Rechnerarchitektur, Foliensatz 2

G. Kemnitz

Institut für Informatik, TU Clausthal (RA-F2.pdf)

13. Dezember 2016

Zusatzspeicher

Datenspeicher ATmega2560

- 32 Arbeitsregister,
- 480 EA- / SF-Register[1]
- 2 kByte interner Datenspeicher,

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

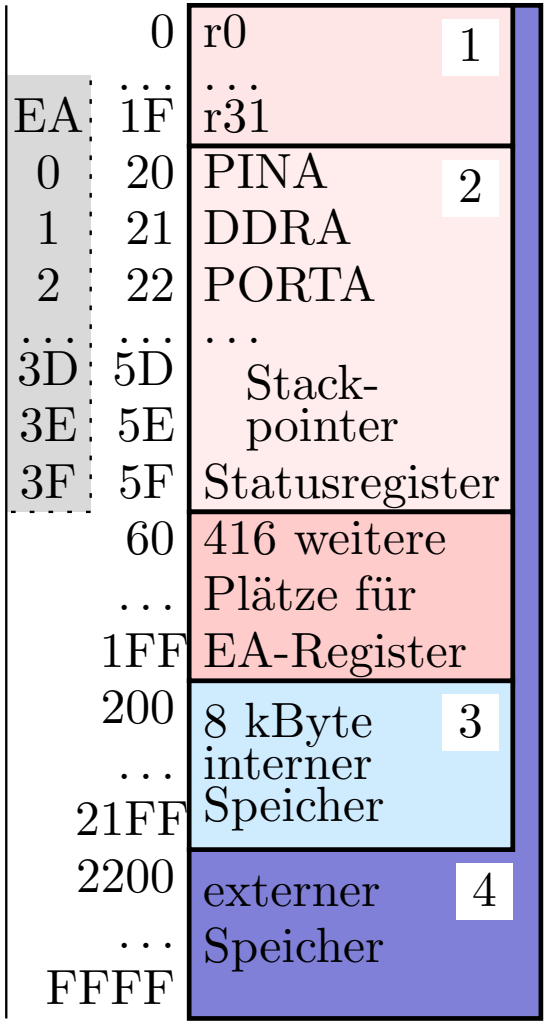
```
lds Rd, k; sts k, Rd
```

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

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

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

```
in Rd, A ; Rd←I/O(A) (Eingabe)
out A, Rd; I/O(A)←Rr (Ausgabe)
```

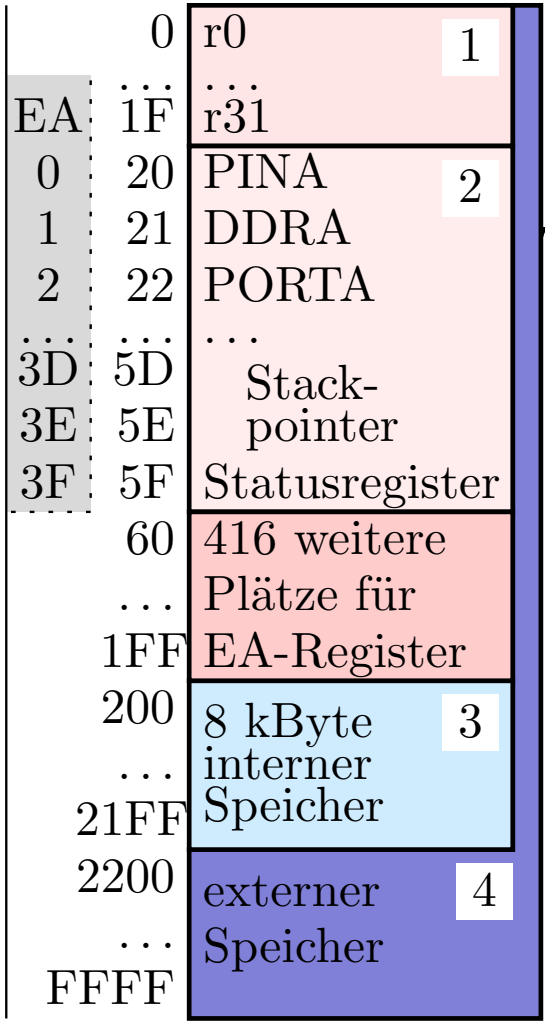


[1] SFR – **S**pezial **F**unction **R**egister.

Weitere Speicher

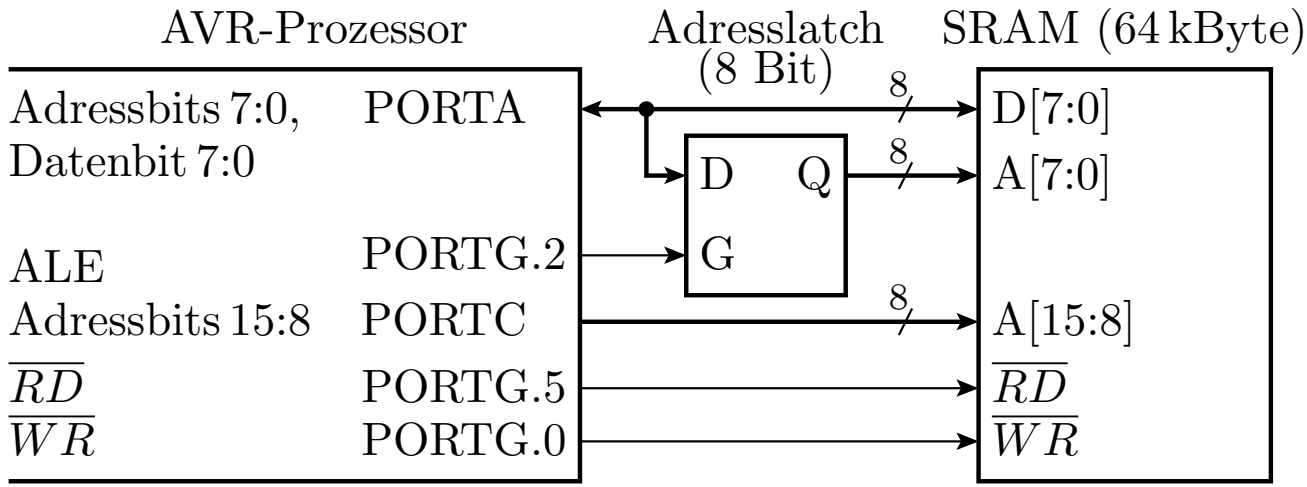
- Möglichkeit, einen bis zu 64-kByte großen externen Speicher anzuschließen.
- Im Adressbereich 0 bis 0x21FF wählt ein SFR-Bit zwischen interem und externem Speicher.
- Programm-Flash: Konstanten.
- EEPROM: Nicht flüchtige Daten[a].

[a] Daten, die auch nach Abschalten der Versorgungsspannung aufzubewahren sind.



Externer Speicher

Anschluss des externen Speichers



ALE – Adress Latch Enable

- Port A: Takt 1 Adressbyte 0, Takt 2 Lese- bzw. Schreibdaten.
- Port C: höherwertiges Adressbyte.
- Port G: Steuersignale.

Bei angeschlossenem externen Speicher: PG5=1 oder DDRA=0xFF[2].

[2]Durch falsche Ansteuerung sind hier im Übungsraum schon mehrere Versuchsbaugruppen kaputt gegangen.

Konstanten im Befehls-Flash

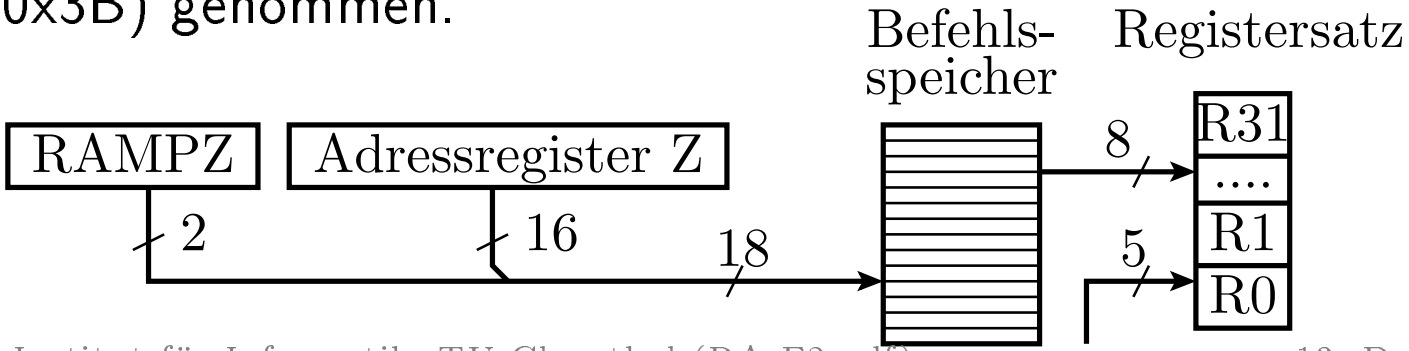
Speichern von Konstanten

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

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

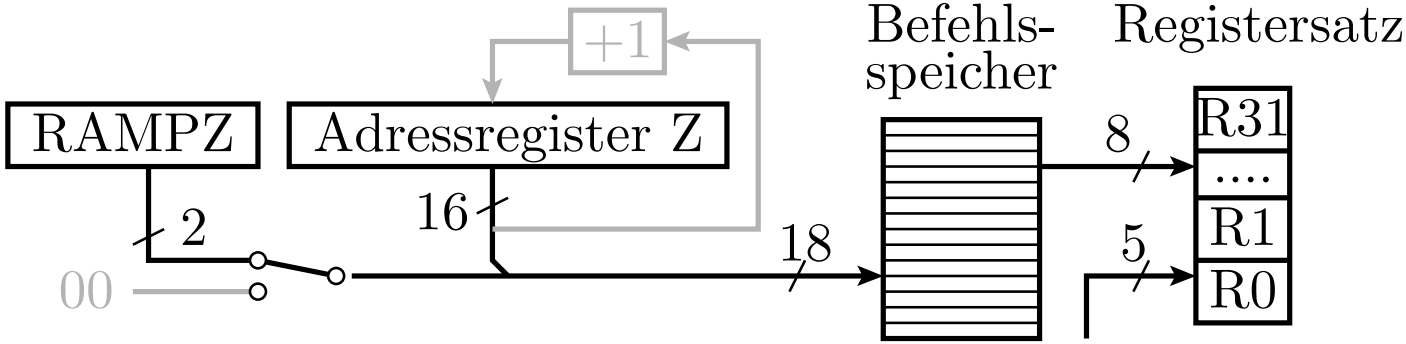
werden im Programmspeicher abgelegt und beim Programmstart in den Datenspeicher kopiert.

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



Befehlsvariationen:

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



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

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

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

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

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

Der Disassembler kann Zeichenketten nicht von Programmcode unterscheiden.

Die nachfolgende Befehlsfolge steht im Start-Up Code vor »main()«. Der Befehlsspeicher wird mit Z und der Datenspeicher mit X jeweils mit Post-Inc. adressiert. Schleifenabruch bei X=0x20C:

```
00007A  LDI R17,0x02
00007B  LDI R26,0x00
00007C  LDI R27,0x02
00007D  LDI R30,0x42
00007E  LDI R31,0x01
00007F  LDI R16,0x00
000080  OUT 0x3B,R16
000081  RJMP PC+0x0003
000082  ELPM R0,Z+
000083  ST X+,R0
000084  CPI R26,0x0C
000085  CPC R27,R17
000086  BRNE PC-0x04
```

$r17 \leftarrow 2$

$X \leftarrow 0x200$

$Z \leftarrow 0x142$

$RAMPZ \leftarrow 0$

$r0 \leftarrow PMem(Z); Z \leftarrow Z+1$

$DMem(X) \leftarrow r0; X \leftarrow X+1$

Teste, ob $X \neq 0x20C$ ist. Wenn ja, springe zu Adresse 0x82

Das mit

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

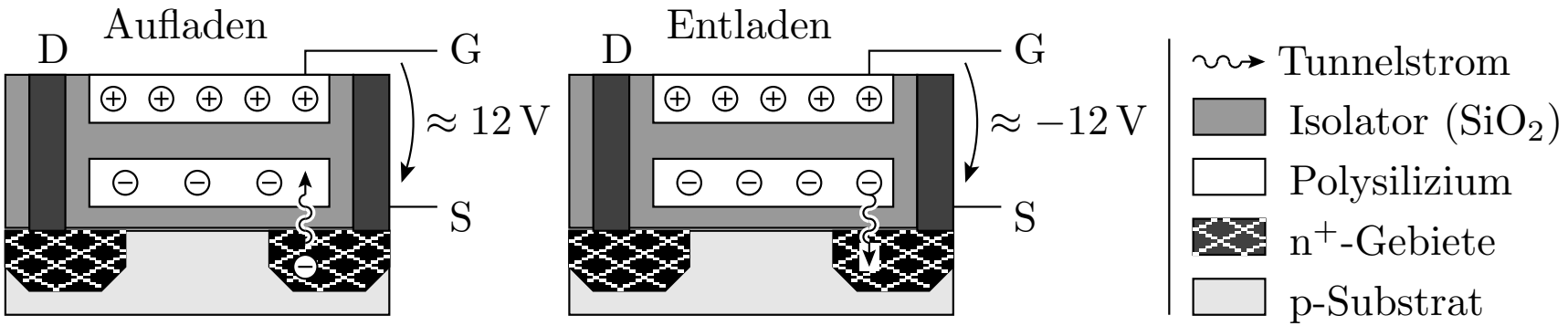
vereinbarte Feld, das der Compiler ab Adresse 0x20C platziert hat, wird mit Nullen initialisiert:

```
//0x0087  LDI R18,0x02    ; r18 = 2
//0x0088  LDI R26,0x0C    ; r27:r26 = 0x20C  (X=&b)
//0x0089  LDI R27,0x02    ;
//0x008A  RJMP PC+0x0002  ; spring zu 0x8C
//0x008B  ST X+,R1        ; dmem(X) = 0; X++
//0x008C  CPI R26,0x16    ; teste r27:r26 - 0x216
//0x008D  CPC R27,R18     ;
//0x008E  BRNE PC-0x03    ; wenn <0, springe zu 0x8B
//0x008F  RCALL PC+0x0003; Aufruf von main()
//0x0090  RJMP PC+0x0009  ; Sprung hinter main()
//0x0091  RJMP PC-0x0091  ; Neustart
int main(void){
//0x0092  LDS R24,0x0200
```

EEPROM

EEPROM

- Der Datenspeicher verliert ohne Spannung die gespeicherten Werte.
- Konstanten stehen deshalb im Befehlsspeicher und werden vom Startup-Code in den RAM kopiert.
- Wo werden während des Betriebs anfallende Daten gespeichert, die nach Abschalten nicht verloren gehen dürfen? ⇒ EEPROM (**E**lectrically **E**rasable and **P**rogrammable **R**ead-**O**nly **M**emory).
- Programmierung über Tunnelströme. Schreibzeit 10^4 bis 10^5 Prozessortakte.



EEPROM-Ansteuerung

EEPROM-Adressregister	–	–	–	–	Bit 11	Bit 10	Bit 9	Bit 8	EEARH (0x42)
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	EEARL (0x41)
Datenregister	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	EEDR (0x40)
Kontrollregister	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	EECR (0x40)

	EEPM1	EEPM0
löschen+schreiben	0	0
nur löschen	0	1
nur schreiben	1	0

Bits zur Aktivierung und Steueung der Programmierung

Ablauf zum Schreiben eines Bytes in den EEPROM:

1. Warte bis EEPE null ist[3].
2. Schreibe die Adresse in EEAR und die Daten in EEDR.
3. Schreibe EEMPE$\leftarrow 1$ und EEPE$\leftarrow 0$.
4. Innerhalb der nächsten 4 Zyklen schreibe auch EEPE$\leftarrow 1$.

[3] Warte, bis die vorherige EEPROM-Schreiboperation abgeschlossen ist. Für Schreiboperation ca. 3.3 ms nach letztem Setzen des Bits EEPE. Bei zeitgleicher Flash-Op. zusätzlich warten, bis SPMEN in SPMCSR null ist.

Schreiboperation als C-Funktion:

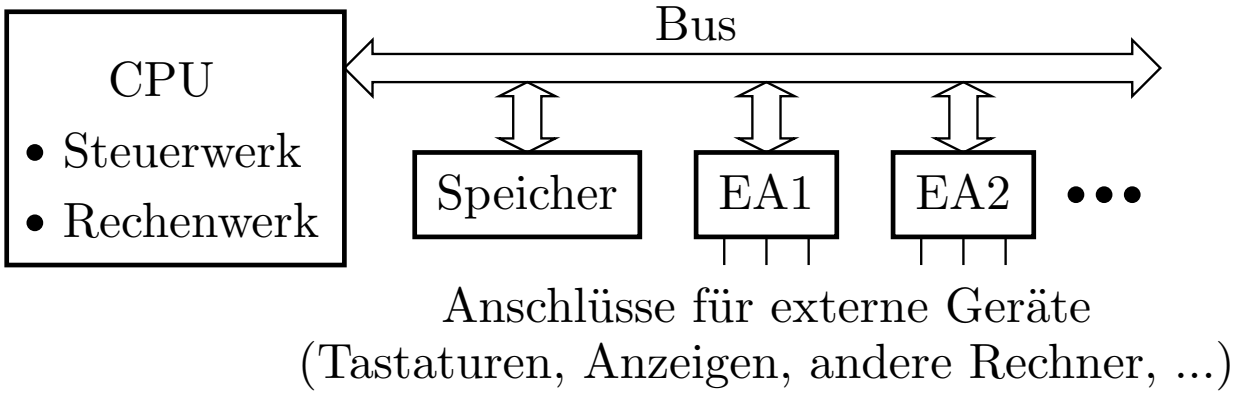
```
void EEPROM_write(uint16_t uiAddress, uint8_t ucData)
  // Warte auf Abschluss letzte Schreiboperation
  while(EECR & (1<<EEPE));
  EEAR = uiAddress;  // Adressübergabe
  EEDR = ucData;     // Datenübergabe
  EECR |= (1<<EEMPE);
  EECR |= (1<<EEPE); // Start der Schreiboperation
}
```

Die Leseoperation muss auf den Abschluss der Schreiboperation warten und ist einen Takt nach Start der Leseoperation fertig:

```
uint8_t EEPROM_read(uint16_t uiAddress) {
 // Warte auf Abschluss letzte Schreiboperation
  while(EECR & (1<<EEPE));
  EEAR = uiAddress;  // Adressübergabe
  EECR |= (1<<EERE); // Start der Leseoperation
  return EEDR;       // Ergebnisrückgabe
}
```

Ports

Prinzip der Ein- und Ausgabe



Ein Prozessor kommuniziert mit seiner Umgebung

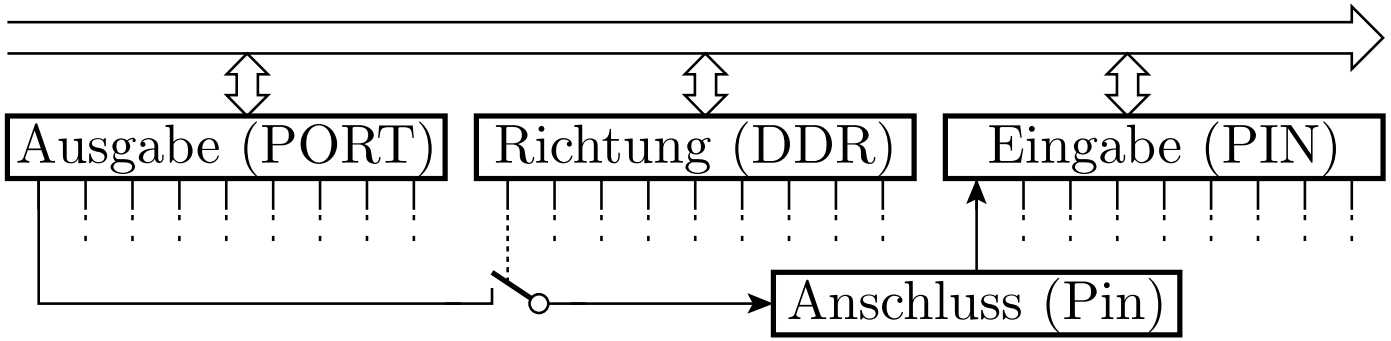
- Benutzer, Sensoren, Aktoren,
- getrennten Werken (Timer, Watchdog, ...),
- anderen Rechnern, ...

über EA-Register.

EA-Register sind für die Umgebung im einfachsten Fall binäre Ausgabesignale z.B. zum Schalten von Anzeigen, Motoren und Eingabesignale, z.B. zum Lesen von Schalter und Sensorwerte.

Ports des ATmega2560

Ports des ATmega2560



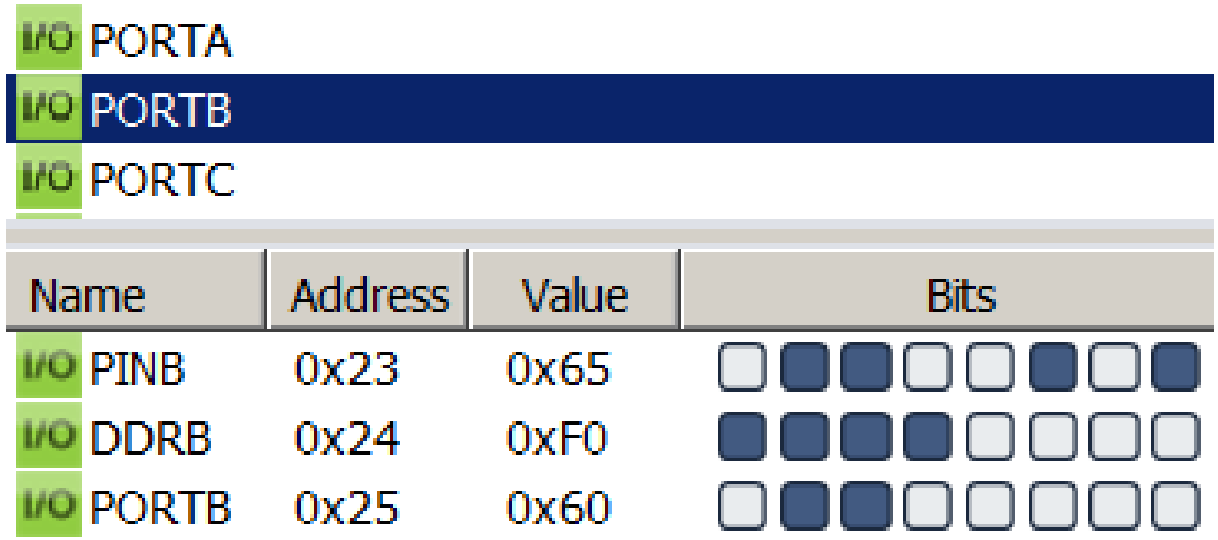
- Ports (Parallel Schnittstellen) sind 8-Bit-, bei größeren Prozessoren auch 16- oder 32-Bit-IO-Register mit anschließbaren Leitungen z.B. für Schalter und LEDs.
- Universelle Ports können bitweise als Eingänge, Ausgänge oder mit umschaltbarer Übertragungsrichtung genutzt werden.
- Bei AVR-Prozessoren gehören zu jedem Port 3 Register mit aufeinanderfolgenden Adressen, eines für die Ausgabe, eines für die Eingabe und eines für die Steuerung der Übertragungsrichtung.

