Rechnerarchitektur 1, Foliensatz 1 Einführung, Rechnermodelle, Verarbeitungsbausteine

G. Kemnitz

Institut für Informatik, TU Clausthal (RA-F1)

21. Januar 2016

Inhalt und Organisation

Funktionsweise und Programmierung von Rechnern am Beispiel eines 8-Bit-Mikrokontrollers:

- wie funktioniert die Hardware,
- wie werden Programme abgearbeitet.

Übung:

- Programmieren in C, Programme testen,
- dissasemblieren und Code verstehen.

Organisation der Lehrveranstaltung:

- ab 17.12.2015 jede Woche Vorlesung,
- jede Woche Hausübungen (Abgabe zur nächsten Vorlesung),
- ab der 2. Vorlesung (07.01.2016) Test am Vorlesungsende über den Inhalt der Hausübungen.
- ab 6./7.01.2016 jede Woche Laborübung.

Leistungsnachweises, Foliensätze

Leistungsnachweises, Erwerb:

- In allen bis auf einem Kurztests mindestens 40% und insgesamt mindesten 50% der Punkte und
- in allen bis auf einer Laborübung mindestens 60% der Punkte.

Alternativ[1] mündliche Kenntnisprüfung über den Übungs- und Vorlesungsstoff[2].

Foliensätze:

1. Einführung, Rechnermodelle, Verarbeitungsbausteine.
2. Zeitabläufe.
3. Schnittstellen und Zusatzwerke.

[1] Auch bei entschuldigter Abwesenheit z.B. Krankheit.

[2] Termine beim Dozent erfragen. Prüfungstermine bis max. Ende Februar.

Inhalt des Foliensatzes

Einführung

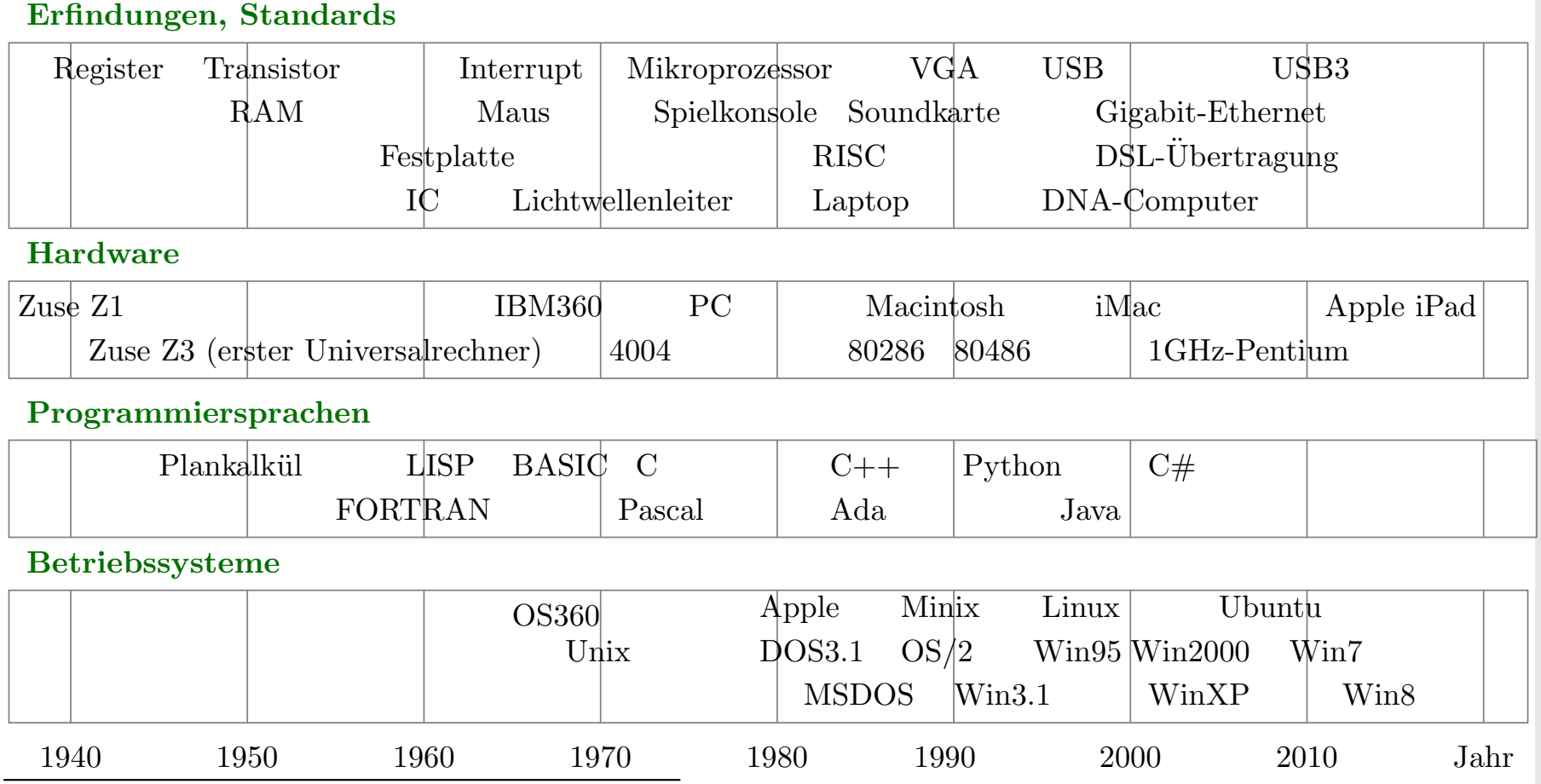
Rechnerachitektur als Teilgebiet der Informatik

Fakt 1

Informatik ist die Wissenschaft von der Automatisierung intellektueller Aufgaben.

Software	Schnittstelle SW ⇔HW	Hardware
Formulierung der Aufgaben in einer von einer Maschine abarbeitbaren Form • Algorithmen • Datenstrukturen • ...	• Programmier-sprachen • Betriebssysteme • Programmier-werkzeuge • ...	Konstruktion von Maschinen zur Ab-arbeitung intellek-tueller Aufgaben • Rechnerarchitektur • Schaltungstechnik • ...

Informatik ist eine junge Wissenschaft[3]



[3] Es gibt auch Quellen, die den Beginn der Informatik auf die Erfindung des Abakus um 1100 v. Chr. im indo-chinesischen Kulturraum vordatieren.

Rechnermodelle

Der Begriff »Modell« in der Informatik

- Selbst die einfachsten Sachverhalte in der Informatik wie die Abarbeitung eines Befehls werden sehr schnell kompliziert, wenn alle Details berücksichtigt werden.

Definition 2

Modell
Ein Modell ist ein Mittel, um einen Zusammenhang zu veranschaulichen. Es stellt die wesentlichen Sachverhalte dar und verbirgt unwesentliche Details.

- Von-Neumann-Modell: Veranschaulicht die Zusammenarbeit zwischen Speicher und Prozessor.
- RT (Register-Transfer-) Modell: Abbildung von Registerzuständen und Eingaben auf Ausgaben und Registerfolgezustände über Verarbeitungsfunktionen.

- Programmiermodell: Register-Transfer-Modell reduziert auf die für den Programmierer sichtbaren Register und Abarbeitungsschritte (keine Beschreibung der unsichtbaren Register und Zwischenschritte, für den 8-Bit-Beispielprozessor der Vorlesung ca. 500 Seiten Dokumentation).
- Assemberprogrammiermodell: Abstraktion der Register-Transfer-Operationen auf symbolische Befehle wie

  ```
  add r1, r2
  ```
- C-Programmiermodell: Architekturunabhängige Beschreibung der Zielfunktion, aus der ein Compiler Programme generiert, die fast so effizient wie handgeschrieben sind.
- Rechner als digitale Schaltung: Tausende bis millionen von Bausteinen. Zu groß für eine ganzheitliche Betrachtung.
- ...

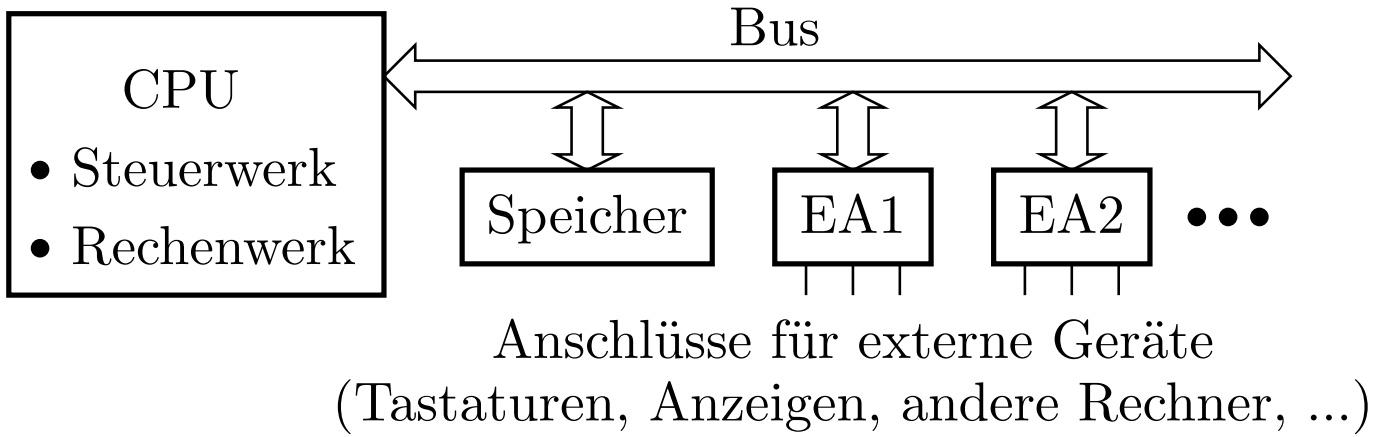
Die Vorlesung wird sich durch all diese Modellebenen schlängeln, mal den einen und mal den anderen Aspekt hervorheben und immer viele Details als unwesentlich vernachlässigen.

Von-Neumann-Architektur

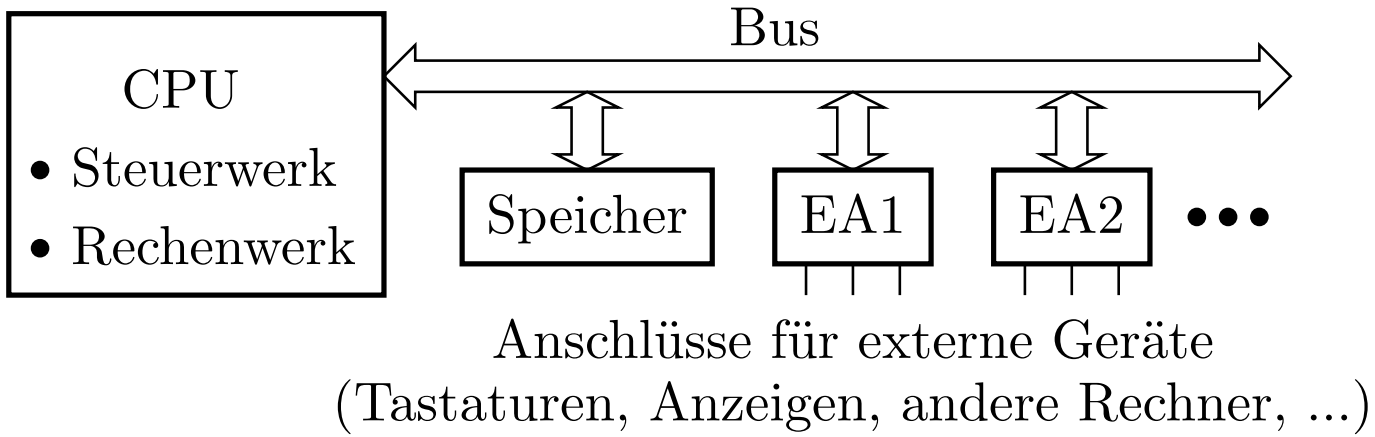
Von-Neumann-Architektur

John von Neumann[4] wird die erste Veröffentlichung des Konzept einer universell programmierbaren Maschine zugeschrieben. Eine solche Maschine besteht aus den Grundbausteinen:

- Zentraleinheit (Central Processing Unit, CPU)
- Speicher für die Befehle und Daten
- EA- (Ein-/Ausgabe-) Einheiten.

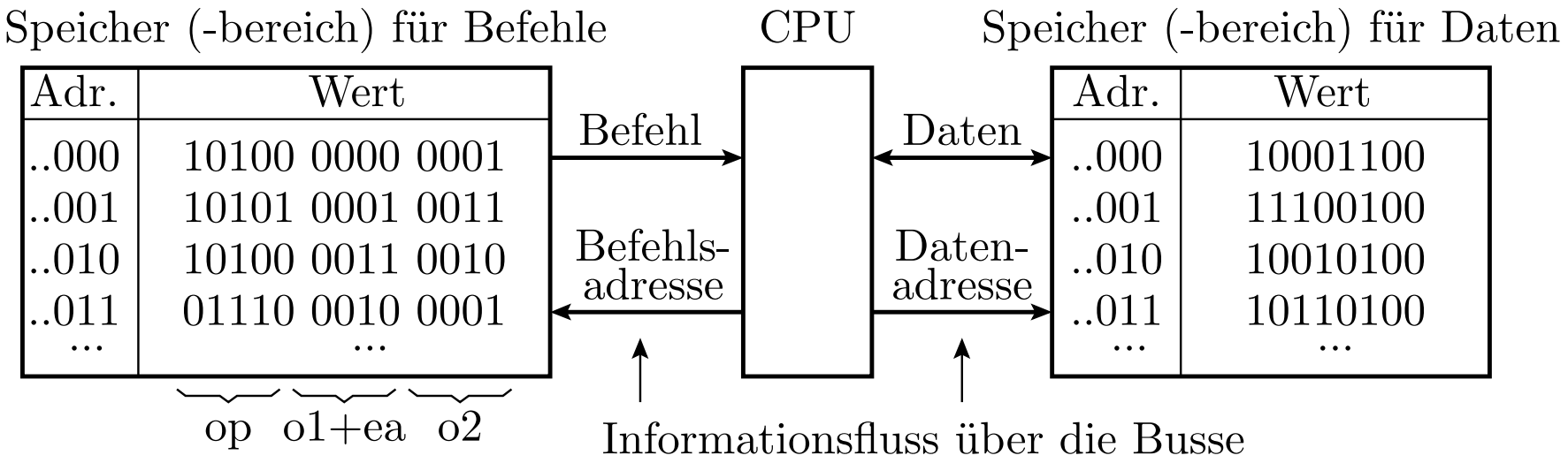


[4] John von Neumann: First Draft of a Report on the EDVAC. (PDF, engl.; 421 kB) 1945



- Die CPU holt nacheinander je einen Befehl aus dem Speicher, holt die Daten dazu, führt den Befehl aus, schreibt das Ergebnis in den Speicher.
- Der Speicher ist eine Tabelle adressierbarer Plätze mit fester Bitanzahl je Speicherplatz.
- Ein-/Ausgabeeinheiten sind aus Rechnersicht schreib- und / oder lesbare Speicherplätze, auf deren Werte außer der CPU auch externe Einheiten zugreifen können.
- Die Funktionseinheiten sind über Busse verbunden, die sich in Steuer-, Adress-, Befehls- und Datenbusse unterteilen.

Abarbeitung einer Befehlsfolge[5]



Der Prozessor wiederholt fortlaufend:

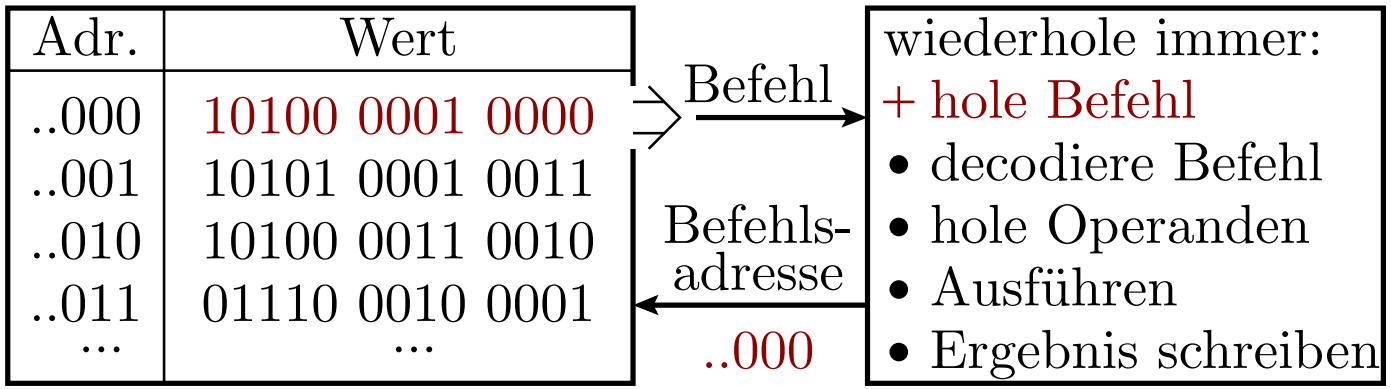
- Befehl aus dem Befehlsspeicher holen.
- Befehl decodieren.
- Operanden aus dem Datenspeicher holen.
- Die Operation ausführen.
- Das Ergebnis schreiben.

o1	Adresse Operand 1
o2	Adresse Operand 2
ea	Ergebnis-adresse
op	Operation
10100	Addition
10101	Subtraktion
...	...

