Rechnerarchitektur, Foliensatz 1 Einführung, Bitverarbeitung

G. Kemnitz

18. Oktober 2021

Lernziele

- Grundverständnis der Funktionsweise von Rechnern.
- Nachbildung Programmierelemente durch Maschinenbefehle.
- Hardware-nahes programmieren, disassemblieren und debuggen.

Beispielrechner:

- Simulationsmodell eines Minimalprozessors zur Demonstration der internen Befehlsabarbeitung.
- 8-Bit-AVR-Prozessor auf einer Versuchsbaugruppe für praktische Experimente. Programmieraufgaben in C und Assembler.

  Parallele und serielle Schnittstellen, Interrupts und Timer.
- Erweiterung des Simulationsmodells des Minimalprozessors zu einem Pipeline-Rechner.

Übungen im Labor am Rechner.

Foliensätze / Vorlesungsthemen

- F1: Einführung, Bitverarbeitung
- F2: Speicher und Rechenwerk
- F3: Kontrollfluss
- F4: ergänzende Aspekte
- F5: Parallele und serielle Schnittstellen
- F6: Interrupts und Timer
- F7: Pipeline-Verarbeitung

Organisation, Leistungsnachweis

- Ab 8 Vorlesungswoche donnerstags 15:15 Vorlesung (7×).
- Jede Woche Hausübungen. Abgabe zur nächsten Vorlesung.
- Ab der 2. Vorlesung Test am Vorlesungsende über den Inhalt der abgegebenen Hausübung.
- Ab 9. Vorlesungswoche jede Woche Übung im Labor in zwei Gruppen.

Leistungsnachweis:

- 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.

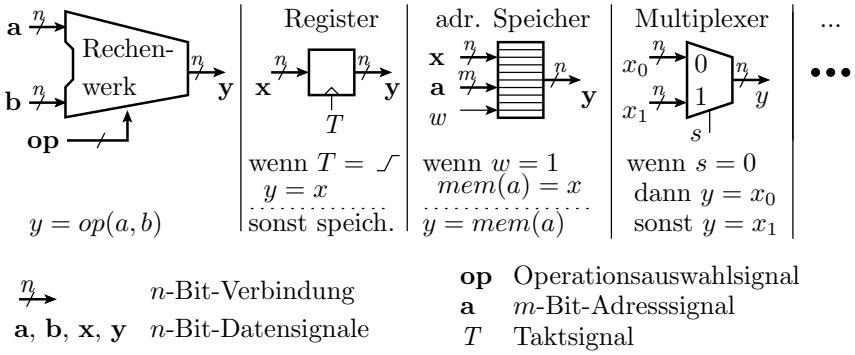
Alternativ[1] mündliche Kenntnisprüfung über den gesamten Übungs- und Vorlesungsstoff[2].

Inhalt F1: Einführung, Bitverarbeitung

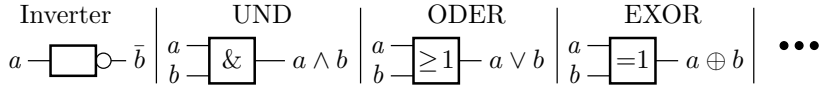
Inhaltsverzeichnis

1 Digitalschaltung zum Rechner

Grundbausteine von Rechnern

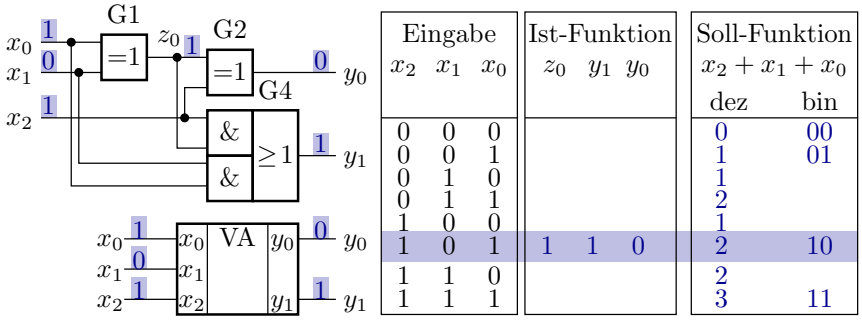


- Rechenwerke, Register, ... bestehen aus Logikgattern:



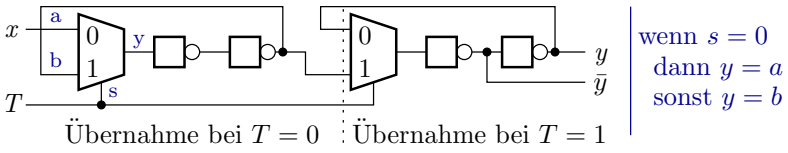
[1] Auch bei entschuldigter Abwesenheit z.B. wegen Krankheit.
[2] Termine beim Dozent erfragen. Prüfungstermine bis max. Ende Februar.