Port-Adressen und Darstellung im Debugger

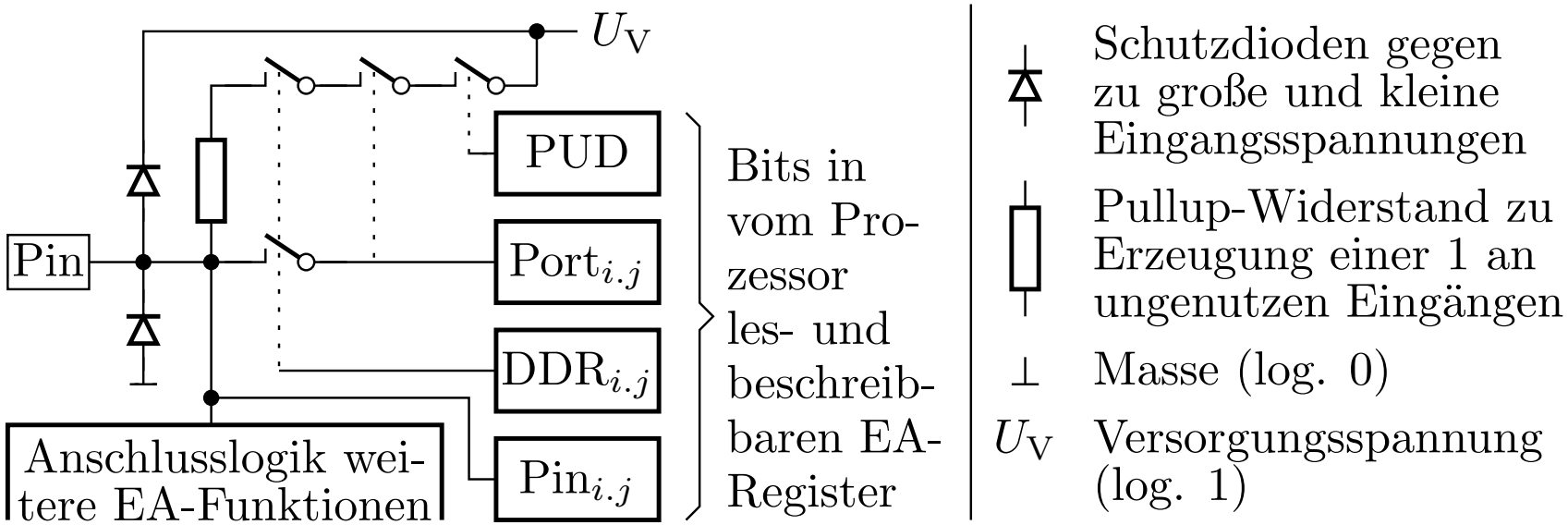
Der ATmega2560 hat 12 Ports mit Richtungsregister DDRx, Ausgaberegister PORTx und Eingaberegister PINx ($x \in \{\mathrm{A}, \mathrm{B}, ..., \mathrm{L}\}$; 0/0x20 – IO-Adresse / Datenspeicheradresse).

	A	B	C	D	E	•
PIN	0 / 0x20	3 / 0x23	6 / 0x26	9 / 0x29	0xC / 0x2C	•
DDR	1 / 0x21	4 / 0x24	7 / 0x27	0xA / 0x2A	0xD / 0x2D	•
PORT	2 / 0x22	5 / 0x25	8 / 0x28	0xB / 0x2B	0xE / 0x2E	•

Debug-Ansicht:

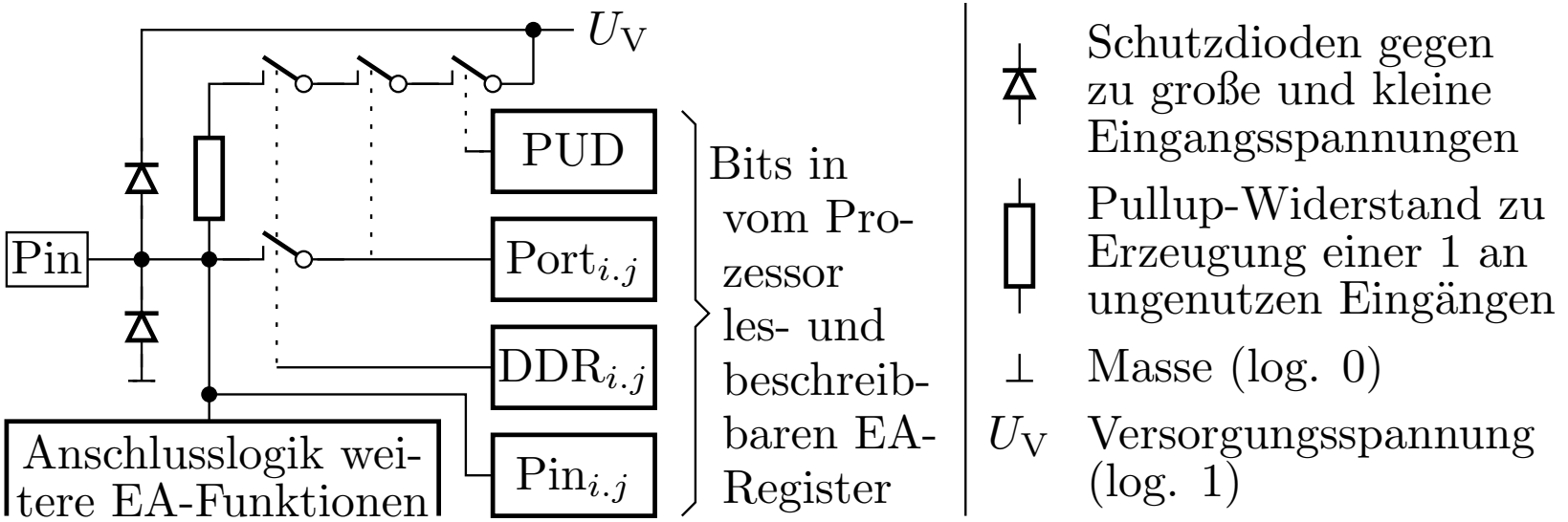


Beschaltung und Nutzung eines einzelnen IO-Pins



Nutzung als Ausgang:

- Richtungsbit eins setzen: $DDR_{i.j} \leftarrow 1$:
- Ausgabe: $PORT_{i.j} \leftarrow$Rd (Rd – Arbeitsregister)
- Rücklesen des Ausgabewertes: Rd$\leftarrow PORT_{i.j}$
- Eingabe: Rd$\leftarrow PIN_{i.j}$. $PIN_{i.j} \neq PORT_{i.j}$ ist möglich und deutet auf Programmier- oder Schaltungsfehler.



Nutzung als Eingang:

- Richtungsbit null setzen: $DDR_{i.j} \leftarrow 0$:
- Werte zwischen 0 und 1, z.B. bei ungenutzten Anschlüssen verursachen erhöhte Stromaufnahme.
- Ausgabewert eins ($PORT_{i.j} \leftarrow 1$) und SFR-Bit »PUD« nicht gesetzt, zieht ungenutzte Eingänge über einen Widerstand auf eins. Zu empfehlen für alle ungenutzten Eingänge.
- Bei externer Signalquelle und vor allem für analoge Eingänge Pullup-Widerstand mit ($PORT_{i.j} \leftarrow 0$) deaktivieren.

Polling und Interrupt

Zur Abstimmung der Ein- und Ausgabezeitpunkte muss ein EA-Gerät warten, bis der Rechner und der Rechner bis das EA-Gerät bereit ist. Dafür gibt es zwei Prinzipien:

- Polling: Zyklische Abfrage aller EA-Geräte durch den Rechner, ob Datenübergabe angefordert oder zur Übernahme bereit. Wenn ja, Verzweigung zum Programmbaustein für den Datenaustausch.
- Interrupt: Gerät fordert Datenaustausch an. Rechner ruft, sobald er dafür bereit ist, eine **I**nterrupt-**S**ervice-**R**outine (ISR) auf.

Interrupts können im Programm global und lokal (für jede Interrupt-Quelle einzeln) gesperrt oder freigegeben werden.

Nach Freigabe kann das externe Gerät das Rechnerprogramm nach jedem Maschinenbefehl unterbrechen.

Polling

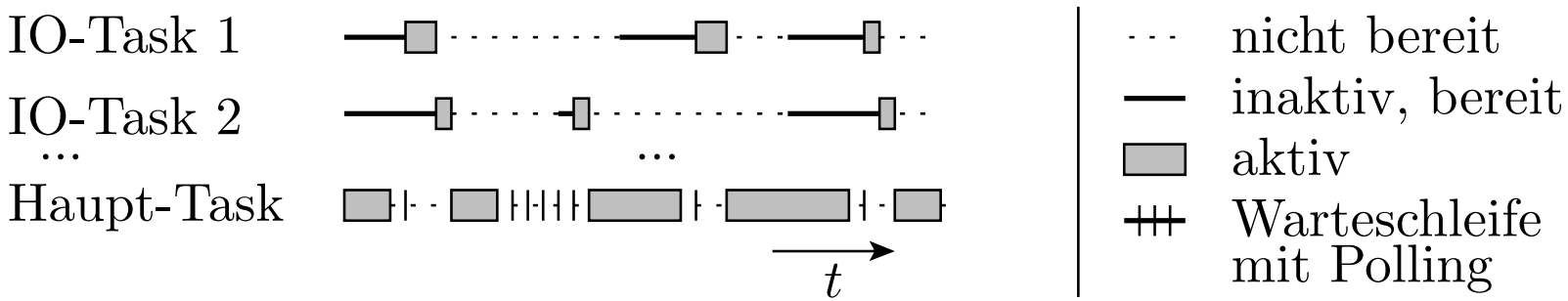
Bedienung von EA-Anforderungen mit Polling

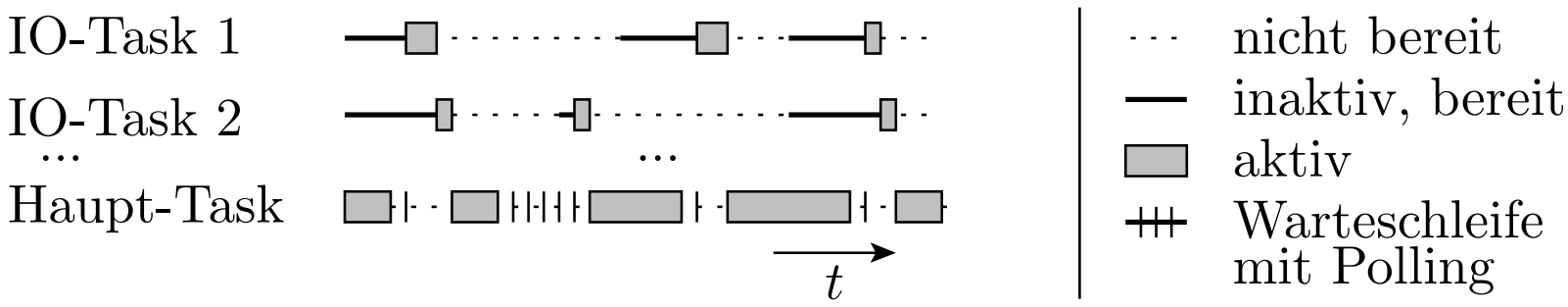
Jede EA-Einheit hat

- ein Ereignisbit zur Signalisierung der Bereitschaft,
- eine Obergrenze für die Reaktionszeit vom Setzen des Ereignisbits bis zum Datenaustausch,

in der der Prozessor hundete bis tausende Befehle abarbeiten kann.
Programme nutzen meist die Dienste mehrere EA-Einheiten.

Software-Ablauf mit einem Haupt-Task und mehreren IO-Tasks zur Bedienung von EA-Einheiten über Polling:





- Der Haupt-Task arbeitet einige tausend Befehle ab und kehrt zur Haupt- (Ereignisabfrage-) Schleife zurück, z.B. wenn auf EA-Operationen gewartet werden muss.
- Die Hauptschleife fragt zyklisch alle Ereignis- (Bereitschafts-) Bits der aktivierten IO-Tasks und des Haupt-Tasks ab.
- Wenn einer bereit, Verzweigung zur Task-Abarbeitung.
- Am Ende der Task-Abarbeitung löscht der Task das eigene und setzt die Ereignisbit der Tasks, die auf diesen gewartet haben + Rückkehr zur Hauptschleife.

Polling funktioniert gut für EA-Operationen mit großen zulässigen Reaktionszeiten.

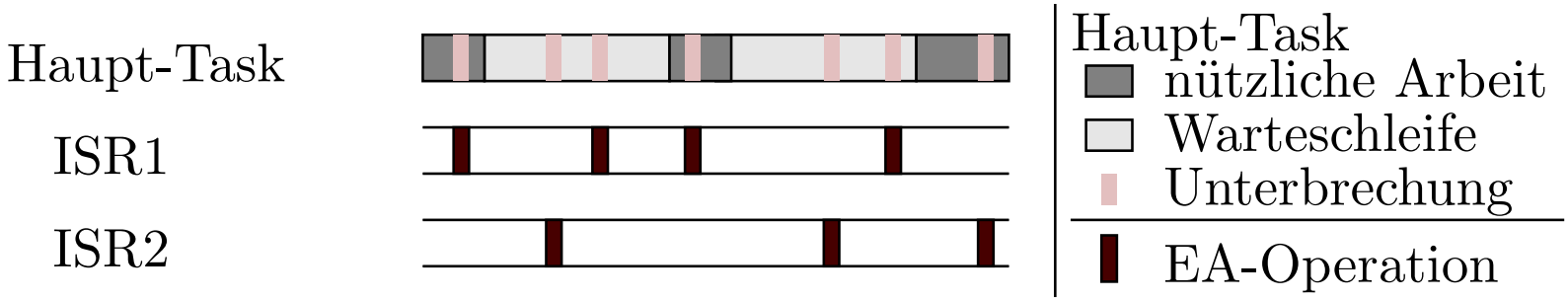
Programmstruktur für EA mit Polling

```
int main(){
  <Initialisierung, Variablen, ...>
  while(1){                  //Endlosschleife
   if (<Haupt-Task bereit>)
     {<Haupt-Task weiter abarbeiten>}
   if (<IO-Task 1 bereit>)
     {<IO-Task 1 abarbeiten>}
   if (<IO-Task 2 bereit>)
     {<IO-Task 2 abarbeiten>}
   ...
  }
}
```

- Der Haupt-Task muss sich nach hinreichend kurzer Zeit für mindesten einen IO-Abfragezyklus unterbrechen.
- IO-Tasks max. wenige hundert abzuarbeitende Befehle.
- Keine Warteschleifen außer der Endlosschleife.

Interrupt

Interrupt-Service-Routinen (ISR)



Haupt-Task:

- Ohne zyklische Abfrage von Ereignisbits.
- Lokale (individuelle) und globale Interrupt-Freigabe.

Interrupt-Behandlung:

- Aufruf der ISR auf einer feste Hardware-Adresse.
- Interrupt sperren. Inhalte genutzter Register incl. Statusregister, Frame-Pointer, ... retten.
- EA-Operation ausführen
- Register wiederherstellen. Interrupt-Freigabe. Rücksprung.

Interrupt-Freigabe, Ereignisbits, ... ATmega 2560

Globale Interrupt-Freigabe: Bit »I« (SREG.7, EA-Adresse 0xFE):

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

C-Anweisungen für des Setzen und löschen von »I«:

```
sei();          // Interrupts global ein
cli();          // Interrupts global aus
```

Adresse und Konfigurationsbits externer Interrupts an Port B:

Interrupt	Adresse*1	Ereignisbit	Freigabebit	weiter Einstellung*2
INT0 (PB0)	0x0002	EIFR.0	EIMSK.0	EICRA[1:0]
INT1 (PB1)	0x0004	EIFR.1	EIMSK.1	EICRA[3:2]
...	...	...	...	...
INT7 (PB7)	0x0010	EIFR.7	EIMSK.7	EICRB[7:6]

*1 Startadresse der Interrupt-Service-Routine (ISR)

*2 Interrupt bei null, steigender und/oder fallender Flanke.

Interrupt-Quellen des ATmega 2560

Ingesamt 57 (ATmega2560.pdf[4], Abschn. 14.1 Interrupts):

- 8× für Port B Anschlüsse. Bei Port B als Eingang auch als Software-Interrupts nutzbar.
- 3× Port-Change-Interrupts für programmierbare Bitänderungen auch an anderen Ports.
- 1× Watchdog (Timeout).
- 26× für Timer-Funktionen.
- 4×3 für die 4 universellen seriellen Schnittstellen (USARTs).
- 1× SPI (serielle Übertragung abgeschlossen).
- 1× Analogkomparator.
- 1× ADC (Digital-Analog-Wandlung abgeschlossen).
- 1× TWI (2-wire Serial Interface).
- 1× EEPROM (Schreiboperation abgeschlossen), ...

[4] http://techwww.in.tu-clausthal.de/site/Lehre/Rechnerarchitektur...

Vor- und Nachteile von Interrupts

Vorteile:

- Schnellere Reaktion auf externe Ereignisse.
- Der Haupt-Task muss nicht aller paar tausend Befehle zur Hauptschleife zurückkehren, darf Warteschleifen enthalten.

Nachteile:

- Haupt-Task wird an zufälligen Programmstellen unterbrochen,
- dadurch kein deterministischer Ablauf.
- Viele zusätzliche Fehlermöglichkeiten.
- Erschwerter Test, erschwerte Fehlersuche.

Einige Regeln für Interrupt-Routinen:

- kurze, vorhersagbare Abarbeitungszeit, keine Warteschleifen.
- keine Änderung von Daten und Registerinhalten, die das unterbrochene Programm möglicherweise gerade bearbeitet.

Interrupt-Sperren

Ein unterbrechbares Programm muss die ISR immer sperren, während es Übergabedaten bearbeitet:

```
...
uint8_t tmp=<Interrupt-Freigaberegister>;
<Interrupt-Freigabebit löschen>
<Bearbeitung der Übergabedaten>
<Interrupt-Freigaberegister> = tmp;
```

Beispiel einer ISR:

```
#include <interrupts.h> // Header für Int.-Nutzung
#include <avr/io.h>
ISR(INT0_vect){         // Int. bei Tastendruck an PB0
  PORTJ^=1;             // PJ0 (LED 1) invertieren
}
```

INT0_vect – Startadresse der Interrupt-Service-Routine (ISR).

Beispiel für ein Hauptprogramm hierzu:

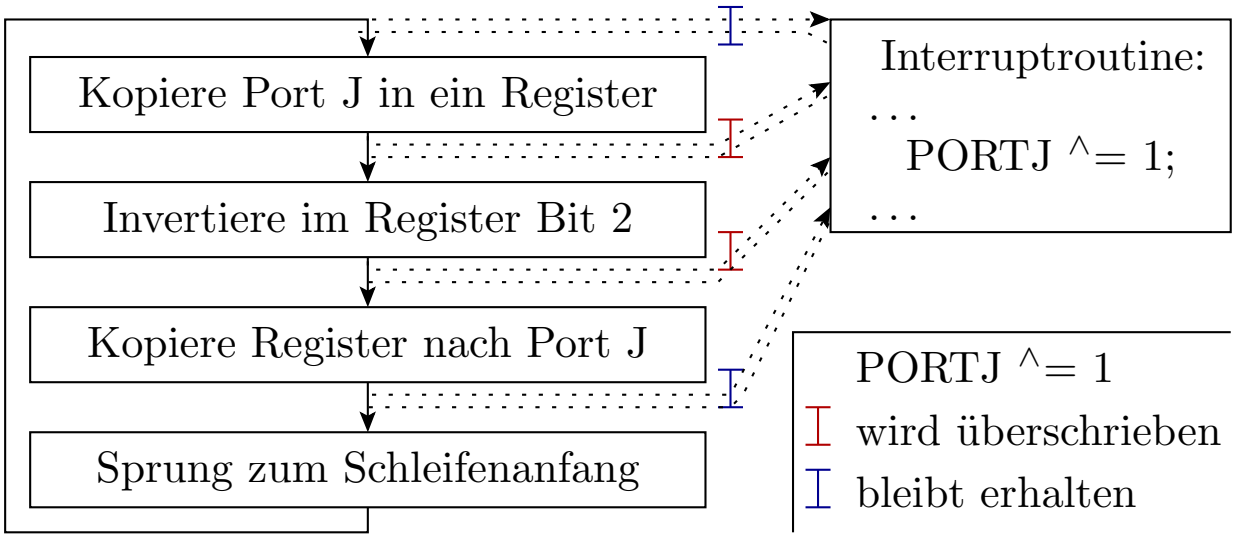
```
int main(){
  DDRB  = 0xFF;  // Port B Eingänge
  DDRJ  = 0;     // Port J Ausgänge
  EIMSK =1<<INT0;// Interrupt-Freigabe PB0
  EICRA = 0b11;  // Interrupt bei 01-Flanke
  sei();         // globale Interrupt-Freigabe
  while (1)      // Endlosschleife
   uint8_t tmp = EIMSK; // Int.-Freigabe speichern
   EIMSK &= ~1;  // INT0 an PB.0 sperren
   PORTJ^=2;     // PJ0 (LED 2) invertieren
   EIMSK=tmp;    // Int-Freigabe wiederherstellen
  }
}
```

Wenn INT0 nicht während »PORTJ^=2« gesperrt wird, wird die Hälfte der Anweisungen »PORTJ^=1« in der ISR bei Tastendruck nicht wirksam. Warum?

$\Rightarrow$ Bearbeitung derselben Daten (PORTJ).

Ohne die drei Anweisungen zur Interrupt-Sperre wird der Schleifenkörper in die Schrittfolge übersetzt:

- Port J lesen,
- Wert bearbeiten,
- Wert schreiben und
- Sprung zum Schleifenbeginn:



An 50% der Unterbrechungsmöglichkeiten wird die Invertierung von PJ0 in der ISR vom Rückschreibwert für die Invertierung von PJ1 überschrieben, d.h. Reaktion nur auf 50% Tastendrucke.

Timer

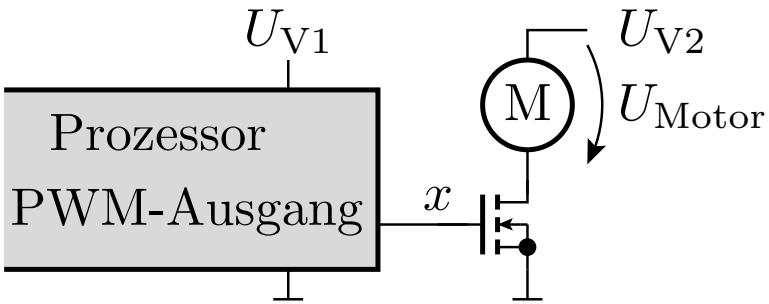
Timer

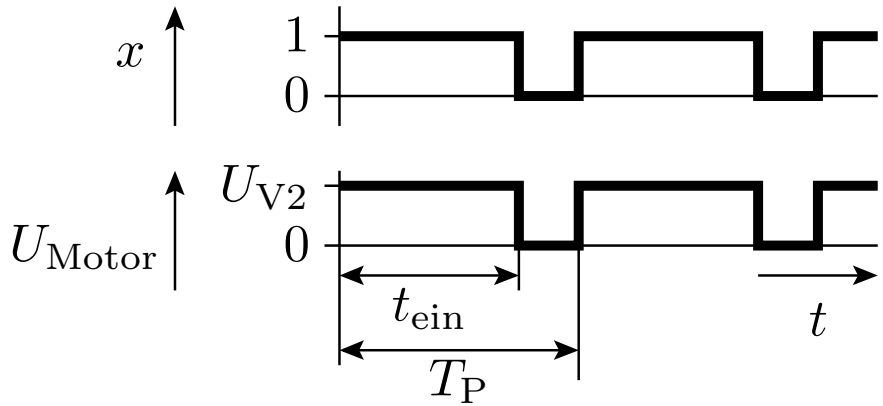
Ein Timer ist eine Hardware-Einheit aus Zähl-, Vergleichs-, Konfigurationsregistern, ... zur

- Erzeugung von Wartezeiten,
- zeitgesteuerten Ereignisabarbeitung,
- Erzeugung pulsweitenmodulierter (PWM-) Signale und
- Pulsweitenmessung.