[5] Auf einem fiktiven Rechnermodell.

Abarbeitung des ersten Befehls

- Befehlsadresse »..000« senden und Befehlswort »101..« holen:

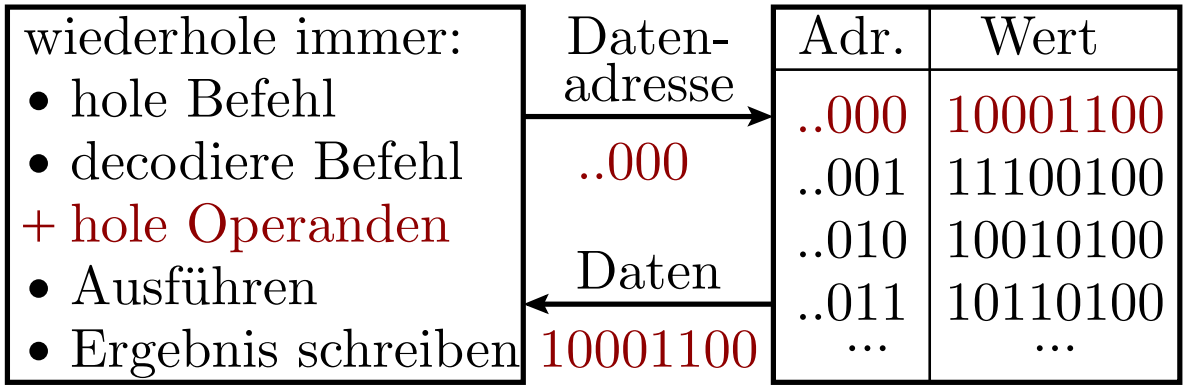


- Decodieren des Befehlsworts:

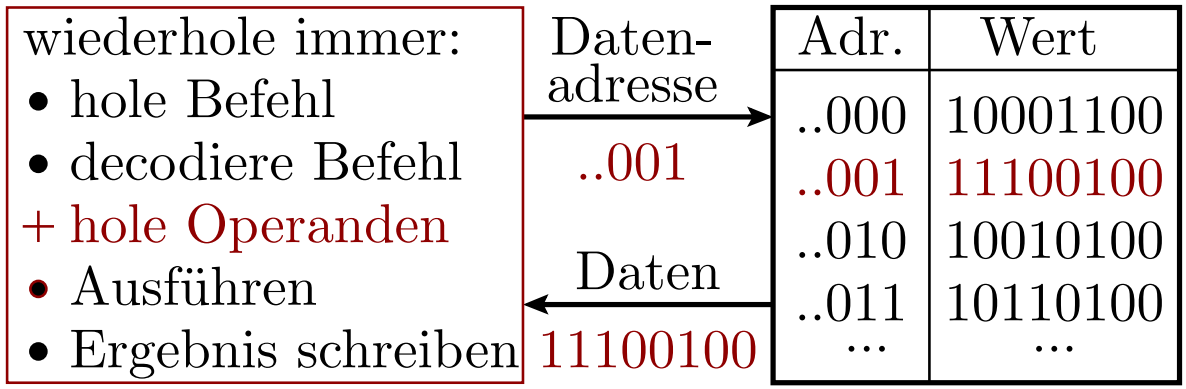
wiederhole immer:
- hole Befehl
+ decodiere Befehl
- hole Operanden
- Ausführen
- Ergebnis schreiben

gesamtes Befehlswort	10100 0001 0000
Operationscode Addion:	10100
Adresse Summand 1 und Summe:	0001
Adresse Summand 2:	0000

- Ersten Summanden von Adresse »0000« lesen:



- Zweiten Summanden von Adresse »0001« lesen:



- Addition ausführen:

wiederhole immer:
• hole Befehl
• decodiere Befehl
• hole Operanden
+ Ausführen
• Ergebnis schreiben

```
   10001100
+  11100100
-----------
(1) 01110000
```

- Ergebnis auf Adresse »0000« schreiben:

wiederhole immer:
• hole Befehl
• decodiere Befehl
• hole Operanden
• Ausführen
+ Ergebnis schreiben

Daten-adresse → ..000

Daten → 01110000

Adr.	Wert
..000	01110000
..001	11100100
..010	10010100
..011	10110100
...	...

Abarbeitung des nächsten Befehls

Nach Abarbeitung des Additionsbefehls wird der Befehlszähler weitergeschaltet und das nächste Befehlswort geholt:

gesamtes Befehlswort	10101 0001 0011
Operationscode Subtraktion:	10101
Adresse Minuend und Ergebnis:	0001
Adresse Subtrahend:	0011

Was passiert in den Folgeschritten?

Lade Operand 1:	Adresse:	Wert:
Lade Operand 2:	Adresse:	Wert:
Operation:		
Schreibe Ergebnis:	Adresse:	Wert:
Hole Befehlswort:	Adresse:	Wert:

RISC-Prozessor

RISC-Prozessoren

Die meisten in den letzten 30 Jahren entwickelten Prozessoren haben eine sog. RISC-Architektur. RISC ist ein Akronym für Reduced Instruction Set Computer. Die Befehlssätze sind auf Befehle reduziert, die sich mit einer oder wenigen Register-Transfer-Funktionen nachbilden lassen. Komplexere Operationen aus vielen hintereinander auszuführenden RT-Operationen bildet der Compiler durch Befehlsfolgen nach.

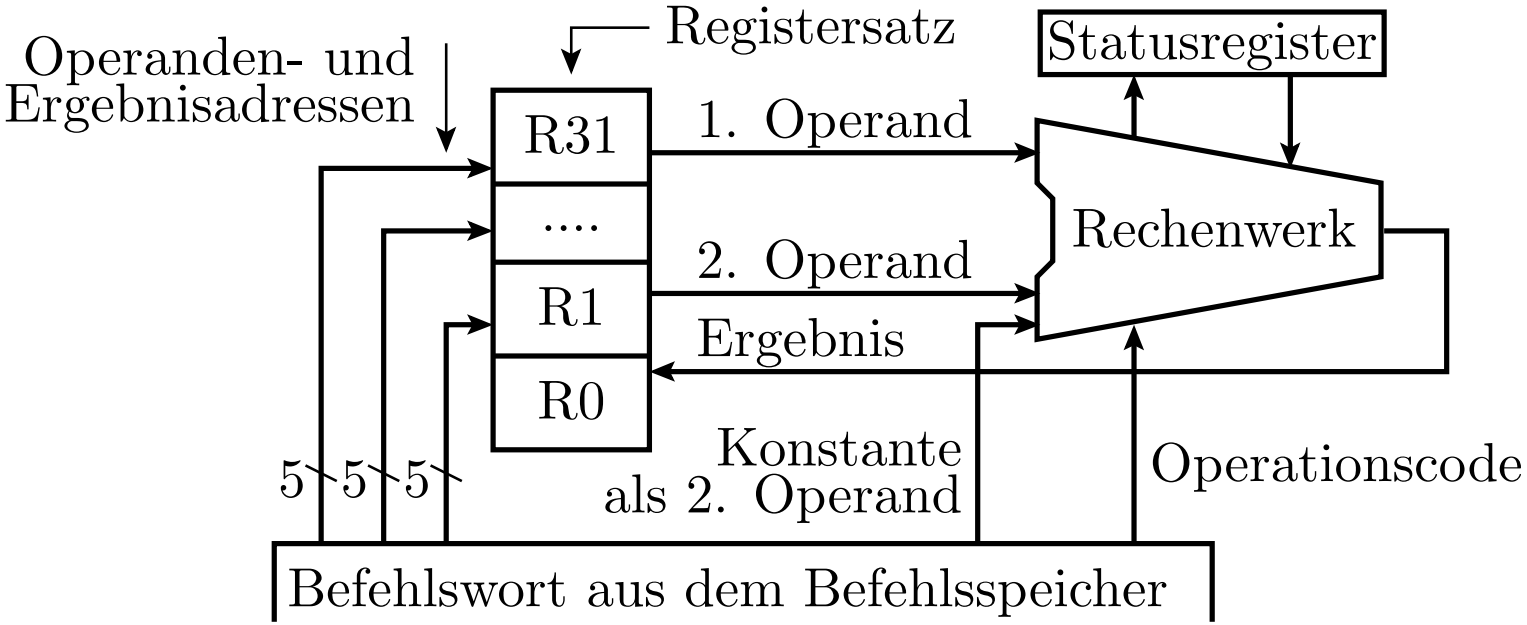
Komplexen Operationsfolge, die auf älteren Prozessoren ein Befehl war und für die ein RISC-Prozessor 3 Befehle braucht:

- Variablenwert in ein Arbeitsregister kopieren,
- Registerinhalt um eins erhöhen und
- geänderten Registerinhalt zurückschreiben.

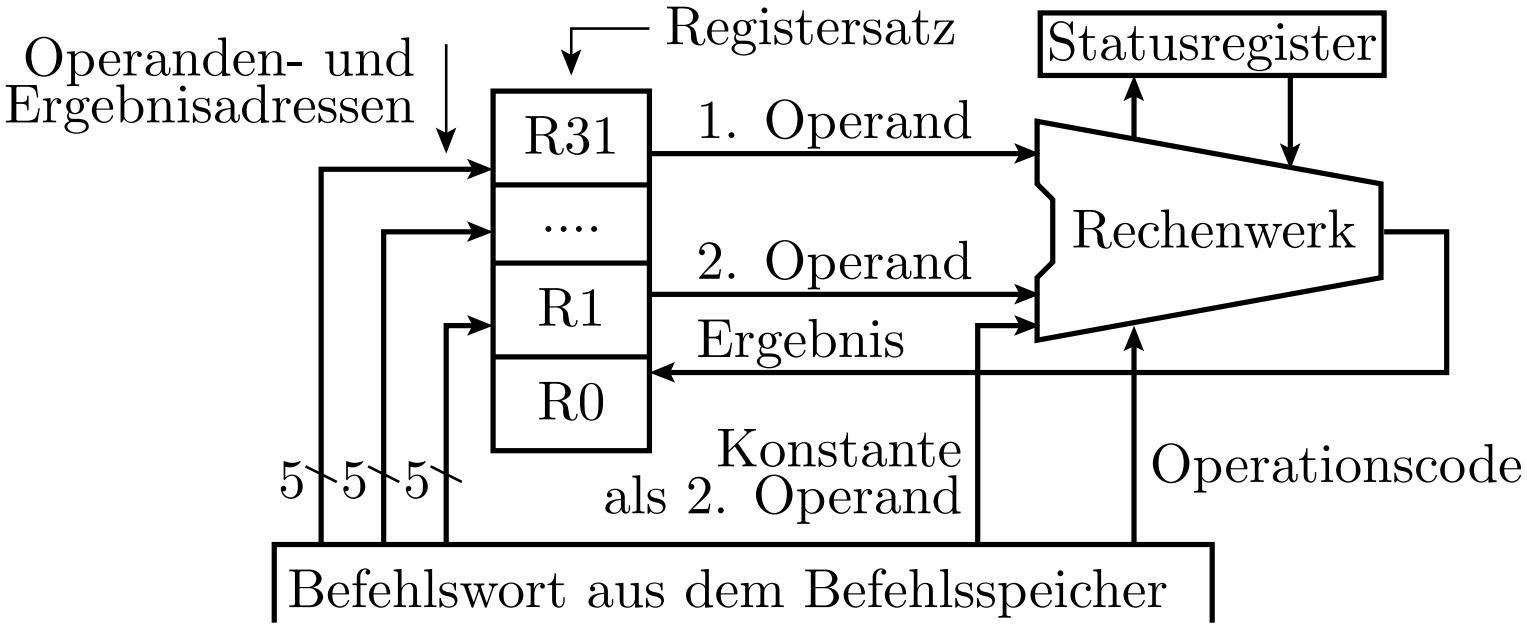
Verarbeitungswerk

Das Verarbeitungswerk

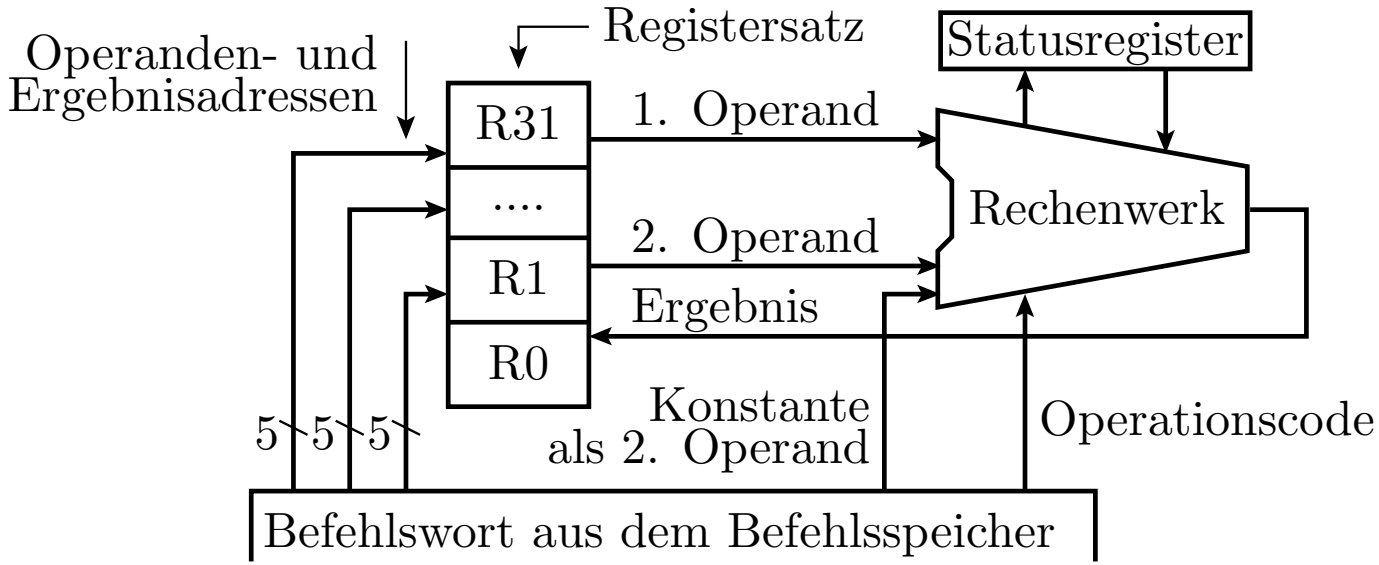
In einem RISC-Prozessor besteht das Verarbeitungswerk aus einem Satz von (meist 32) Registern, einem Rechenwerk und einem Statusregister.



Der Registersatz erlaubt das zeitgleiche Lesen von zwei Operanden und das Schreiben eines Ergebnisses.



- Das Rechenwerk führt eine durch den Operationscode ausgewählte Operation aus. Der zweite Operand kann auch eine Konstante sein, die im Befehlswort steht.
- Das Statusregister speichert Zusatzdaten zum Ergebnis:
  - den Übertrag von Additionen und Subtraktionen für Nachfolgeoperationen,
  - >0, =0, ... für nachfolgende bedingte Sprünge,
  - ...



Für die Befehlsausführung mit einer (wenigen) RT-Funktionen müssen alle Befehlsbestandteile (Adressen, Op-Code und Konstante in ein Befehlswort passen[6]. Typ. Befehlsaufbau:

- 32 Bit Befehlswort: 3×5 Bits für Adressen, 6-Bit Op-Code, alternativ 16-Bit-Konstante statt einer 5-Bit-Adresse.
- 16 Bit Befehlswort: 2×5 Adressbits[7], 6-Bit Op-Code; bzw. 1×4 Adressbits, 4-Bit-Op-Code, 8-Bit-Konstante.

[6] Damit das Befehlswort mit einem Speicherzugriff gelesen werden kann.
[7] Zieladresse fest oder gleich einer der Operandenadressen.

Speicherzugriff

Datenspeicher

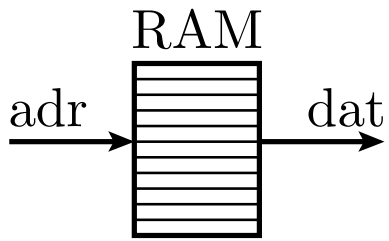
- Die 32 Arbeitsregister sind für die meisten Aufgaben zu wenig. Erweiterung um einen schreib- und lesbaren Blockspeicher (RAM).
- Ein RAM hat Adress- und Steuereingänge sowie Datenanschlüsse und wird als programmierbare Tabelle modelliert.
- Zum Lesen wird die Adresse und der Steuercode für Lesen angelegt und am Datenanschluss die Daten übernommen.
- Zum Schreiben werden die Adresse, die Daten und der Steuercode für Schreiben angelegt.

Speicher als Tabelle

Adresse	Daten
..000	0110 1000
..001	0101 1101
..010	1100 1000
...	...