- 1-Bit-Addierer (Volladdierer):

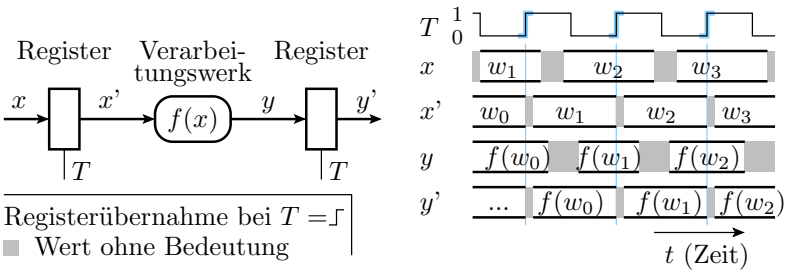


Eingabe x_2	x_1	x_0	Ist-Funktion z_0	y_1	y_0	Soll-Funktion $x_2+x_1+x_0$ dez	bin
0	0	0				0	00
0	0	1				1	01
0	1	0				1	
0	1	1				2	
1	0	0				1	
1	0	1	1	1	0	2	10
1	1	0				2	
1	1	1				3	11

- 1-Bit-Registerzelle: Übernahme bei $T: 0 \rightarrow 1$, sonst speichern

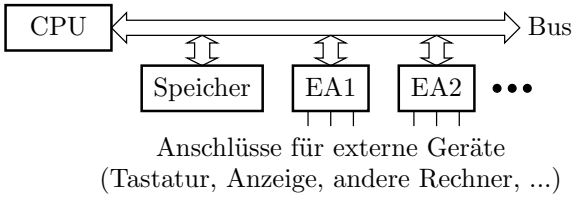


Ein Verarbeitungsschritt dauert einen Takt



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

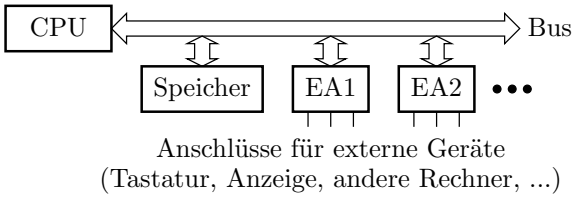
Funktionsweise eines Universalrechners



- Befehle und Daten stehen in einem Speicher.
- Der Prozessor (CPU **C**entral **P**rocessing **U**nit) führt für jeden Befehl eine Folge von Aktionen aus:

- Befehlswort lesen (IF **I**nstruction **F**etch)
- Operanden Laden (OF **O**perand **F**etch)
- Operation ausführen (EX **Ex**ecute)
- Ergebnis schreiben (RW **R**esult **W**rite).

Ein- und Ausgabe



Ein- und Ausgabegeräte:

- für den Prozessor adressierbare Speicher, von denen er Daten lesen und auf die er Daten schreiben kann, und
- für die EA-Geräte les- und beschreibbare Register mit spezieller angeschlossener Hardware z.B. für den seriellen Datenaustausch.

Zeitliche Abstimmung zwischen Programm und EA-Gerät, wann Daten bereit / übernommen ... (siehe später Foliensatz 5 und 6).

2 CISC und RISC

CISC- und RISC-Prozessoren

Ältere Prozessoren haben ein Mikroprogrammsteuerwerk. Befehlsabarbeitung in einer Schleife:

- Wiederhole immer:
  - Laden der Befehlsnummer und starte Mikroprogramm.
    - Laden beliebig vieler Befehlsbestandteile (Adressen, Konstanten, ...)
    - Laden beliebig vieler Operanden.
    - Beliebige Anzahl von Berechnungsschritten.
    - Beliebige Anzahl zu speichernder Ergebnisse.
    - Ein Mikroprogramm kann sogar eine Schleife z.B. zum Kopieren von Datenblöcken variabler Größe enthalten.
  - Berechnung der Adresse für den Folgebefehl.

Dieser Typ von Prozessoren wurde in Abgrenzung zu später entwickelten RISC-Architekturen nachträglich als CISC[3] bezeichnet. CISC-Architekturen haben kaum noch Bedeutung.

[3] CISC – **C**omplex **I**nstruction **S**et **C**omputer

RISC-Architektur[4]

(Fast) nur Befehle, die in einem Schritt[5] abgearbeitet werden:

- einheitliche Befehlswortgröße,
- parallel nutzbare Hardware für jeden Teilschritt (IF, OF, ...).

Vorteile:

- weniger Hardware und schneller als CISC,
- kleinere, einheitlichere Befehlssätze,
- einfachere Compiler-Algorithmen.