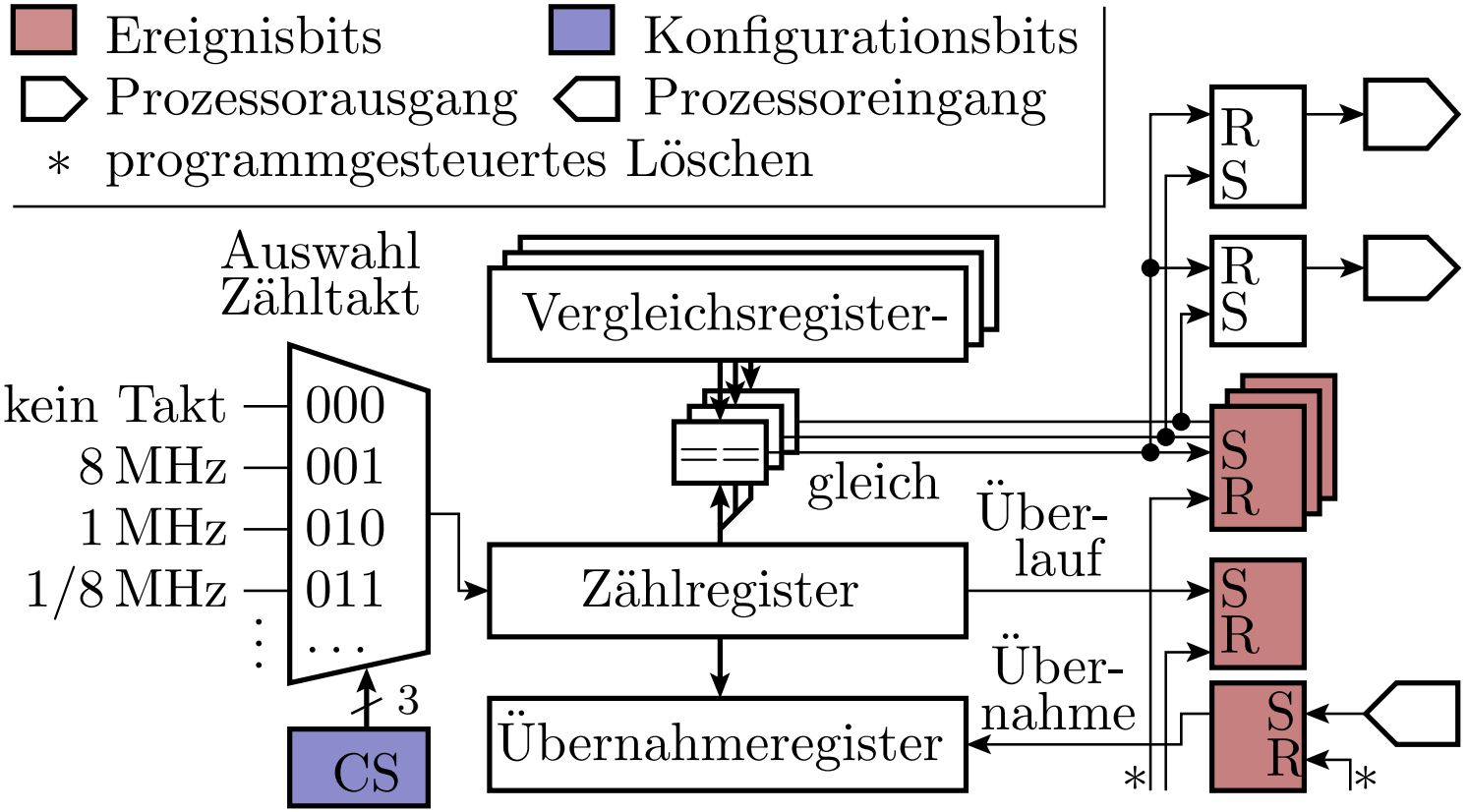
PWM-Signale dienen

- zur Informationsübertragung z.B. an Modellbauservos und
- zur stufenlosen Leistungssteuerung, z.B. von Elektromotoren.





Aufbau und Funktionsweise eines Timers



- Kern eines Timers ist ein Zählregister mit einem vom Programm zuschaltbaren programmierbaren Takt.
- Die Ereignisbits (Überlauf, Gleichheit, externe Flanke) sind vom Programm les- und löschbar.

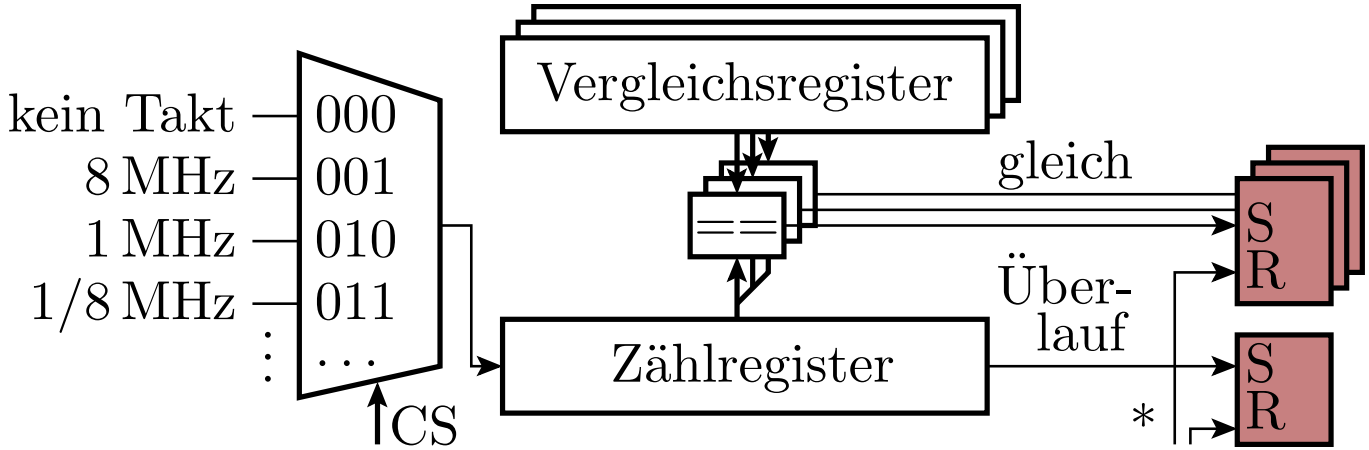
Timer des ATMega 2560

- Zwei 8-Bit Timer (Tmr0 und Tmr2).
- Vier 16-Bit-Timer (Tmr1, Tmr3, Tmr4 und Tmr5).

Die Bit-Anzahl beschreibt die Größe der Zähl- und Vergleichsregister.

Normalmodus, Zeitmessung

Normalmodus



- Zählregister zählt zyklisch bis zum Überlauf.
- Beim Überlauf wird eine Überlaufbit und bei Gleichheit mit einem Vergleichsregister ein Gleichheitsbit gesetzt.
- Beispiel Wartefunktion:

```
void wait(uint32_t tw){
 <berechne+setze Taktauswahl+Vergleichswert>
 <Lösche Zähler und Gleichheitsbit>
 <warte bis Gleichheitsbit wieder gesetzt ist>
 <schalte Zähltakt aus> }
```

Zeitmessung für Float-Operationen

Gleitkommaoperationen (+, *, ...) werden auf unserm Prozessor mit Unterprogrammen realisiert und dauern lange. Wie lange?

```
#include <avr/io.h>
float a=26.34516, b=1045.6734;
uint16_t t;
int main(void){
    TCNT1 = 0;    // Normalmodus, Zähltakt 8MHz
    TCCR1B = 1;   // WGM=0b0000, CS=0b001
    a = (a + b) * 22.9856;
    TCCR1B = 0;   // Zähltakt aus
    t = TCNT1;
}
```

Zählwert in t am Programmende: 317 Takte (Befehlszyklen)

- davon ca. 20 für den Unterprogammaufruf und
- je 150 für die Abarbeitung einer Gleitkommaoperation.

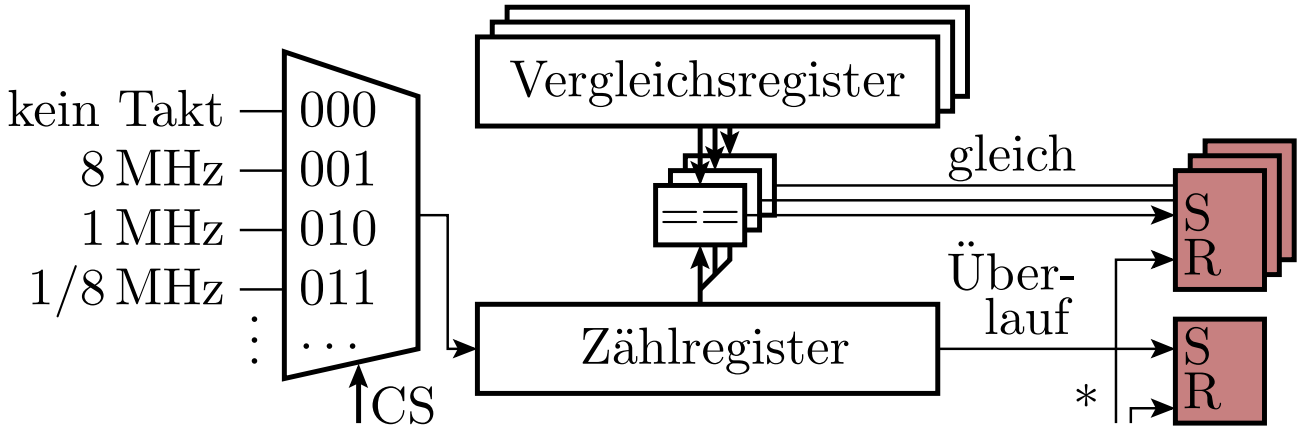
Disassemblierter Programmausschnitt

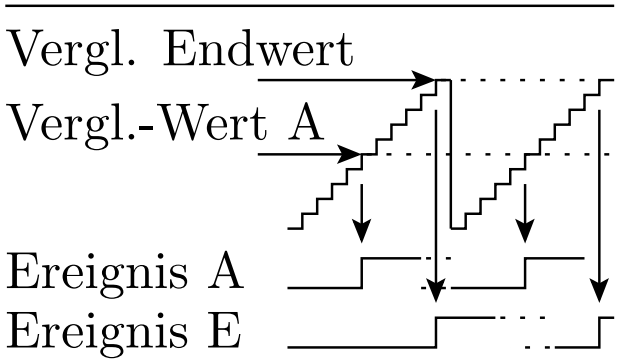
```
int main(void){
  TCNT1 = 0;
  TCCR1B = 1;  // WGM=0b0000, CS=0b001
  a = (a + b) * 22.9856;
// 0x009F  LDS R18,0x0200 ; r21:r20:r19:r18 = a
// 0x00A1  LDS R19,0x0201 ; 2. Befehl
// 0x00A3  LDS R20,0x0202 ; 3. Befehl
// 0x00A5  LDS R21,0x0203 ; 4. Befehl
// 0x00A7  LDS R22,0x0204 ; r25:r24:r23:r22 = b
// 0x00A9  LDS R23,0x0205 ; 2. Befehl
// 0x00AB  LDS R24,0x0206 ; 3. Befehl
// 0x00AD  LDS R25,0x0207 ; 4. Befehl
// 0x00AF  RCALL PC+0x001E; Gleitkommaaddition
// ... Konstante 22.9856 in r21:r20:r19:r18 laden
// ... Gleitkommamultiplikation
// ... a = r25:r24:r23:r22
```

```
// in r25:r24:r23:r22 steht a + b
// 0x00B0  LDI R18,0x82    ; r21:r20:r19:r18 = 22.9856
// 0x00B1  LDI R19,0xE2    ; 2. Befehl
// 0x00B2  LDI R20,0xB7    ; 3. Befehl
// 0x00B3  LDI R21,0x41    ; 4. Befehl
// 0x00B4  RCALL PC+0x00CE; Gleitkommamultiplikation
// 0x00B5  STS 0x0204,R22  ; a = r25:r24:r23:r22
// 0x00B7  STS 0x0205,R23  ; 2. Befehl
// 0x00B9  STS 0x0206,R24  ; 3. Befehl
// 0x00BB  STS 0x0207,R25  ; 4. Befehl
    TCCR1B = 0;   // Zähler aus
    t = TCNT1;    // Zählwert speichern
}
```

CTC-Modus

CTC- (Clear on Compare) Modus



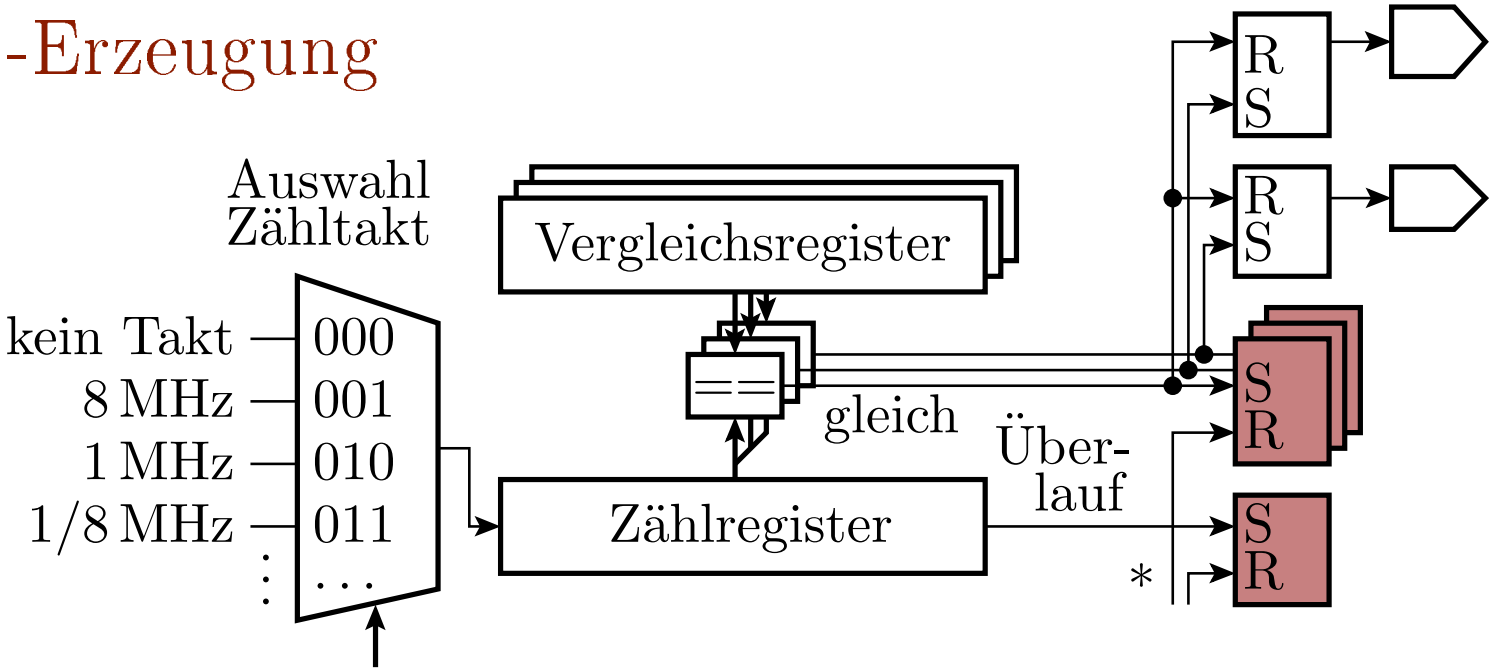


- Zähler wird bei Gleichheit mit einem der Vergleichsregister rückgesetzt.
- Auslösung zyklischer Ereignisse, z.B. Uhrenprozess:

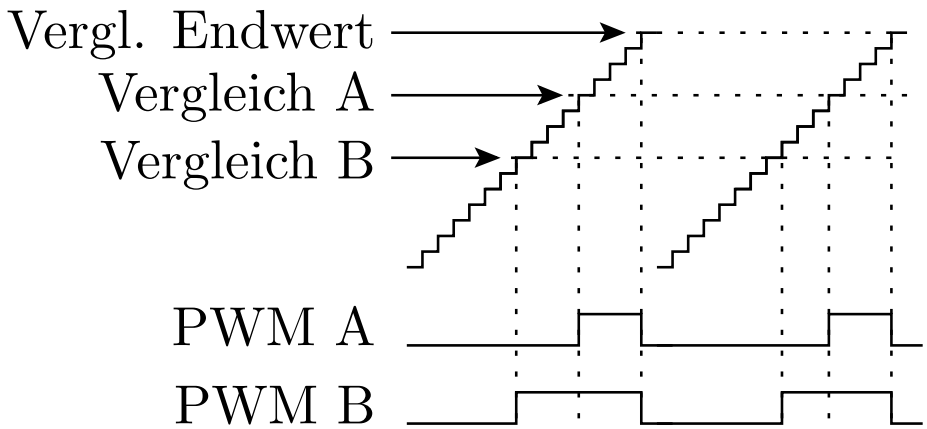
```
void Schrittfunktion Uhr(){
  if (<Vergleichs-Rücksetz-Ereignis>)
    <lösche Ereignisbit(s) und schalte Uhr weiter>
}
```

PWM-Erzeugung

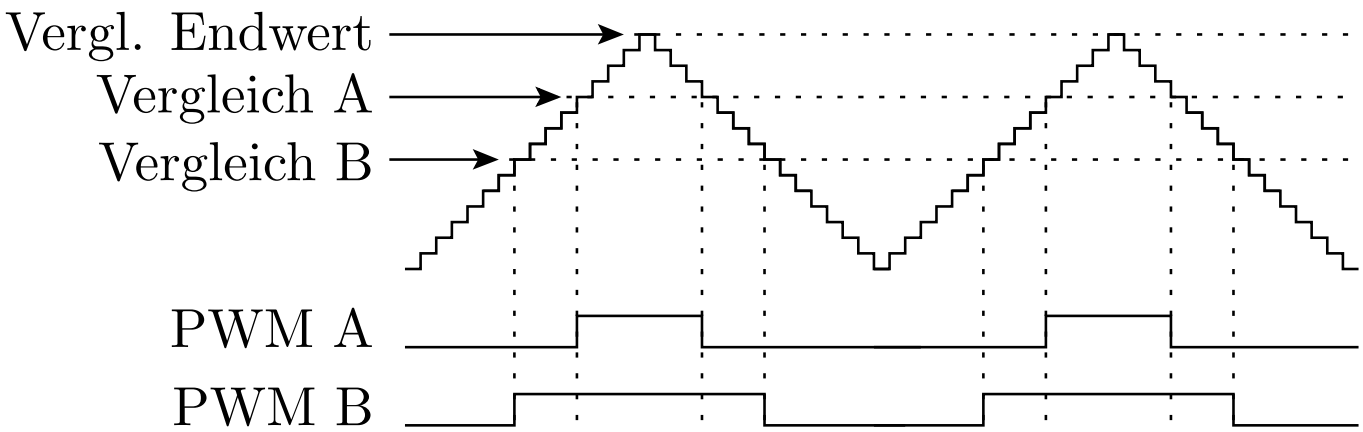
PWM-Erzeugung



- Zählerüberlauf setzt Ausgabebit.
- Vergleichsereignis löscht Ausgabebit.
- Pulsgenerierung z.B. zur Motoransteuerung ohne Polling und ISR.



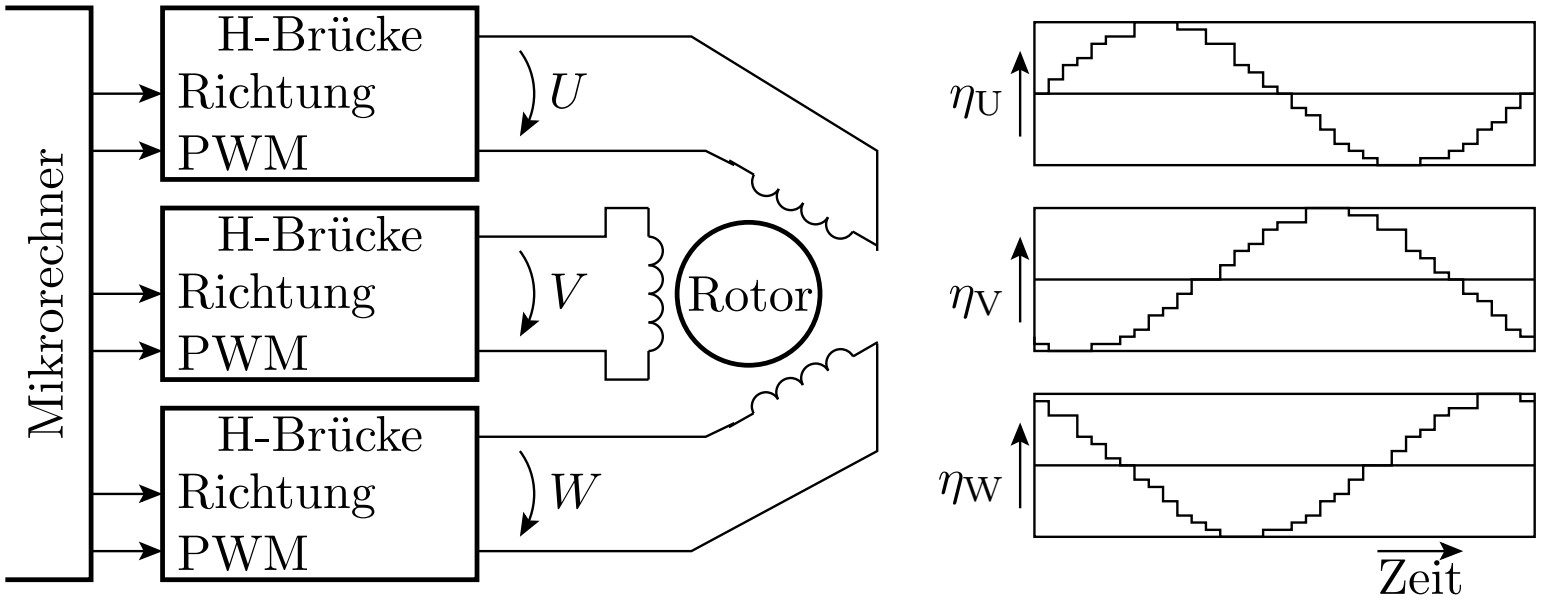
Symmetrische PWM[5]



- An den Endwerten schaltet die Zählrichtung um.
- Bei Gleichheit und Hochzählen wird die Ausgabe ein- und bei Gleichheit und Abwärtszählen ausgeschaltet.
- Bei dieser und der vorherigen PWM kann auch eine invertiert Ausgabe programmiert werden, so dass der Vergleichswert statt der Ausschalt-, die Einschaltzeit festlegt.

[5] Im Datenblatt unseres Prozessors ist das die phasenrichtige und die vorhergehende normale PWM die schnelle (Fast-) PWM.

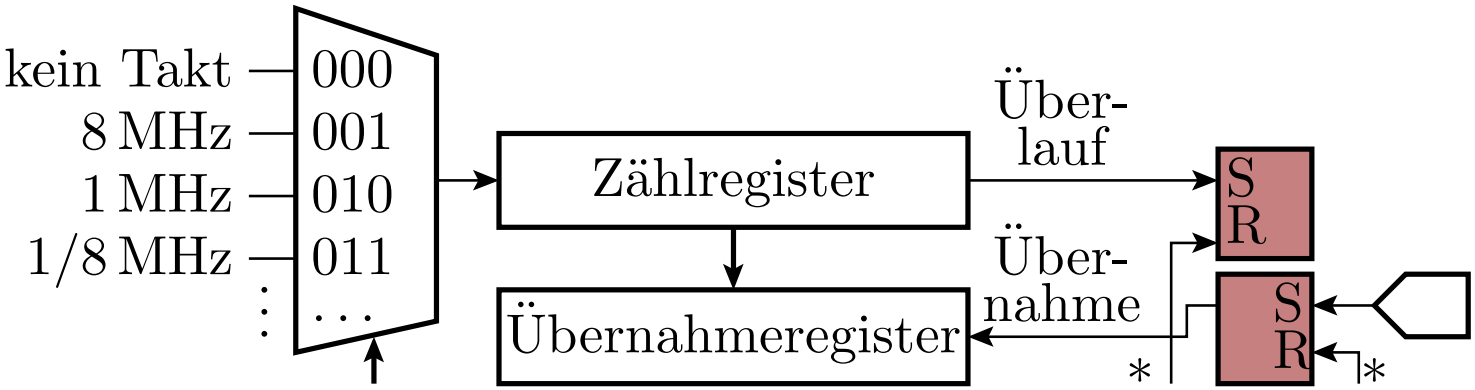
Typische Motoransteuerung



- Die Erzeugung von 3 sinusförmigen Mittelwertverläufen erfordert eine PWM-Einheit mit drei Vergleichsregistern.
- Ansteuerung über H-Brücken.
- Stufenlose Positions-, Geschwindigkeits- und Drehmomentsteuerung für viele Typen von Elektromotoren.

