das C-Programm zu verändern, damit die Schleife abbricht, wenn a nicht mehr kleiner als 256 ist?
4. Wie viele Bytes werden bei Compileroptimierung -O0 beim Aufruf von main() auf dem Stack reserviert und welchen Wert hat der Stackpointer innerhalb von main() nach Einrichtung des Stackframes?

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

Lösung

1. Endlosschleife, da a vom Typ uint8_t immer kleiner 256 und damit die Wiederholbedingung immer wahr ist.
2. Mit -O0 auch Endlosschleife:

```
      push r28       ; Frampointer auf Stack
      push r29       ;
      push r1        ; Platz für lokale Var. a
      in r28,0x3D    ; Frampointer := Stackpointer
      in r29,0x3E    ;
M1:   ldd r24,Y+1    ; r24 := a
      subi r24,0xFF  ; r24 :=r24+1
      std Y+1,r24    ; a   := r24
      rjmp PC-3      ; springe zu M1
      ret
```

3. Vergrößerung des Schleifenzählers auf 16 Bit.
4. 6 Bytes, Stackpointer 0x21FF-6 = 0x21F9