Nachteile:

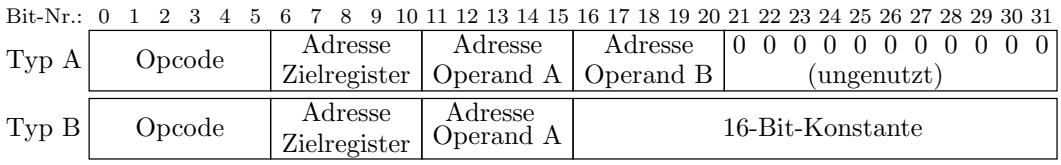
- Mehr Befehle pro Aufgabe und
- erschwerte Programmierung ohne Compiler.

Ein RISC-Prozessor ist nicht besser als sein Compiler.

RISC-Befehlssätze

- Befehlswortgröße typ. 16 bis 32 Bit. Typ. 6 Bit Opcode (Befehlsnummer), 2/3 Registeradressen oder 1/2 Registeradressen + Konstante.
- Load/Store-Prinzip: Zu verarbeitende Daten werden mit Load-Befehlen in Arbeitsregister und Ergebnisse mit Store-Befehlen in den Datenspeicher kopiert.
- 3-Port-Registersatz: In jedem Zeitschritt werden gleichzeitig bis zu 2 Operanden aus dem Registersatz gelesen und ein Ergebnis geschrieben. ...

Befehlsformate Microblaze (32-Bit-Softprozessor für FPGAs):



3 Minimalprozessor MiPro

Befehlsformate MiPro[6]

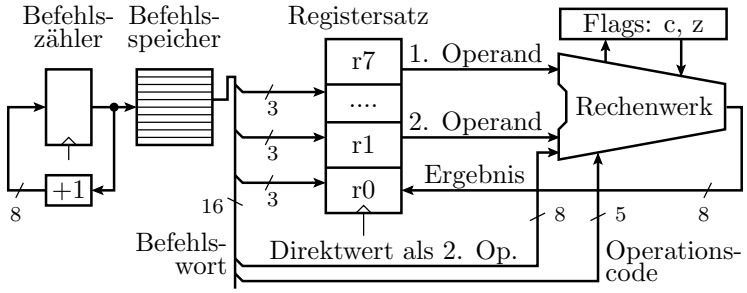
Der Minimalprozessor hat 16-Bit Befehlsworte, 8-Bit-Datenworte und 8-Bit-Adressen, für Befehle und Daten getrennt.

[4] RISC – **R**educed **I**nstruction **S**et **C**omputer.
[5] Genauer in einer Pipeline-Zeitscheibe, siehe Foliensatz 7.
[6] VHDL-Simulationsmodell eines Minimalprozessors.

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

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

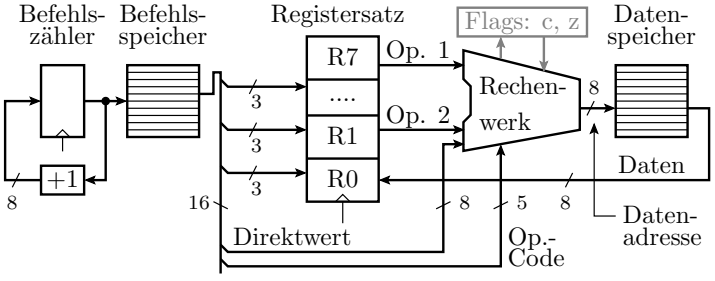
Die Hardware für Verarbeitungsbefehle



Die Hardware für Verarbeitungsbefehle besteht aus

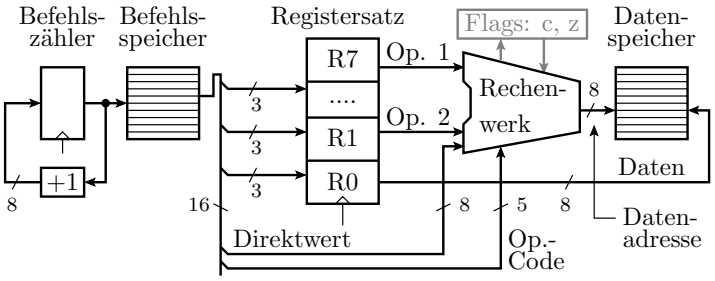
- Befehlszähler, der nach jedem Takt um eins weiterzählt,
- Befehlsspeicher, der das Befehlswort zur Befehlsadresse liefert,
- 8-Bit-Registersatz, der für bis zu 2 Operanden Daten liefert und zum Abschluss der Befehlsausführung das Ergebnis übernimmt,
- Rechenwerk (Ergänzungen für weitere benötigte Befehle folgen).

