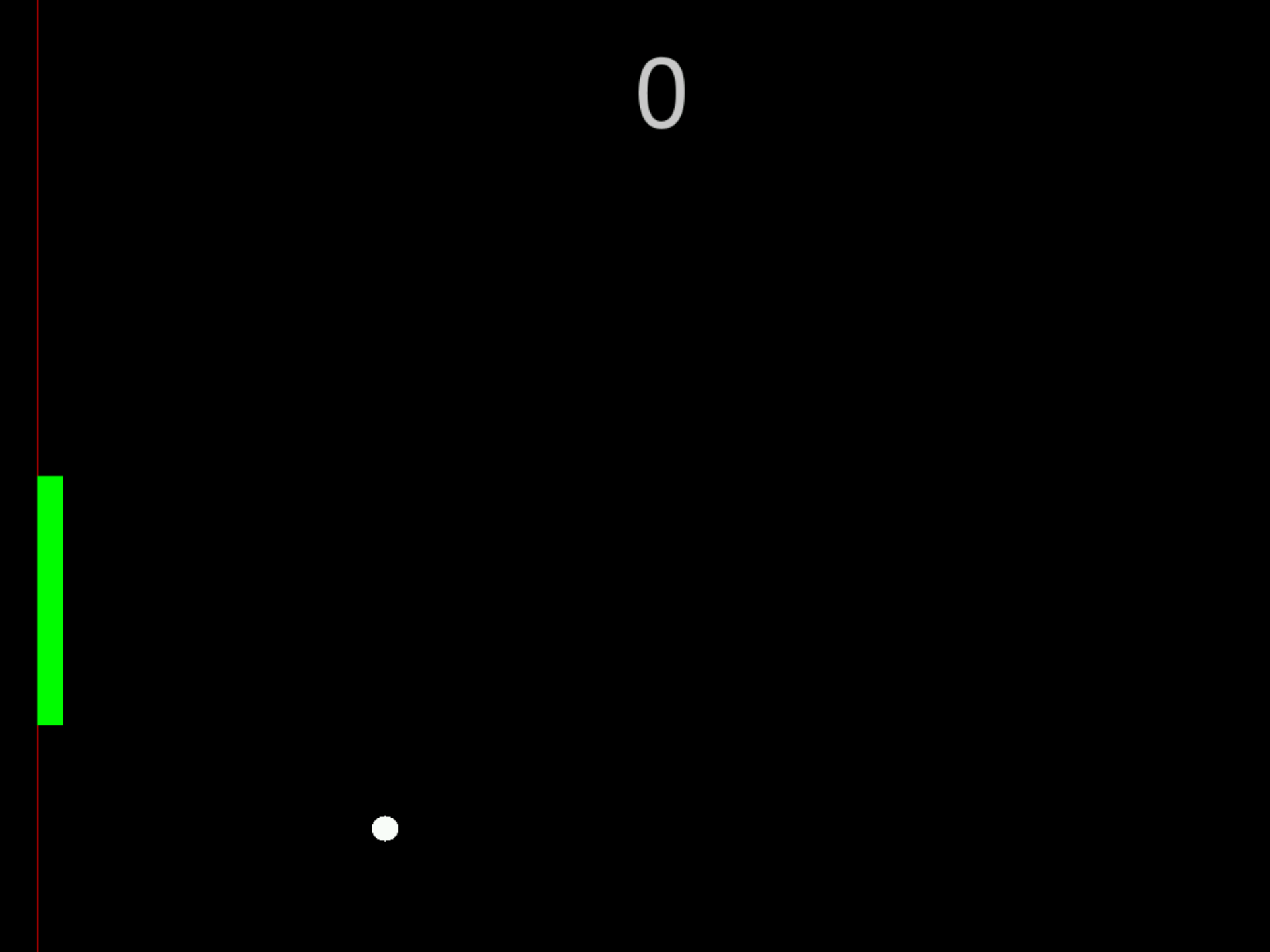
03-05-H -709.53 Informatik für Nichtinformatiker (11)

Prof. Dr. Udo Frese
Tobias Hammer

Wie funktioniert ein Computer?