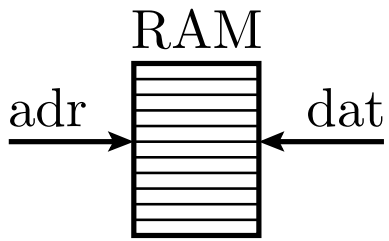
Leseoperation

$dat \leftarrow mem(adr)$

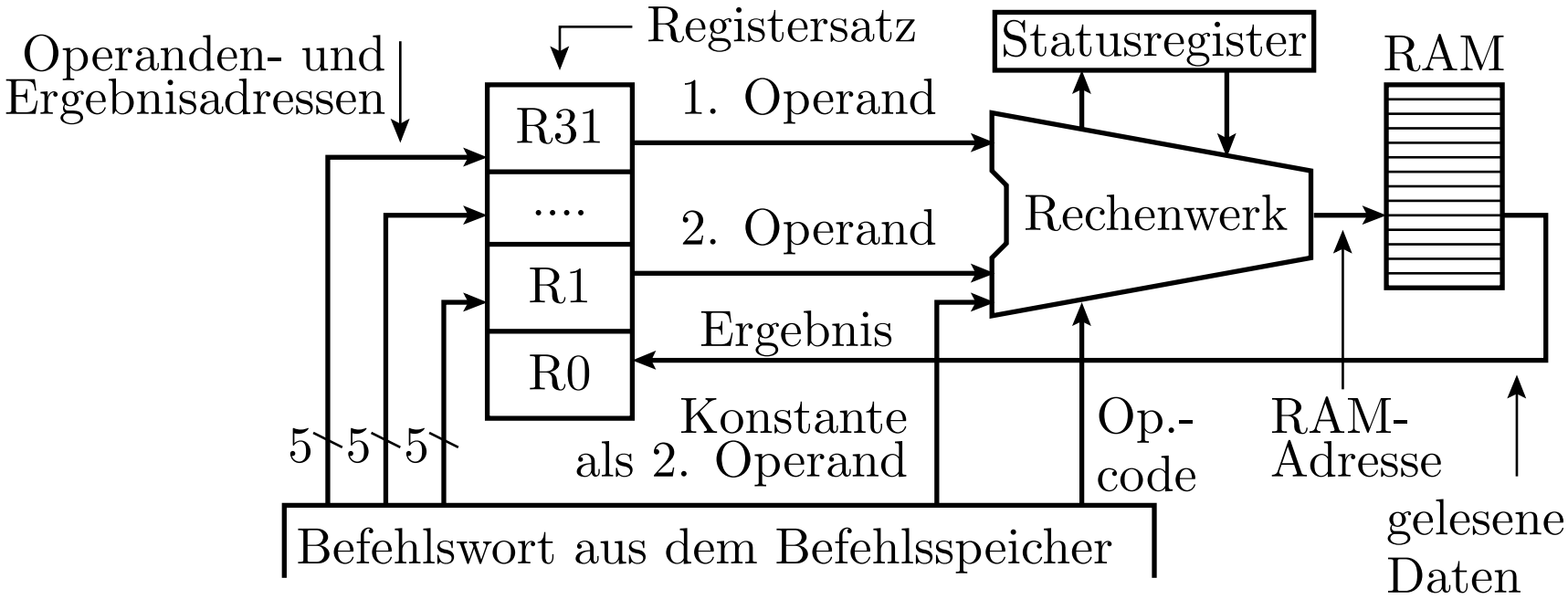


Schreiboperation

$mem(adr) \leftarrow dat$

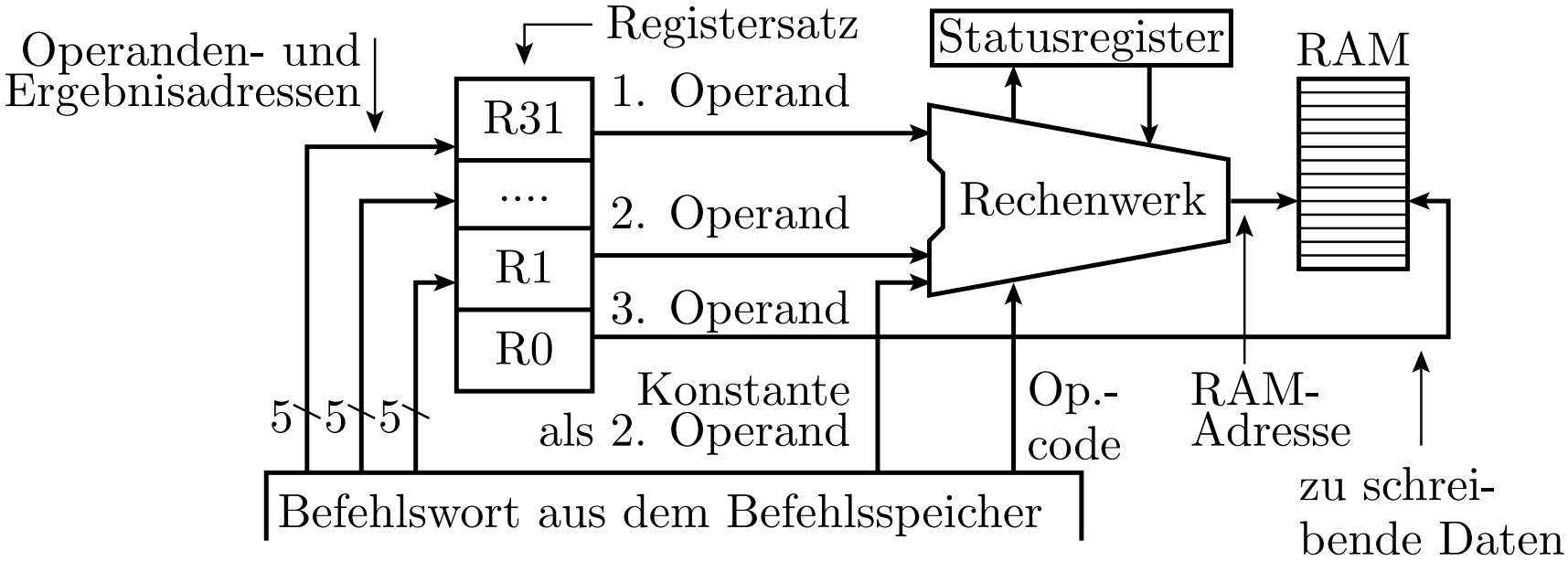


Ladeoperation



- Das Rechenwerk bildet die Adresse, z.B. Durchreichen eines Registerinhalts, Addition von zwei Registerinhalten oder Registerinhalt plus Konstante.
- Der gelesene Wert wird in das Ergebnisregister geschrieben.

Speicheroperation



- Wie Ladeoperation, nur dass die Ergebnisadresse für die Auswahl eines dritten Operanden verwendet wird, den zu speichernden Wert.

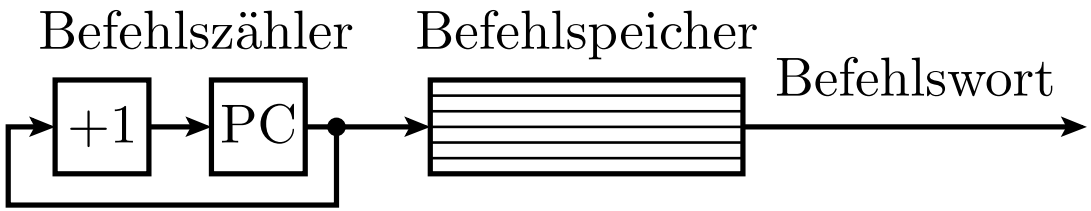
EEPROM

- Ein RAM und auch die Prozessorregister verlieren ihre Daten bei Abschaltung der Versorgungsspannung. Moderne Rechner haben deshalb oft einen zusätzlichen nicht flüchtigen Speicher, einen batteriegestützten RAM oder einen EEPROM (Electrical Programmable and Erasable Read Only Memory).
- Ein EEPROM unterscheidet sich von einem RAM darin, dass das Schreiben tausende Zeitschritte (Takte) benötigt.
- EEPROMs werden deshalb vom Rechner wie Ein- und Ausgabegeräte angesprochen, d.h. über EA-Register (siehe später Foliensatz 3).

Befehlsspeicher

Der Befehlsspeicher wird bei Programmabarbeitung nur gelesen. Moderne Mikrokontroller verwenden dafür vorzugsweise Flash-Speicher. Ein Flash-Speicher ähnelt in seiner Funktion einem EEPROM, nur werden beim Löschen große Speicherbereiche statt einzelne Worte auf einmal gelöscht.

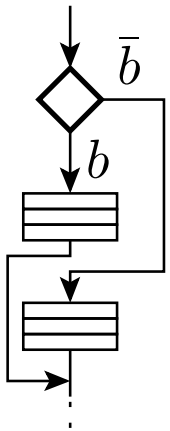
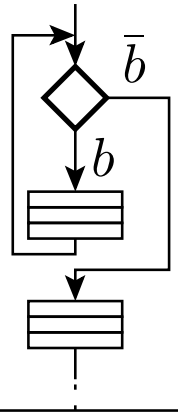
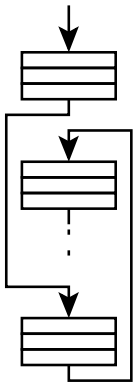
- Ansteuerung zur Programmierung wie ein Ein-/Ausgabegerät.
- Zur Programmabarbeitung Adressierung durch den Befehlszähler, der nach jedem Schritt (im Normalfall) um eins erhöht wird:



Sprungbefehle

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
Arbeite, wenn b wahr ist, eine sonst eine andere Befehlsfolge ab.	Wiederhole solange b wahr ist eine Befehlsfolge.	Füge eine Befehlsfolge ab einer anderen Adresse ein.
		
bedingter Sprung unbedingter Sprung	bedingter Sprung unbedingter Sprung	Unterprogrammaufruf (Sprung) Rücksprung

Register-Transfer-Operationen bei Sprüngen

- Absoluter Sprung:

  `PC` $\leftarrow$ `Rr`^{1}

- relativer Sprung

  `PC` $\leftarrow$ `PC + 1 + Konstante`

- bedingter Sprung

  `wenn b dann PC` $\leftarrow$ `PC + 1 + Konstante`
  `sonst PC` $\leftarrow$ `PC + 1`

- Unterprogrammaufruf:

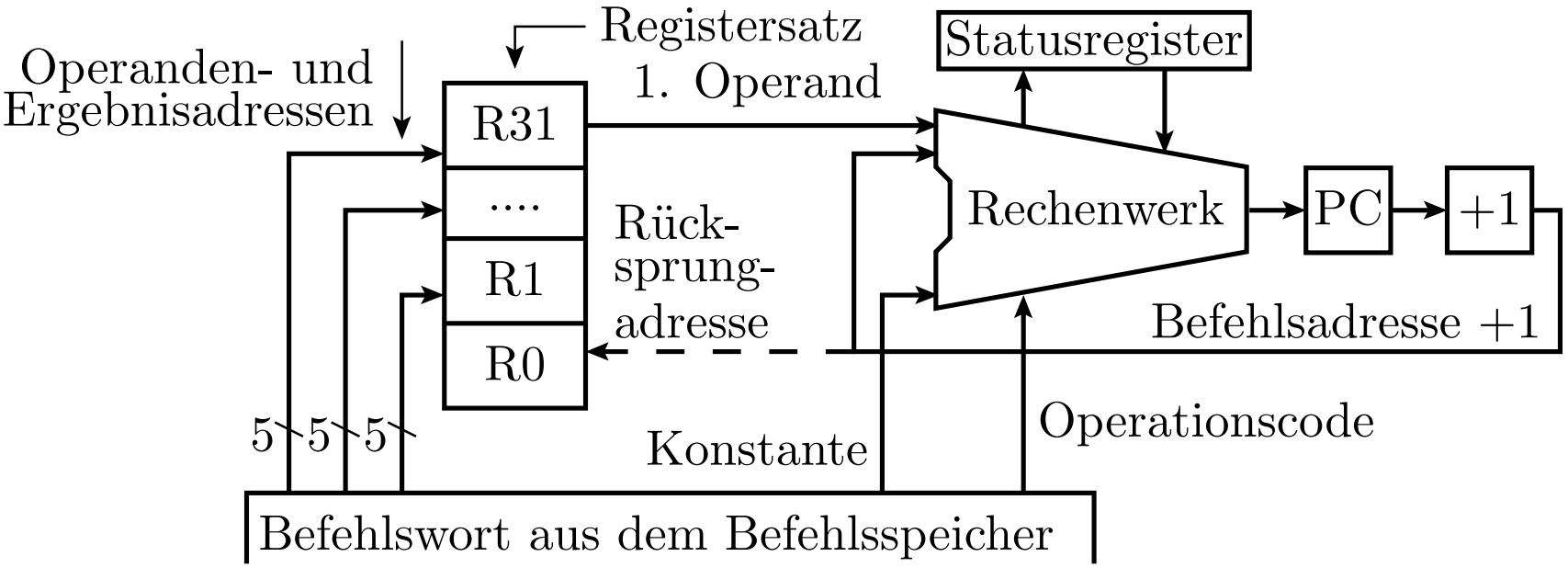
  `Rd`$^2\leftarrow$ `PC + 1; PC` $\leftarrow$ `Rr`^{1}

- Rücksprung aus einem Unterprogramm:

  `PC` $\leftarrow$ `Rr`^{3}

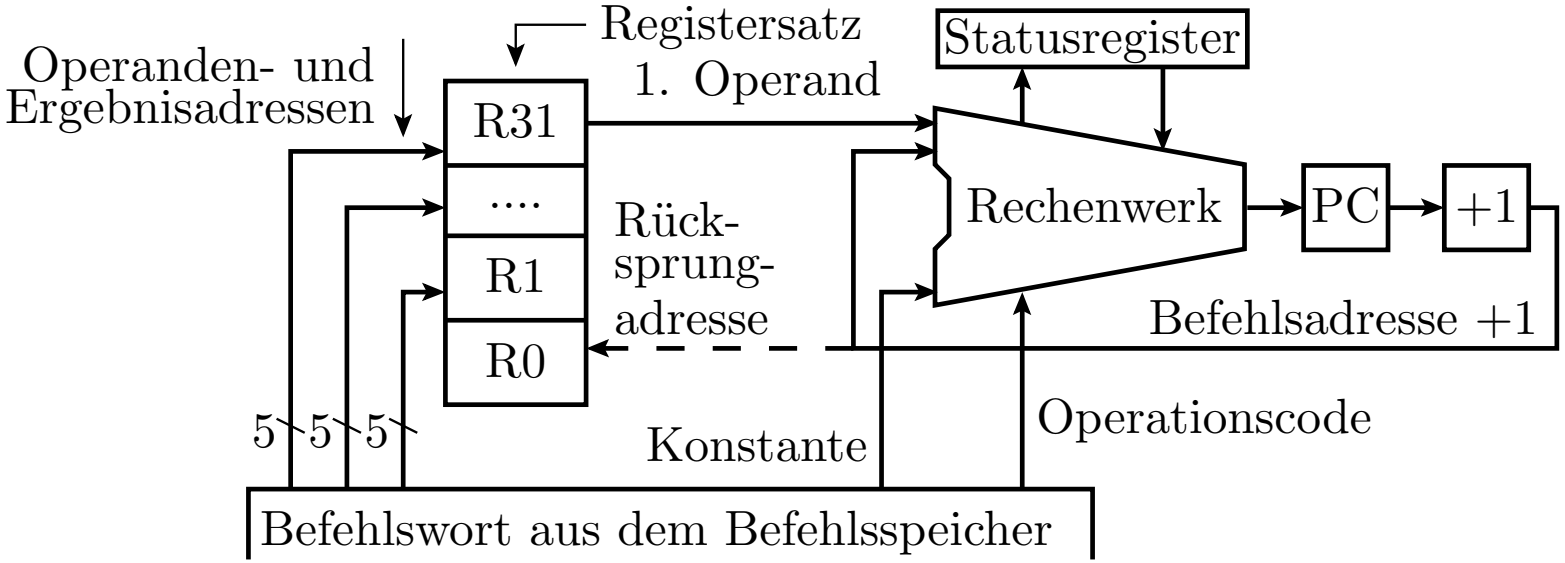
Rr^1 – Register, in das zuvor das Sprungziel geladen wird, Rd^2 – Register, in dem die Rücksprungadresse gespeichert wird; Rr^3 – Register mit der Rücksprungadresse).

Sprungwerk



- Bei absoluten Sprüngen ist der 1. Operanden des Rechenwerks ein Registerinhalt, bei relativen Sprüngen »Befehlsadresse +1«. Das Zielregister ist der Befehlszähler (PC[8]).
- Operationen sind unverändertes Durchreichen des ersten Operanden oder Addition der Konstanten.

[8] Program Counter.



- Bei bedingten Sprüngen sind die Bedingungen, ob oder ob nicht addiert wird, Bitwerte aus dem Statusregister, die anzeigen, ob das vorherige Ergebnis null, größer null etc. war.
- Bei Unterprogrammaufrufen wird zusätzlich die Folgeadresse als Rücksprungadresse gespeichert.
- Komplexe Operationen wie das Ablegen und Zurückladen der Rücksprungadresse auf und vom Stack oder der Sprung zu einer Adresskonstanten verlangen Befehlsfolgen.