Erweiterung um Ladeoperationen



- Aus max. zwei Registerinhalten oder einem Registerinhalt und einer Konstanten wird die Adresse berechnet.
- Das Zielregister übernimmt statt des Berechnungsergebnisses den aus dem Datenspeicher gelesenen Wert.

Erweiterung um Speicheroperationen



- Aus max. zwei Registerinhalten oder einem Registerinhalt und einer Konstanten wird die Adresse berechnet.
- Das »Zielregister« wird gelesen und sein Wert unter der berechneten Adresse im Datenspeicher abgelegt.

In späteren Abschnitten: Sprünge, Unterprogramme, ...

4 Bitverarbeitung

Bit, Byte, Wort, ...

Operandengrößen zur Verarbeitung mit einem Rechenwerk:

- Bit: Wertebereich $\{0, 1\}$, z.B. Übertrags-, Überlaufsbit, ...
- Byte: 8 Bit, Interpretation als ganze Zahl:
  - ohne Vorzeichen WB 0 bis $2^8 - 1$,
  - mit Vorzeichen (Zweierkomplement) WB -2^7 bis $2^7 - 1$.
- Wort: 2 Byte, Interpretation als ganze Zahl:
  - ohne Vorzeichen WB 0 bis $2^{16} - 1$,
  - mit Vorzeichen (Zweierkomplement) WB -2^{15} bis $2^{15} - 1$.
- Weitere Typen: 4 Byte, 8 Byte, Gleitkomma, Feld, Struktur, ...

Praktisch jeder Prozessor hat Befehle

- zu Bitverarbeitung: bitweise Logikoperationen, Verschiebebefehle, Einzelbitverarbeitung, ...
- Verarbeitung ganzer Zahlen: Addition, Subtraktion, ...
- ...

Bitweise Logikbefehle

a	b	$a \wedge b$	$a \vee b$	$a \oplus b$	$\bar{b}$
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

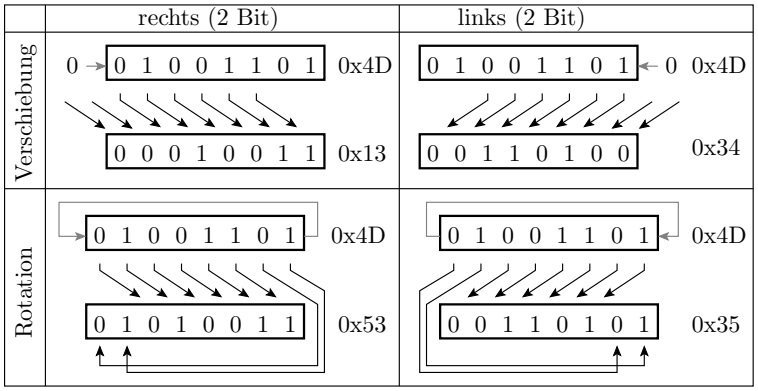
Beispiele:

$$\begin{array}{rr} a & 1011\ 0110\ \ (0xB6) \\ b & 1001\ 0011\ \ (0x93) \\ \hline a \wedge b & 1001\ 0010\ \ (0x92) \end{array} \qquad \begin{array}{rr} a & 1011\ 0110\ \ (0xB6) \\ b & 1001\ 0011\ \ (0x93) \\ \hline a \vee b & 1011\ 0111\ \ (0xB7) \end{array}$$

$$\begin{array}{rr} a & 1011\ 0110\ \ (0xB6) \\ b & 1001\ 0011\ \ (0x93) \\ \hline a \oplus b & 0010\ 0101\ \ (0x25) \end{array} \qquad \begin{array}{rr} b & 1001\ 0011\ \ (0x93) \\ \hline \bar{b} & 0110\ 1100\ \ (0x6C) \end{array}$$

- Einfachste mit Logikgattern realisierbare Funktionen.
- Nutzung innerhalb von Programmen z.B. zum Setzen, Löschen, Invertieren einzelner Bits in Datenobjekten.

Verschiebung und Rotation



- Verschiebung: Füllen freiwerdender Bits mit null.
- Rotation: Verschiebung im Kreis.

Bit- und Verschiebeoperationen in C

```
uint8_t a=0xB6, b=0x93, x = 0x4D, y;
...
y = a & b; // bitweises UND  (y: 0x92)
y = a | b; // bitweises ODER (y: 0xB7)
y = a ^ b; // bitweises EXOR (y: 0x25)
y =    ~b; // bitw. Negation (y: 0x6C)
y = x >> 2;// Rechtsverschiebung um 2 Bit (y: 0x13)
y = x << 2;// Linksverschiebung um 2 Bit  (y: 0x34)
y = (x>>2)|(x<<6);// Rechtsrot. um 2 Bit  (y: 0x53)
```

- Für 1, 2, 4, 8 Byte-Datenobjekte.
- Die Verschiebung kann auch eine Variable sein.