Pong in Ruby


0

Pong in Ruby

- **Eingabe**
  - Wie merkt der Computer, wenn man eine Taste drückt?
  - Taste → @events.each
- **Verarbeitung**
  - Wie „malt“ der Computer ein Rechteck im Speicher?
  - @screen.draw_box_s
- **Ausgabe**
  - Wie können wir das Rechteck aus dem Speicher sehen?
  - @screen.update→Bildschirm

```
@events.each do |event|
    if event.is_a?
   (Rubygame::KeyDownEvent) and
   event.key==Rubygame::K_UP then …
End

@screen.draw_box_s(
   [@club_x,@club_y],
   [@club_x+@club_width,
   @club_y+@club_height], [0,255,0])

@screen.update
```

Pong in Ruby

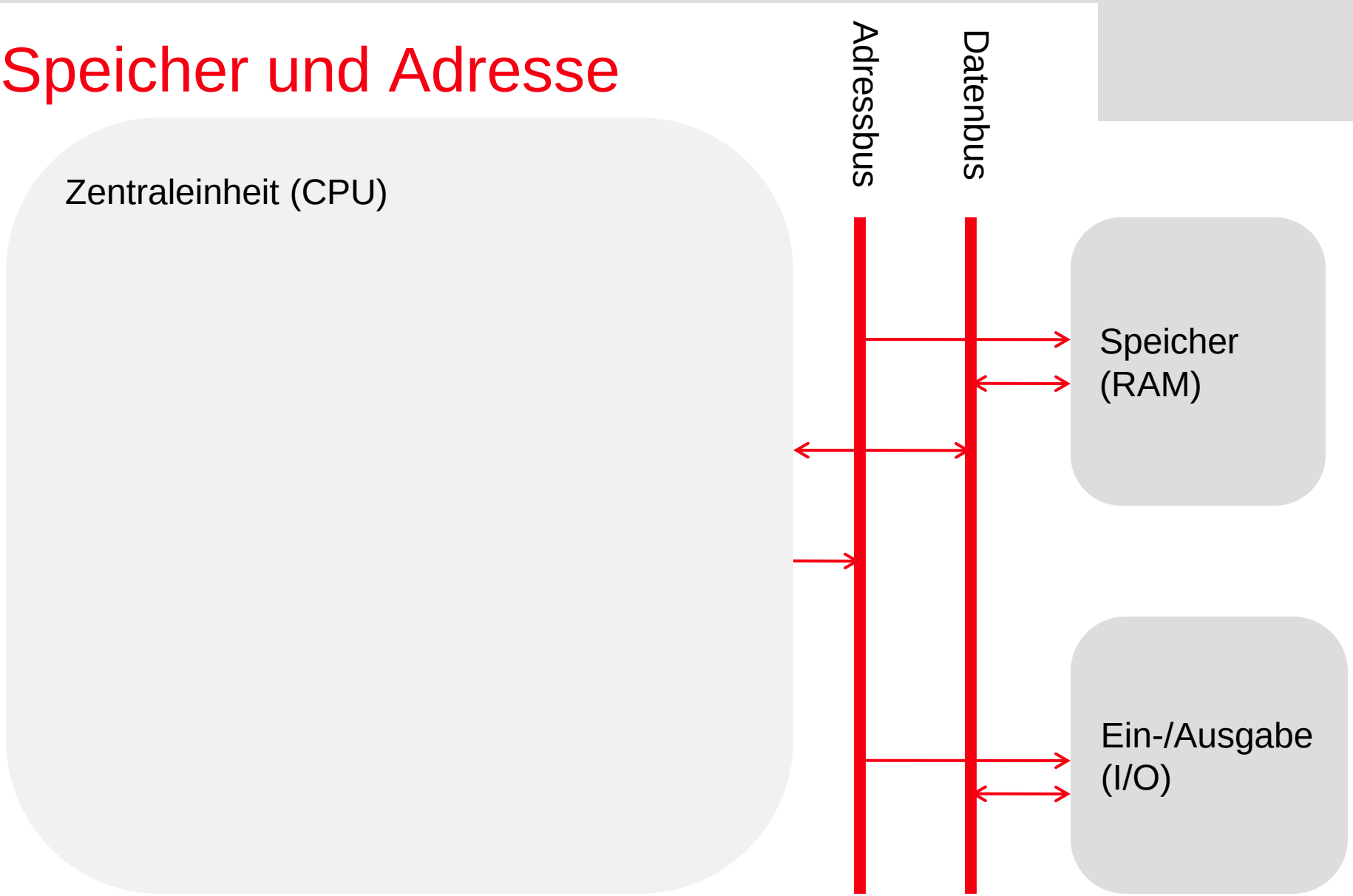
Eingabe (Tastatur)	Verarbeitung (Rechteck)	Ausgabe (Bild)
	Speicher und Adresse	
	Assembler/Maschinensprache	
Tastatur		
Ereignisse (Interrupt)		
	von Neumann Architektur	
	Addierer	
	Gatter und Speicher	
		Grafik (LCD)