Befehlssatz

Typische Beispielbefehle

Der Befehlssatz beschreibt für alle Befehlsworte die Operation, die ausgeführt wird. Die Lehrveranstaltung verwendet den AVR-Befehlssatz[9]. Befehlswortgröße 16 Bit. Einige Beispielbefehle:

Operation	Op.-Code	Assembler
Rd←I/O(A) (Eingabe)	1011 0AAd dddd AAAA	`in Rd, A`
I/O(A) ←Rr (Ausgabe)	1011 1AAr rrrr AAAA	`out A, Rd`
`Rd← 0 (Register löschen)`	0010 01dd dddd dddd[(1)]	`clr Rd`
`Rd← 0xFF (Register setzen)`	1110 1111 dddd 1111[(2)]	`ser Rd`
`PC←PC+k+1 (relativer Sprung)`	1100 kkkk kkkk kkkk	`rjmp k`
`PC←PC+k+1 (rel. UP-Aufruf`[(3)]`)`	1101 kkkk kkkk kkkk	`rcall k`
`I`[(4)]`←0 (Interrupts aus)`	1001 0100 1111 1000	`cli`

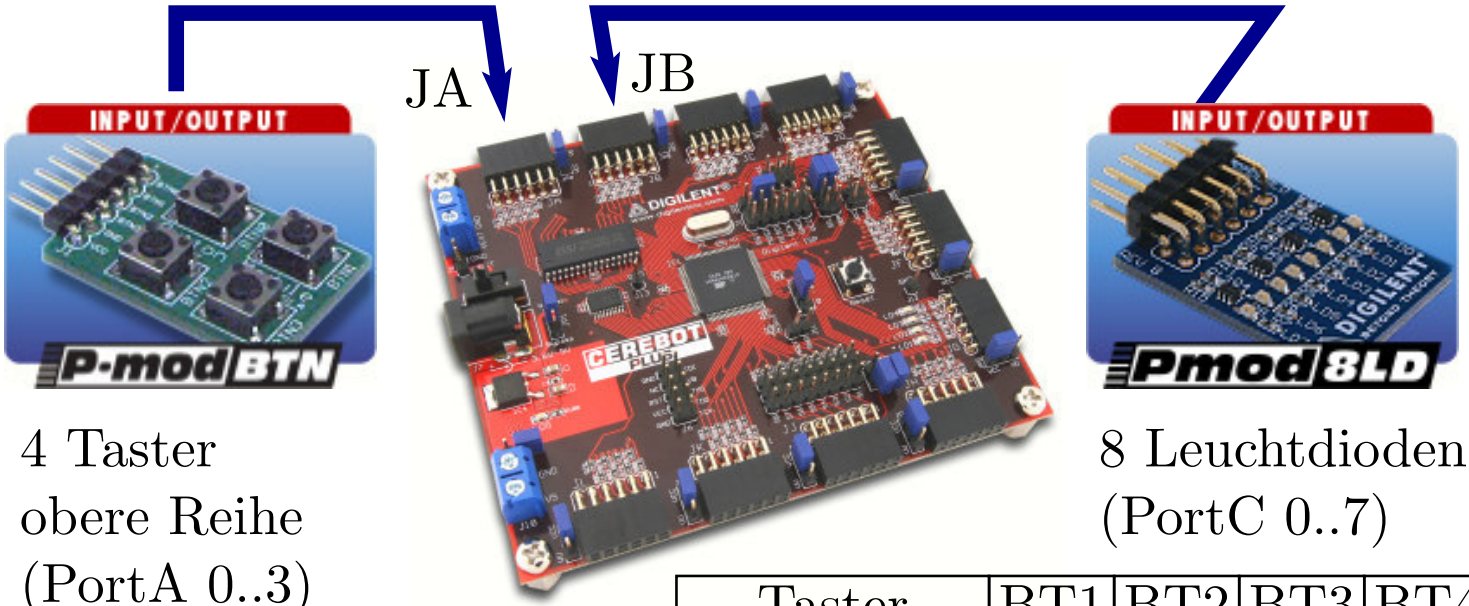
A – I/O-Adresse 0..63; k – Sprungdistanz -2024..2047; [(1)]EXOR mit gleichen Registern; [(2)]nur für R16 bis R31; [(3)]plus Rücksprungadresse auf Stack; [(4)] Spezielles EA-Registerbit.

[9]Programmieren in C und Reengineering des Maschinenprogramms.

Erstes Beispielprogramm

Testobjekt Mikrorechnerbaugruppe

Ein Mikrokontroller ist zwar ein kompletter Rechner, aber ohne Tastatur und Bildschirm.



4 Taster
obere Reihe
(PortA 0..3)

8 Leuchtdioden
(PortC 0..7)

Taster	BT1	BT2	BT3	BT4
Anschluss	PA0	PA1	PA2	PA3

Leuchtdiode	LD0	LD1	···	LD7
Anschluss	PC0	PC1	···	PC7

Anschluss von

- 4 Tasten
- 8 Leuchtdioden

Programmierung der Ports

- Der in der Übung verwendete Mikrorechner ATmega2560 hat 12 Ports. Jeder Port $x \in \{\mathrm{A}, \mathrm{B}, ..., \mathrm{L}\}$ hat drei Register:
  - Richtungsregister: DDRx
  - Ausgaberegister: PORTx
  - Eingaberegister: PINx
- Eingänge: DDx, Bit i null lassen. Wert von PINx lesen.
- Ausgänge: DDx, Bit i eins setzen. Wert auf PORTx schreiben.

IO-Adressen der Port-Register des ATmega2560:

	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	•

Die zweite um 0x20 größere 16-Bit Adresse ist die für normale Lese-/Schreiboperationen des Datenspeichers.

Das erste Programmbeispiel

C-Programm, das fortlaufen die Tasterwerte liest und auf die LEDs ausgibt:

```
 9  #include <avr/io.h>
10  void main(){
11      DDRA = 0;           // Port A als Eingänge (Schalter)
12      DDRC = 0xFF;        // Port C als Ausgänge (LEDs)
13          while(1){       // Endlosschleife
14              PORTC = PINA; // Kopiere von Port A nach Port C
15          }
16  }
```

Aufgaben für die Laborübung:

- Programmieren, Übersetzen.
- Anschauen des disassemblierten Programms.
- Schrittweise Programmabarbeitung.

Das disassemblierte Hauptprogramm

```
  10: void main(){
  11:     DDRA = 0;
0x007D  OUT 0x01,R1       Out to I/O location
  12:     DDRC = 0xFF;
0x007E  SER R24           Set Register
0x007F  OUT 0x07,R24      Out to I/O location
  14:         PORTC = PINA;
0x0080  IN R24,0x00       In from I/O location
0x0081  OUT 0x08,R24      Out to I/O location
0x0082  RJMP PC-0x0002    Relative jump
```

Kommentare zu Start-Up-Code auf der nächsten Folie:
Initialisierung des Statusregisters SREG (EA-Adresse 0xFE) mit null, des Registers R1 mit null, des Stackpointers SP (Adresse 0xFE,0xFD) mit 0x21FF. Aufruf des Hauptprogramms auf Adresse 0x7D.

Start-Up-Code

```
0x0000  RJMP PC+0x0072      Relative jump
0x0072  CLR R1              Clear Register
0x0073  OUT 0x3F,R1         Out to I/O location
0x0074  SER R28             Set Register
0x0075  LDI R29,0x21        Load immediate
0x0076  OUT 0x3E,R29        Out to I/O location
0x0077  OUT 0x3D,R28        Out to I/O location
0x0078  LDI R16,0x00        Load immediate
0x0079  OUT 0x3C,R16        Out to I/O location
0x007A  RCALL PC+0x0003     Aufruf des Hauptprogramms
0x007B  RJMP PC+0x0008      Relative jump
```

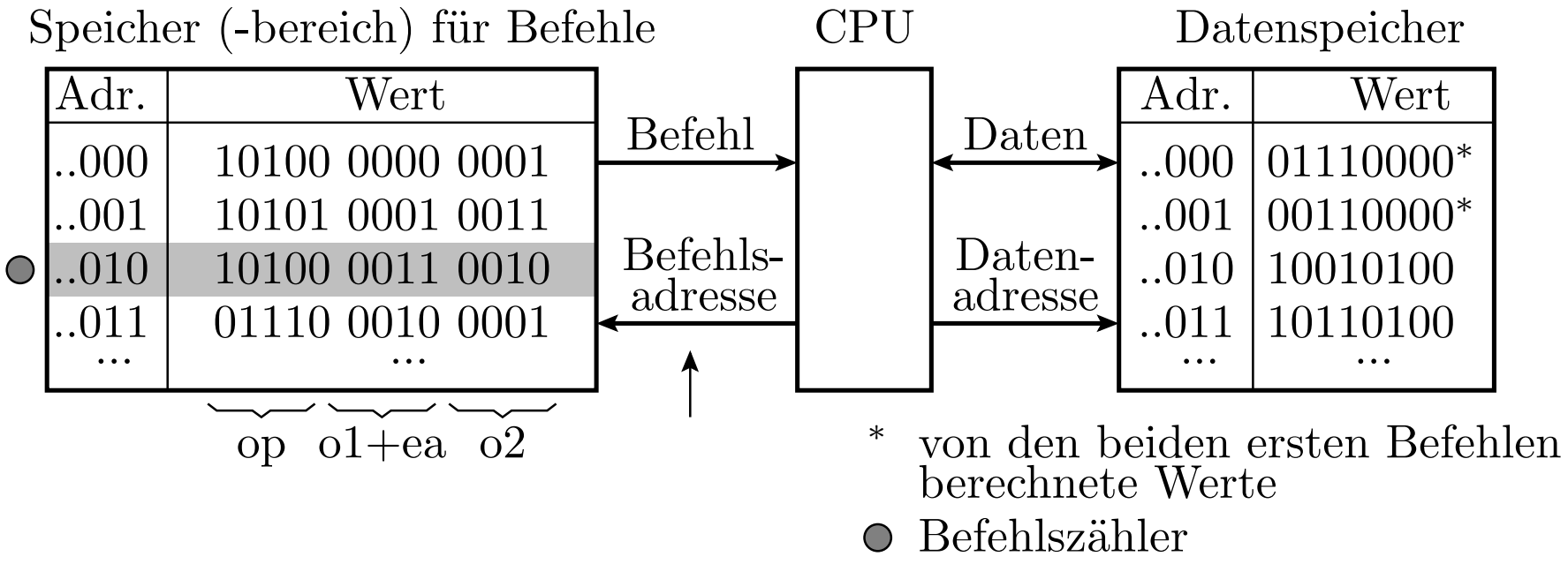
...7D Beginn des Hauptprogramms

falls das Hauptprogramm mit einem Rücksprung endet

```
0x0083  CLI                 Global Interrupt Disable
0x0084  RJMP PC-0x0000      Sprung zu sich selbst
```

Aufgaben

Aufgabe 1.1



- Welche Operation ist im dritten Befehl auf Folie 14 codiert?
- Welche Adresse haben die Operanden?
- Was passiert in den Folgeschritten nach dem Holen des Befehlswortes?

Füllen Sie hierzu die Tabelle auf der nächsten Folie aus.

Lade Operand 1:	Adresse:	Wert:
Lade Operand 2:	Adresse:	Wert:
Operation:		
Schreibe Ergebnis:	Adresse:	Wert:
Hole Befehlswort:	Adresse:	Wert:

Aufgabe 1.2

Suchen Sie in der AVR-Befehlssatzbeschreibung AVR_Instructions.pdf auf der Web-Seite der Vorlesung die Befehle für

- die Addition zweiter Registerinhalte (add)
- die bitweise ODER-Verknüpfung von zwei Operanden (or)
- die Multiplikation zweier Registerinhalte (mul).

Wie lauten für diese Befehle die Operationscodes und in welchen Bits des Befehlsworts stehen die Adressen der Operanden?

	Operationscode	Adresse 1. Operand*	Adresse 2. Operand*
add			
or			
mul			

(*– Bitnummern im Befehlswort, z.B. Bit 9 bis 5).

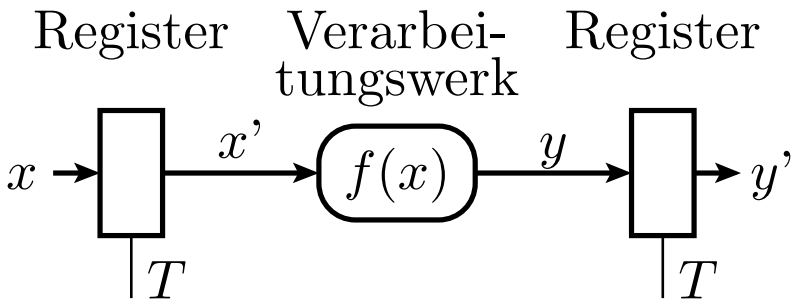
- In welche Register werden welche Ergebnisteile gespeichert?

Verarbeitungswerke

RT-Funktionen und Verarbeitungswerke

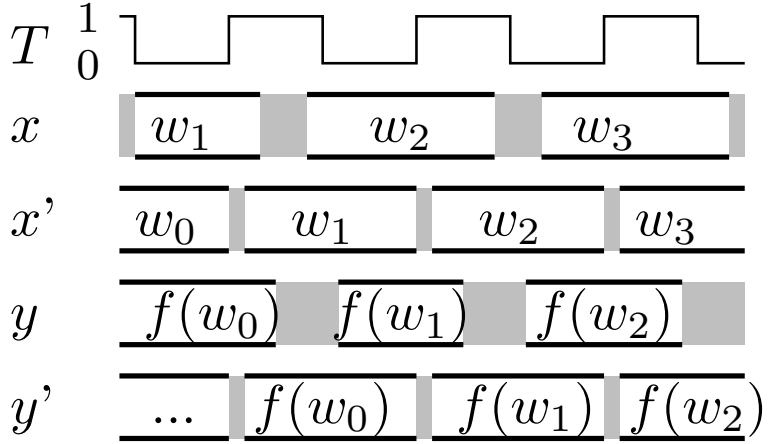
Im RT- Modell werden aus Eingaben und Registerwerten die Ausgaben und Registerfolgezustände gebildet.

- Die Register stellen die Operanden bereit und tasten die Ergebnisse ab.
- Die Übernahmebedingungen sind Taktflanken optional UND-verknüpft mit Freigabesignalen.
- Die Übergangsfunktionen zwischen den Registerzuständen werden durch Verarbeitungswerke realisiert.



Registerübernahme bei $T = \lrcorner$

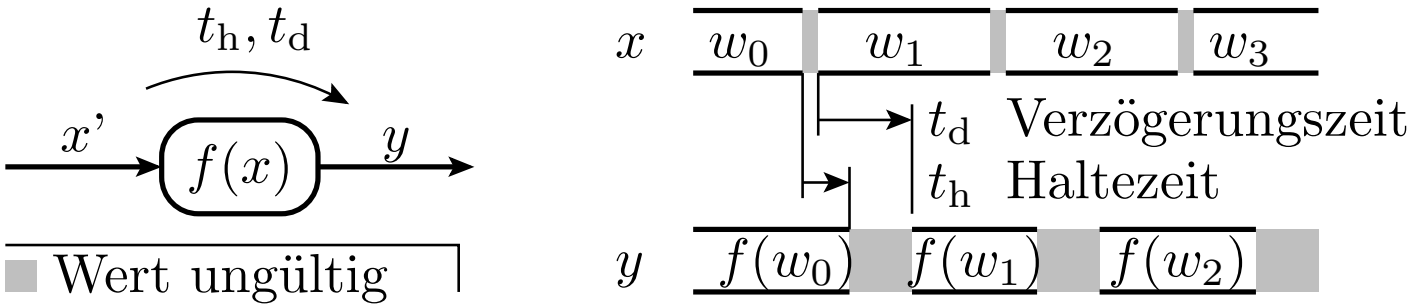
- Wert ohne Bedeutung



Zeitverhalten von Verarbeitungswerken

Verarbeitungswerke (Gatterschaltungen, Rechenwerke, ...) sind kombinatorische Schaltungen, die

- Ausgaben aus Eingaben bilden,
- selbst keine Zustände speichern und
- eine Halte- und eine Verzögerungszeit[10] haben.



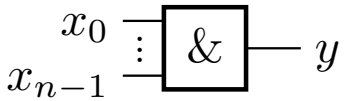
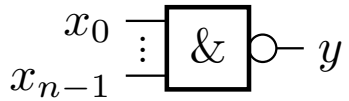
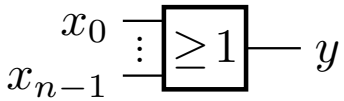
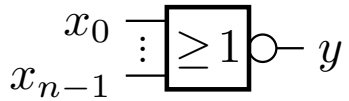
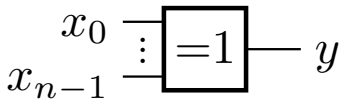
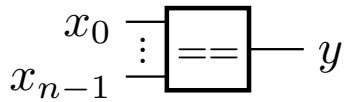
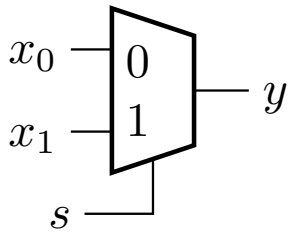
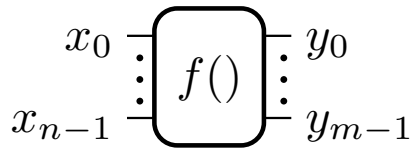
Das Ergebnis muss bei Abtastung gültig sein[11].

[10] Die Haltezeit ist die minimale Zeit, die die Ausgabe nach einer Eingabeänderung unverändert bleibt und die Verzögungszeit ist die maximale Zeit nach Anlegen einer neuen Eingabe, nach der die Ausgabe verfügbar ist.