Logik- und Verschiebebefehle werden genutzt:

- Verarbeitung von Schaltereingaben,
- Verkettung und Trennen von Teilbitvektoren,
- Nachbildung der Multiplikation und Division, ...

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

Logikbefehle des Minimalprozessors

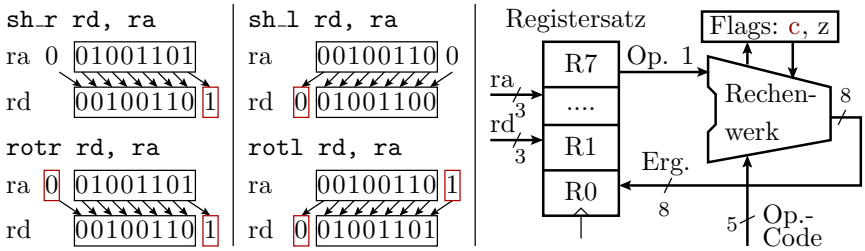
Befehl	Operation	Flags *	cnr
andr rd,ra,rb	rd := ra & rb	z	28
andi rd,imm	rd := rd & imm	z	12
or_r rd,ra,rb	rd := ra | rb	z	29
or_i rd,imm	rd := rd | imm	z	13
xorr rd,ra,rb	rd := ra ^ rb	z	30
xori rd,imm	rd := rd ^ imm	z	14
notr rd,ra	rd := ~ra	z	23

& – UND; | – ODER; ^ – EXOR; ~ – Negation, jeweils bitweise; * – veränderte Flags: $z = 1$ wenn Ergebnis '0' sonst '1' (c unverändert).

Befehle zum Initialisieren und Kopieren von Registerinhalten:

ld_i rd,imm	rd := imm		5
move rd,ra	rd := ra		16

Verschiebe- und Rotationsbefehle



sh_r rd,ra	rd := 0:ra >> 1	c,z	20
rotr rd,ra	rd := c:ra >> 1	c,z	22
sh_l rd,ra	rd := ra:0 << 1	c,z	19
rotl rd,ra	rd := ra:c << 1	c,z	21

- 0:ra – Verkettung von null und ra zu einem 9-Bit-Wert.
- 1-Bit-Verschiebung, rausgeschobenes Bit in c.
- Verschiebebefehle übernehmen in das freiwerdende Bit null, Rotationsbefehle den bisherigen Wert von c.

Beispielaufgabe

Welche Werte stehen nach Befehlsausführung in den Registern?

Befehl	Werte nach Befehlsausführung				
	r0	r1	r2	c	z
	.....	.....	.....	.	.
ld_i r0,45					
ld_i r1,6E					
move r2,r0					
andr r0,r0,r1					
notr r0,r1					
or_r r2,r2,r1					
xorr r0,r0,r2					
sh_l r2,r0					
rotl r0,r1					

(. – unbekannter Bitwert; *** – Wert unbeeinflusst).

Lösung

Befehl	Werte nach Befehlsausführung				
	r0	r1	r2	c	z
ld_i r0,45	01000101	.....	.....	.	.
ld_i r1,6E	********	01101110	.....	.	.
move r2,r0	********	********	01000101	.	.
andr r0,r0,r1	01000100	********	********	.	0
notr r0,r1	10010001	********	********	.	0
or_r r2,r2,r1	********	********	01101111	.	0
xorr r0,r0,r2	11111110	********	********	.	0
sh_l r2,r0	********	********	11111100	1	0
rotl r0,r1	11011101	********	********	0	0
Endwert:	11011101	01101110	11111100	0	0

(... – Wert unbestimmt; *** – Wert unbeeinflusst).

5 Vorbereitung Übung 1

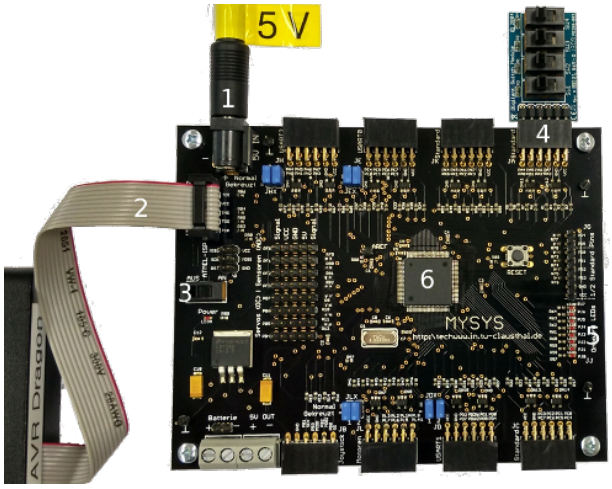
Ziel und Inhalt der ersten Übung

Entwicklung und Test eines Programms mit Logikbefehlen

- in C. Test mit Debugger, Schaltern und LEDs.
- Inspektion des dissasemblierten Codes.
- Programmieren in Assembler, ...