Zielsetzung

- Informatik-Systeme haben viele aufeinander aufbauenden Ebenen
- Ebenen eines Computers schlaglichtartig gesehen haben
- Darstellung vereinfacht

Speicher und Adresse

Speicher und Adresse



Speicher und Adresse

Von Neumann-Architektur (Grob)

- **Zentraleinheit (CPU)**
  - Aktiver Teil des Computers
  - Führt das Programm aus
  - Rechnet
  - Steuert Speicher und Ein- / Ausgabe an
- **Speicher**
  - Bewahrt Daten auf
  - Speichert zu jeder Adresse eine Zahl (feste Anzahl Binärstellen)
- **Ein- / Ausgabe (E/A)**
  - Verschiedenste Geräte (Tastatur, Bildschirm, Festplatte, Drucker, Audio)
  - Von Zentraleinheit wie Speicher angesprochen
  - In den E/A-Speicher geschriebene Zahlen bewirken etwas
  - Externe Ereignisse verändern Zahlen in E/A Speicher

Speicher und Adresse

Speicherbenutzung

Adresse →

Daten ↔

Speicher (RAM)

	+0	+1	+2	+3	+4	+5	+6	+7
0	17	4	103	45	2	7	234	120
8	43	192	171	88	1	0	0	0
16	200	201	205	141	102	102	43	3
24	67	69	9	7	121	207	3	7
32	121	67	19	9	65	67	69	111
40	134	200	187	167	201	202	207	1
48	2	3	4	10	12	14	16	18
56	100	101	9	200	200	0	0	0
64	1	1	2	201	202	207	209	1

Speicher und Adresse

Speicherbenutzung

- **Lesen**

10 →

← 171

Speicher (RAM)

	+0	+1	+2	+3	+4	+5	+6	+7
0	17	4	103	45	2	7	234	120
8	43	192	**171**	88	1	0	0	0
16	200	201	205	141	102	102	43	3
24	67	69	9	7	121	207	3	7
32	121	67	19	9	65	67	69	111
40	134	200	187	167	201	202	207	1
48	2	3	4	10	12	14	16	18
56	100	101	9	200	200	0	0	0
64	1	1	2	201	202	207	209	1

Speicher und Adresse

Speicherbenutzung

- **Schreiben**

38 →

100 →

Speicher (RAM)

	+0	+1	+2	+3	+4	+5	+6	+7
0	17	4	103	45	2	7	234	120
8	43	192	171	88	1	0	0	0
16	200	201	205	141	102	102	43	3
24	67	69	9	7	121	207	3	7
32	121	67	19	9	65	67	**100**	111
40	134	200	187	167	201	202	207	1
48	2	3	4	10	12	14	16	18
56	100	101	9	200	200	0	0	0
64	1	1	2	201	202	207	209	1

Speicher und Adresse

Speicherbenutzung

- **Lesen**

38 →

← 100

Speicher (RAM)

	+0	+1	+2	+3	+4	+5	+6	+7
0	17	4	103	45	2	7	234	120
8	43	192	171	88	1	0	0	0
16	200	201	205	141	102	102	43	3
24	67	69	9	7	121	207	3	7
32	121	67	19	9	65	67	**100**	111
40	134	200	187	167	201	202	207	1
48	2	3	4	10	12	14	16	18
56	100	101	9	200	200	0	0	0
64	1	1	2	201	202	207	209	1