Pulsweitenmessung

Pulsweitenmessung



- Externes Ereignis (Schaltflanke) bewirkt Übernahme des Zählwerts in das Übernahmeregister und setzt das Ereignisbit.
- Polling auf oder ISR-Aufruf bei Ereignisbitaktivierung.
- Programmgesteuerte Differenzbildung der Übernahmewerte zwischen den Übernahmeereignissen.

Watchdog

Watchdog-Timer (WDT)

Jedes größere Programm enthält statisch gesehen Fehler, die unter anderem auch dazu führen, dass das Programm abstürzt. Der WDT begrenzt die Dauer der Nichtverfügbarkeit durch Abstürze auf wenige ms bis s.

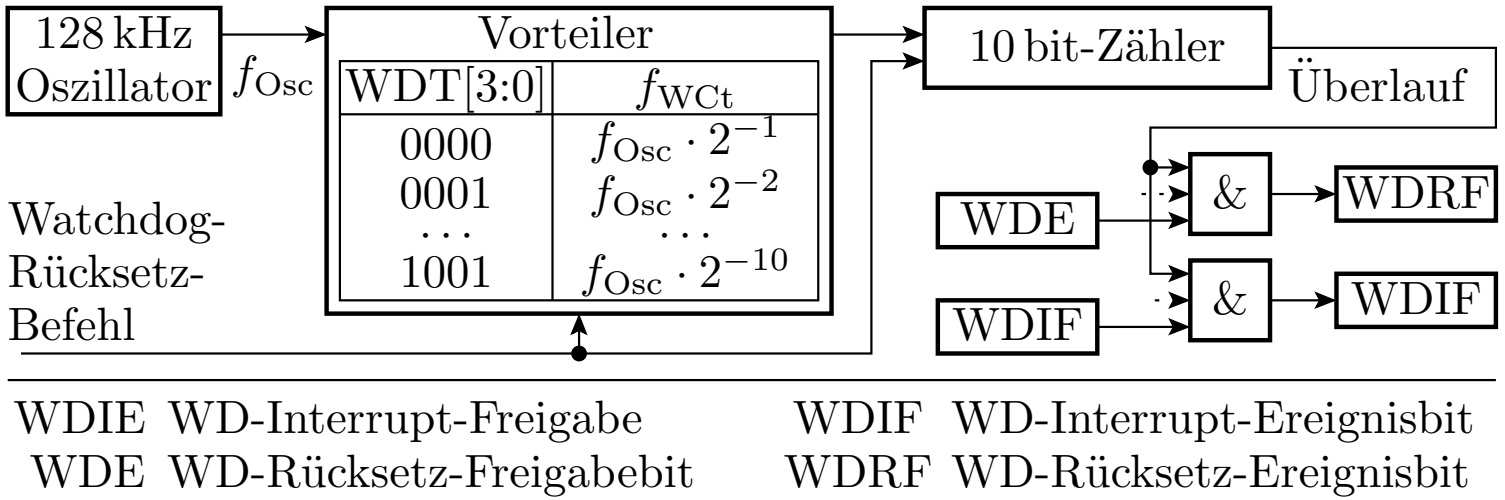
Funktionsprinzip:

- Zeitzähler, der Zeitimpulse zählt und bei Überlauf das System neuinitialisiert (und/oder Interrupt auslöst).
- Um Überläufe (Neuinitialisierungen) zu verhindern, muss das Programm in einer vorprogrammierten Mindestzeit Rücksetzbefehle für den Watchdog ausführen.

Programmierbar sind:

- die Zeit bis zum Überlauf und
- die Reaktion bei Überlauf (Interrupt, Neustart).

Watchdog-Timer (WDT) des ATmega 2560



- Zeit bis zum Überlauf: programmierbar von 16 ms bis 8 s.
- Nur-Interrupt: »Wiederbelebung« per Software.
- Interrupt + Rücksetzen: Datenretten + Neustart.
- WDT-Reset mit Fuse-Bit »WDTON« auch dauerhaft aktivierbar. Dann keine Deaktivierung durch Software (-Fehler) möglich.

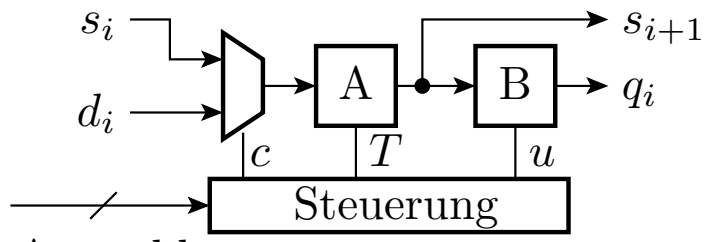
Serielle Schnittstellen

Serieller Datenaustausch

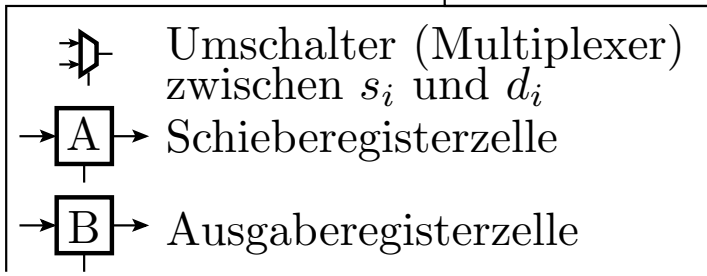
Der Datenaustausch zwischen Rechnern erfolgt in der Regel seriell[6].
Grundbaustein Schieberegister mit den Funktionen

- parallele Übernahme der zu übertragenden Daten,
- serielle Übertragung und
- parallele Übergabe.

Schaltung einer Schieberegisterzelle:



Steueraufgabe	c	T	u	Operation
Schieben	0	$\int$	0	$s_{i+1} \leftarrow s_i$
Übernahme	1	$\int$	0	$s_{i+1} \leftarrow d_i$
Übergabe	–	0	$\int$	$q_i \leftarrow s_{i+1}$
keine	–	0	0	——

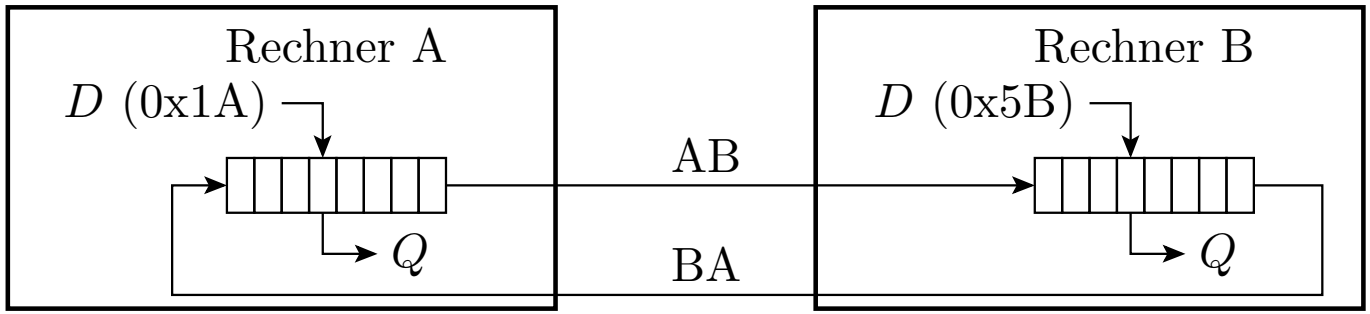


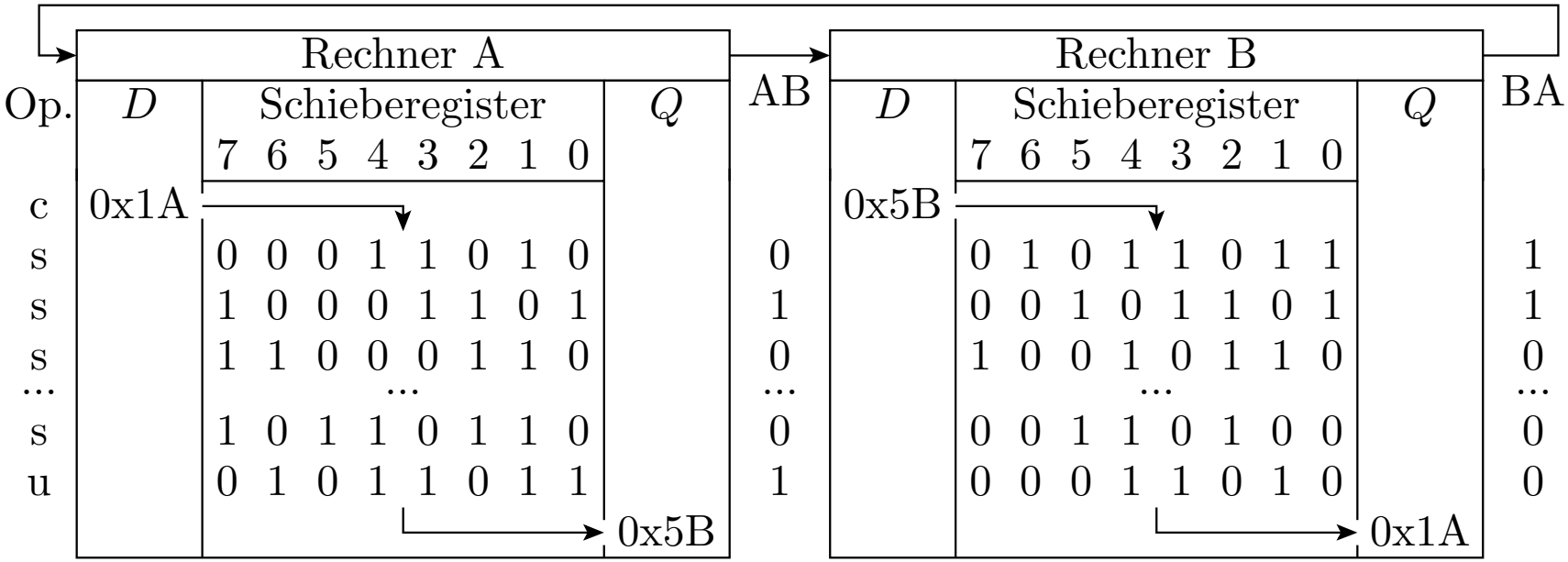
interne Takt- und Steuersignale:

u Update
c Capture
T Takt

[6] Seriell, d.h. hintereinander über eine, statt parallel über viele Leitungen.

Bidirektionale Kopplung zweier Rechner



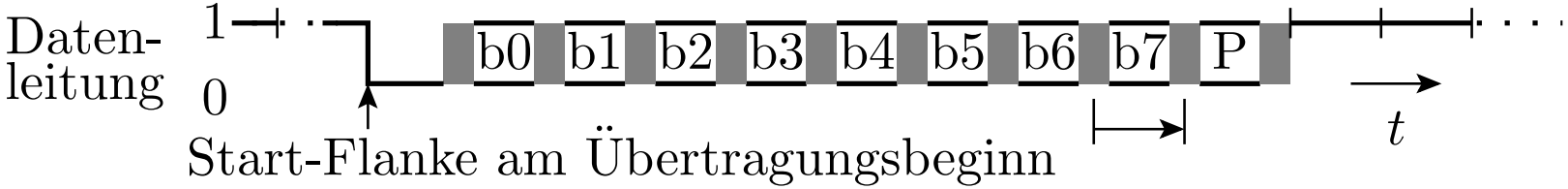


c Übernahme (Capture) s Schieben (Shift) u Übergabe (Update)

USART

USART (**U**niversal **S**ynchronous or **A**synchronous **R**eceiver and **T**ransmitter)

Übertragung ohne Takt und Steuersignale.



$b \in \{0,1\}$	Datenbits	$\cdots$	Übertragungspause
$P \in \{0,1\}$	Paritätsbit	$\longmapsto$	Bitzeit, z.B. $t_{\mathrm{Bit}} \approx 0{,}1\,\mathrm{ms}$
$\longmapsto\!\!\shortmid$	Stoppbit (Wert 1)		Übertragungsdauer: $12 \cdot t_{\mathrm{Bit}}$

Der Empfänger erkennt den Übertragungsbeginn an der Stopp-/Start-Flanke und übernimmt die Werte nach 1,5, 2,5 etc. Bitzeiten. Voraussetzung: Gleich eingestellte Bitzeit, Bitanzahl, Stoppbitanzahl und Parität bei Sender und Empfänger. Die Baudrate b als Kehrwert der Bitzeit t_{Bit} wird mit einem Teiler aus dem Prozessortakt gebildet.

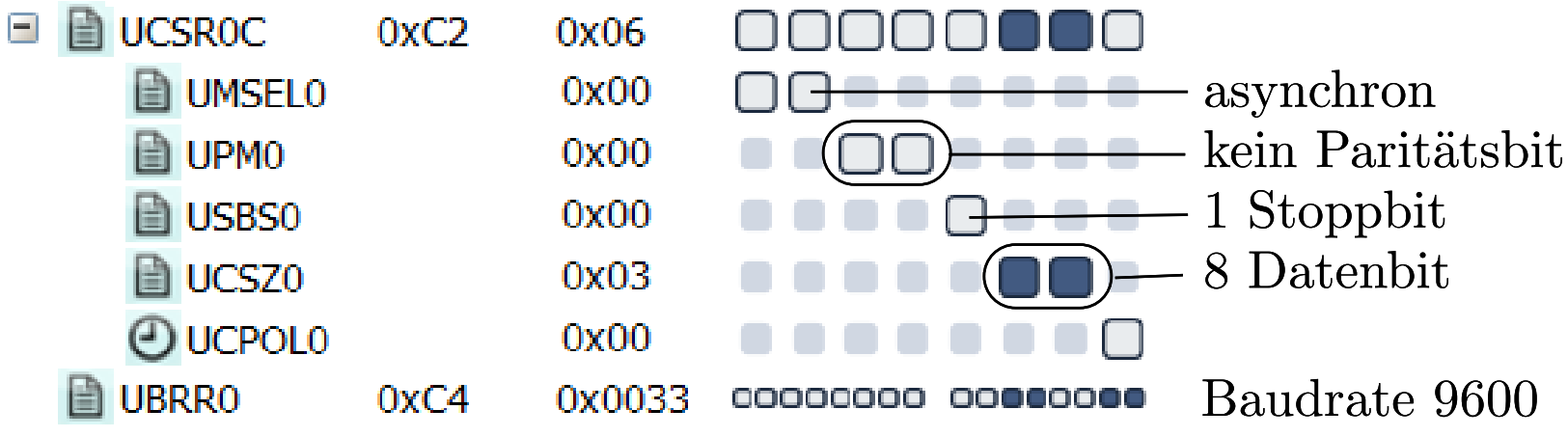
Baudrate b, Masseinheit Bd (Baud), ergibt sich aus der Prozessortaktfrequenz f_{Proz} und einem programmierten Teilerfaktor T:

$$b = \frac{f_{\mathrm{Proz}}}{16 \cdot (T+1)}$$

Teilerfaktor für $f_{\mathrm{Proz}} = 8\,\mathrm{MHz}$ und $b = 9600\,\mathrm{Bd}$:

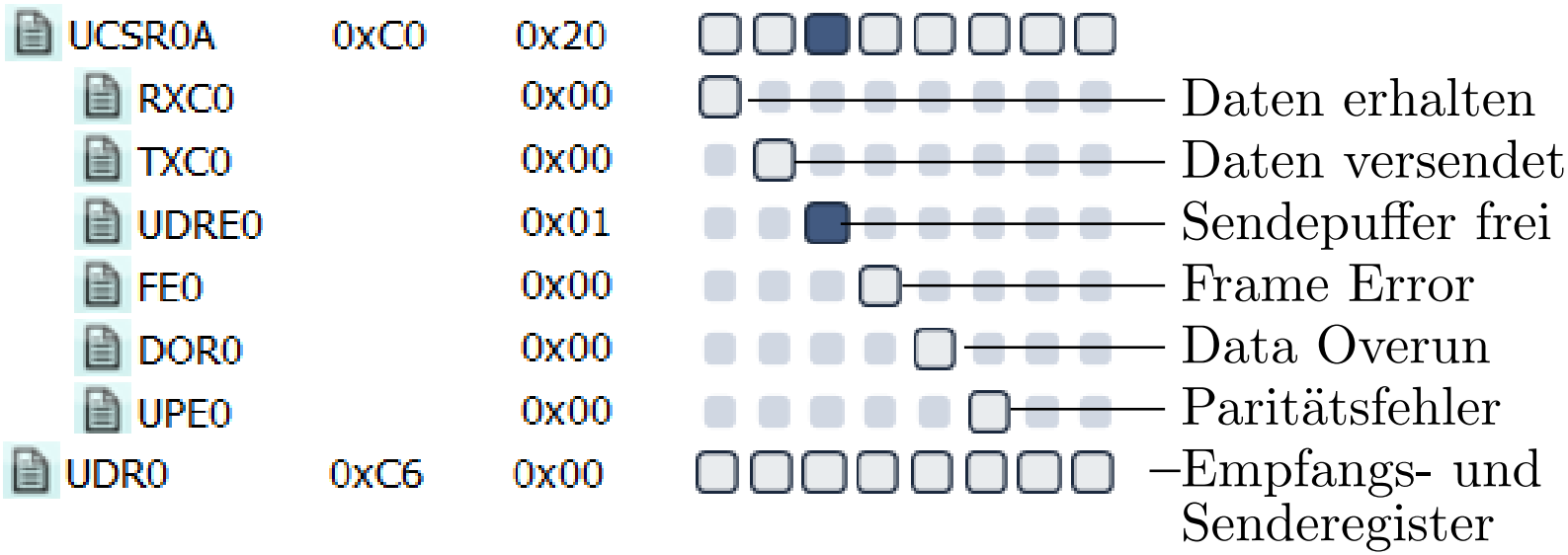
$$T = \frac{8\,\mathrm{MHz}}{16 \cdot 9600\,\mathrm{Bd}} - 1 = 51 = 0\mathrm{x}33$$

Initialisierung: 9600 Bd, 8 Daten-, 1 Stoppbit, keine Parität:




```
UBRR0 = 0x33;          Teilerwert für 9600 baud
UCSR0B = 0b00011000;   Sender und Empfänger einschalten
UCSR0C = 0b00000110;   Datenformat 8n1
```

Senden und Empfang



Funktionen für den Empfang und das Versenden eines Bytes

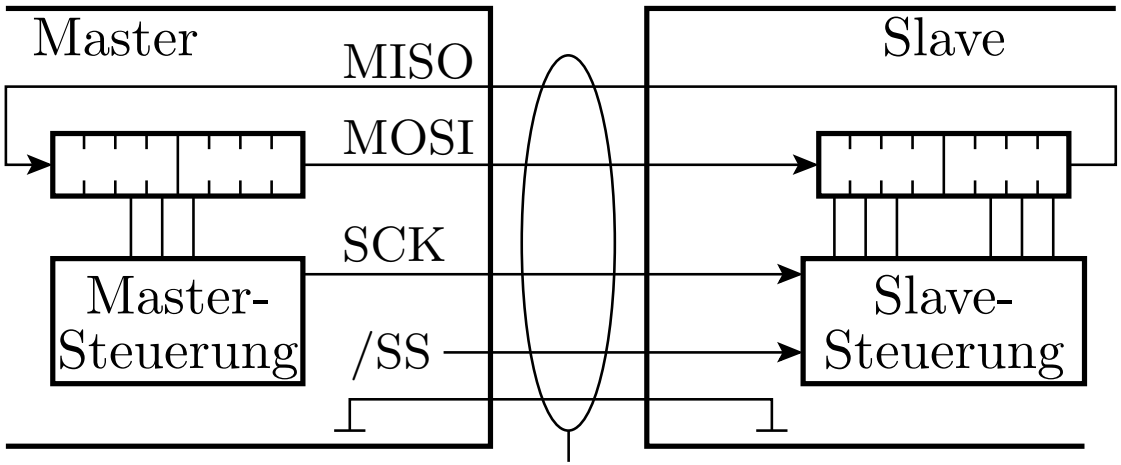
```
uint8_t getChar(){
  while (!(UCSR1A & (1<<RXC0)));   // Warte auf Empfang
  return UDR0;                     // Rückgabe Empfangsbyte
}

void putChar(uint8_t c){
  while (!(UCSR1A & (1<<UDRE0)));  // Warte bis Sendepuffer
  UDR0 = c;                        // frei. Byte versenden
}
```

SPI-Bus

SPI-Bus

Serieller Bus zur Vernetzung von Schaltkreisen.