Versuchs-Hardware: Mikrorechner-Board mit ATmega 2560, LEDs, Schaltern, ...

1. Anschluss 5V-Netzteil
2. Anschluss Programmer
3. Ein-Ausschalter
4. Schaltermodul
5. Ausgabe LEDs
6. Prozessor



Test mit Schaltereingabe und LED-Ausgabe

```
#include <avr/io.h>
uint8_t a, b, x, y;
int main(void){
  DDRA = 0x00;            // PORTA als Eingang
  DDRJ = 0xFF;            // PORTJ als Ausgang
  while(1){
    x = PINA;             // x(3:0) := Schalter(3:0)
    a = 0b11 & x;         // a(1:0) := Schalter(1:0)
    b = 0b11 & (x >> 2);// b(1:0) := Schalter(3:2)
    y = (a & b) | (a|b)<<2 | (a^b)<<4 | ~a <<6;
    PORTJ = y;            // Ausgabe auf die 8 LEDs
  }
}
```

LED-Ausgabe (s_i – Wert von Schalter i):

LED8	LED7	LED6	LED5	LED4	LED3	LED2	LED1
$\bar{s}_1$	$\bar{s}_0$	$s_1 \oplus s_3$	$s_0 \oplus s_2$	$s_1 \vee s_3$	$s_0 \vee s_2$	$s_1 \wedge s_3$	$s_0 \wedge s_2$

AVR-Logikbefehle

AVR-Prozessoren haben wie der Minimalprozessor bitweise Logikbefehle, Verschiebe- und Rotationsbefehle um ein Bit und einige Spezialbitbefehle. Bitweise Logikbefehle:

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 := ~Rd	1001 010d dddd 0000	com Rd

(Rd – Zielregister und erster Operand[7]; Rr – 2. Operand; K – 8-Bit-Konstante, * Nur anwendbar auf Register R16 bis R31).

[7] Eine dritte 5-Bit-Adresse passt nicht in das Befehlswort.

ATmega-Verschiebebefehle

Operation	Operationscode	Assembler
C ← 0	1001 0100 1000 1000	clc
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

(clc – **Cl**ear **C**arry Flag; sec – **Se**t **C**arry Flag; lsl – **L**ogical **S**hift **L**eft, identisch mit add Rd, Rd; rol – **Ro**tate **L**eft, identisch mit adc Rd, Rd)

Dissassemblierte Programmausschnitte

```
   uint8_t   a,       b,       x,       y; ...
//Adressen: &a=0x203 &b=0x200 &x=0x201 &y=0x202
   while(1){
    x = PINA;              // x := sw (Schalter)
    a = 0b11 & x;          // a(1:0) := sw(1:0)
//  00098  LDS R24,0x0201  ; r24 := x
//  0009A  ANDI R24,0x03   ; r24 := r24 & 0b11
//  0009B  STS 0x0203,R24  ; a   := r24
    b = 0b11 & (x>>2);     // b(1:0) := sw(3:2)
//  0009D  LDS R24,0x0201  ; r24 := x
//  0009F  LSR R24         ; r24 := r24 >> 1
//  000A0  LSR R24         ; r24 := r24 >> 1
//  000A1  ANDI R24,0x03   ; r24 := r24 & 0b11
//  000A2  STS 0x0200,R24  ; b   := r24
    y = (a&b)|(a|b)<<2;    // Verarbeitung
    PORTJ = y;             // Ausgabe auf 8 LEDs
   }
```

```
// Adressen: &a=0x203 &b=0x200 &x=0x201 &y=0x202
   y = (a & b) | (a | b)<<2;
// y(1:0):=sw(1:0)&sw(3:2); y(3:2):=sw(1:0)|sw(3:2)
// 0x00A4  LDS R25,0x0203 ; r25 := a
// 0x00A6  LDS R24,0x0200 ; r24 := b
// 0x00A8  AND R24,R25    ; r24 := r24 & r25
// 0x00A9  MOV R18,R24    ; r18 := r24
// 0x00AA  LDS R25,0x0203 ; r25 := a
// 0x00AC  LDS R24,0x0200 ; r24 := b
// 0x00AE  OR R24,R25     ; r24 := r24 | r25
// 0x00AF  MOV R24,R24    ; r24 := r24 (sinnlos)
// 0x00B0  LDI R25,0x00   ; r25 := 0   (sinnlos)
// 0x00B1  LSL R24        ; r24:r25 >> 1
// 0x00B2  ROL R25        ;
// 0x00B3  LSL R24        ; r24:r25 >> 1
// 0x00B4  ROL R25        ;
```

```
// 0x00B5  OR R24,R18      ; r24 := r24 | r18
// 0x00B6  STS 0x0202,R24  ; y   := r24
```