[11] Wird im Weiteren immer als erfüllt betrachtet.

Gatterschaltungen

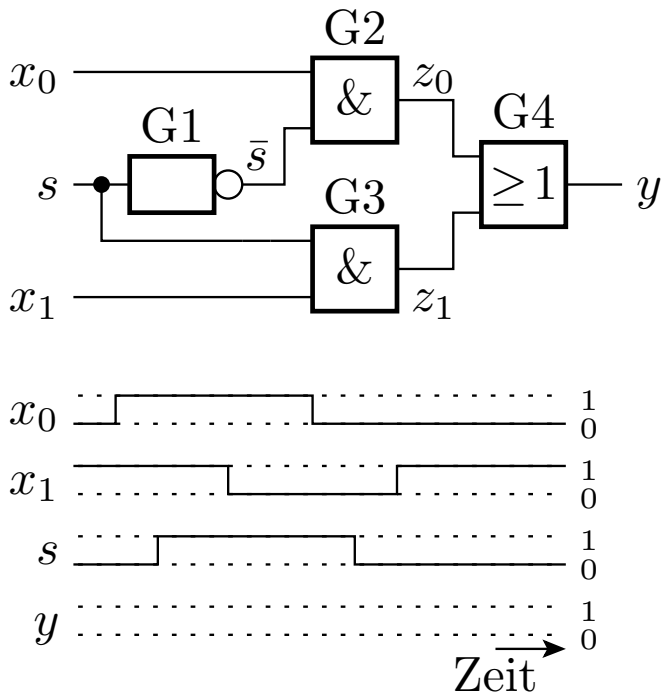
Logikgatter

logische Funktion	Symbol	logische Funktion	Symbol
Treiber (Identität): $y \leftarrow x$		Inverter: $y \leftarrow \bar{x}$	
UND: $y \leftarrow x_0 \wedge x_1 \wedge \ldots$		NAND: $y \leftarrow \overline{x_0 \wedge x_1 \wedge \ldots}$	
ODER: $y \leftarrow x_0 \vee x_1 \vee \ldots$		NOR: $y \leftarrow \overline{x_0 \vee x_1 \vee \ldots}$	
XOR: $y \leftarrow x_0 \oplus x_1 \oplus \ldots$		XNOR: $y \leftarrow \overline{x_0 \oplus x_1 \oplus \ldots}$	
Multiplexer: wenn $s = 0$ dann $y \leftarrow x_0$ sonst $y \leftarrow x_1$		allg. Funktion: $\mathbf{y} \leftarrow f(\mathbf{x})$	

n – Anzahl der Eingabebits; m – Anzahl der Ausgabebits.

Schaltungen aus Gattern

Aus Gattern werden komplexere Funktionsbausteine zusammengesetzt, z.B. Multiplexer:

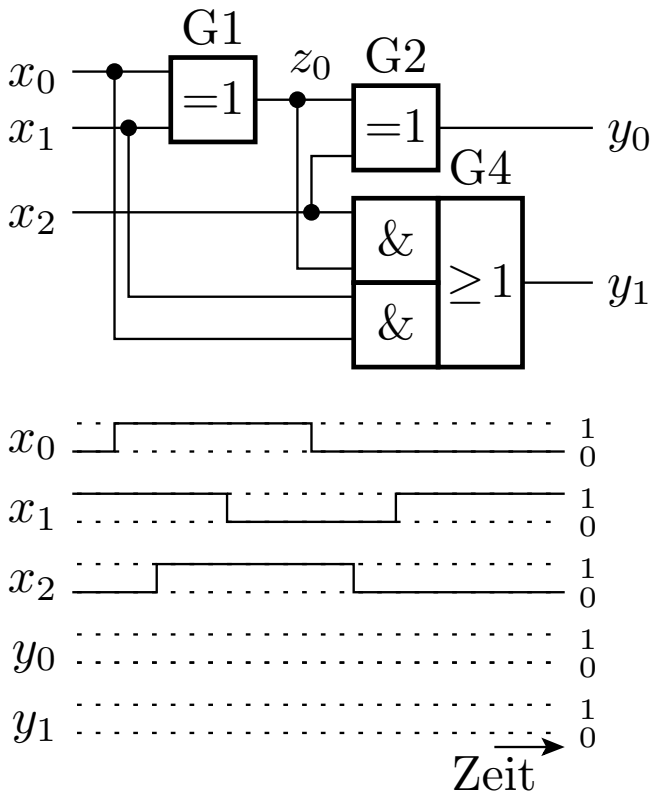


Eingabe			Ist-Funktion				Soll-Funktion
s	x_1	x_0	$\bar{s}$	z_0	z_1	y	wenn $s=0$ dann $y=x_0$ sonst $y=x_1$
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

Beim Zeichnen der Zeitabläufe werden in dieser Vorlesung die Halte- und Verzögerungszeiten sowie »ungültig« vernachlässigt.

Volladdierer

Ein Volladdierer bildet die Summe von drei Bits. Das Ergebnis hat den Wertebereich 0 bis 3 und wird als 2-Bit-Vektor dargestellt:



Eingabe x_2	x_1	x_0	Ist-Funktion z_0	y_1	y_0	Soll-Funktion $x_2+x_1+x_0$ dez	bin
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

Bitverarbeitung

Logikoperationen in Rechnern

Rechnerbefehle führen die logischen Grundoperationen (UND, ODER, EXOR und Negation) bitweise aus. Sie dienen zum

- Setzen, Löschen, Invertieren einzelner Bits und
- zur Programmierung logischer Verknüpfungen.

Logikbefehle des ATmega-Befehlssatzes:

Operation	Op.-Code	Assembler
Rd←Rd∧Rr	0010 00rd dddd rrrr	and Rd, Rr
Rd←Rd∧K	0111 kkkk dddd kkkk	andi Rd*, K
Rd←Rd∨Rr	0010 10rd dddd rrrr	or Rd, Rr
Rd←Rd∨K	0110 kkkk dddd kkkk	ori Rd*, K
Rd←Rd⊕Rr	0010 10rd dddd rrrr	eor Rd, Rr
RD ← $\overline{\text{Rd}}$	1001 010d dddd 0000	com Rd

(Rd – Zielregister und erster Operand; Rr – 2. Operand; K – 8-Bit-Konstante, * Nur anwendbar auf Register R16 bis R31).

Beispielprogramm

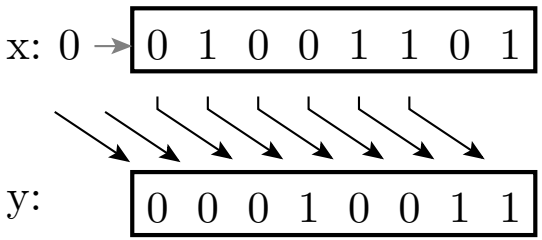
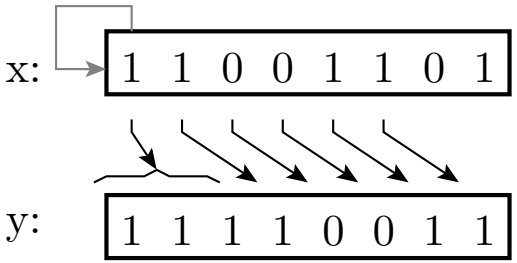
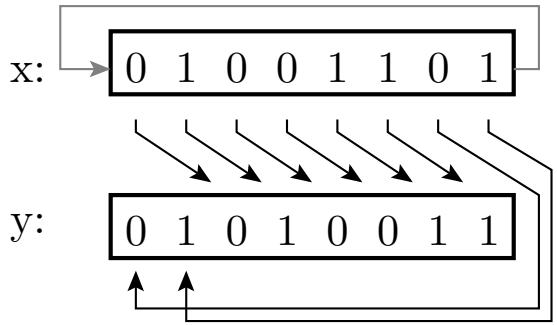
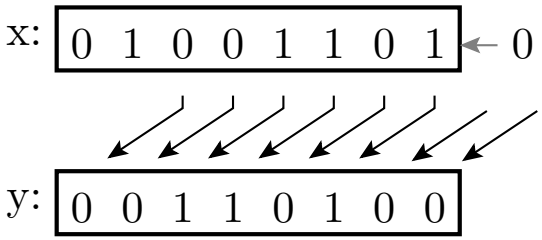
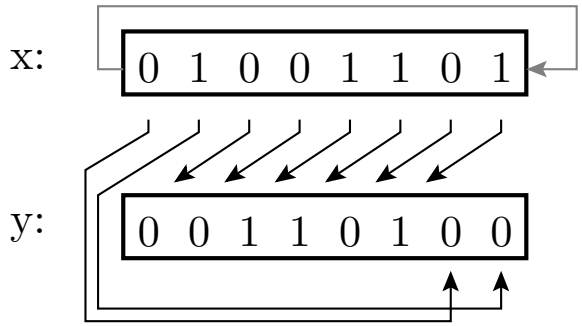
Welche Werte werden den Register r16 und r17 zugewiesen?

```
10   .global Logikprogramm
11   Logikprogramm:
12     ldi r16, 0x18   ; r16 = 0b.... ....
13     ldi r17, 0x24   ; r17 = 0b.... ....
14     ori r16, 0x03   ; r16 = 0b.... ....
15     andi r16, 0xFE  ; r16 = 0b.... ....
16     eor r17, r16    ; r17 = 0b.... ....
17     com r17         ; r17 = 0b.... ....
18     ret
```

»Logikprogramm:« ist eine Marke, ein Textsymbol für eine Adresse. Mit der Definition als »global« ist die Adresse auch in der Datei mit dem Hauptprogramm bekannt und in Funktionsaufrufen als Unterprogrammadresse nutzbar.

```
---       .././Assembly1.s ---
    12:     ldi r16, 0x18       ; r16 = 0b.... ....
0000007D 08.e1                  LDI R16,0x18
    13:     ldi r17, 0x24       ; r17 = 0b.... ....
0000007E 14.e2                  LDI R17,0x24
    14:     ori r16, 0x03       ; r16 = 0b.... ....
0000007F 03.60                  ORI R16,0x03
    15:     andi r16, 0xFE  ; r16 = 0b.... ....
00000080 0e.7f                  ANDI R16,0xFE
    16:     eor r17, r16        ; r17 = 0b.... ....
00000081 10.27                  EOR R17,R16
    17:     com r17                 ; r17 = 0b....
00000082 10.95                  COM R17
    18:     ret
00000083 08.95                  RET
---       .././Logikprogramm.c
    12: int main(void){         ; Hauptprogramm
    13:     while(1){           ; Endlosschleife
    14:         Logikprogramm(); ; Funktionsaufruf
00000084 f8.df                  RCALL PC-0x0007
00000085 fe.cf                  RJMP PC-0x0001
```

Verschiebung und Rotation

logische Verschiebung	arithmetische Verschiebung	Rotation
uint.. x,y; y = x >> 2;	int8_t x,y; y = x >> 2;	2 Stellen nach rechts
		
	-0b1101 = -13	
(u)int8_t x,y; y = x << 2;	Verschiebung mit Erhalt des Vorzeichens. Im Beispiel: 0xCD >> 2 entspricht: -0x33/4 exakt -12,75; ohne Nachkommastellen -13	2 Stellen nach links
		

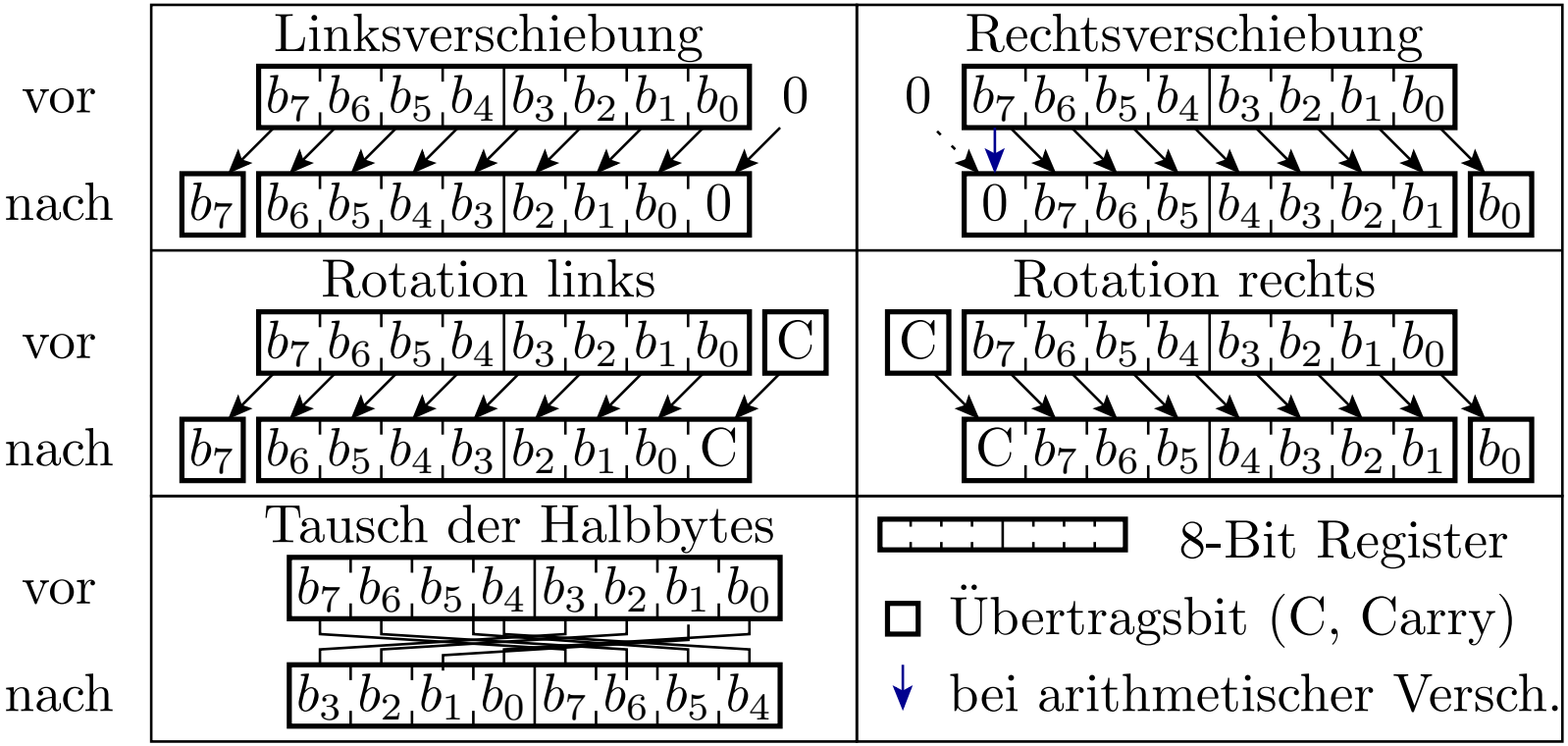
Auffüllen der freiwerdenden Bitstellen mit

- Logische Verschiebung: null.

- Arithmetische Verschiebung: Vorzeichenbits[12].
- Rotation: rausgeschobene Bits.

Unser Beispielprozessor kann

- 8 Bit um eine Stell nach rechts/links verschieben und
- Halbbytes tauschen (Rotation um 4 stellen).



[12] Siehe später vorzeichenbehafteter Zahlen ab Folie 73).

ATmega-Verschiebebefehle

Operation	Operationscode	Assembler
C ← 0	1001 0100 1000 1000	cls
C ← 1	1001 0100 0000 1000	sec
C ← $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ ← 0	0000 11dd dddd dddd	lsl Rd
0 → $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ → C	1001 010d dddd 0110	lsr Rd
$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ → C	1001 010d dddd 0101	asr Rd
C ← $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ ← C	1001 11dd dddd dddd	rol Rd
C → $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$ → C	1001 010d dddd 0111	ror Rd
$b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$	1001 010d dddd 0010	swap Rd

(cls – Clear Carry Flag; sec – Set Carry Flag; lsl – Logical Shift Left, identisch mit add Rd, Rd; rol – Rotate Left, identisch mit adc Rd, Rd)

Wie werden C-Verschiebeoperation, z.B.

```
uint16_t a, b;
...
a = b << 2;
```

durch Prozessorbefehle nachgebildet?