Speicher und Adresse

Prinzip der Digitalisierung

- **Alle Daten, egal welchen Typs auf Folge von Zahlen abgebildet**
  - Zahl → direkt
  - Zeichen → nach Codetabelle (ASCII) in Zahl umgewandelt
  - Zeichenkette → Folge von Zahlen
  - (Objekt -) Referenz → Adresse
  - Objekt →
    - *Referenz der Klasse als Beschreibung*
    - *Instanzvariablen hintereinander*
  - Array → Länge, Folge von Referenzen
  - Bild → Zeilenweise Folge von 3 Zahlen (R,G,B) je Pixel
  - Sound→ Folge von Druckwerten
- **Inhalt der Daten ist Frage der Interpretation**
- **⇒ beliebige Daten können von der selben Hardware gespeichert / verarbeitet / verwaltet werden**

Speicher und Adresse

Bild im Speicher

- **Bild w×h hat h Zeilen á w Pixel á 3 Zahlen (R, G, B)**
- **Adresse von (x,y) → Basis + 3*x + 3*w*y**
- **Rechnen mit Adressen für systematischen Zugriff auf Daten**

	+0	+3	+6	+9	+12	+15	+18	+21
0w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
3w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
6w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
9w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
12w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
15w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
18w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
21w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0

Speicher und Adresse

Bild im Speicher

- **Rechteck füllen**
- **@screen.draw_box_s([1,2],[6,3],[255,0,0])**

	+0	+3	+6	+9	+12	+15	+18	+21
3w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
6w	0,0,0	0,,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
9w	0,0,0	255,0,0	255,0,0	255,0,0	255,0,0	255,0,0	255,0,0	0,0,0
12w	0,0,0	255,0,0	255,0,0	255,0,0	255,0,0	255,0,0	255,0,0	0,0,0
15w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
18w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
21w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0
24w	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0	0,0,0

Assembler/Maschinensprache

Assembler/Maschinensprache

- **Programmiersprache bestehend aus Befehlen, die die Zentraleinheit des Computers direkt versteht**
- **Maschinensprache: Binär in Zahlen codiert für die Maschine**
- **Assembler: Symbolisch als Text codiert für den Menschen**
- **Umständlich und fehleranfällig**
- **Aber direkt und sehr schnell**

Assembler/Maschinensprache

Draw_box_s

- **@screen.draw_box_s ([1,2],[6,3],[255,0,0])**
- **(fiktive) Rubymethode zum Füllen eines Rechteckes**

```
def draw_box_s (p1, p2, color)
  r = color[0]
  g = color[1]
  b = color[2]
  y = p1[1]
  while y<=p2[1] do
     x = p1[0]
     while x<=p2[0] do
        address = basis+3*x+3*width*y
        [address]  = r
        [address+1]= g
        [address+2]= b
        x += 1
     end
     y += 1
  end
end
```

Assembler/Maschinensprache

Draw_box_s

- @screen.draw_box_s ([1,2],[6,3],[255,0,0])
- (fiktive) Rubymethode zum Füllen eines Rechteckes
- Optimierung: Adresse nicht immer neu ausrechnen

```
def draw_box_s (p1, p2, color)
  r = color[0];   g = color[1]
  b = color[2];   y = p1[1]
  xEnd = p2[0]
  while y<=p2[1] do
     x = p1[0]
     address = basis+3*x+3*width*y
     while x<=xEnd do
        [address]  = r
        [address+1]= g
        [address+2]= b
        x+=1
        address+=3
     end
     y+=1
  end
end
```

Assembler/Maschinensprache

Draw_box_s

- (fiktive) Rubymethode zum Füllen eines Rechteckes
- Optimierung: Adresse nicht immer neu ausrechnen
- Innerste Schleife in Assembler

```
def draw_box_s (p1, p2, color)
  r = color[0];   g = color[1]
  b = color[2];   y = p1[1]
  xEnd = p2[0]
  while y<=p2[1] do
     x = p1[0]
     address = basis+3*x+3*width*y
     while x<=xEnd do
        [address]  = r
        [address+1]= g
        [address+2]= b
        x+=1
        address+=3
     end
     y+=1
  end
end
```

Assembler/Maschinensprache

Grundkonzepte

- **Register (z.B.: R0-7)**
  - Spezieller Variablen/Speicher in der Zentraleinheit für eine Zahl
  - Schneller und flexibler Zugriff
- **Akkumulator (z.B. R0)**
  - Besonderes Register für Ergebnisse von Rechnungen
- **Befehl**
  - Anweisung die eine Elementaroperation ausführt
  - Arithmetisches Verknüpfen von Akkumulator und Register
  - Laden/Speichern von Register nach Speicher
  - Übertragen von Register nach Register
  - (Bedingter) Sprung