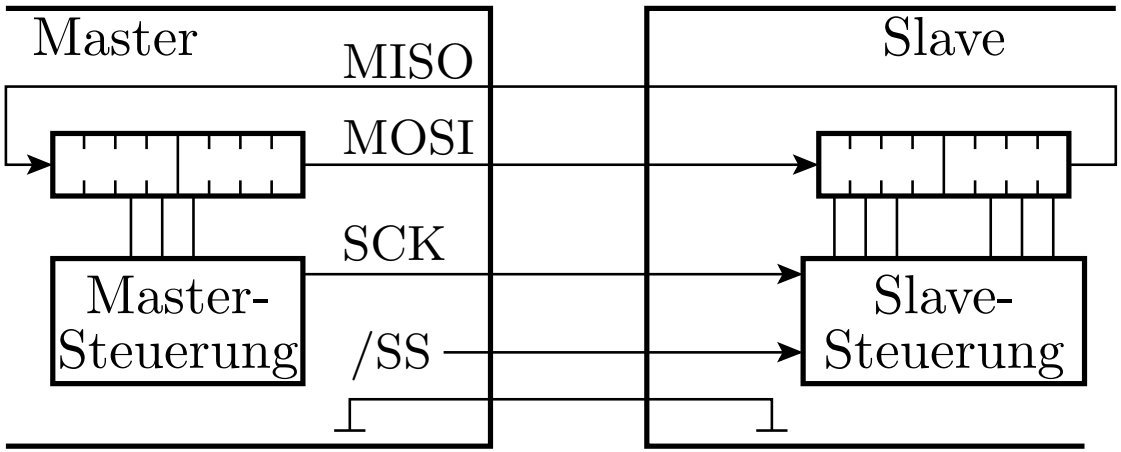


Leitungen zwischen den Schaltkreisen

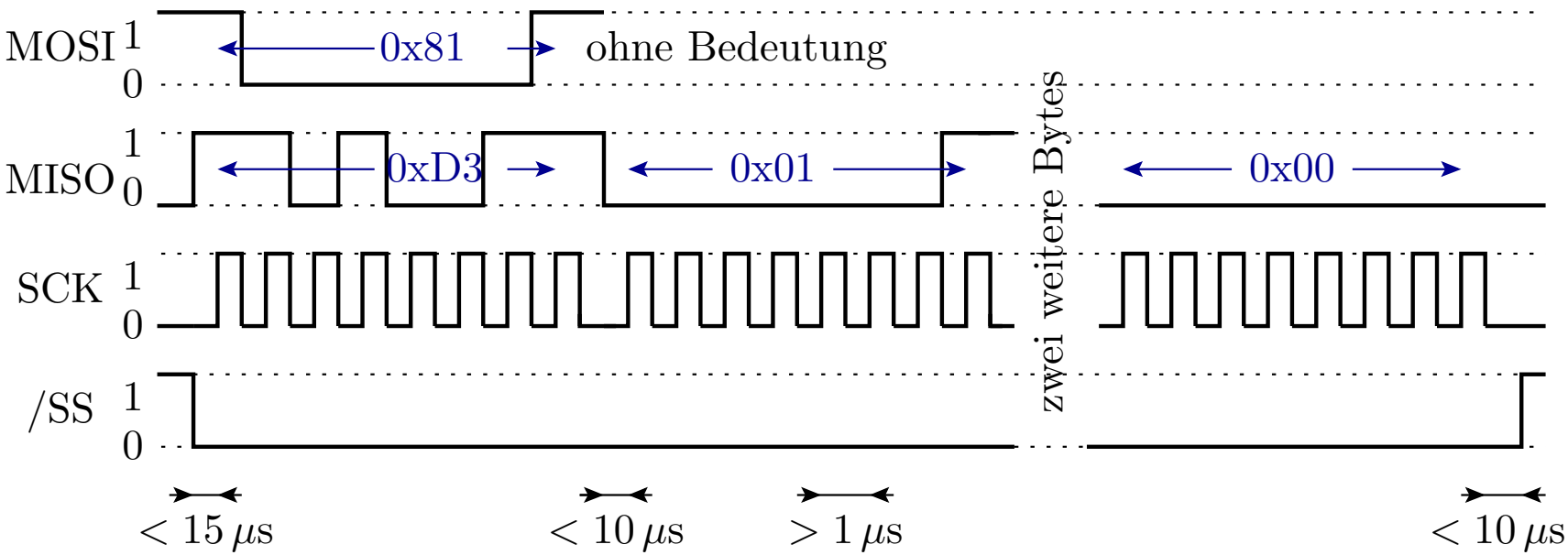
MOSI **M**aster **O**ut **S**lave **I**n
MISO **M**aster **I**n **S**lave **O**ut
SCK Schiebetakt
/SS Slave-Auswahl

Aktion	/SS	SCK
Übernahme	⊓	
Schieben	0	⊓
Ausgabe	⊓	
keine	sonst	

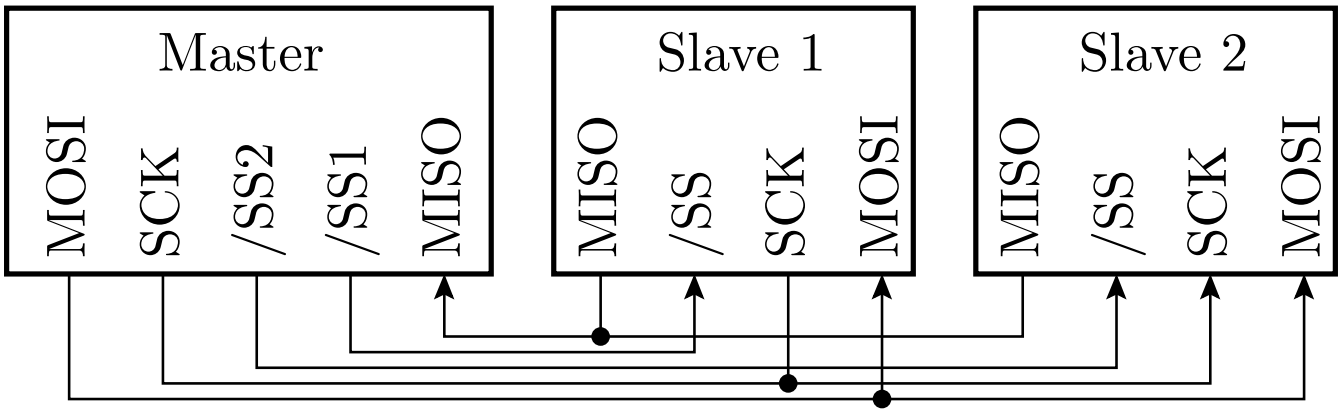
- Ein Schaltkreis ist der Master, der den Takt SCK und die Slave-Auswahlsignale erzeugt, die anderen sind Slaves, die diese



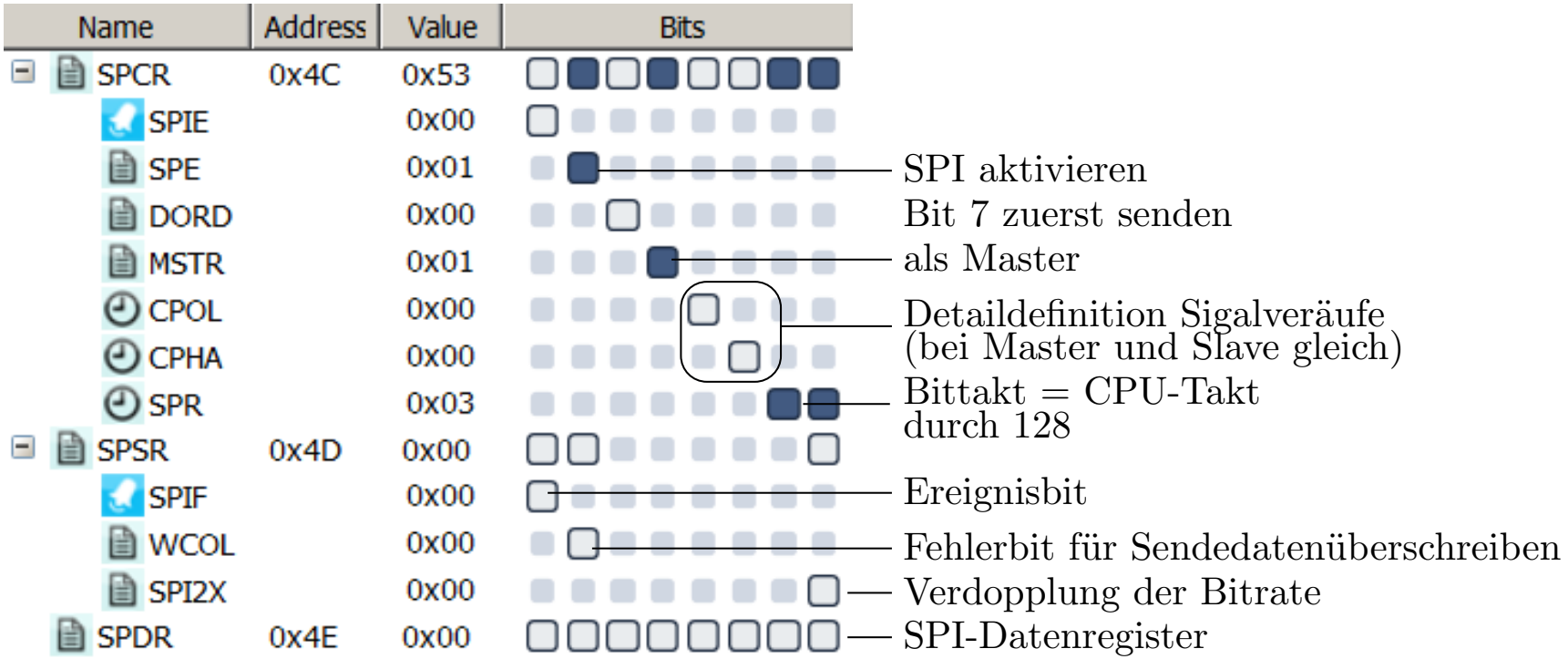
Beispiel für die Übertragung 0x81 vom Master zum Slave und von 0xD201...00 vom Slave zum Master:



- Eine Übertragung beginnt mit Aktivierung von /SS=0 (Übernahme), gefolgt von n Schiebetakten und endet mit Deaktivierung /SS=1.
- Die /SS-Signale des Masters werden über Ausgänge paralleler Schnittstellen ausgegeben.
- Ein Master kann mehrere Slaves mit unterschiedlichen /SS-Signalen angesteuern.



Konfiguration des SPI-Controllers



- Einschalten als Master oder Slave.
- Festlegen der Bitrate und Protokollparameter.
- Pins für /SS Signale konfigurieren, beim Master als Ausgänge mit Wert eins, beim Slave als Eingänge.

Algorithmus für den Datenaustausch

Name	Address	Value	Bits
SPSR	0x4D	0x00	
SPIF		0x00	
SPDR	0x4E	0x00	

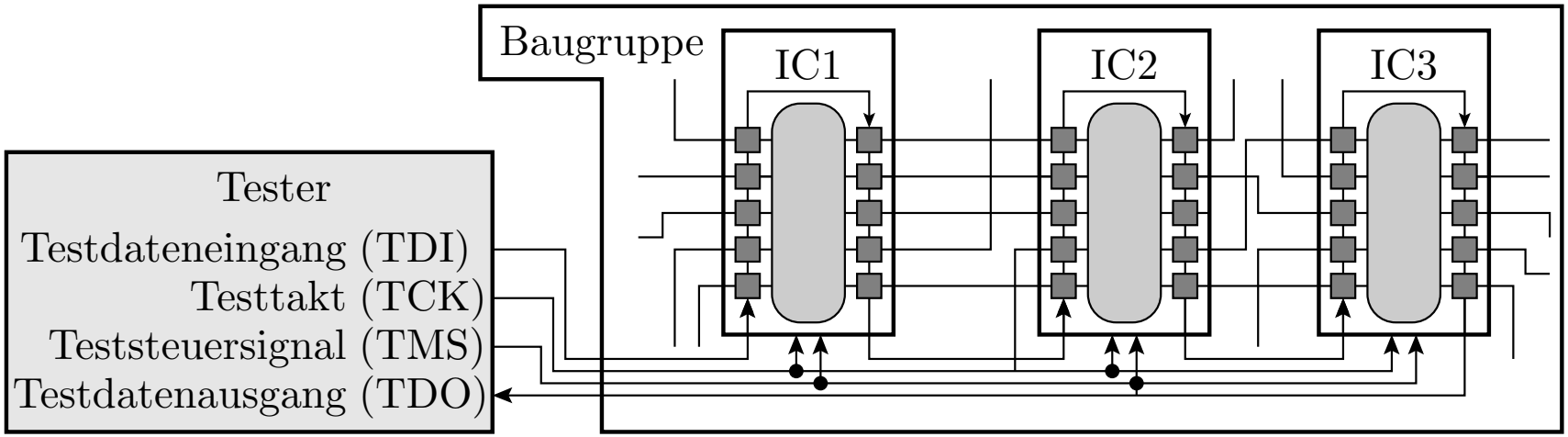
Für jede n-Byte-Übertragung

- Aktiviere das Slave-Auswahlsignal /SS (Master) bzw. warte auf /SS=0 (Slave).
- Für jedes Byte
  - Schreibe Sendewert in das Datenregister.
  - Warte bis Ereignisbit SPIF eins ist.
  - Lese empfangenes Byte aus und schreibe nächstes zu sendende Byte in das SPI-Datenregister SPDR.
- Deaktiviere das Slave-Auswahlsignal.

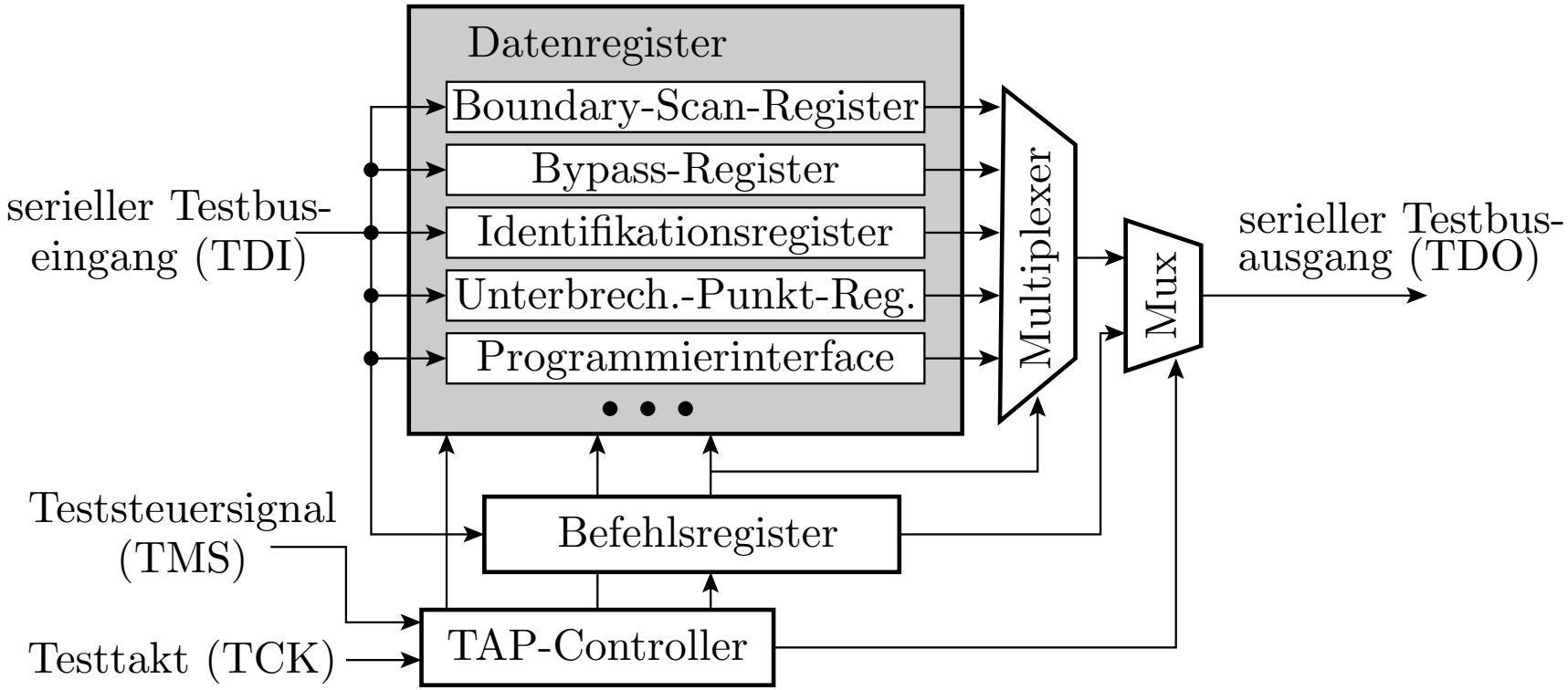
JTAG (Testbus)

JTAG

Test-, Diagnose-, Debug- und Programmierbus.

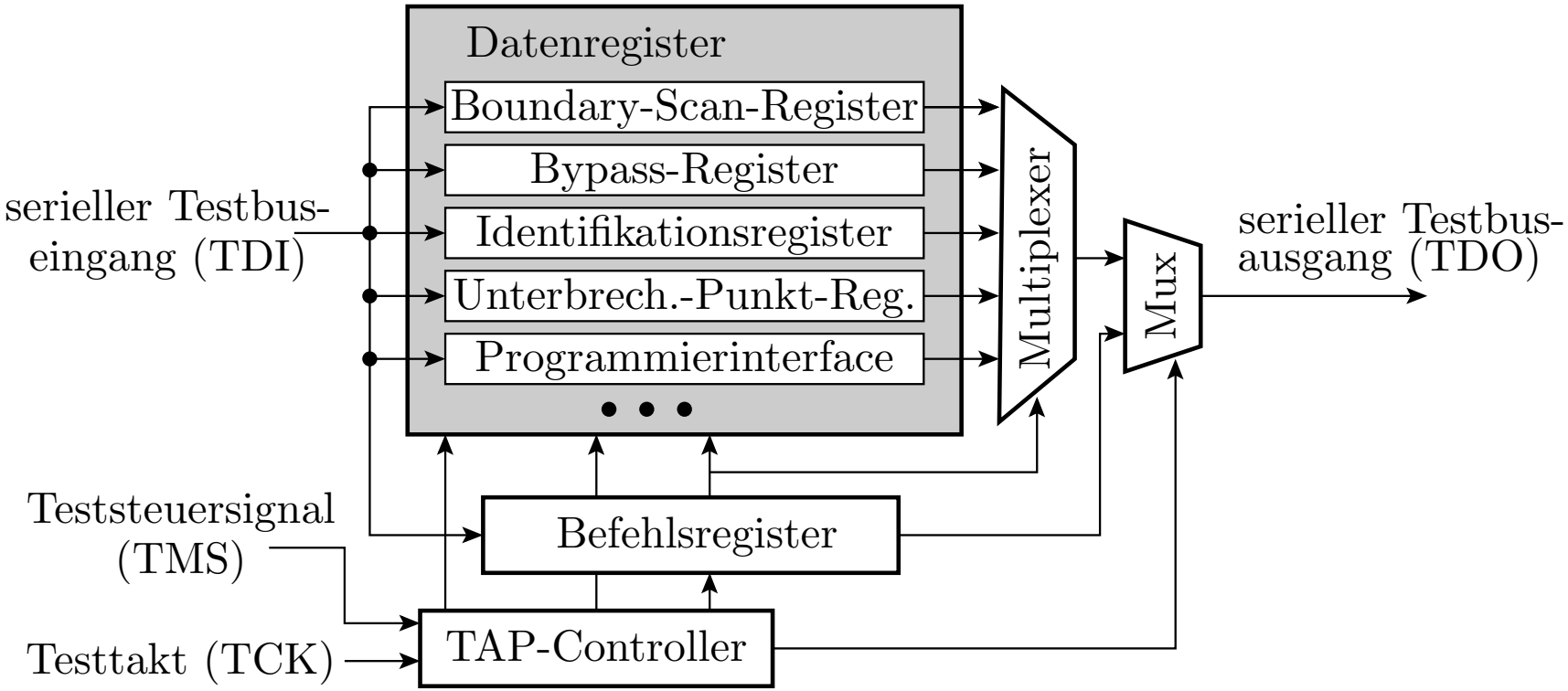


- Verschaltung aller Schaltkreise zu einer Kette.
- In der Ursprungsidee (Boundary-Scan) waren die Schieberegister mit den Funktionen Übernahme, Schieben und Übergabe am Schaltkreisrand angeordnet, um die Verbindungen zwischen den Schaltkreisen ohne mechanische Kontaktierung testen zu können.



Ein Schaltkreis mit JTAG-Bus hat mehrere über ein Befehlswort auswählbare Datenregister:

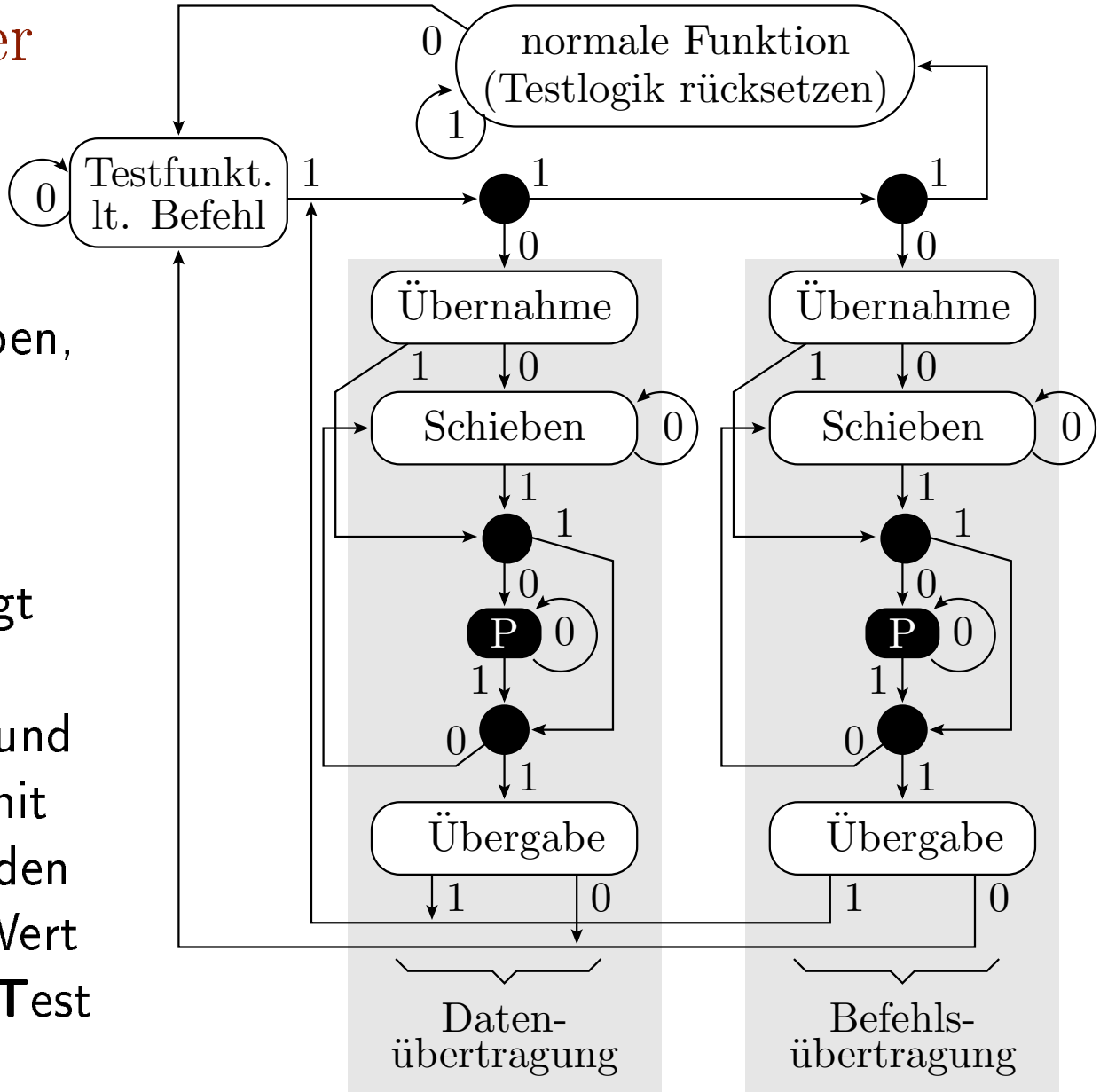
- Bypass-Register zur Verkürzung der Länge des Schieberegisters durch den Schaltkreis auf 1 Bit,
- Identifikationsregister mit Hersteller- und Bauteilnummer,



- Programmier-Interface: Schnittstelle zum Lesen und Schreiben des Befehls-Flashes, des Daten-EEPROMs und der Fuse-Register.
- Schnittstellenregister zum OCD (**O**n-**C**hip **D**ebugger), ...

TAP-Controller

Die Auswahl der 6 Busaktionen: Übernahme, Schieben, Übergabe für das Befehls- und das ausgewählte Datenregister erfolgt über ein 1-Bit-Steuersignal und einen Automaten mit 16 Zuständen. An den Kanten steht der Wert des Signals TMS (**T**est **M**ode **S**elect).



Von der JTAG-Implementierung in unserem Prozessor sind nur die standardisierten Testfunktionen, die für den Bestückungstest von Baugruppen vorgesehen sind, veröffentlicht. Die Befehle für die Programmierung und den OCD (On-Chip Debugger) sind in den Dokumentationen nicht beschrieben.