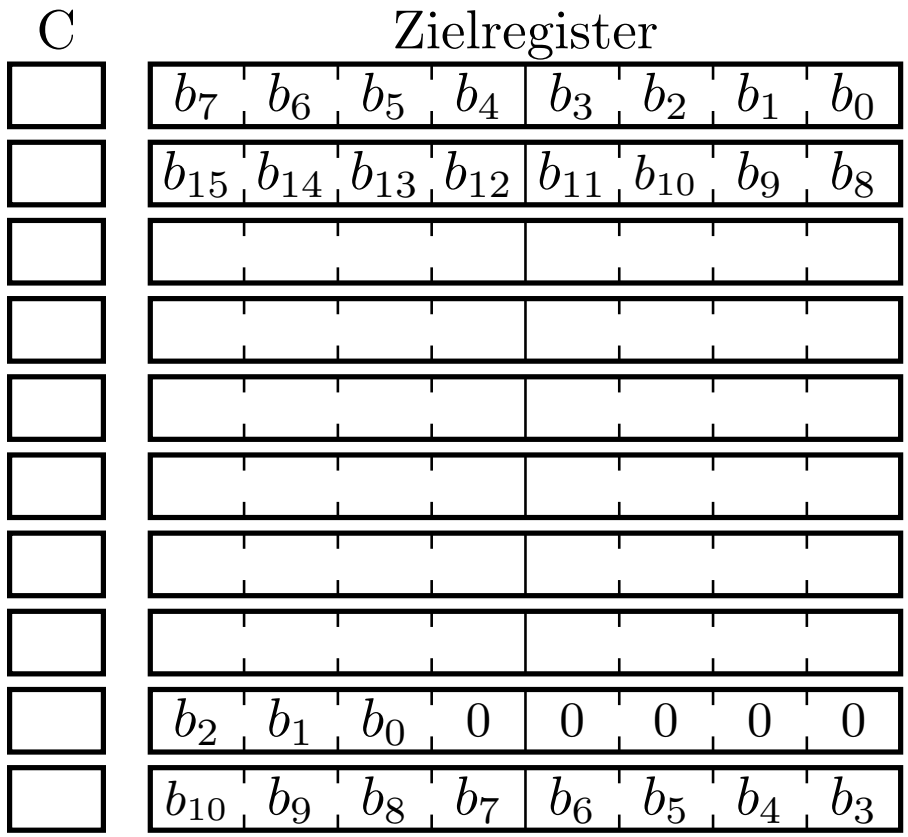
Ausschnitt aus dem disassemblierten Programm:

```
    a = b<<2;
LDS R24,0x0200   ; r24 ← b Byte 0
LDS R25,0x0201   ; r25 ← b Byte 1
LSL R24    ; C: [b7]  r24: [b6 b5 b4 b3 b2 b1 b0 0]
ROL R25    ; C: [15]  r25: [14 13 12 11 10 b9 b8 b7]
LSL R24    ; C: [b6]  r24: [b5 b4 b3 b2 b1 b0 0 0]
ROL R25    ; C: [14]  r25: [13 12 11 10 b9 b8 b7 b6]
STS 0x0203,R25   ; a Byte 1 ← r25
STS 0x0202,R24   ; a Byte 0 ← a r24
```

Für die Verschiebung um vier verwendet der Compiler den Swap-Befehl:

```
uint16_t a, b;

    a = b<<5;
LDS R24,0x0200
LDS R25,0x0201
LSL R24
ROL R25
SWAP R24
SWAP R25
ANDI R25,0xF0
EOR R25,R24
ANDI R24,0xF0
EOR R25,R24
STS 0x0203,R25
STS 0x0202,R24
```



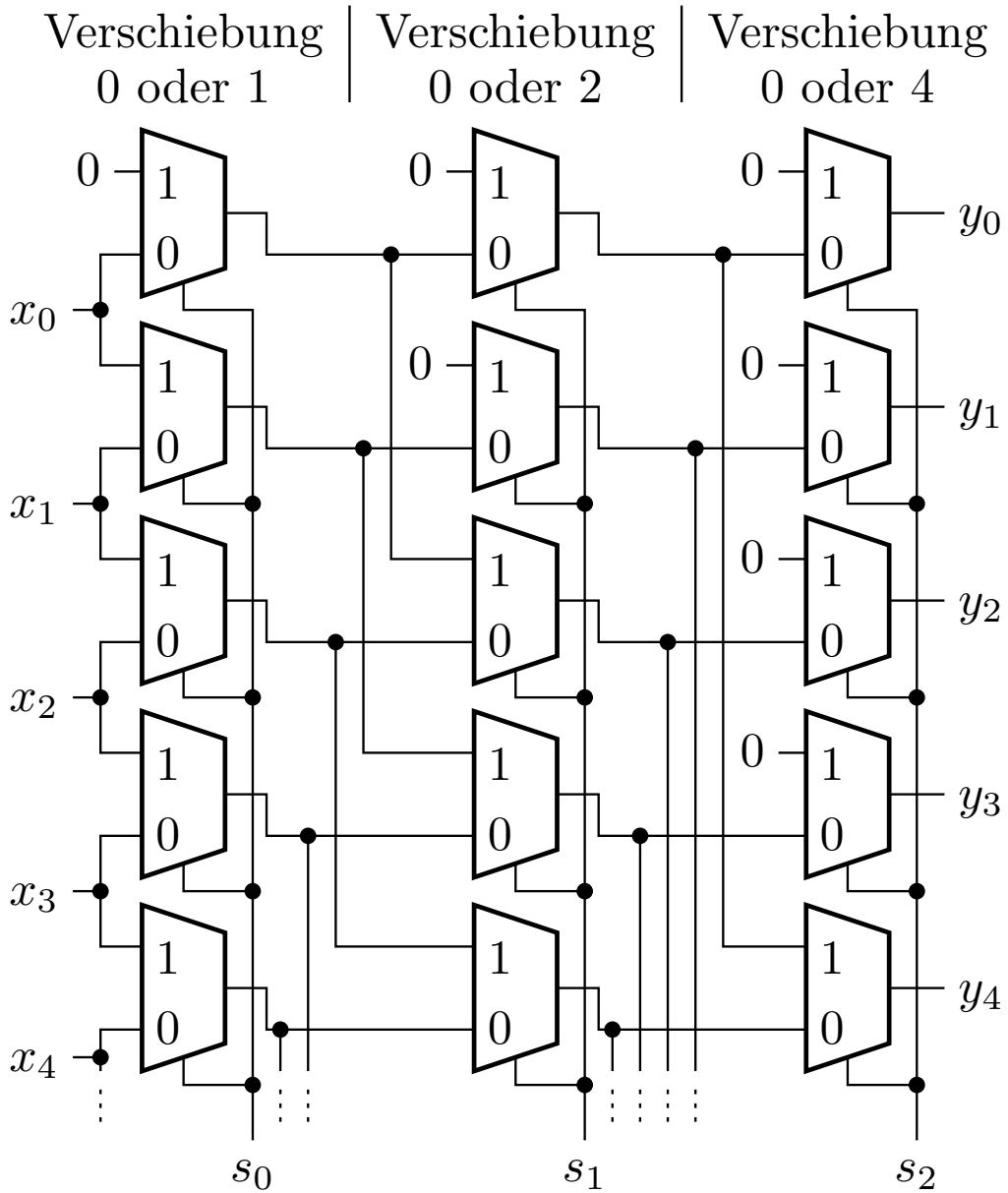
Block-Shifter

32-Bit-Prozessoren haben oft einen Blockshifter zur Ausführung von Verschiebeoperationen:

```
(u)int32_t a, b
int8_t s;
...
a = b << s;
```

($0 \leq s < 32$) in einem Schritt.

$\Rightarrow$ umschaltbare Verbindungen (Multiplexer), Block-Shifter



Spezielle Bitverarbeitungsoperationen

Viel Prozessoren haben Spezialbefehle, die ihen in bestimmten Anwendungen Geschwindigkeitsvorteile verschaffen.
ATmega-Prozessoren haben z.B. die Befehle:

```
sbi A, b ; (Set Bit I/O) setze Bit b in EA-Register A
cbi A, b ; (Clear Bit I/O) lösche Bit b in EA-Register A
bld Rd, b; (Bit LoaD) kopiere Bit b aus Register Rd in
         ; das Statusbit T
bst Rd, b; (Bit STore) kopiere T nach Rd Bit b
```

Der Befehl

```
sbi 8, 3; setze Bit 3 in Port C«
```

ersetzt z.B. eine Befehlsfolge:

```
in r16, 8
ori r16, 0b00001000
out 8, r16
```

Das Kopieren von PINA, Bit 2 nach PORTB, Bit 7 verlangt mit den allgemein üblichen Logikbefehlen eines 8-Bit Prozessors eine Befehlsfolge von mindestens 8 Befehlen:

```
in r0, 0              ; r0 ← PINA
swap r0               ; Bit 2 nach Bit 6 tauschen
lsl r0                ; Bit 2 weiter nach Bit 7 schieben
andi r0, 0b10000000 ; alle anderen Bits null setzen
in r1, 5              ; r1 ← PORTB
andi r1, 0b01111111 ; Bit 7 löschen
or r0, r1,            ; Registerinhalte zusammenfassen
out 5, r0             ; PORTB ← r0
```

Mit den Spezialoperationen genügen fünf Befehle:

```
in r0, 0  ; r0 ← PINA
in r1, 5  ; r1 ← PORTB
bld r0, 2 ; T  ← r0, Bit 3
bst r0, 7 ; r1, Bit 5 ← T
out 5, r0 ; PORTB ← r0
```

Nutzt der Compiler diese Spezialbefehle? Ausprobieren!

Skip-Operationen (Bedingtes Überspringen)

Skip Operationen überprüfen eine Bedingung und überspringen den nächsten Befehl, wenn die Bedingung erfüllt ist:

```
cpse Rd, Rr; Bedingung: Rr gleich Rd
sbrc Rr, b ; Bedingung: Bit b in Rr gleich null
sbrs Rr, b ; Bedingung: Bit b in Rr gleich eins
sbic A, b  ; Bedingung: Bit b in A gleich null
sbis A, b  ; Bedingung: Bit b in A gleich eins
```

(Rr – Prozessorregister; A – Ein-/Ausgaberegister)

Mit Skip-Befehlen lässt sich das Kopieren von PINA, Bit 2 nach PORTB, Bit 7 mit noch weniger Befehlen lösen:

```
sbis 0, 2; wenn PINA, Bit 2 gleich 1, überspringe
cbi 5, 7 ; lösche PORTB, Bit 7
sbic 0, 2; wenn PINA, Bit 2 gleich 0, überspringe
sbi 5, 7 ; setze PORTB, Bit 7
```

Addierer, Subtrahierer

Addierer

Die Addition ist eine der meist benötigten Operationen in einem Rechner:

- Additions- und Increment-Operationen im Programm,
- (Schleifen-) Zähler,
- Weiterschalten der Befehlsadresse, Skip-Operation und relative Sprünge,
- Indexrechnung für Feldzugriffe, indirekte Adressierung mit Verschiebung[13],
- Subtraktion als Addition mit dem negierten Subtrahenden,
- größer/kleiner-Vergleich als Subtraktion mit Auswertung des Ergebnisvorzeichens.

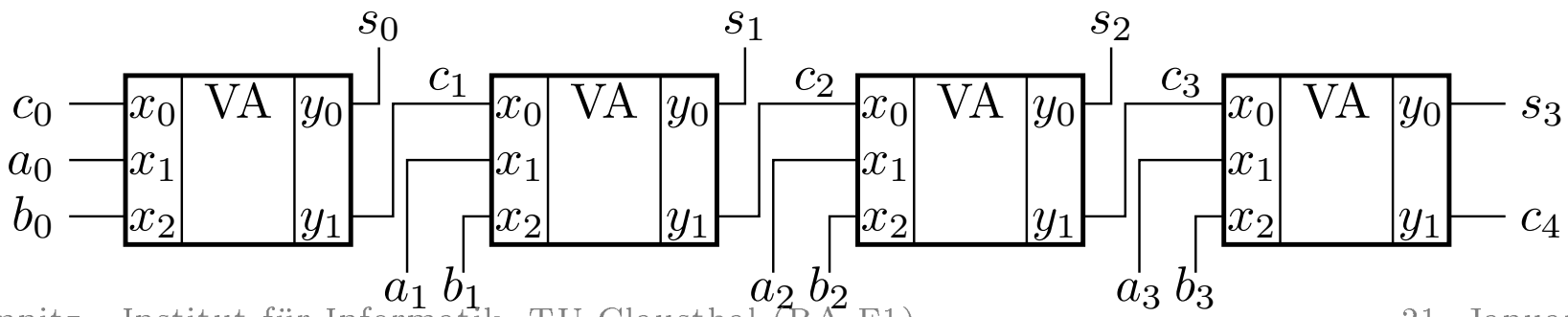
[13] Datenadresse gleich Registerinhalt plus Konstante im Befehlswort.

Algorithmus der Addition

	0	1	1	0	0	1	0	1	$(-0 \cdot 2^8)$ Erweiterung
+	1	0	0	0	0	0	1	1	$(-1 \cdot 2^8)$ zu vorzei-
									chenbehaf-
	1	1	1	0	$1^{(1)}$	$0^{(1)}$	$0^{(1)}$	0	$(-1 \cdot 2^8)$ tete Zahlen

- Wiederhole für alle Bitstellen beginnend mit der niederwertigsten
  - Addition der Ziffern + Übertrag der vorherigen Stelle (Funktion eines Volladdierers)

Addierer als Schaltung (im Beispiel für 4-Bit-Operanden):



- Die Bitbreite eines Addierers ist skalierbar. In Rechnern richtet sich die Breite nach den Operanden, für Adressrechnungen Adressbusbreite, für die Addition von Daten Datenbusbreite.
- Das Ergebnis ist um das Übertragsbit größer als der größte Operand. Speicherung des Übertrags im C (Carry-) Bit des Statusworts.
- ATmega hat einen Additionsbefehl mit und einen ohne einlaufenden Übertrag:

Operation	Op.-Code	Assembler
C,Rd← Rd+Rr	`0000 11rd dddd rrrr`	`add Rd, Rr`
C,Rd← Rd+Rr+C	`0001 11rd dddd rrrr`	`addc Rd, Rr`

Bei Addition mehrerer Bytes werden die niederwertigen Bytes mit add und die höherwertigen mit addc addiert.

Addition zweier 16-Bit-Zahlen

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

```
    a = b+c;
LDS R18,0x0200   ; r18 ← b Byte 0
LDS R19,0x0201   ; r19 ← b Byte 1
LDS R24,0x0202   ; r24 ← c Byte 0
LDS R25,0x0203   ; r25 ← c Byte 1
ADD R24,R18      ; Addition Byte 0
ADC R25,R19      ; Addition Byte 1
STS 0x0205,R25   ; Speichere a Byte 1
STS 0x0204,R24   ; Speichere a Byte 0
```

disassemblierter Code der Anweisung

Testbeispiel

r18: 0x 32
r19: 0x 1A
r24: 0x 7A
r25: 0x 15
r24: 0x
r25: 0x

Vorzeichenbehaftete Summanden und Subtraktion

Statt durch Vorzeichen und Betrag werden vorzeichenbehaftete Zahlen in Rechnern durch »Stellenkomplement +1« dargestellt.
Mathematische Grundlage:

- Das Stellenkomplement zu einer Ziffer b_i ist die Differenz zur größten darstellbaren Ziffer mit dem Wert $B-1$:

$$\bar{b}_i = B - 1 - b_i$$

(B – Basis des Zahlensystems, für Dezimalzahlen $B = 10$).

Beispiel: $\overline{437} = 562$

- Zahl plus Stellenkomplement gleich größte darstellbare Zahl.

Beispiel: $437 + \overline{437} = 437 + 562 = 999$

- plus Eins gleich kleinste nicht darstellbare Zahl:

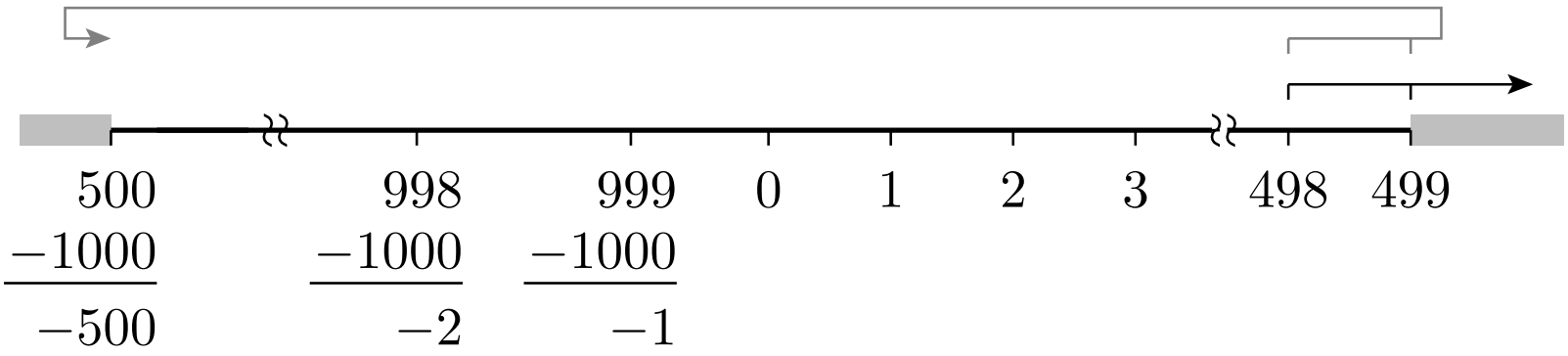
$$Z + \bar{Z} + 1 = B^n$$

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

- Auflösung nach $-Z$:

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

- Die Zählreihenfolge bleibt, nur der Darstellungsbereich verschiebt sich:



- Die Negation ist das Stellenkomplement plus 1:

$$-Z = \bar{Z} + 1$$

- Die Subtraktion ist die Addition mit dem Stellenkomplement plus 1:

$$Z_1 - Z_2 = Z_1 + \bar{Z}_2 + 1$$

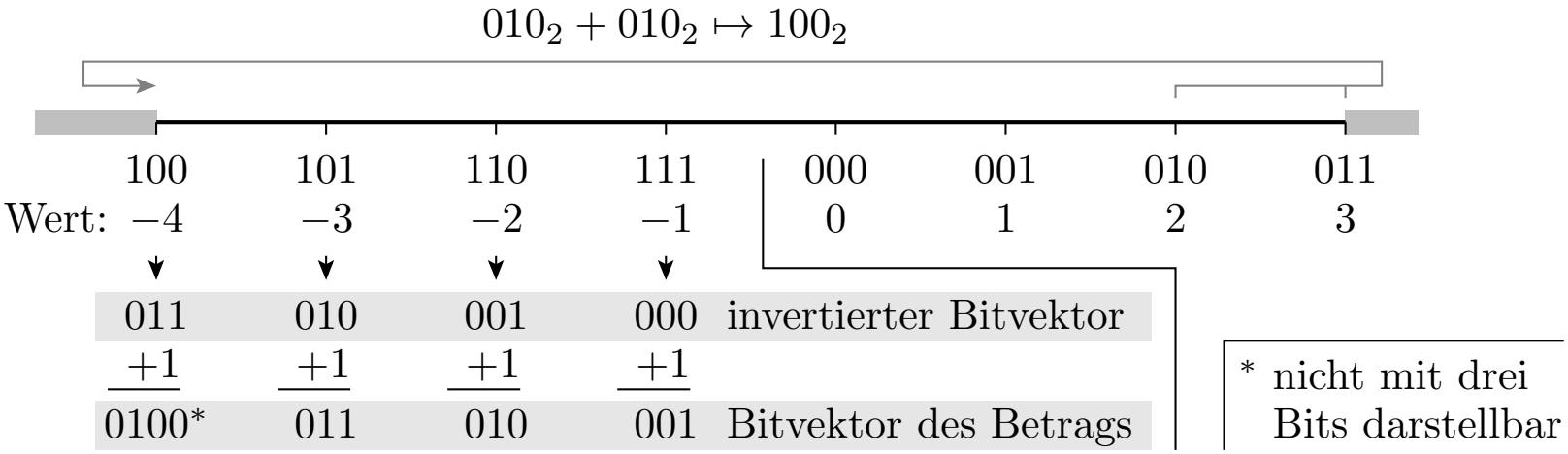
  Das gilt für alle Stellensysteme: Dezimalsystem, Binärsystem, ...