Assembler/Maschinensprache

Liste der (fiktiven) Assemblerbefehle

Befehl	Bedeutung	Erklärung
`MOV Ra, Rb`	Ra = Rb	Kopiere Inhalt von Register Rb nach Ra
`LOD Ra, Rb`	Ra = [Rb]	Kopiere Inhalt der Speicherstelle mit Adresse Rb nach Ra
`STO Ra, Rb`	[Ra]=Rb	Kopiere Inhalt von Rb in die Speicherstelle mit Adresse Ra
`ADD Ra`	R0=R0+Ra	Addiere Ra auf R0 (Ergebnis in R0)
`SUB Ra`	R0=R0-Ra	Subtrahiere Ra von R0 (Ergebnis in R0)
`CMP Ra`	Ra<=>R0	Vergleich Ra mit R0
`JLE Ziel`		Wenn Vergleich <= ergab, springe nach Ziel
`JMP Ziel`		Springe nach Ziel

Assembler/Maschinensprache

Innerste Schleife in Assembler

```
# R1=address, R2=x, R3=r, R4=g, R5=b, R6=xEnd
Loop:
  MOV R0, R1   # R0=R1
  STO R0, R3   # [R0]=R3
  ADD 1        # R0=R0+1
  STO R0, R4   # [R0]=R4
  ADD 1        # R0=R0+1
  STO R0, R5   # [R0]=R5
  ADD 1        # R0=R0+1
  MOV R1, R0   # R1=R0
  MOV R0, R2   # R0=R2
  ADD 1        # R0=R0+1
  MOV R2, R0   # R2=R0
  CMP R6       # R0<=R6
  JLE Loop     # if R0<=R6 then goto Loop
```

```
while x<=xEnd do
   [address]  = r
   [address+1]= g
   [address+2]= b
   x+=1
   address+=3
end
```

Assembler/Maschinensprache

Übersetzung in Maschinensprache

- **Kann direkt von der Zentraleinheit ausgeführt werden**
- **Befehl durch eine Binärzahl**
- **Befehl selbst (LOD, STO, MOV, ADD) in Bits nummerieren**
- **Register mit 3-Bits nummerieren**
  - LOD A, B: 00 AAA BBB
  - STO A, B: 01 AAA BBB
  - MOV A, B: 10 AAA BBB
  - ADD R: 11000 RRR
  - ADD Zahl: 11001000 ; Zahl
  - CMP R: 11010RRR
  - JLE: 11011000; wohin

Assembler/Maschinensprache

Übersetzung in Maschinensprache

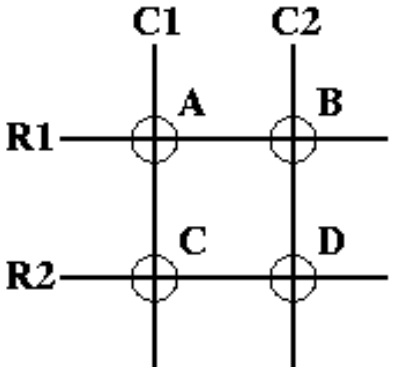
```
# R1=address, R2=X, R3=r, R4=g, R5=b, R6=xEnd
Loop:
  MOV R0, R1   # R0=R1                          10 000 001
  STO R0, R3   # [R0]=R3                        01 000 011
  ADD 1        # R0=R0+1                        1100100        00000001
  STO R0, R4   # [R0]=R4                        01 000 100
  ADD 1        # R0=R0+1                        1100100        00000001
  STO R0, R5   # [R0]=R5                        01 000 101
  ADD 1        # R0=R0+1                        1100100        00000001
  MOV R1, R0   # R1=R0                          10 001 000
  MOV R0, R2   # R0=R2                          10 000 010
  ADD 1        # R0=R0+1                        1100100        00000001
  MOV R2, R0   # R2=R0                          10 010 000
  CMP R6       # R0<=R6                         11010 110
  JLE Loop     # if R0<=R6 then goto Loop       11011 000      11101111
```