Analoge Eingabe

Messung und Überwachung von analogen Werten

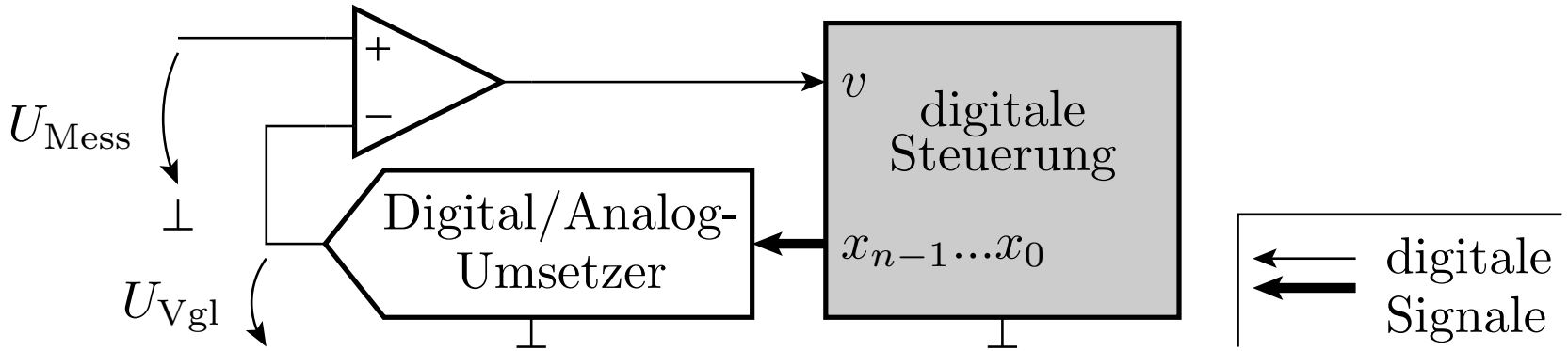
Analog-Digital-Wandler:

- Umwandlung einer analogen Eingangsspannung oder Eingangsspannungsdifferenz in einen Bitvektor (beim ATmega 2560 10 Bit).
- Wandlungsdauer 13 bis 25 Takte mit einer Taktperiode $t_{\mathrm{AClk}} \geq 1\mu\mathrm{s}$. Gesamte Wandlungsdauer $\geq 13 \ldots 25\mu\mathrm{s}$.
- Über einen programmierbaren Eingabemultiplexer kann zwischen unterschiedlichen Signalquellen ausgewählt werden.

Analog-Komparator:

- Vergleich zweier analoger Eingangsspannungen.
- Das 1-Bit-Vergleichergebnis kann programmgesteuert ausgewertet, Interrupts auslösen oder Zeitmessungen mit Timern steuern.

Prinzip eines seriellen Analog-Digital-Wandlers



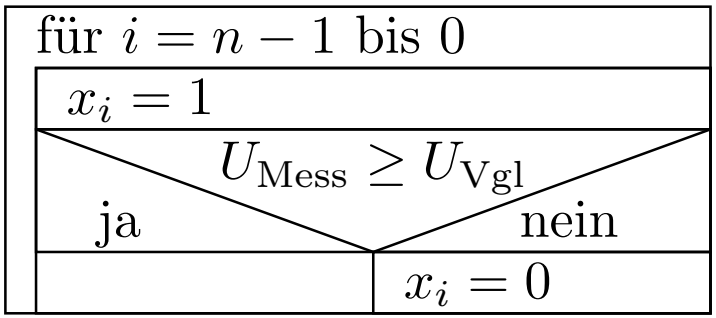
$$v = \begin{cases} 0 & \text{wenn } U_{\text{Mess}} < U_{\text{Vgl}} \\ 1 & \text{sonst} \end{cases}$$

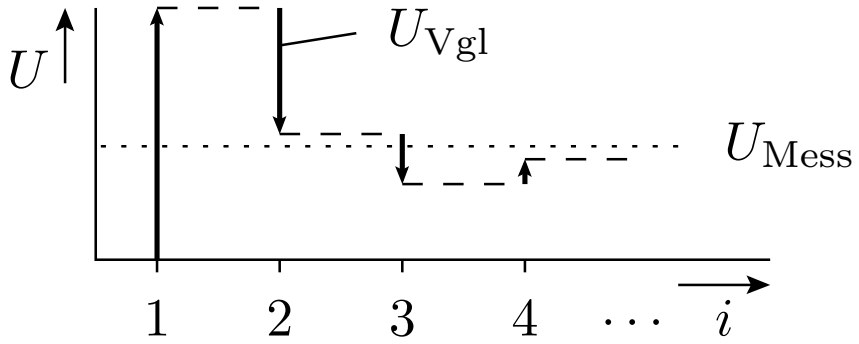
Die Vergleichsspannung, die der DAU (**D**igital-**A**nalog-**U**msetzer) ausgibt:

$$U_{\text{Vgl}} = U_{\text{ref}} \cdot \frac{\mathbf{x}}{2^n}$$

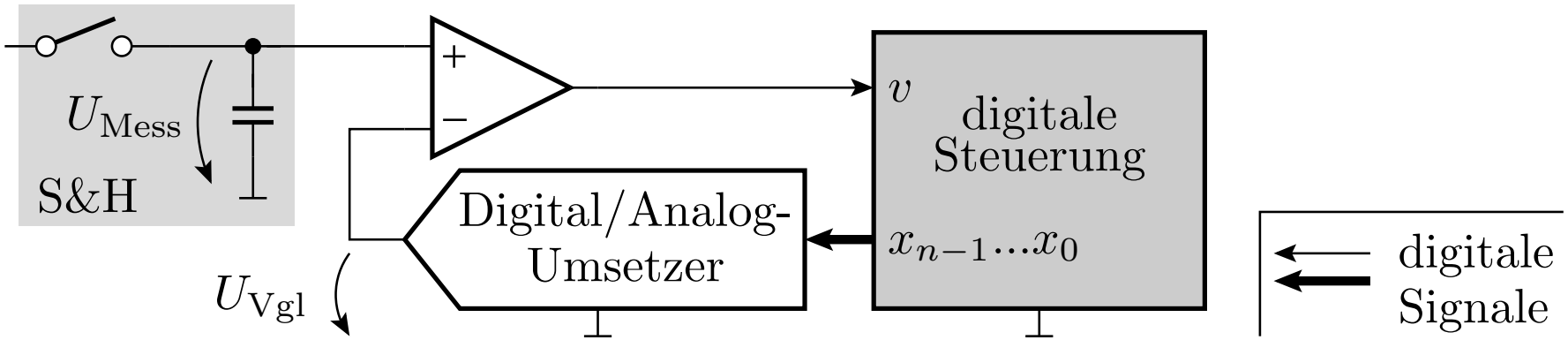
(U_{ref} – Referenzspannung; $\mathbf{x}$ – Ausgabewert (Bitvektor); n – Bitanzahl des Ausgabewerts).

Sukzessive Approximation





Ein Vergleich je Bit. Der Messwert darf sich während der Wandlung nicht ändern. Deshalb wird U_{Mess} mit einer Sample-and-Hold-Schaltung (S&H) abgetastet und während der Messung gespeichert.



Beispiel zur ADC-Initialisierung

```
11  void initADC(){
12    ADMUX = (3<<MUX0); // Kanal 3 auswählen
13    // Einschalten mit Wandlungstaktteiler 64
14    ADCSRA = (1<<ADEN)|(0b110<<ADPS0);
15    DDRE &= ~0x80;  // Sensoreingang als Eingang
16    PORTE &=~0x80;  // Ausgabewert 0 (hochohmig)
17  }
```

- Der Sensor ist an ADC3 (PF3) (Kanal 3 auswählen).
- Der Wandlertakt als CPU-Takt durch Teilerwert

$$f_{\mathrm{ADC}} = \frac{f_{\mathrm{CPU}}}{64} \approx 117\,\mathrm{kHz}$$

- Um den Analogwert nicht zur verfälschen, ist PF3 als Eingang mit Ausgabewert 0 (Pullup aus) zu konfigurieren.

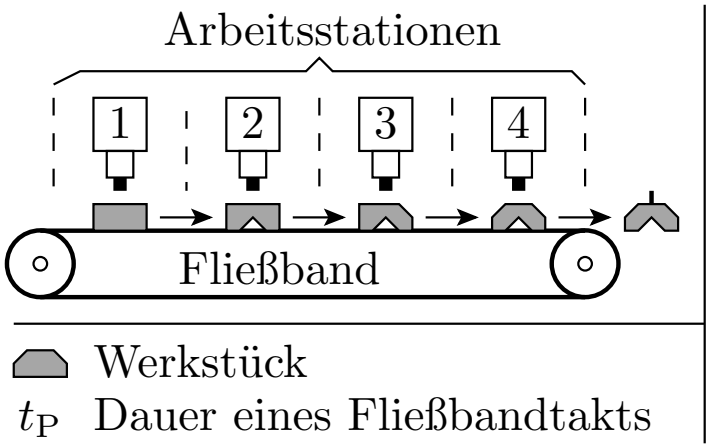
Funktion zur Messung eines Analogwerts

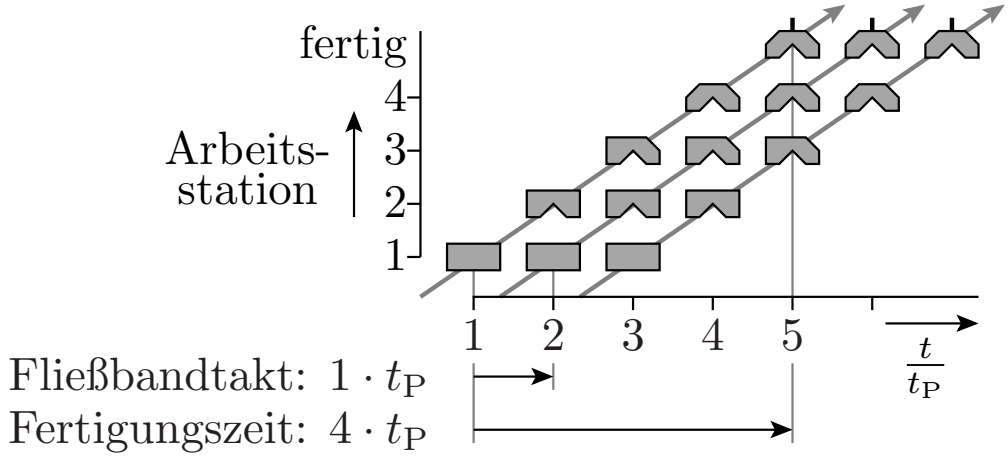
```
uint16_t getADC(){
 uint16_t wert;
 ADCSRA |= (1<<ADSC);                  //Wandlung starten
 while(!(ADCSRA & (1<<ADIF)));//auf ADIF warten
 ADCSRA |= (1<<ADIF);                  //ADIF löschen
 wert = ADC;                           //Ergebnisrückgabe
 return wert;
}
```

- Wandlungsstart durch Setzen von ADSC in ADCSRA.
- Bei Wandlungsabschluss setzt der Prozessor ADIF=1.
- ADIF wird durch Schreiben einer Eins gelöscht.

Pipeline-Verarbeitung

Pipeline-Verarbeitung

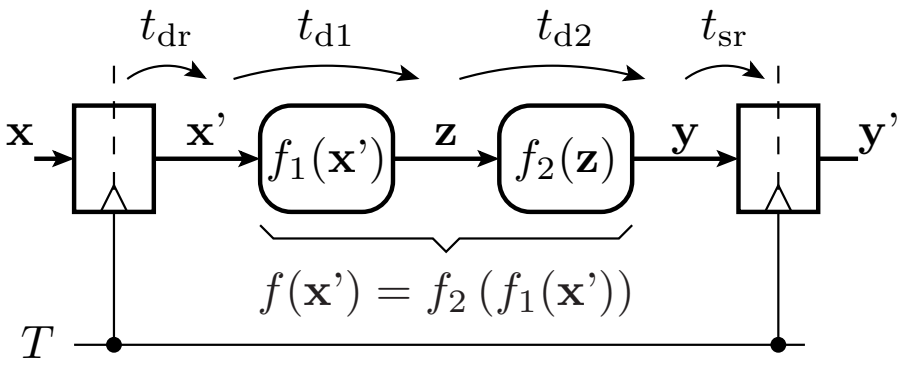


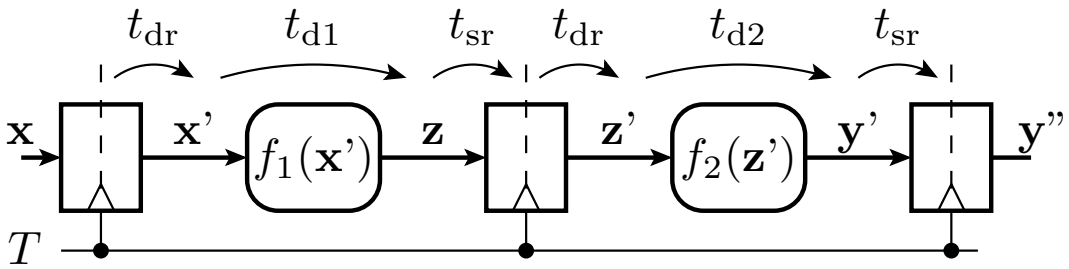


- Aufteilung einer Gesamtaufgabe in N_P Arbeitsschritte.
- Gesamtaufwand je Objekt: $N_P \cdot t_T$ (t_T – Periode Schritttakt)
- Je Takt wird ein Objekt fertig.

Parallelverarbeitung ohne viel Zusatzaufwand.

Angewandt auf Hardware





t_{dr}	Registerverzögerungszeit
t_{d1}	Verzögerungszeit von f_1
t_{d2}	Verzögerungszeit von f_2
t_{sr}	Registervorhaltezeit
$\mathbf{x}$	Eingabesignal
$\mathbf{z}$	Zwischensignal
$\mathbf{y}$	Ausgabesignal
...'	abgetastetes Signal

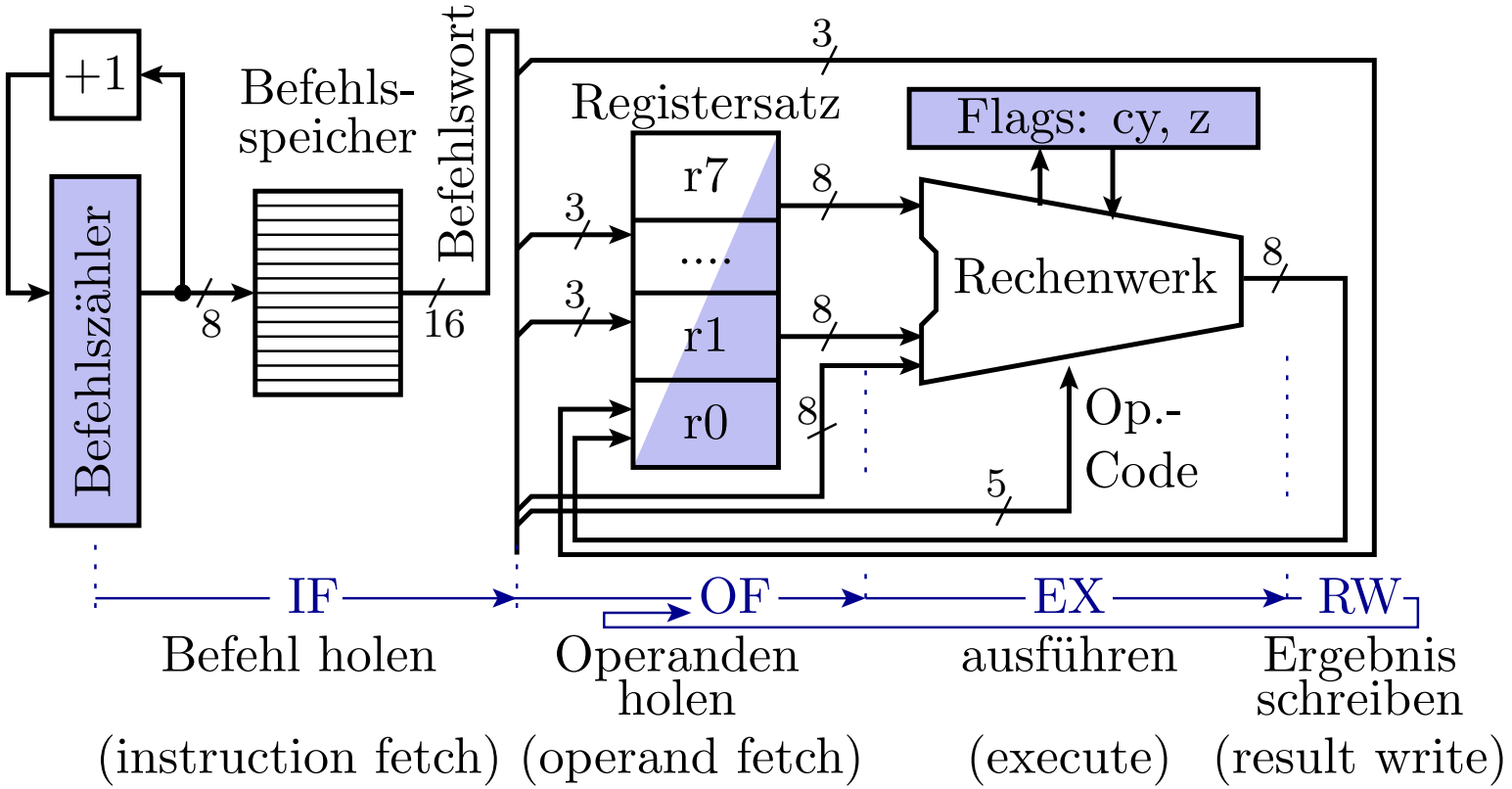
minimale Taktperiode

- ohne Pipeline
  $t_{dr} + t_{sr} + t_{d1} + t_{d2}$
- mit Pipeline
  $t_{dr} + t_{sr} + \max(t_{d1}, t_{d2})$

- Aufwand: ein zusätzliches Register je Pipeline-Stufe.
- Viel billiger als mehrfache Hardware. Vorzuglösung für Parallelverarbeitung.

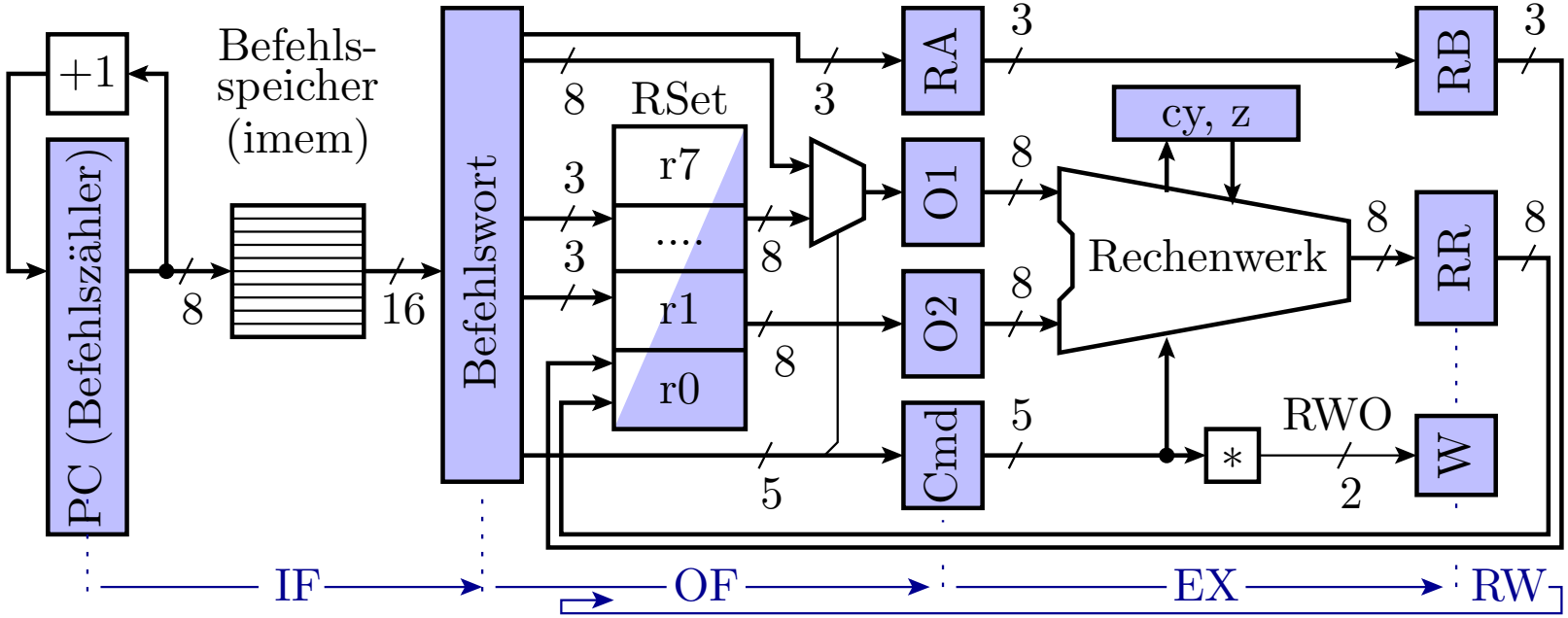
MiPro mit Pipeline

Der Kern des Minimalprozessors ohne Pipeline

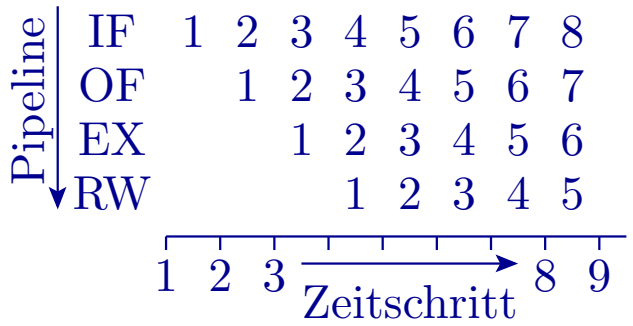


- Quellregister: Operanden- und Statusregister, Befehlszähler.
- Zielregister: Ergebnis- und Statusregister, Befehlszähler.
- Verarbeitungsschritte: Befehl holen, Operanden holen, ...

Aufteilung in Pipeline-Phasen



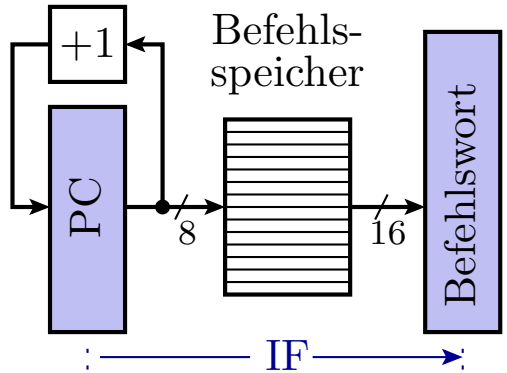
r0 bis r7	Arbeitsregister
O1, O2	Operandenregister
Cmd	Befehlsregister
RA, RB	Ergebnisadressreg.
RR	Ergebnisregister
W	OP-Code RW-Phase



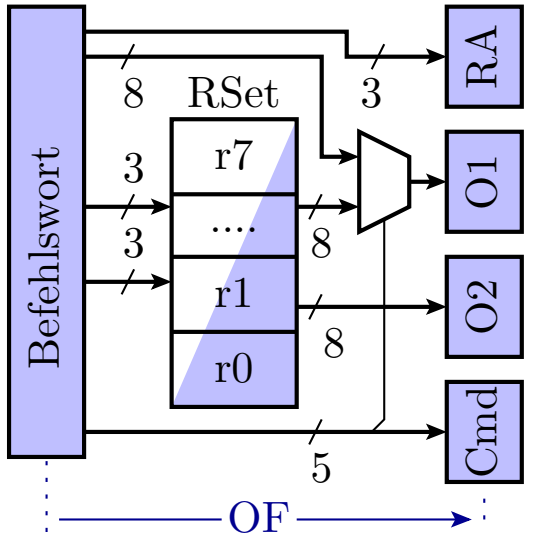
Die Aufteilung der Verarbeitungsschritte in Pipeline-Phasen erfolgt durch Einbau getakteter Register für die Zwischenergebnisse.