Vorteile »Stellenkompl.+1« gegenüber »Vorzeichen und Betrag«:

- Gleiche Zählreihenfolge für vorzeichenbehaftete und vorzeichenfreie Zahlen (00, 01, ..., 99, 00, ...).
- Die Addition und Subtraktion, die ja auf »Mehrfachzählen« basieren, sind für Zahlen mit und ohne Vorzeichen gleich.

Zweierkomplement

Stellenkomplement für Binärzahlen mit führendem Bit gleich Vorzeichenbit:



- Stellenkomplement für Binärziffern: $\bar{0} \mapsto 1$; $\bar{1} \mapsto 0$ (Invertierung).
- Wertebereichsüberlauf bei Übertrag $C \neq$ Vorzeichenbit S:
  - größte Zahl +1: 011...1+1 → 100...0, $C = 0$, $S = 1$
  - kleinste Zahl -1: 100...0+111...1→011...1, $C = 1$; $S = 0$

Subtraktion zweier 16-Bit-Zahlen

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

```
    a = b-c;
LDS R18,0x0200   ; r18 ← b Byte 0
LDS R19,0x0201   ; r19 ← b Byte 1
LDS R24,0x0202   ; r24 ← c Byte 0
LDS R25,0x0203   ; r25 ← c Byte 1
MOVW R20,R18
SUB R20,R24      ; Subtraktion Byte 0
SBC R21,R25      ; Subtraktion Byte 1
MOVW R24,R20
```

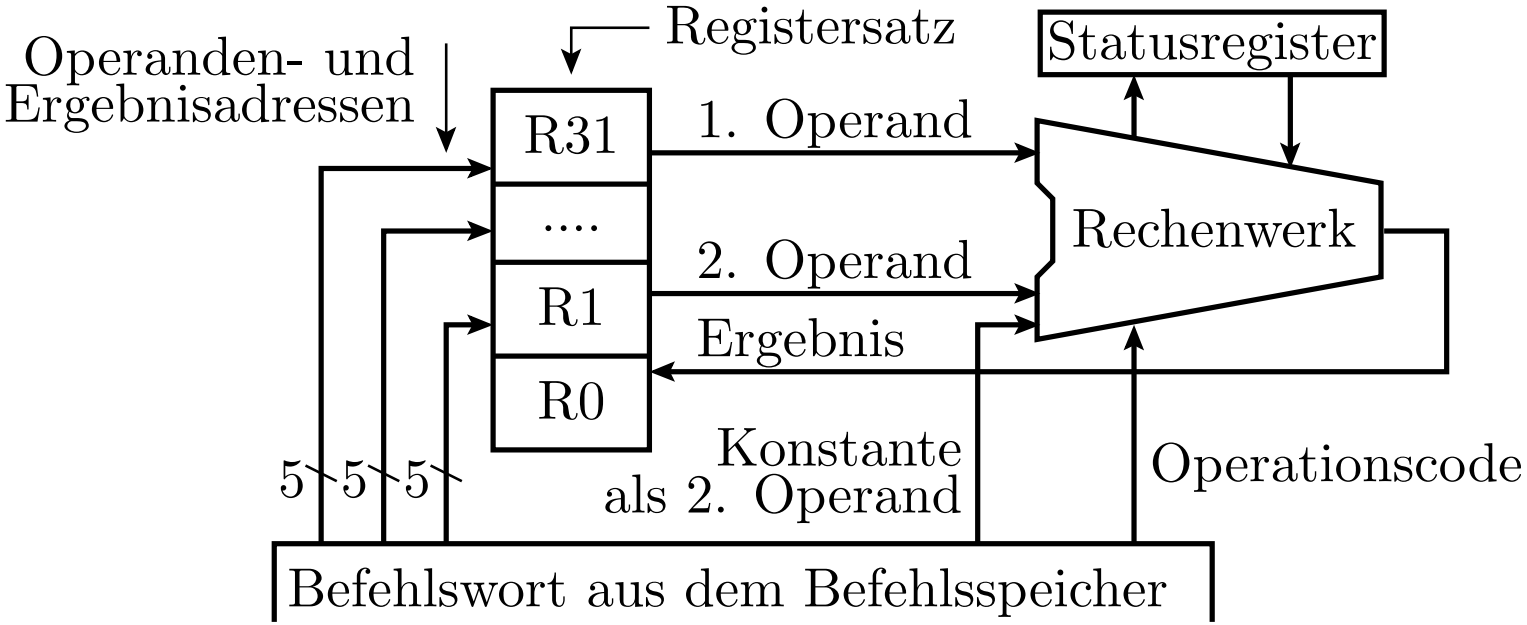
disassemblierter Code der Anweisung

Testbeispiel

r18: 0x A3
r19: 0x 31
r24: 0x 21
r25: 0x 10
r20: 0x r21: 0x
r20: 0x
r21: 0x
r24: 0x r25: 0x

Statusregister

Das Statusregister



- Das Statusregister speichert Zusatzdaten zum Ergebnis:
- den Übertrag von Additionen und Subtraktionen für Nachfolgeoperationen,
  - >0, =0, ... für nachfolgende bedingte Sprünge,
  - ...

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, Übertrag.
- Z Zero Flag: 1 bei Ergebnis null.
- N Negative Flag (unsigned): 1 bei neg. Differenz.
- V Zweierkomplement Überlauf.
- S $(N \oplus V)$ Vorzeichen Zweierkomplement.
- H Half Carry: Übertrag nach dem ersten Halbbyte.
- T Transferbit: Zwischenspeicher für Bitkopieren.
- I Globales Interrupt-Freigabebit.

Befehlsarten und ihr Einfluss auf die Statusbits:

- add, sub, ... setzen C, Z, N, V, S, H nach Ergebniswert.
- Logische Op. setzen Z, N, S nach Ergebniswert und V=0.
- Lade-, Speicher- und Sprungbefehle lassen die Statusbits unverändert.

Compare-Befehle[14] beeinflussen nur Statusbits:

```
cp  Rd, Rr; Rd - Rr, Flags: Z,C,N,V,S,H
cpc Rd, Rr; Rd-Rr-C, Flags: Z,C,N,V,S,H
cpi Rd, K; Rd - K    Flags: Z,C,N,V,S,H
```

Statusbits dienen als Bedingungen für Sprünge:

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

Jeden dieser 16 Befehle gibt es auch mit speziellem Namen statt der Nummer des zu testenden Statusbits:

```
breq k   ; BRanch if EQual: Sprungbedingung Z=1
brne k   ; BRanch if Not Equal: Sprungbed. Z=1
brcs k   ; BRanch if Carry Set: Sprungbed. C=1
brcc k   ; BRanch if Carry Clear: Sprungbed. C=0
...
```

(Weitere bedingte Sprünge siehe AVR-Befehlssatz auf der Web-Seite oder auf Foliensatz 2).

[14] Das sind Subtraktionen ohne Speichern der Differenz in Rd.

Multiplikation, Division

Multiplikation vorzeichenfreier Binärzahlen

$$(a_3 \cdot 2^3 + a_2 \cdot 2^2 + a_1 \cdot 2^1 + a_0 \cdot 2^0) \cdot (b_3 \cdot 2^3 + b_2 \cdot 2^2 + b_1 \cdot 2^1 + b_0 \cdot 2^0) =$$

$$\begin{array}{r}
a_3b_0 \cdot 2^3 + a_2b_0 \cdot 2^2 + a_1b_0 \cdot 2^1 + a_0b_0 \cdot 2^0 \\
a_3b_1 \cdot 2^4 + a_2b_1 \cdot 2^3 + a_1b_1 \cdot 2^2 + a_0b_1 \cdot 2^1 \\
a_3b_2 \cdot 2^5 + a_2b_2 \cdot 2^4 + a_1b_2 \cdot 2^3 + a_0b_2 \cdot 2^2 \\
a_3b_3 \cdot 2^6 + a_2b_3 \cdot 2^5 + a_1b_3 \cdot 2^4 + a_0b_3 \cdot 2^3 \\
\hline
p_7 \quad p_6 \quad p_5 \quad p_4 \quad p_3 \quad p_2 \quad p_1 \quad p_0
\end{array}$$

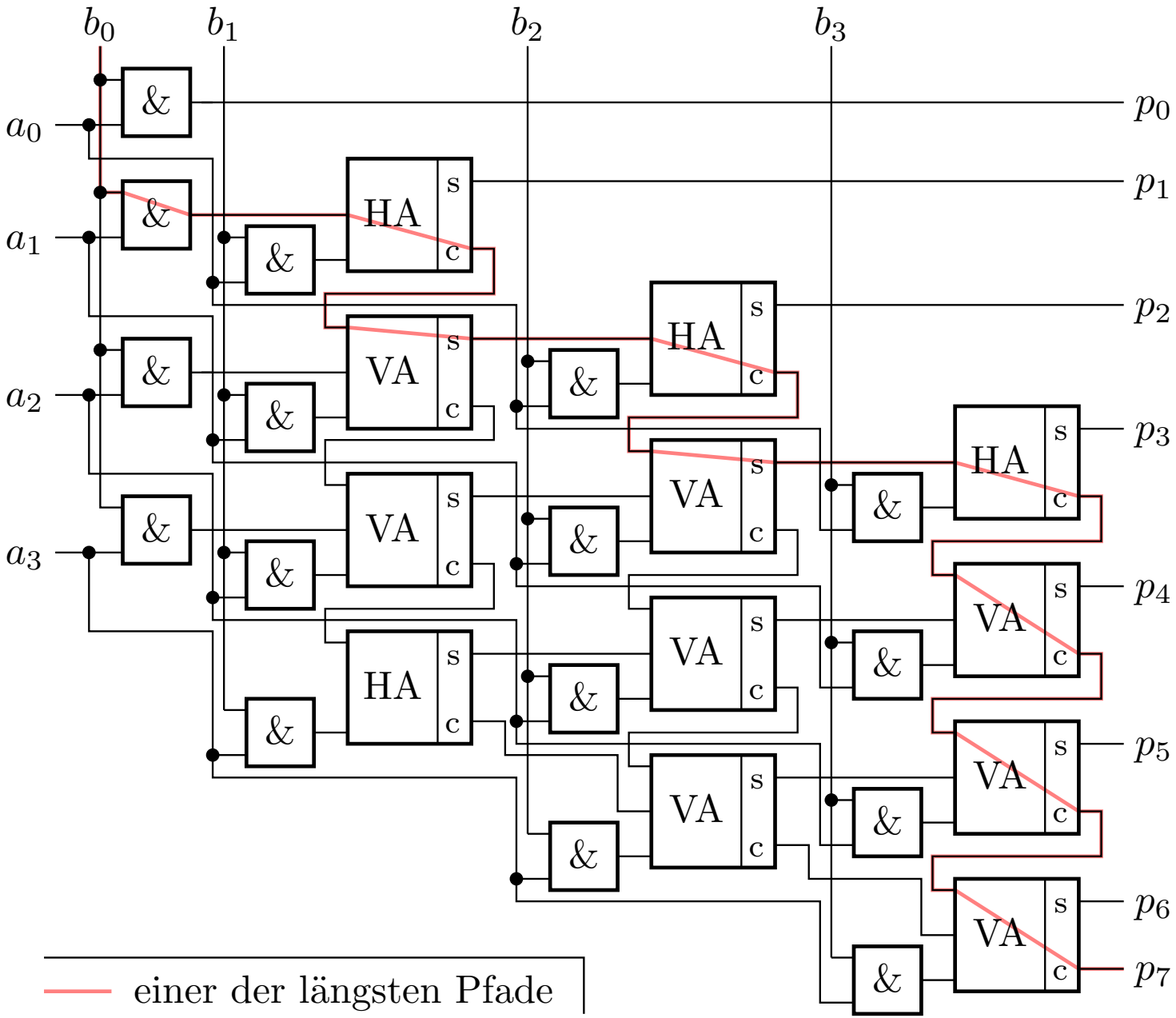
Eine Multiplikation ist nachbildbar aus:

- $m \times n$ 1-Bit-Multiplikation (UND-Verknüpfungen) und
- zeilen- und spaltenweisen Additionen mit Halb- und Volladdierern.

Das kleinste Produkt ist $0 \cdot 0 = 0$. Das größte Produkt ist

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

und benötigt $m + n$ Ergebnisbits.


b_0
b_1
b_2
b_3
a_0
a_1
a_2
a_3
&
HA
VA
s
c
p_0
p_1
p_2
p_3
p_4
p_5
p_6
p_7
einer der längsten Pfade

- Der Schaltungsaufwand eines $n \times n$-Bit Multiplizierers wächst mit n^2.
- Die Verzögerungszeit entlang des längsten Datenpfades, bis das Ergebnis garantiert fertig gebildet ist, wächst nur mit n.
- Ein $n \times n$-Bit-Matrixmultiplizierer hat etwa die 2...3-fache Verzögerung eines normalen n-Bit-Addierers[15].
- Prozessoren haben deshalb oft einen Multiplizierer, der eine Multiplikation in einem oder wenigen Takten ausführt.

ATmega-Multiplikationsbefehl für vorzeichenfreie Zahlen:

Operation	Op.-Code	Assembler
R1:R0←Rd · Rr	`1001 11rd dddd rrrr`	`mul Rd, Rr`

Wählbare Operandenregister. Das höherwertige Ergebnisbyte wird immer in r1 und das niederwertige in r0 gespeichert.

[15] Bei der Nachbildung durch bedingte Additionen wächst sowohl die Anzahl als auch die Dauer der Additionen mit n, d.h. die Gesamtdauer mit n^2.

Nachbildung 16-Bit- durch 8-Bit Multiplikationen

Zerlegung der Operanden und des Ergebnisses in Polynome von einzelnen Bytes a_i, b_i, ... :

$$a_3 \cdot 2^{24} + a_2 \cdot 2^{16} + a_1 \cdot 2^8 + a_0 = \left(b_1 \cdot 2^8 + b_0\right) \cdot \left(c_1 \cdot 2^8 + c_0\right)$$

Berechnung der Ergebnisbytes:

$$\begin{aligned} a_0 &= L\left(b_0 \cdot c_0\right) \\ a_2^* a_1 &= H\left(b_0 \cdot c_0\right) + L\left(b_1 \cdot c_0\right) + L\left(b_0 \cdot c_1\right) \\ a_3^* a_2 &= a_2 + H\left(b_1 \cdot c_0\right) + H\left(b_0 \cdot c_1\right) + L\left(b_1 \cdot c_1\right) \\ a_3 &= a_3 + H\left(b_1 \cdot c_1\right) \end{aligned} \tag{1}$$

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

Gegeben sei folgendes C-Programm, das Watch-Fenster mit den Variablenadressen und eine Tabelle mit einer Adress-Byte-Zuordnung:

```
11   uint16_t b, c;
12   uint32_t a;
13 int main(void){
14     a=b*c;
15   }
```

Name	Value	Type
a	0	uint32_t{data}@0x0204
b	0	uint16_t{data}@0x0200
c	0	uint16_t{data}@0x0202

Adresse	0x200	0x201	0x202	0x203	0x204	0x205	0x206	0x207
Byte	b_0	b_1	c_0	c_1	a_0	a_1	a_2	a_3

Der disassemblierte Programm sollte enthalten:

- vier Ladeoperationen für Operandenbytes in Register,
- vier 8×8-Bit Multiplikationen,
- mehrere Additionen und
- vier Speicheroperationen für Ergebnisbytes.

```
a=b*c;
00089  LDS R20,0x0200   r20 ← b0
0008B  LDS R21,0x0201   r21 ← b1
0008D  LDS R18,0x0202   r18 ← c0
0008F  LDS R19,0x0203   r19 ← c1
00091  MUL R20,R18       r1:r0 ← b0 · c0
00092  MOVW R24,R0       r25 ← H(b0 · c0), r24 ← L(b0 · c0)
00093  MUL R20,R19       r1:r0 ← b0 · c1
00094  ADD R25,R0        r25 ← H(b0 · c0) + L(b0 · c1)
00095  MUL R21,R18       r1:r0 ← b1 · c0
00096  ADD R25,R0        r25 ← H(b0 · c0) + L(b0 · c1) + L(b1 · c0)
00099  LDI R26,0x00      r26 ← 0
0009A  LDI R27,0x00      r27 ← 0
0009B  STS 0x0204,R24    a0 ← L(b0 · c0)
0009D  STS 0x0205,R25    a1 ← H(b0 · c0) + L(b0 · c1) + L(b1 · c0)
0009F  STS 0x0206,R26    a2 = 0
000A1  STS 0x0207,R27    a3 = 0
```

Nicht das erwartete Ergebnis! C berechnet ein 16-Bit Ergebnis und hängt vorn 2 Nullen an.