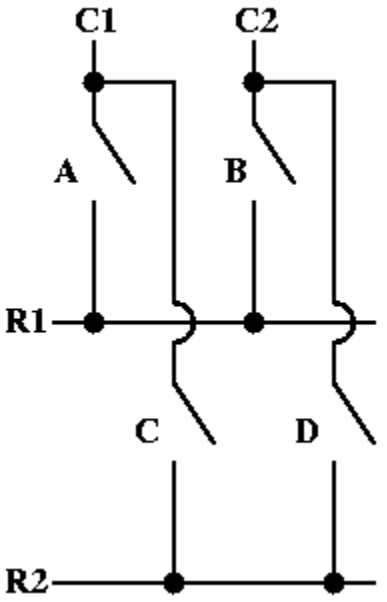
Tastatur

Tastatur

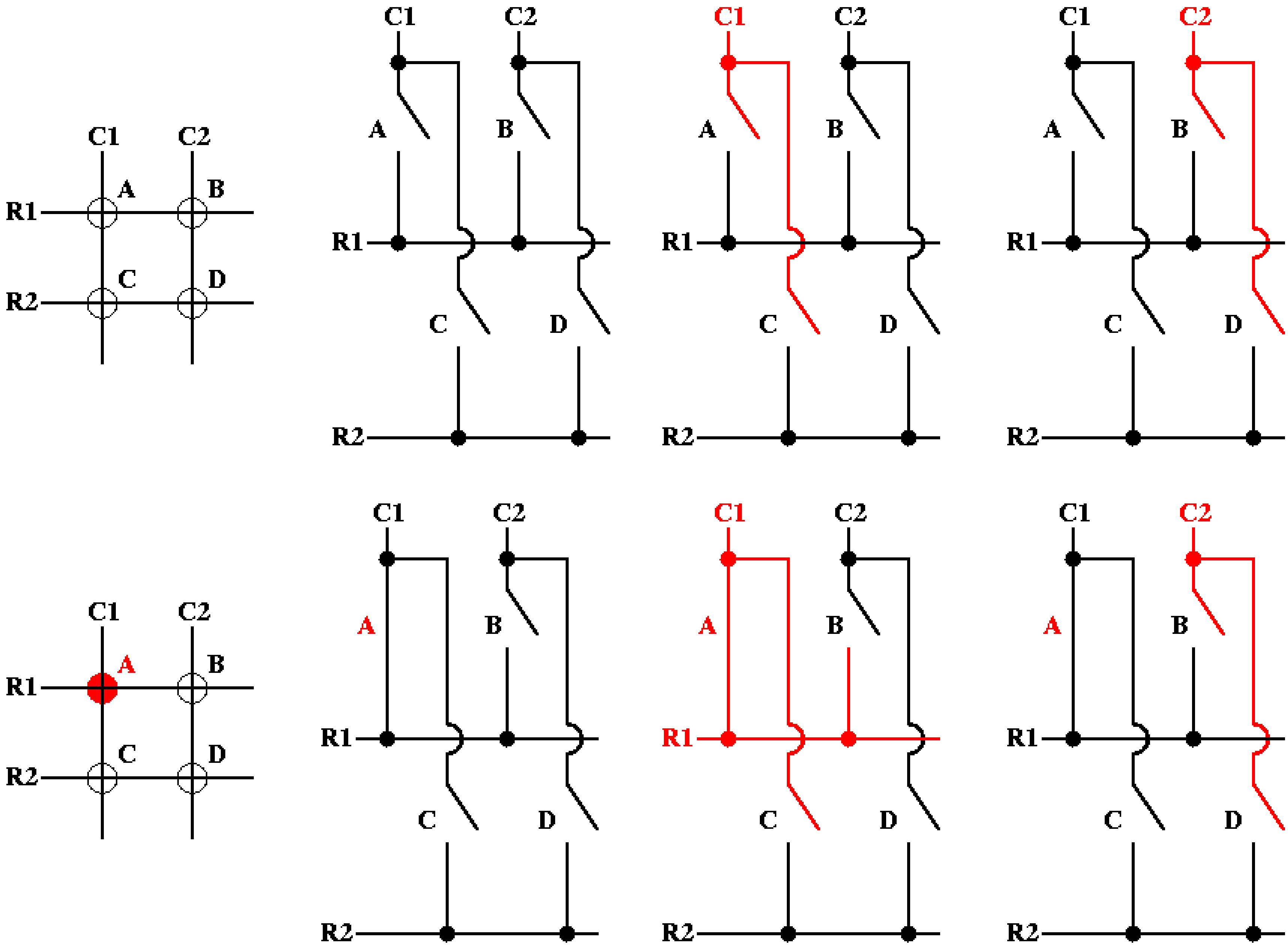
Auslesen einer Matrix von Schaltern

- **Wie spricht man 104 Schalter an?**
- **Nicht mit 104 Kabeln und 104 Eingängen**
- **Schalter elektrisch in Reihen (16) und Spalten (8) anordnen**
- **Nacheinander Strom auf Reihen geben**
- **Prüfen, in welchen Spalten Strom ankommt**
- **www.dribin.org/dave/keyboard/one_html/**