Operationen der einzelnen Pipeline-Phasen:

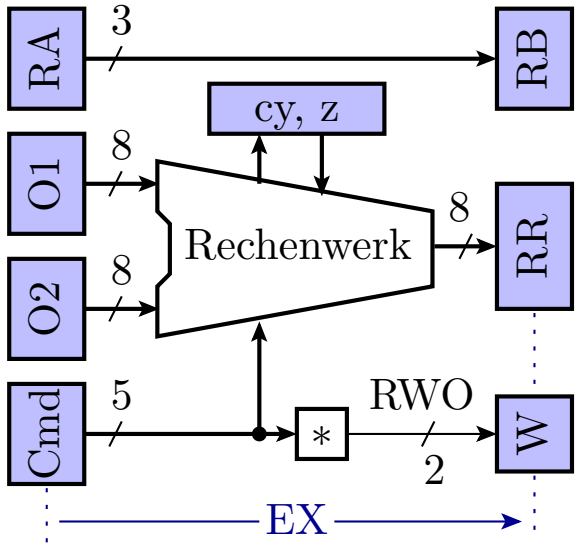
- IF (**I**nstruction **F**etch) Befehl holen: Adressierung des Befehlsspeichers vom Befehlszähler und Übernahme des Befehlswortes in das Befehlswortregister.



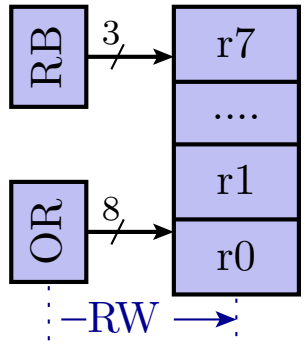
- OF (**O**perand **F**etch) Operanden holen: Adressierung des Registersatzes mit den Operandenadressen und Übernahme der Registerinhalte in die Operandenregister. Weitergabe der Ergebnisadresse und des Operationscodes an die nächste Pipeline-Phase.



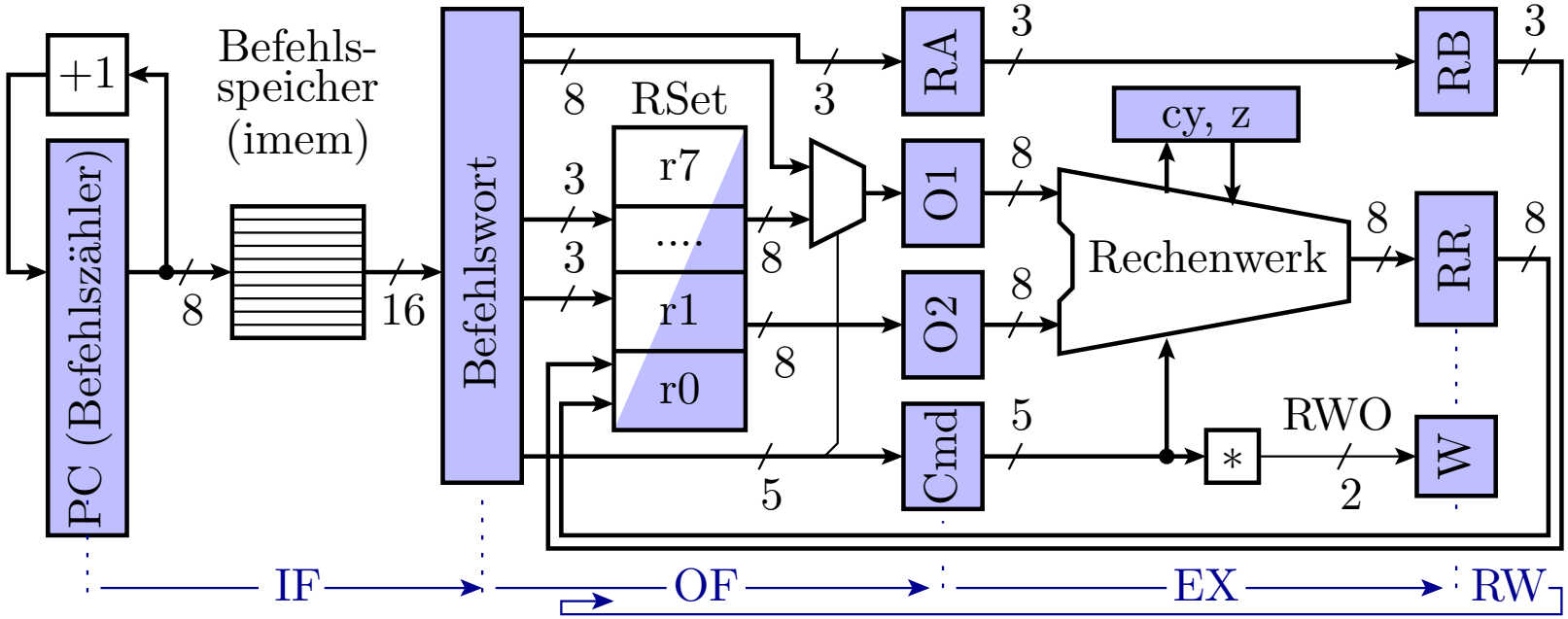
- EX (**Ex**ecute): Befehle ausführen: Aus den Operanden und dem Operationscode bildet das Rechenwerk das Ergebnis und Weitergabe der Ergebnisadresse und eines Operations-Codes »RWO∈{-,R,L,S}« an die RW-Pipeline-Phase.



- RW (**R**esult **W**rite) Ergebnis schreiben: Wenn (W)=R« Adressierung des Registersatzes mit der Ergebnisadresse und Speichern des Ergebnisses.



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 mindestens genauso lange wie ohne Pipeline.
- Pipeline-Ergänzungen für LS-Befehle, Sprünge, ... folgen noch.

Pipeline-Auslastung

Pipeline-Auslastung

Verarbeitungsergebnisse werden erst zwei Takte nach Lesen der Operanden geschrieben. Verursacht Probleme. Beispiel:

Addition von vier Registerinhalten

Adr	Befehl
0	B1: addr r0,r0,r1
1	B2: addr r0,r0,r2
2	B3: addr r0,r0,r3

*	Register lesen
*n	lesen und mit n überschreiben
BW	Befehlswort

IF → OF → RW → EX

PC	BW	r_0	r_1	r_2	r_3	O1	O2	RR
0		7	2	11	17			
1	B1							
2	B2	*	*			7	2	
3	B3	*		*		7	11	9
4		*9			*	7	17	18
5		18						24
6		24						

In r_0 steht in Takt 5 $r_0 + r_1$, in Takt 6 $r_0 + r_2$ und in Takt 7 $r_0 + r_3$. Statt $r_0 + r_1 + r_2 + r_3$ wird $r_0 + r_3$, d.h. ein falsches Ergebnis berechnet.

Einfügen von noop[7]-Befehlen

```
Adr  Befehl
 0   B1: addr r0,r0,r1
 1   noop
 2   noop
 3   B2: addr r0,r0,r2
 4   noop
 5   noop
 6   B3: addr r0,r0,r3
```

*	Register lesen
*n	lesen und mit n überschreiben

PC	BW	r_0	r_1	r_2	r_3	O1	O2	RR
0		7	2	11	17			
1	B1							
2	noop							
3	noop							
4	B2							
5	noop							
6	noop							
7	B3							
8	noop							
9	noop							
10	noop							
11	noop							

- Wird die Summe der 4 Registerwerte richtig berechnet?
- Wie lange dauern jetzt die drei Additionen?

[7] noop – **No Op**eration, für den Beispielprozessor Op-Code 0x0000.

Lösung

Adr Befehl
0 B1: addr r0,r0,r1
1 noop
2 noop
3 B2: addr r0,r0,r2
4 noop
5 noop
6 B3: addr r0,r0,r3

*	Register lesen
*n	lesen und mit n überschreiben

IF → OF → RW → EX

PC	BW	r_0	r_1	r_2	r_3	O1	O2	RR
0		7	2	11	17			
1	B1							
2	noop	*	*			7	2	
3	noop							9
4	B2	9						
5	noop	*		*		9	11	
6	noop							20
7	B3	20						
8	noop	*			*	20	17	
9	noop							37
10	noop	37						
11	noop							

- Die Summe der 4 Registerwerte wird richtig berechnet
- Die drei Additionen benötigen bis zum Abschluss 11 Takte.

Optimierte Berechnungsreihenfolge

```
0  B1: addr r0,r0,r1
1  B2: addr r2,r2,r3
2  noop
3  noop
4  B3: addr r0,r0,r2
```

*	Register lesen
*n	lesen und mit n überschreiben

IF → OF → RW → EX

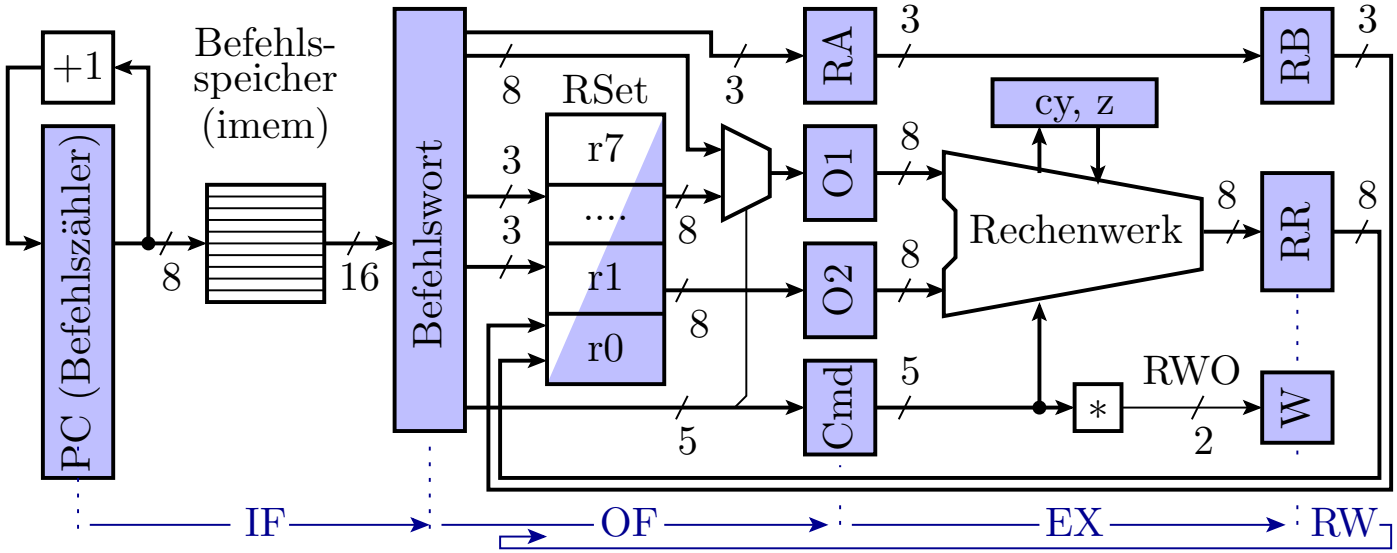
PC	BW	r_0	r_1	r_2	r_3	O1	O2	RR
0		7	2	11	17			
1	B1							
2	B2	*	*			7	2	
3	noop			*	*	11	17	9
4	noop	9						28
5	B3			28				
6	noop	*		*		9	28	
7	noop							37
8	noop	37						

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

Die Abarbeitungszeit und Größe von Programmen hängen erheblich von der Compiler-Optimierung ab.

Verarbeitungs-Pipeline

Darstellung des Prozessorzustands




```
PC| Befehlswort |Cmd |O1|O2|RR|c|z|RA|RB|W|r0 r1 r2 r3
01|ld_i r1,4a,..|noop|00|00|00|0|0|r0|r0|-|00 00 00 00
02|ld_i r0,73,..|....|..|..|..|.|.|..|..|.|.. .. .. ..
```

Register: Cmd – Operation; O1, O2 – Operanden; c, z – Flags; RR – Ergebnis; RA, RB – Ergebnisadresse; W∈{-, R, L, S} – RW-Operation; r0, r1, ... – Arbeitsregister.

Pipeline-Verarbeitung für Logikoperationen

Ergänzen Sie die Registerwerte.

PC	Befehlswort	Cmd	O1	O2	RR	c	z	RA	RB	W	r0	r1	r2	r3
01	ld_i r1,4a,..	noop	00	00	00	0	0	r0	r0	-	00	00	00	00
02	ld_i r0,73,..	....	..	..	..	.	.	..	..	.	..	..	..	..
03	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..
04	move r3,r1,..	....	..	..	..	.	.	..	..	.	..	..	..	..
05	move r2,r0,..	....	..	..	..	.	.	..	..	.	..	..	..	..
06	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..
07	andi r3,f0,..	....	..	..	..	.	.	..	..	.	..	..	..	..
08	andi r2,0f,..	....	..	..	..	.	.	..	..	.	..	..	..	..
09	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..
0a	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..
0b	or_r r4,r2,r3	....	..	..	..	.	.	..	..	.	..	..	..	..
0c	xorr r2,r3,r3	....	..	..	..	.	.	..	..	.	..	..	..	..
0d	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..
0e	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..

Lösung

PC	Befehlswort	Cmd	O1	O2	RR	c	z	RA	RB	W	r0	r1	r2	r3
01	ld_i r1,4a,..	noop	00	00	00	0	0	r0	r0	-	00	00	00	00
02	ld_i r0,73,..	ld_i	4a	00	00	0	0	r1	r0	-	00	00	00	00
03	noop ..,..,..	ld_i	73	00	4a	0	0	r0	r1	R	00	00	00	00
04	move r3,r1,..	noop	00	00	73	0	0	r0	r0	R	00	4a	00	00
05	move r2,r0,..	move	4a	00	73	0	0	r3	r0	-	73	4a	00	00
06	noop ..,..,..	move	73	00	4a	0	0	r2	r3	R	73	4a	00	00
07	andi r3,f0,..	noop	00	00	73	0	0	r0	r2	R	73	4a	00	4a
08	andi r2,0f,..	andi	f0	4a	73	0	0	r3	r0	-	73	4a	73	4a
09	noop ..,..,..	andi	0f	73	40	0	0	r2	r3	R	73	4a	73	4a
0a	noop ..,..,..	noop	00	00	03	0	0	r0	r2	R	73	4a	73	40
0b	or_r r4,r2,r3	noop	00	00	03	0	0	r0	r0	-	73	4a	03	40
0c	xorr r2,r3,r3	or_r	03	40	03	0	0	r4	r0	-	73	4a	03	40
0d	noop ..,..,..	xorr	40	40	43	0	0	r5	r4	R	73	4a	03	40
0e	noop ..,..,..	noop	00	00	00	0	1	r0	r5	R	73	4a	03	40

Cmd – Operationscode; O1, O2 – Operandenregister; c, z – Flags; RA, RB – Ergebnisadresse; W∈{-, R, L, S} – RW-Operation; r0, r1, ... – Arbeitsregister.

Beispielaufgabe 2-Byte-Addition

```
r0:r1 = 733A; r2:r3 = 13E7
r4:r5 = r0:r1 + r2:r3     (Ergebnis: 8721)
r4:r5 = 8623 - r4:r5      (Ergebnis: FF02)
```

PC	Befehlswort	O1	O2	RR	c	z	RA	RB	W	r0	r1	r2	r3	r4	r5
01	ld_i r1,3a,..	00	00	00	0	0	r0	r0	-	00	00	00	00	00	00
02	ld_i r0,73,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
03	ld_i r3,e7,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
04	ld_i r2,13,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
05	noop ..,..,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
06	addr r5,r1,r3	..	..	..	.	.	..	..	.	..	..	..	..	..	..
07	adcr r4,r0,r2	..	..	..	.	.	..	..	.	..	..	..	..	..	..
08	noop ..,..,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
09	subi r5,23,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
0a	sbci r4,86,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
0b	noop ..,..,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
0c	noop ..,..,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..
0d	noop ..,..,..	..	..	..	.	.	..	..	.	..	..	..	..	..	..

Lösung

```
r0:r1 = 733A; r2:r3 = 13E7
r4:r5 = r0:r1 + r2:r3     (Ergebnis: 8721)
r4:r5 = 8623 - r4:r5      (Ergebnis: FF02)
```

PC	Befehlswort	01	02	RR	c	z	RA	RB	W	r0	r1	r2	r3	r4	r5
01	ld_i r1,3a,..	00	00	00	0	0	r0	r0	-	00	00	00	00	00	00
02	ld_i r0,73,..	3a	00	00	0	0	r1	r0	-	00	00	00	00	00	00
03	ld_i r3,e7,..	73	00	3a	0	0	r0	r1	R	00	00	00	00	00	00
04	ld_i r2,13,..	e7	00	73	0	0	r3	r0	R	00	3a	00	00	00	00
05	noop ..,..,..	13	00	e7	0	0	r2	r3	R	73	3a	00	00	00	00
06	addr r5,r1,r3	00	00	13	0	0	r0	r2	R	73	3a	00	e7	00	00
07	adcr r4,r0,r2	3a	e7	13	0	0	r5	r0	-	73	3a	13	e7	00	00
08	noop ..,..,..	73	13	21	1	0	r4	r5	R	73	3a	13	e7	00	00
09	subi r5,23,..	00	00	87	0	0	r0	r4	R	73	3a	13	e7	00	21
0a	sbci r4,86,..	23	21	87	0	0	r5	r0	-	73	3a	13	e7	87	21
0b	noop ..,..,..	86	87	02	0	0	r4	r5	R	73	3a	13	e7	87	21
0c	noop ..,..,..	00	00	ff	1	0	r0	r4	R	73	3a	13	e7	87	02
0d	noop ..,..,..	00	00	ff	1	0	r0	r0	-	73	3a	13	e7	ff	02

LS-Pipeline

Lade- / Speicherbefehle des Minimalprozessors

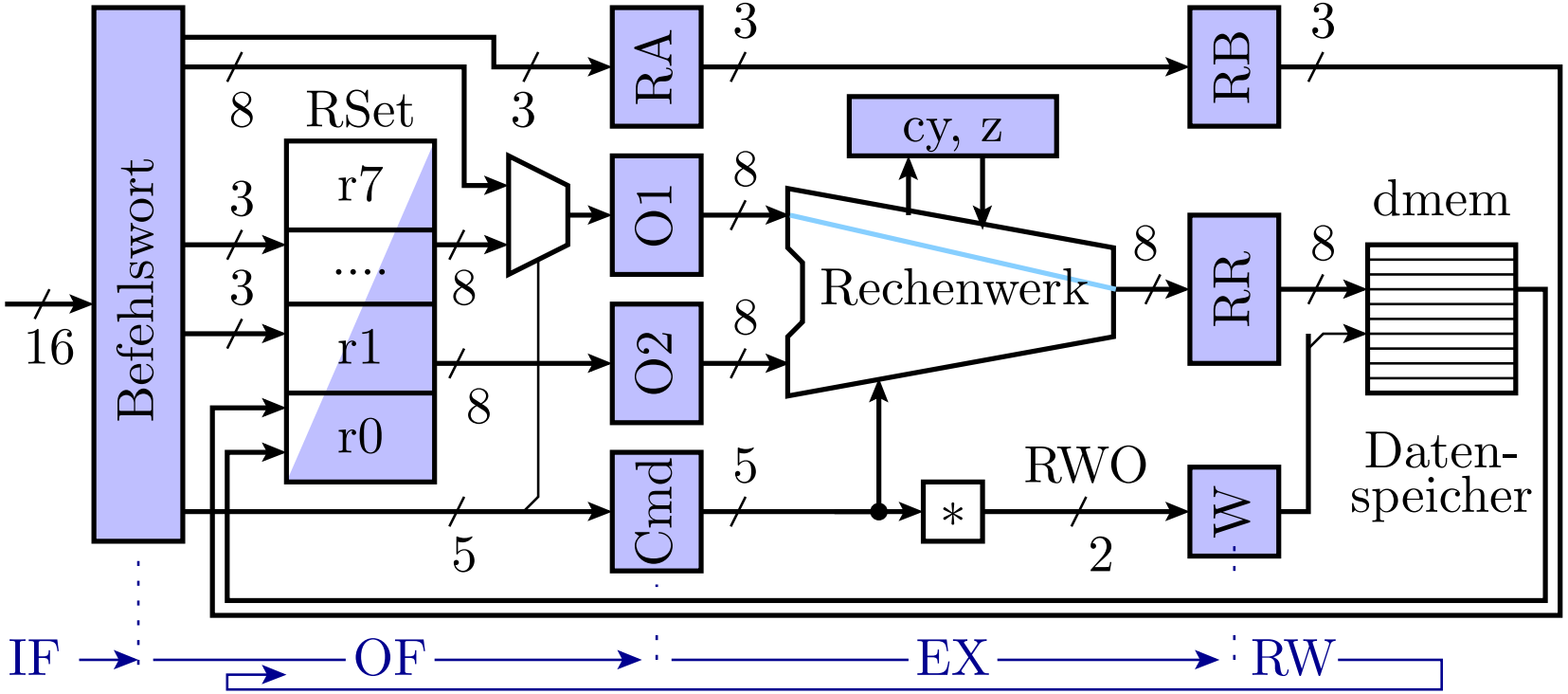
Teilbitvektor	15 ... 11	10 9 8	7 6 5	4 3 2	1 0
cmd rd,ra	cnr	rd	ra		
cmd rd,imm	cnr	rd	imm		

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

Unterstütze Adressierungsarten:

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

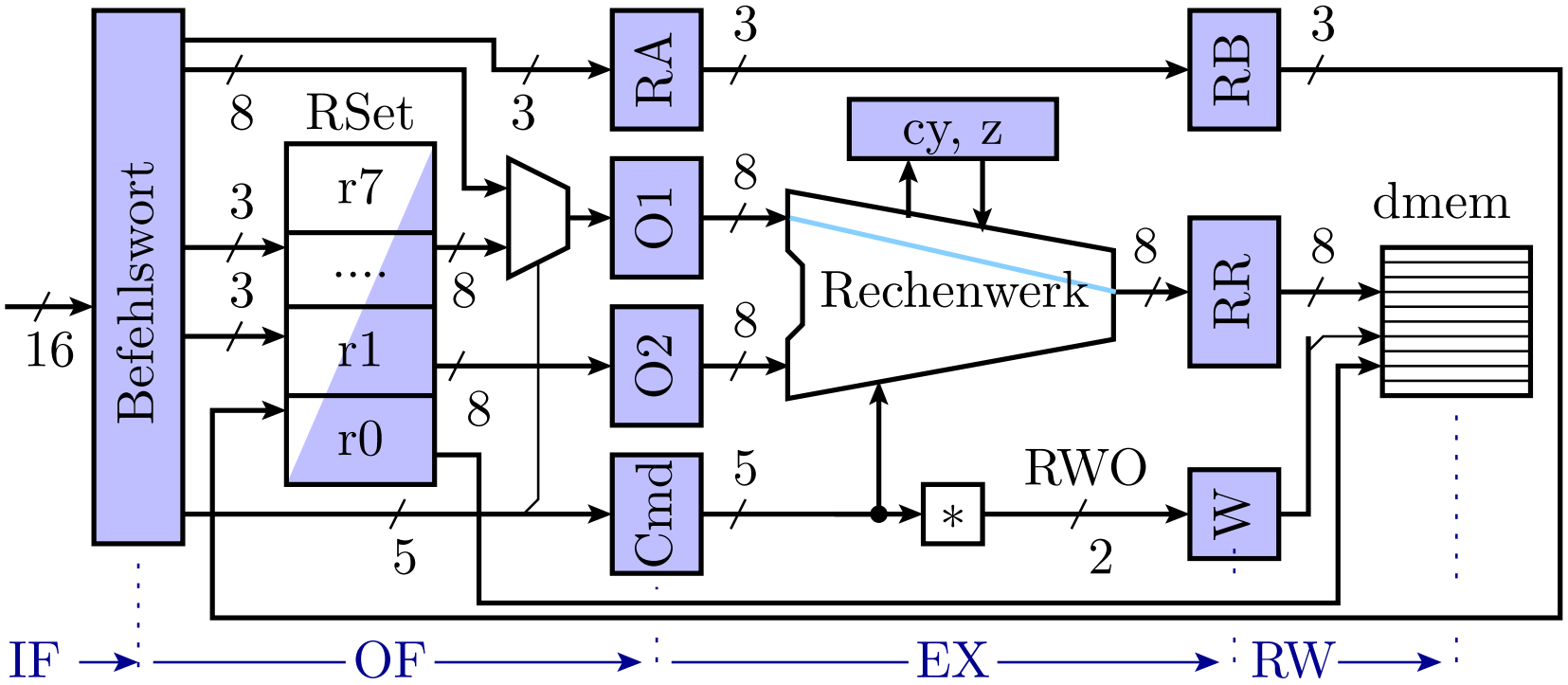
Lade-Pipeline



* Bildung des 2-Bit-Operationswortes für die RW-Phase

- EX-Phase: Adressrechnung, Weitergabe RWO-Code für Laden.
- RW-Phase: Laden des Ergebnisregisters mit Speicherinhalt, gesteuert durch RWO=L.