Anmerkungen

- Variablen (globale) erhalten bei der Vereinbarung eine feste Adresse (&a – Adresse von a):

```
uint8_t a, b, x, y;
//Adressen: &a: 0x203; &b: 0x200;
//          &x: 0x201; &y: 0x202
```

- Das Programm wurde mit Compiler-Optimierung »-O0« übersetzt. Bei dieser Optimierungsstufe wird jede C-Anweisung einzeln übersetzt, beginnend mit Laden der Variablen aus dem Speicher bis zum Zurückspeichern der Ergebnisse.

```
    a = 0b11 & x;
// 0x0098  LDS R24,0x0201  ; r24 := x
// 0x009A  ANDI R24,0x03   ; r24 := r24 & 0b11
// 0x009B  STS 0x0203,R24  ; a   := r24
```

Programmieren in Assembler

```
.global main
main:           ; Marke für Programmstartadresse
 ldi r16, 0x18  ; r16 := 0b.... ....
 ldi r17, 0x24  ; r17 := 0b.... ....
 ori r16, 0x03  ; r16 := 0b.... ....
 andi r16, 0xFE ; r16 := 0b.... ....
 eor r17, r16   ; r17 := 0b.... ....
 ret            ; Programmende (Rücksprung)
```

»main« ist die Marke für die Startadresse eines C-Programms und muss als »global« vereinbart sein. Jede Assembler-Anweisung wird in einen Maschinenbefehl übersetzt:

```
Adresse   Befehl        |Adresse   Befehl
0x0007D  LDI R16,0x18  |0x00080  ANDI R16,0xFE
0x0007E  LDI R17,0x24  |0x00081  EOR R17,R16
0x0007F  ORI R16,0x03  |0x00082  RET
```

Spezielle Bitverarbeitungsoperationen

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

```
sbi A, b ; setze Bit b in EA-Register A
cbi A, b ; lösche Bit b in EA-Register A
bld Rd, b; Bit LoaD, kopiere Bit b aus Register
         ; Rd in das Statusregisterbit T
bst Rd, b; Bit STore, kopiere T nach Rd Bit b
```

Der Befehl

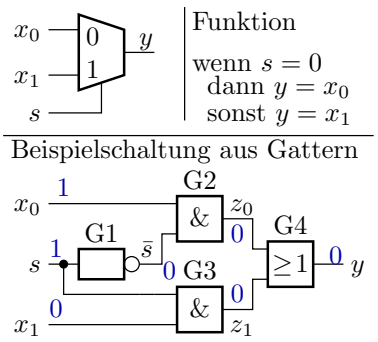
```
sbi 8, 3; setze Bit 3 in Port C
```

ersetzt z.B. die Befehlsfolge:

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

6 Aufgaben

Aufgabe 1.1: Multiplexer



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

Der dargestellte 1-Bit-Umschalter (Multiplexer) soll für $s = 0$ den Wert von x_0 und für $s = 1$ den Wert von x_1 weiterleiten. Kontrollieren Sie die Soll-Ausgaben und vervollständigen Sie die Ist-Ausgaben.

Aufgabe 1.2: Bits und Bytes

1. Wie viele Register sind mit 5-Bit-Adressen auswählbar?
2. Wie viele Befehls- und wie viele Datenspeicherplätze kann ein AVR-Prozessor mit 17-Bit-Befehls- und 16-Bit-Datenadressen maximal adressieren?
3. Bis zu wie viele Datenbytes und bis zu wie viele Befehlsworte kann ein 32-Bit-Prozessor direkt adressieren?
4. Bis zu wie viele Datenbytes und bis zu wie viele Befehlsworte kann ein 64-Bit-Prozessor direkt adressieren?

Lösung

1. 5-Bit-Registeradressen: $2^5 = 32$ Register.
2. 17-Bit-Befehlsadresse: $2^{17} = 128\,\text{k}$ Befehlsworte. 16-Bit-Datenadressen: $2^{16} = 64\,\text{k}$Bytes Daten.
3. 32-Bit-Prozessor: $2^{32} = 4\,\text{G}$Byte Daten und 2^{30} Befehlsworte.
4. 64-Bit-Prozessor: $2^{64} = 4 \cdot 2^{32}\,\text{G}$Byte Daten und 2^{62} Befehlsworte.