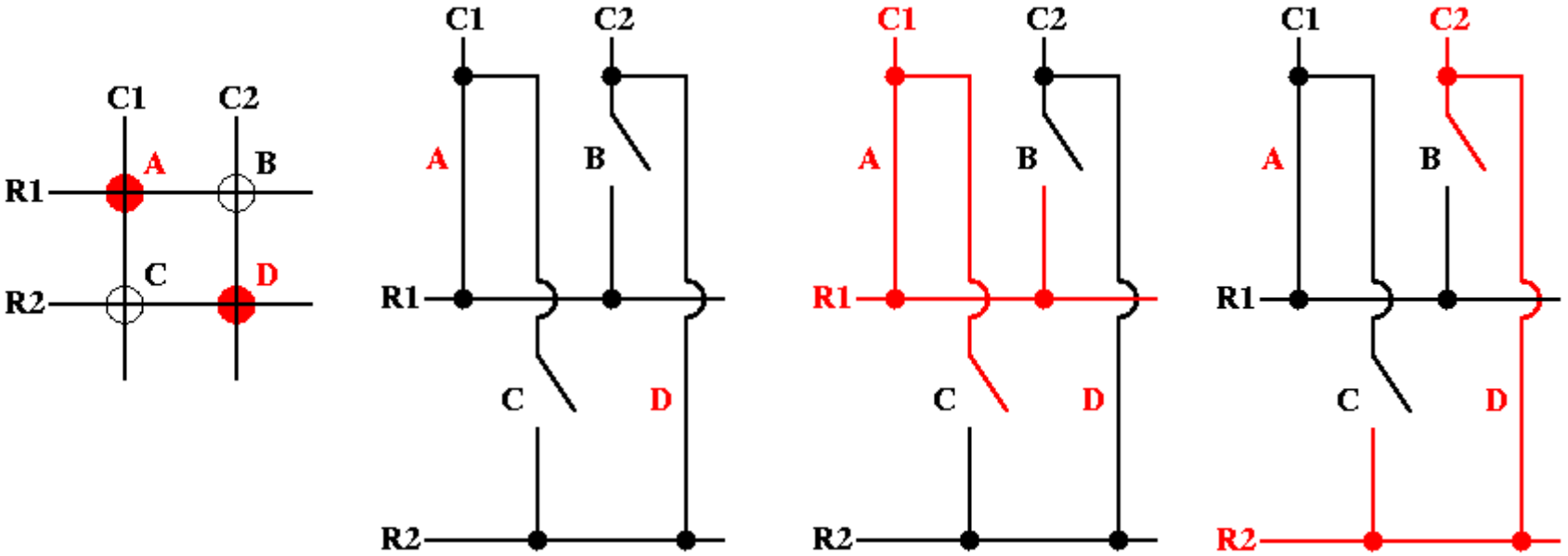




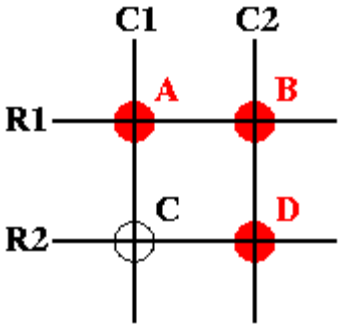
Abfragen der Tasten in einer Tastenmatrix

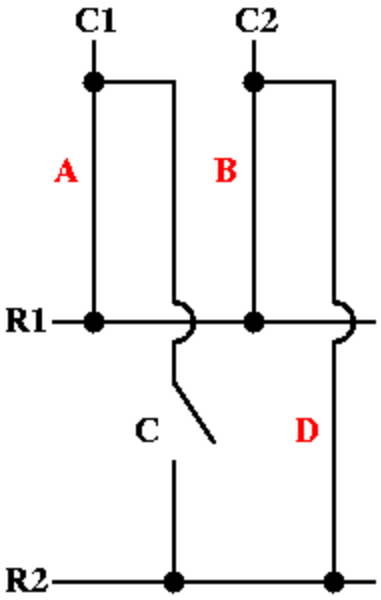


Abfragen der Tasten in einer Tastenmatrix



Frage an das Auditorium: Was passiert, wenn A-B-D gedrückt sind?