Speicher-Pipeline



- EX-Phase: Adressrechnung, Weitergabe RWO-Code Speichern.
- RW-Phase: Kopieren Ergebnisregisterinhalt in den Speicher, gesteuert durch RWO=S.

Testbeispiel mit Lade- und Speicherbefehlen

```
dmem(5) = 0x48;
r1 = 6; dmem(r1)= 0x31;
r3 = dmem(5);
r4 = dmem(r1);
```

Programmiert für Pipeline-Verarbeitung:

```
0000: ld_i r0,48,..; r0 = 0x48
0001: stor r0,05,..; dmem(5) = r0
0002: ld_i r1,06,..; r1 = 0x06 (Adresse)
0003: noop ..,..,..
0004: ld_i r2,31,..; r2 = 0x31 (Daten)
0005: noop ..,..,..
0006: st_r r2,r1,..; dmem(r1) = r2
0007: load r3,05,..; r3 = dmem(5)
0008: ld_r r4,r1,..; r4 = dmem(r1)
0009: noop ..,..,..
```

Aufgabe

Ergänzen Sie die Register- und Datenspeicherwerte.

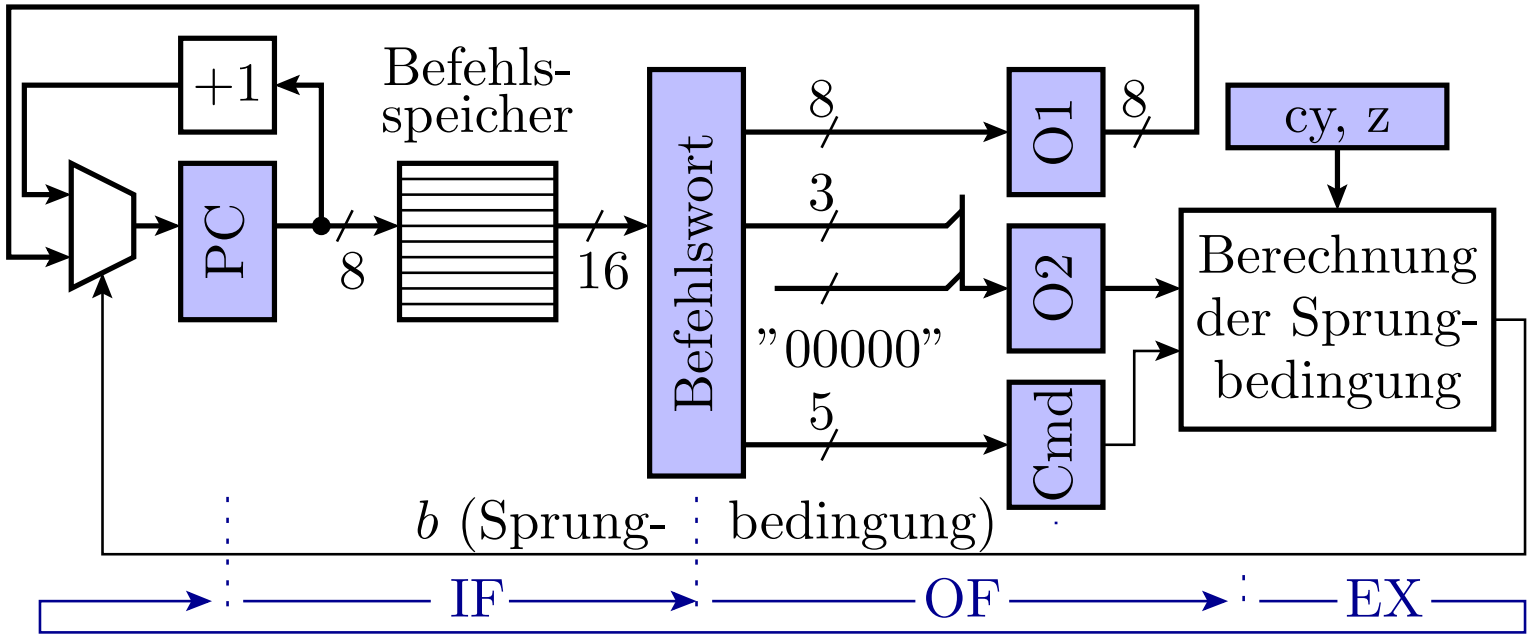
PC	Befehlswort	Cmd	O1	O2	RR	c	z	RA	RB	W	r0	r1	r2	r3	r4
01	ld_i r0,48,..	noop	00	00	00	0	0	r0	r0	-	00	00	00	00	00
02	stor r0,05,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
03	ld_i r1,06,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
04	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
05	ld_i r2,31,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
dmem[0:7] =						[..	..	..	..	..	..	..	..	..]	
06	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
07	st_r r2,r1,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
08	load r3,05,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
09	ld_r r4,r1,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
0a	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
dmem[0:7] =						[..	..	..	..	..	..	..	..	..]	
0b	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..
0c	noop .. ,.. ,..	....	..	..	..	.	.	..	..	.	..	..	..	..	..

Lösung

```
PC| Befehlswort |Cmd |O1|O2|RR|c|z|RA|RB|W|r0 r1 r2 r3 r4
01|ld_i r0,48,..|noop|00|00|00|0|0|r0|r0|-|00 00 00 00 00
02|stor r0,05,..|ld_i|48|00|00|0|0|r0|r0|-|00 00 00 00 00
03|ld_i r1,06,..|stor|05|00|48|0|0|r0|r0|R|00 00 00 00 00
04|noop ..,..,..|ld_i|06|00|05|0|0|r1|r0|S|48 00 00 00 00
05|ld_i r2,31,..|noop|00|00|06|0|0|r0|r1|R|48 00 00 00 00
dmem[0:7] =                          [00 00 00 00 00 48 00 00]
06|noop ..,..,..|ld_i|31|00|06|0|0|r2|r0|-|48 06 00 00 00
07|st_r r2,r1,..|noop|00|00|31|0|0|r0|r2|R|48 06 00 00 00
08|load r3,05,..|st_r|06|00|31|0|0|r2|r0|-|48 06 31 00 00
09|ld_r r4,r1,..|load|05|00|06|0|0|r3|r2|S|48 06 31 00 00
0a|noop ..,..,..|ld_r|06|00|05|0|0|r4|r3|L|48 06 31 00 00
dmem[0:7] =                          [00 00 00 00 00 48 31 00]
0b|noop ..,..,..|noop|00|00|06|0|0|r0|r4|L|48 06 31 48 00
0c|noop ..,..,..|noop|00|00|06|0|0|r0|r0|-|48 06 31 48 31
```

Sprung-Pipeline

Sprung-Pipeline




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

- OF-Phase: Sprungziel in O1 und Sprungbedingung in O2 laden.
- EX-Phase: bedingte Übernahme von O1 in den Befehlszähler.
- RW-Phase: keine Operation.

Beispielprogramm mit einer Schleife

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

Programmiert für Pipeline-Verarbeitung:

```
0x00: ld_i r0,01,..  ; r0 = 0x01
0x01: ld_i r1,34,..  ; r1 = 0x34
0x02: noop ..,..,..  ; warte, bis r0 geladen ist
0x03: comp r0,03,..  ; vergleiche r0 mit 0x3
0x04: st_r r1,r0,..  ; dmem(r0) = r1;
0x05: jump 02,lth..  ; wenn r0<0x3 springe zu 0x2
0x06: addi r0,01,..  ; r0 = r0+1  (Delayslot 1)
0x07: subr r1,r1,r0  ; r1 = r1+r0 (Delayslot 2)
0x08: noop ..,..,..  ; 1. Anw. nach der Schleife
```

Programmabarbeitung

```
PC| Befehlswort |Cmd |O1|O2|RR|c|z|RA|RB|W|r0 r1 r2 r3 r4
01|ld_i r0,01,..|noop|00|00|00|0|0|r0|r0|-|00 00 00 00 00

1. Schleifendurchlauf:

02|ld_i r1,34,..|ld_i|01|00|00|0|0|r0|r0|-|00 00 00 00 00
03|noop ..,..,..|ld_i|34|00|01|0|0|r1|r0|R|00 00 00 00 00
04|comp r0,03,..|noop|00|00|34|0|0|r0|r1|R|01 00 00 00 00
05|st_r r1,r0,..|comp|03|01|34|0|0|r0|r0|-|01 34 00 00 00
06|jump 02,lth..|st_r|01|00|02|0|0|r1|r0|-|01 34 00 00 00
07|addi r0,01,..|jump|02|07|01|0|0|r7|r1|S|01 34 00 00 00

2. Schleifendurchlauf:

02|subr r1,r1,r0|addi|01|01|01|0|0|r0|r7|-|01 34 00 00 00

Abschluss 1. Schreibop.: dmem[0:7]=[00 34 00 00 00 00 00]

03|noop ..,..,..|subr|34|01|02|0|0|r1|r0|R|01 34 00 00 00
04|comp r0,03,..|noop|00|00|33|0|0|r0|r1|R|02 34 00 00 00
05|st_r r1,r0,..|comp|03|02|33|0|0|r0|r0|-|02 33 00 00 00
```

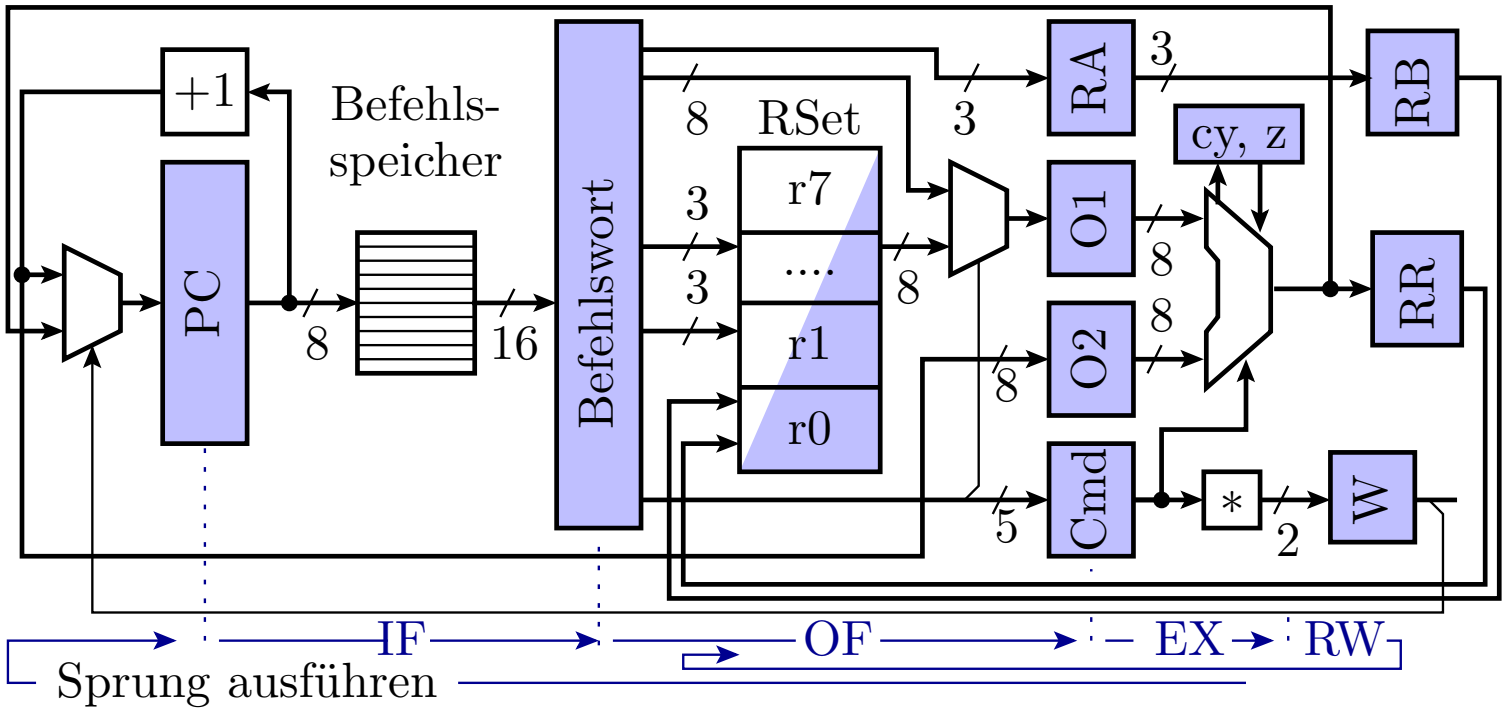
Fortsetzung der Programmabarbeitung

```
PC| Befehlswort |Cmd |O1|O2|RR|c|z|RA|RB|W|r0 r1 r2 r3 r4
05| Fortsetzung nach Laden des Spungbefehls
06|jump 02,lth..|st_r|02|00|01|0|0|r1|r0|-|02 33 00 00 00
07|addi r0,01,..|jump|02|07|02|0|0|r7|r1|S|02 33 00 00 00
3. Schleifendurchlauf:
02|subr r1,r1,r0|addi|01|02|02|0|0|r0|r7|-|02 33 00 00 00
Abschluss 2. Schreibop.: dmem[0:7]=[00 34 33 00 00 00 00]
03|noop ..,..,..|subr|33|02|03|0|0|r1|r0|R|02 33 00 00 00
04|comp r0,03,..|noop|00|00|31|0|0|r0|r1|R|03 33 00 00 00
05|st_r r1,r0,..|comp|03|03|31|0|0|r0|r0|-|03 31 00 00 00
06|jump 02,lth..|st_r|03|00|00|0|1|r1|r0|-|03 31 00 00 00
07|addi r0,01,..|jump|02|07|03|0|1|r7|r1|S|03 31 00 00 00
08|subr r1,r1,r0|addi|01|03|03|0|1|r0|r7|-|03 31 00 00 00
Abschluss 3. Schreibop.: dmem[0:7]=[00 34 33 31 00 00 00]
09|noop ..,..,..|subr|31|03|04|0|0|r1|r0|R|03 31 00 00 00
0a|noop ..,..,..|noop|00|00|2e|0|0|r0|r1|R|04 31 00 00 00
```

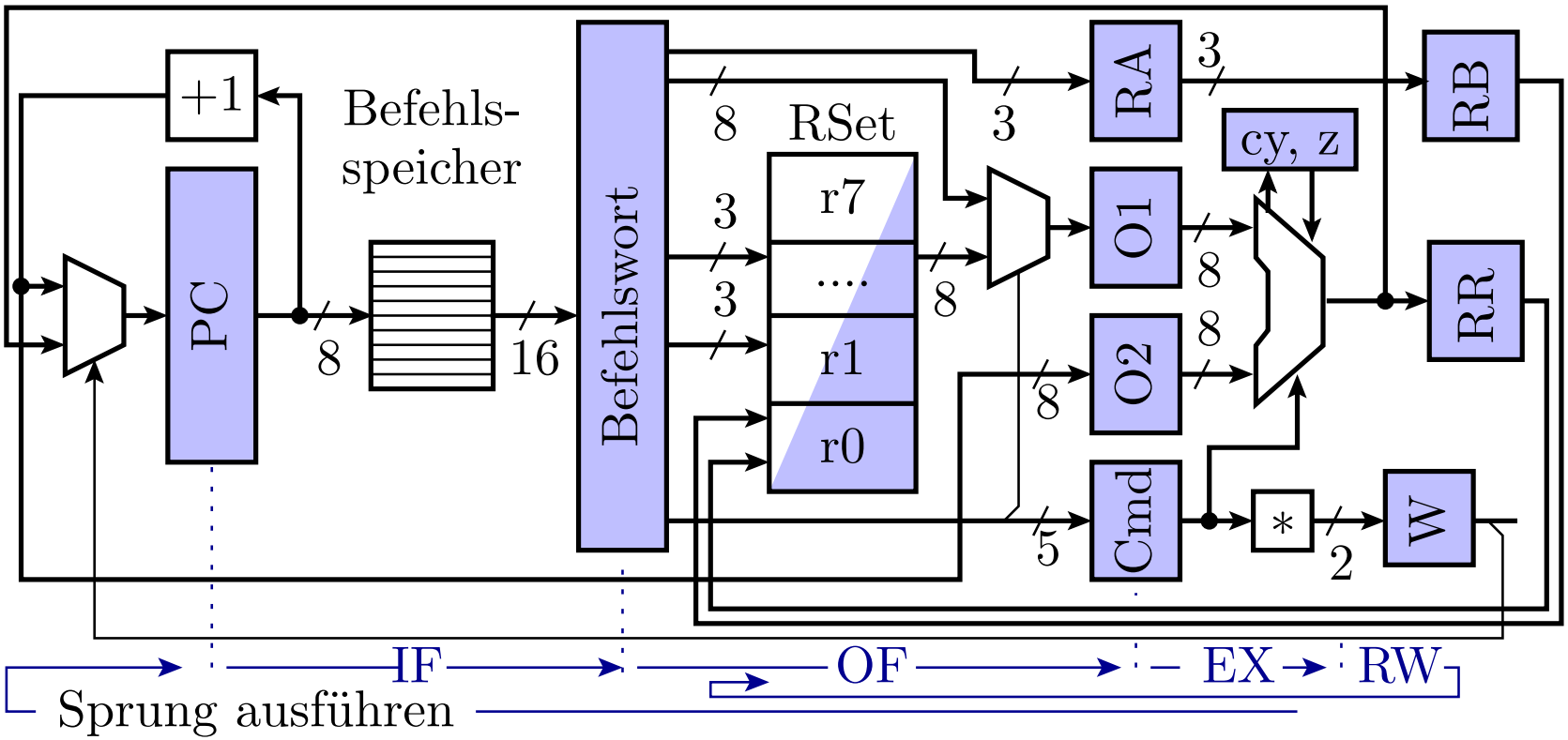
Unterprogramme

Unterprogrammaufruf und Rücksprung

Ein Unterprogrammaufruf speichert die Rückkehradresse in einem Register und der Rücksprung liest das Sprungziel aus einem Register.



call rd,imm	rd:=pc+1, pc:=imm
retu rd	pc := rd



- OF-Phase: Direktwert (call) oder Registerinhalt (retu), in O1. Befehlszähler+1 in O2, Zieladresse in RA (nur retu).
- EX-Phase: Sprungausführung. Rückkehradresse und Registeradresse weiterreichen (nur call).
- RW-Phase: Rücksprungadresse in Register speichern (nur call).

Unterprogrammaufrufe mit Pipeline

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

```
0000: ld_i r0,35,..          Unterprogramm:
0001: stor r0,01,..          0010: load r3,01,..
0002: ld_i r1,02,..          0013: addi r3,13,..
0003: call r5,10,..          0014: st_r r3,r1,..
0006: ld_i r0,46,..          0015: retu r5,..,..
0007: stor r0,01,..
0008: ld_i r1,04,..          (Restliche Befehle noop)
0009: call r5,10,..
000c: jump 08,alw.. ; Endlosschleife
```

Testbeispiele:

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

```
PC| Befehlswort |Cmd |O1|O2|RR|RA|RB|W|r0 r1 r2 r3 r4 r5
01|ld_i r0,35,..|noop|00|00|00|r0|r0|-|00 00 00 00 00 00
02|stor r0,01,..|ld_i|35|00|00|r0|r0|-|00 00 00 00 00 00
03|ld_i r1,02,..|stor|01|00|35|r0|r0|R|00 00 00 00 00 00

Unterprogrammaufruf mit dmem(1)=0x35 und r1=2

04|call r5,10,..|ld_i|02|00|01|r1|r0|S|35 00 00 00 00 00
05|noop ..,..,..|call|10|04|02|r5|r1|R|35 00 00 00 00 00

Abschluss Schreibop. Aufruf 1: dmem = [00 35 00 00 00 ..]

10|noop ..,..,..|noop|00|00|04|r0|r5|R|35 02 00 00 00 00
11|load r3,01,..|noop|00|00|04|r0|r0|-|35 02 00 00 00 04
12|noop ..,..,..|load|01|00|04|r3|r0|-|35 02 00 00 00 04
13|noop ..,..,..|noop|00|00|01|r0|r3|L|35 02 00 00 00 04
14|addi r3,13,..|noop|00|00|01|r0|r0|-|35 02 00 35 00 04
15|st_r r3,r1,..|addi|13|35|01|r3|r0|-|35 02 00 35 00 04
16|retu r5,..,..|st_r|02|00|48|r3|r3|R|35 02 00 35 00 04
17|noop ..,..,..|retu|04|00|02|r5|r3|S|35 02 00 48 00 04
04|noop ..,..,..|noop|00|00|02|r0|r5|-|35 02 00 48 00 04

Abschluss Schreibop. UP1: dmem = [00 35 48 00 00 00 ..]
```

```
PC| Befehlswort |Cmd |O1|O2|RR|RA|RB|W|r0 r1 r2 r3 r4 r5
... Abschluss Schreibop. UP1:
05|noop ..,..,..|noop|00|00|02|r0|r0|-|35 02 00 48 00 04
06|noop ..,..,..|noop|00|00|02|r0|r0|-|35 02 00 48 00 04
07|ld_i r0,46,..|noop|00|00|02|r0|r0|-|35 02 00 48 00 04
08|stor r0,01,..|ld_i|46|00|02|r0|r0|-|35 02 00 48 00 04
09|ld_i r1,04,..|stor|01|00|46|r0|r0|R|35 02 00 48 00 04

Unterprogrammaufruf mit dmem(1)=0x46 und r1=4

0a|call r5,10,..|ld_i|04|00|01|r1|r0|S|46 02 00 48 00 04
0b|noop ..,..,..|call|10|0a|04|r5|r1|R|46 02 00 48 00 04

Abschluss Schreibop. Aufruf 2: dmem = [00 46 48 00 00 ..]

10|noop ..,..,..|noop|00|00|0a|r0|r5|R|46 04 00 48 00 04
11|load r3,01,..|noop|00|00|0a|r0|r0|-|46 04 00 48 00 0a
12|noop ..,..,..|load|01|00|0a|r3|r0|-|46 04 00 48 00 0a
13|noop ..,..,..|noop|00|00|01|r0|r3|L|46 04 00 48 00 0a
14|addi r3,13,..|noop|00|00|01|r0|r0|-|46 04 00 46 00 0a
15|st_r r3,r1,..|addi|13|46|01|r3|r0|-|46 04 00 46 00 0a
16|retu r5,..,..|st_r|04|00|59|r3|r3|R|46 04 00 46 00 0a
...
```

```
PC| Befehlswort |Cmd |O1|O2|RR|RA|RB|W|r0 r1 r2 r3 r4 r5
...
17|noop ..,..,..|retu|0a|00|04|r5|r3|S|46 04 00 59 00 0a
0a|noop ..,..,..|noop|00|00|04|r0|r5|-|46 04 00 59 00 0a
Abschluss Schreibop. UP2: dmem = [00 46 48 00 59 00 ..]
0b|noop ..,..,..|noop|00|00|04|r0|r0|-|46 04 00 59 00 0a
0c|noop ..,..,..|noop|00|00|04|r0|r0|-|46 04 00 59 00 0a
Endlosschleife mit 2 Delay-Slots
0d|jump 0c,alw..|noop|00|00|04|r0|r0|-|46 04 00 59 00 0a
0e|noop ..,..,..|jump|0c|01|04|r1|r0|-|46 04 00 59 00 0a
0c|noop ..,..,..|noop|00|00|04|r0|r1|-|46 04 00 59 00 0a
0d|jump 0c,alw..|noop|00|00|04|r0|r0|-|46 04 00 59 00 0a
```