Aufgabe 1.3: Trace-Tabelle ausfüllen (MiPro)

Ergänzen Sie in der nachfolgenden Trace-Tabelle für die Programmabarbeitung auf dem Minimalprozessor die in die Register geschriebenen Werte als 2-stellige Hexadezimalzahlen:

PC	Befehl assem.: hex	r0	r1	r2	r3	r4	r5	r6	r7	z
00	ld_i r1,4a,..:294a	..	..	..	..	..	..	..	..	.
01	ld_i r0,73,..:2873	..	..	..	..	..	..	..	..	.
02	move r2,r0,..:8200	..	..	..	..	..	..	..	..	.
03	move r3,r1,..:8320	..	..	..	..	..	..	..	..	.
04	andi r2,0f,..:620f	..	..	..	..	..	..	..	..	.
05	andi r3,f0,..:63f0	..	..	..	..	..	..	..	..	.
06	or_r r4,r2,r3:ec4c	..	..	..	..	..	..	..	..	.
07	xorr r5,r4,r4:f590	..	..	..	..	..	..	..	..	.

(** – Wert unverändert; .. – Wert gesucht). Wenn das Ergebnis einer logischen Operation 0x00 ist, wird das Zero-Flag »z« auf eins und sonst auf null gesetzt.

Lösung

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|z|
00|ld_i r1,4a,..:294a|.. 4a .. .. .. .. .. ..|.|
01|ld_i r0,73,..:2873|73 ** .. .. .. .. .. ..|.|
02|move r2,r0,..:8200|** ** 73 .. .. .. .. ..|.|
03|move r3,r1,..:8320|** ** ** 4a .. .. .. ..|.|
04|andi r2,0f,..:620f|** ** 03 ** .. .. .. ..|0|
05|andi r3,f0,..:63f0|** ** ** 40 .. .. .. ..|*|
06|or_r r4,r2,r3:ec4c|** ** ** ** 43 .. .. ..|*|
07|xorr r5,r4,r4:f590|** ** ** ** ** 00 .. ..|1|
```

Aufgabe 1.4: Logikprogramm mit MiPro

Schreiben Sie für den Minimalprozessor eine Befehlsfolge, die den Bitvektor **x**=0xB3 in Register r0 schreibt und daraus einen Bitvektor **y** in Register r1 nach folgender Vorschrift bildet:

y_7	y_6	y_5	...	y_2	y_1	y_0
$x_7 \oplus x_6$	$x_6 \oplus x_5$	$x_5 \oplus x_4$	...	$x_2 \oplus x_1$	$x_1 \oplus x_0$	$x_0 \oplus x_7$

1. Welchen Wert erhält y für **x**=0xB3?
2. Kontrollieren Sie durch Ausfüllen der nachfolgenden Tabelle, dass das Programm dieses Ergebnis liefert.

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|z|
00|ld_i r0,b3,..:28b3|b3 00 00 00 00 00 00 00|0|
01|.... .. ,.. ,  :  |.. .. .. .. .. .. .. ..|.|
02|.... .. ,.. ,  :  |.. .. .. .. .. .. .. ..|.|
03|.... .. ,.. ,  :  |.. .. .. .. .. .. .. ..|.|
04|.... .. ,.. ,  :  |.. .. .. .. .. .. .. ..|.|
```

Lösung

1. Sollwertberechnung für **y**:

$$\begin{array}{rcccccccc} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \quad \text{0xb3} \\ \oplus & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \quad \text{0x67} \\ \hline & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \quad \text{0xd4} \end{array}$$

2. Programm zur Bildung von **y** in r1 und Simulation:

```
PC|Befehl assem.: hex|r0 r1 r2 r3 r4 r5 r6 r7|z|
00|ld_i r0,b3,..:28b3|b3 00 00 00 00 00 00 00|0|
01|sh_l r2,r0,..:9a00|** ** 66 ** ** ** ** **|*|
02|rotl r3,r0,..:ab00|** ** ** 67 ** ** ** **|*|
03|xorr r1,r0,r3:f10c|** d4 ** ** ** ** ** **|*|
04|.... .. ,.. ,  :  |.. .. .. .. .. .. .. ..|.|
```

Aufgabe 1.5: Logikverarbeitung mit ATmega 2560

Welche Werte werden den Register r16 und r17 zugewiesen?

```
.global main
main:
 ldi r16, 0x18 ; r16 := 0b.... ....
 ldi r17, 0x24 ; r17 := 0b.... ....
 ori r16, 0x03 ; r16 := 0b.... ....
 andi r16, 0xFE; r16 := 0b.... ....
 eor r17, r16  ; r17 := 0b.... ....
 ret           ; Programmende
```

Lösung

```
.global main
main:
 ldi r16, 0x18 ; r16 := 0b0001 1000
 ldi r17, 0x24 ; r17 := 0b0010 0100
 ori r16, 0x03 ; r16 := 0b0001 1011
 andi r16, 0xFE; r16 := 0b0001 1010
 eor r17, r16  ; r17 := 0b0011 1110
 com r17       ; r17 := 0b1100 0001
 ret           ; Programmende
```