Multiplikation für Vorzeichenzahlen

$$(A - a_{n-1} \cdot 2^n) \cdot (B - b_{n-1} \cdot 2^n) = A \cdot B - (A \cdot b_{n-1} + B \cdot a_{n-1}) \cdot 2^n + \underbrace{a_{n-1} \cdot b_{n-1} \cdot 2^{2n}}_{\text{mit 2·n Bits nicht darstellbar}}$$

Zusätzliche bedingte Subtraktion der um n Bit linksverschobenen Faktoren vom »vorzeichenfreien« Produkt, wenn das jeweils andere Vorzeichenbit eins ist.

Der AVR-Befehlssatz hat Befehle für die Multiplikationen vorzeichenbehafteter Zahlen, die allerdings der Compiler nicht unbedingt nutzt:

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

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

Aufgabe:

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

Lösung mit Compiler-Optimierung -O1:

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

Gibt r25=H(b*c) und r24=L(b*c)« zurück (H(..) – High-Byte; L(..) – Low-Byte). »CLR R1«, weil in R1 immer null erwartet wird.

Mit Compileroptimierung -O0:

r20 ← b; r18 ← c

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

Die berechneten niederwertigen Produktbytes sind bei vorzeichenbehafteter und vorzeichenfreier Multiplikation gleich.

Fakt 3

Ein Prozessor ist nicht besser als sein Compiler!

Division

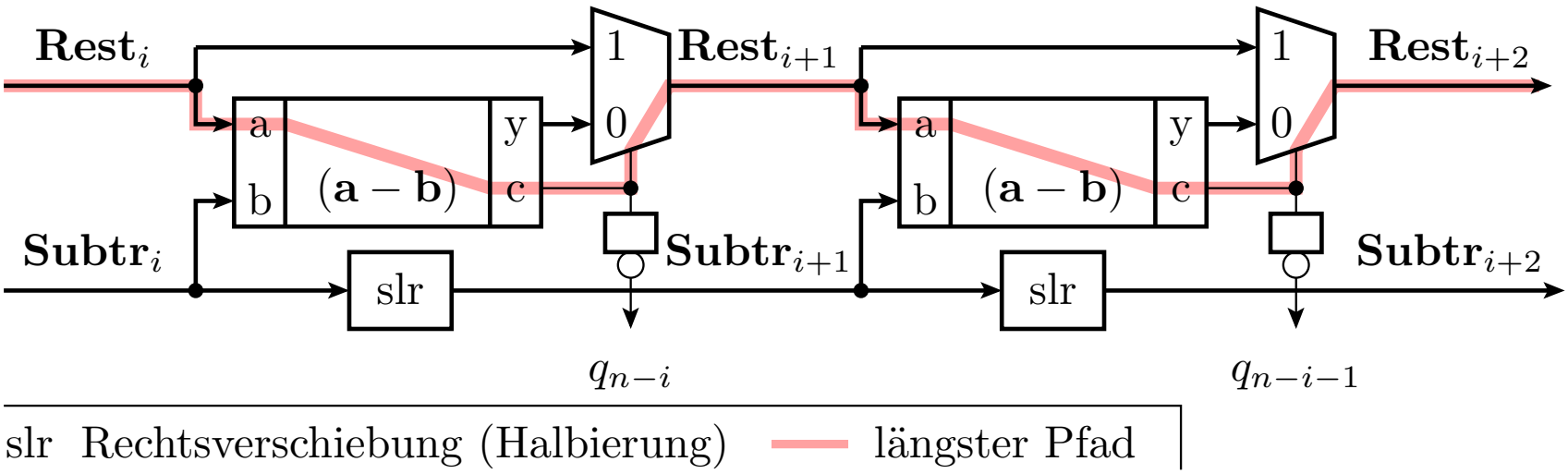
$$\frac{\mathbf{a}}{\mathbf{b}} = \mathbf{q} + \frac{\mathbf{r}}{\mathbf{b}}$$

- Berechnung des Bitvektors q für den ganzzahligen Quotienten und des Bitvektors r für den Divisionsrest

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

$$\frac{11}{3} = 3 + \frac{2}{3}$$

Datenfluss Matrix-Dividierer



- Längster Pfad geht durch alle Subtrahierer. Quadratische Zunahme der Laufzeit mit der Bitanzahl.
- Im Vergleich dazu nimmt die Laufzeit eines Matrixmultiplizierers nur linear mit der Bitbreite zu.
- Die Division wird deshalb in der Regel in einer Schleife bitweise ausgeführt.

C-Operatoren für Division und Divisionsrest

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

- In C werden Divisionsrest und Quotient mit unterschiedlichen Operatoren, d.h. in getrennten Anweisungen bestimmt.
- Wie die nächste Folie zeigt, ruft das Beispielprogramm zweimal desselben Unterprogramm mit denselben Operanden auf.
- Der Divisionsrest wird vom Unterprogramm offenbar in r25 und der Quotient in r24 zurückgegeben.

```
      r = a%b;    // Berechnung des Divisionsrests
00000089  LDS R24,0x0202          r24 ← a
0000008B  LDS R25,0x0200          r25 ← b
0000008D  MOV R22,R25             r22 ← b
0000008E  RCALL PC+0x0011         Unterprogrammaufruf
0000008F  MOV R24,R25
00000090  STS 0x0201,R24          Ergebnis in q speichern
      q = a/b;    // Berechnung des Quotienten
00000092  LDS R24,0x0202          r24 ← a
00000094  LDS R25,0x0200          r25 ← b
00000096  MOV R22,R25             r22 ← b
00000097  RCALL PC+0x0008         Unterprogrammaufruf
00000098  STS 0x0203,R24          Ergebnis in q speichern

--- No source file ------------------------------
0000009F  SUB R25,R25     1. Unterprogrammbefehl: r25← 0
                • • •
```

- Das Unterprogramm können Sie in der Übung analysieren.

Kommazahlen

Festkommazahlen

- Erweiterung ganzer Zahlen um m Nachkommastellen.
- Nachkommastellen haben einen negativen Stellenindex.
- Wert positiver Festkommazahlen:

$$Z = \sum_{i=-m}^{n-1} b_i \cdot B^i$$

(b_i – Ziffernwerte; B – Basis des Zahlensystems)

Beispiel: $23{,}89 = 2 \cdot 10^1 + 3 \cdot 10^0 + 8 \cdot 10^{-1} + 9 \cdot 10^{-2}$

- Wert negativer Festkommazahlen:

$$Z = \sum_{i=-m}^{n-1} b_i \cdot B^i - B^n$$

Beispiel: $-23{,}89 = 7 \cdot 10^1 + 6 \cdot 10^0 + 1 \cdot 10^{-1} + 1 \cdot 10^{-2} \left(-10^2\right)^*$

($*$ – mit n Vorkommastellen nicht darstellbarer Summand)

Für gebrochene Binärzahlen im Zweierkomplement ist das führende Bit das Vorzeichenbit. Wert:

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

Die größte mit n Vorkomma- und m Nachkommabits darstellbare Zahl 01..1,11... hat dem Wert $2^n - 2^{-m}$ und die kleinste darstellbare Zahl 10...0,00... den Wert -2^n.

Fakt 4

Festkommazahlen werden mit den arithmetischen Operationen für ganze Zahlen bearbeitet: Addition, Subtraktion, Verschiebebefehle und Multiplikation.

Es gibt zwei Multiplikationsbefehle für zwei ganz spezielle Festkommaformate:

Operation	Op.-Code	Assembler
R1:R0←Rd$^{(s)}$ · Rr$^{(s)}$	`1000 0010 dddd rrrr`$^{(1)}$	`fmuls Rd, Rr`
R1:R0←Rd$^{(s)}$ · Rr$^{(u)}$	`1000 0011 0ddd 0rrr`$^{(2)}$	`fmulsu Rd, Rr`

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

Schauen Sie in der Befehlssatzbeschreibung nach,

- für welche Festkommaformate diese Befehle gedacht sind und
- was ihre genaue Funktion ist.

Wertebereich vs. Rundungsfehler

Kommaposition	Wertebereich	Rundungsfehler
$n = 2, m = 6$	0 bis $2^2 - 2^{-6}$	$\pm 2^{-7}$
$n = 4, m = 4$	0 bis $2^4 - 2^{-4}$	$\pm 2^{-5}$
$n = 6, m = 2$	0 bis $2^6 - 2^{-2}$	$\pm 2^{-3}$

Bei einer Festkommazahl bestimmt die Anzahl

- der Vorkommastellen den Wertebereich und
- die Anzahl der Nachkommastellen den Rundungsfehler.

Die Wahl der Kommaposition ist ein Kompromiss zwischen der Größe des Wertebereichs und der Genauigkeit.

Gleitkommazahlen (variable Kommaposition)

- Mantisse M: Wertebereich (normiert) $1 \leq Z(M) < 2$
- Charakteristik c: Kommaverschiebung, ganzzahlig; c_0 – Wert von c für Kommaverschiebung Null
- Vorzeichenbit s

Wert für $0 < c < c_{\max}$ (normierte Darstellung[16]):

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

Wert für $c = 0$ (denormiert[17]):

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

[16] In der normierten Darstellung ist die Mantisse M eine vorzeichenfreie Zahl mit einem Vorkommabit gleich eins, das nicht mit gespeichert wird.

[17] In der denormierten Darstellung kann das Vorkommabit auch null sein und wird mit gespeichert.

Sonderwerte $c = c_{\max}$:

$$Z = \begin{cases} \infty & \text{für } s = 0 \text{ und } m = 0 \\ -\infty & \text{für } s = 1 \text{ und } m = 0 \\ \text{nan} & \text{für } m \neq 0 \end{cases}$$

(nan, not a number – ungültig; $\pm\infty$ – positiver/negativer Wertebereichsüberlauf)

- 32-Bit-Format »IEEE-754 single«:

Bitvektor (31 … 24 | 23 … 16 | 15 … 8 | 7 … 0, Bitnummer ←)								Wert
s	c				M			
0	1000001	1	0010010	00000000	00000000			$+1.240000_{16} \cdot 2^{83_{16}-7f_{16}}$
+	$c = 83_{16}$		$M = 1,240000_{16}$					$= 18{,}25$
1	0111100	1	1100101	10011101	00001110			$-1.CB3A1C_{16} \cdot 2^{79_{16}-7f_{16}}$
−	$c = 79_{16}$		$M = 1,CB3A1C_{16}$					$\approx -2{,}8029 \cdot 10^{-2}$
0	0000000	0	0001100	10111101	00011001	00		$+0,32F464_{16} \cdot 2^{0-7f_{16}}$
	0/denorm.		$M = 0,32F464_{16}$					$\approx 1{,}170 \cdot 10^{-39}$

Nutzung von Gleitkommazahlen in C

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

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

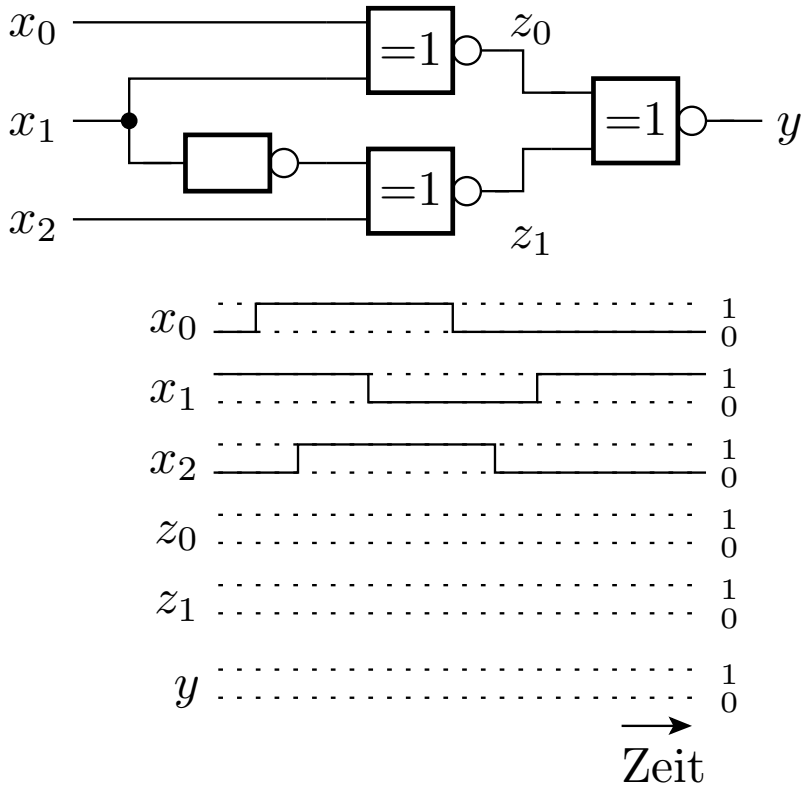
```
data 0x0200  0a d7 6b 41
data 0x0204  5c 8f bd c3
data 0x0208  d4 a1 ae c5
```

- Gleitkommazahlen werden unterstützt und ihre Werte sind im Debugger darstellbar.
- Die Byte-Darstellung ist im Speicher einsehbar.
- Die für die Addition genutzten Unterprogramme umfassen insgesamt über 180 Befehle.
- 32-Bit-Prozessoren haben oft ein Gleitkommarechenwerk, das Gleitkommaoperationen in einem Schritt ausführt.

Aufgaben

Aufgabe 1.3

Füllen Sie für die nachfolgende Schaltung die Wertetabelle aus und bestimmen Sie die Zwischensignalverläufe und den Ausgabesignalverlauf:



x_2	x_1	x_0	z_0	z_1	y
0	0	0			
0	0	1			
0	1	0			
0	1	1			
1	0	0			
1	0	1			
1	1	0			
1	1	1			

Aufgabe 1.4

Bestimmen Sie für jede Zeile der nachfolgenden Tabelle die Registerwerte nach Ausführung der Operation:

	r0	r1	r2	r3	
...	0001 0011	1100 1010	1011 0111	0101 1100	
and r2, r3					
or r0, r1					
eor r0, r2					
and r0, 0b01100111					

Aufgabe 1.5

Es soll folgende vorzeichenbehaftete (arithmetische) Rechtsverschiebung programmiert werden:

```
int16_t a, b;
a = b >> 3;
```

1. Welche Ergebnis soll die Operation für die nachfolgenden Eingaben liefert?

a	0xF3A2	0x41D3
b		

2. Schreiben Sie ein Assemblerprogramm mit folgender Registerzuordnung:

r18	r19	r20	r21
b_0	b_1	a_0	a_1

Lösungshinweise: Wandeln Sie die vorgegebenen hex-Werte in Binärwerte um, verschieben Sie diese. Bei einer arithmetischen Rechtsverschiebung wird in das freiwerdende höchste Bit der Vorzeichenwert geschrieben. Sie benötigen den arithmetischen Rechtsverschiebebefehl »asr Rd«, der in der Vorlesung nicht behandelt wurde.

Aufgabe 1.6

Entwickeln Sie nach dem Algorithmus Gl. 1

$$\begin{aligned} a_0 &= L(b_0 \cdot c_0) \\ \downarrow: a_1 &= H(b_0 \cdot c_0) + L(b_1 \cdot c_0) + L(b_0 \cdot c_1) \\ \downarrow: a_2 &= a_2 + H(b_1 \cdot c_0) + H(b_0 \cdot c_1) + L(b_1 \cdot c_1) \\ a_3 &= a_3 + H(b_1 \cdot c_1) \end{aligned}$$

ein Assemblerprogramm, dass das 4-Byte-Produkt aus den 2-Byte-Faktoren bildet. Registerzuordnung:

r18	r19
b_0	b_1

r20	r21
c_0	c_1

r22	r23	r24	r25
a_0	a_1	a_2	a_3

Aufgabe 1.7

Warum werden Multiplizierwerke in Hardware realisiert, aber Dividierwerke nicht?

Aufgabe 1.8

Stellen Sie den dezimalen Zahlenwert 26,75

1. als vorzeichenfreie Festkommazahl mit 5 Bit vor und 3 Bit nach dem Komma und
2. im 32-Bit-Format »IEEE-754 single«

dar.