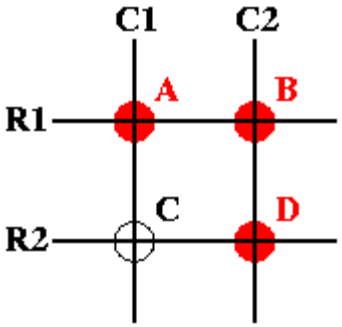


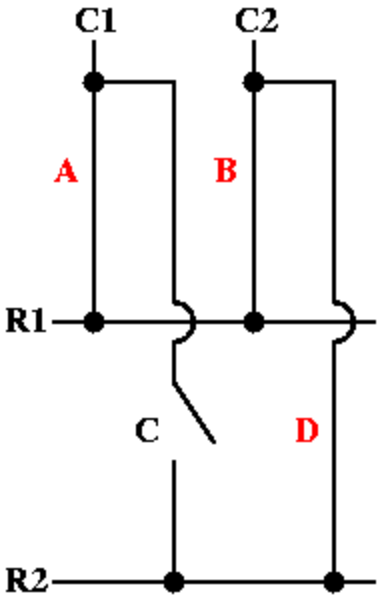


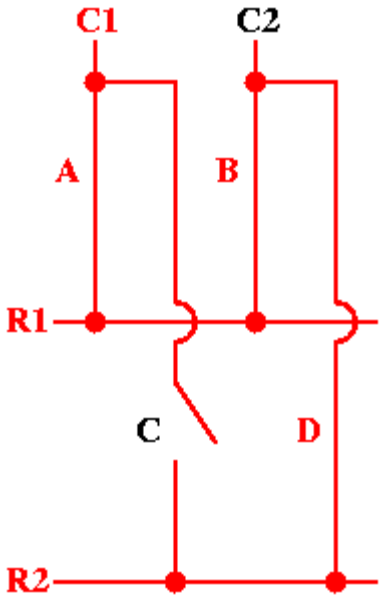
Frage an das Auditorium: Was passiert, wenn A-B-D gedrückt sind?

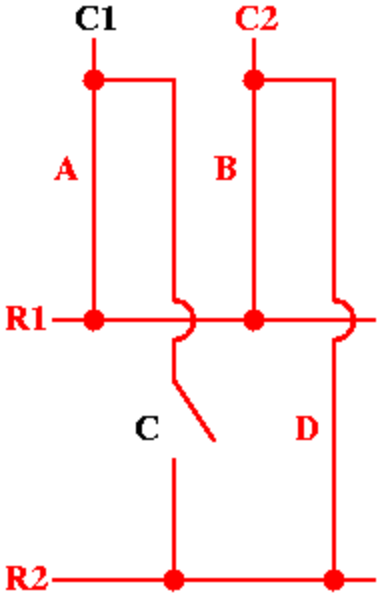
Es besteht über B-C2-D eine Verbindung zwischen R1 und R2.

Dadurch wird auch C erkannt (C1-A-R1-B-C2-D), obwohl nicht gedrückt.









Interrupt

Prinzip der Anbindung an die Zentraleinheit

- **Ein- / Ausgabe wird wie Speicher behandelt**
- **An einer speziellen Adresse…**
- **…erhält man nicht den zuletzt dort gespeicherten Wert, sondern…**
- **…die letzte gedrückte/losgelassene Taste.**
- **Von einer speziellen Elektronik, …**
- **… die sich unter genau der Adresse angesprochen fühlt.**
- **⇒ Kein spezieller Befehl in Zentraleinheit nötig**
- **Woher weiß der Computer, dass eine Taste gedrückt wurde?**
  - Periodisch nachschauen (Polling)
    - *Langsam, unelegant, mit Verzögerung*
  - Interrupt

Interrupt

Interrupt

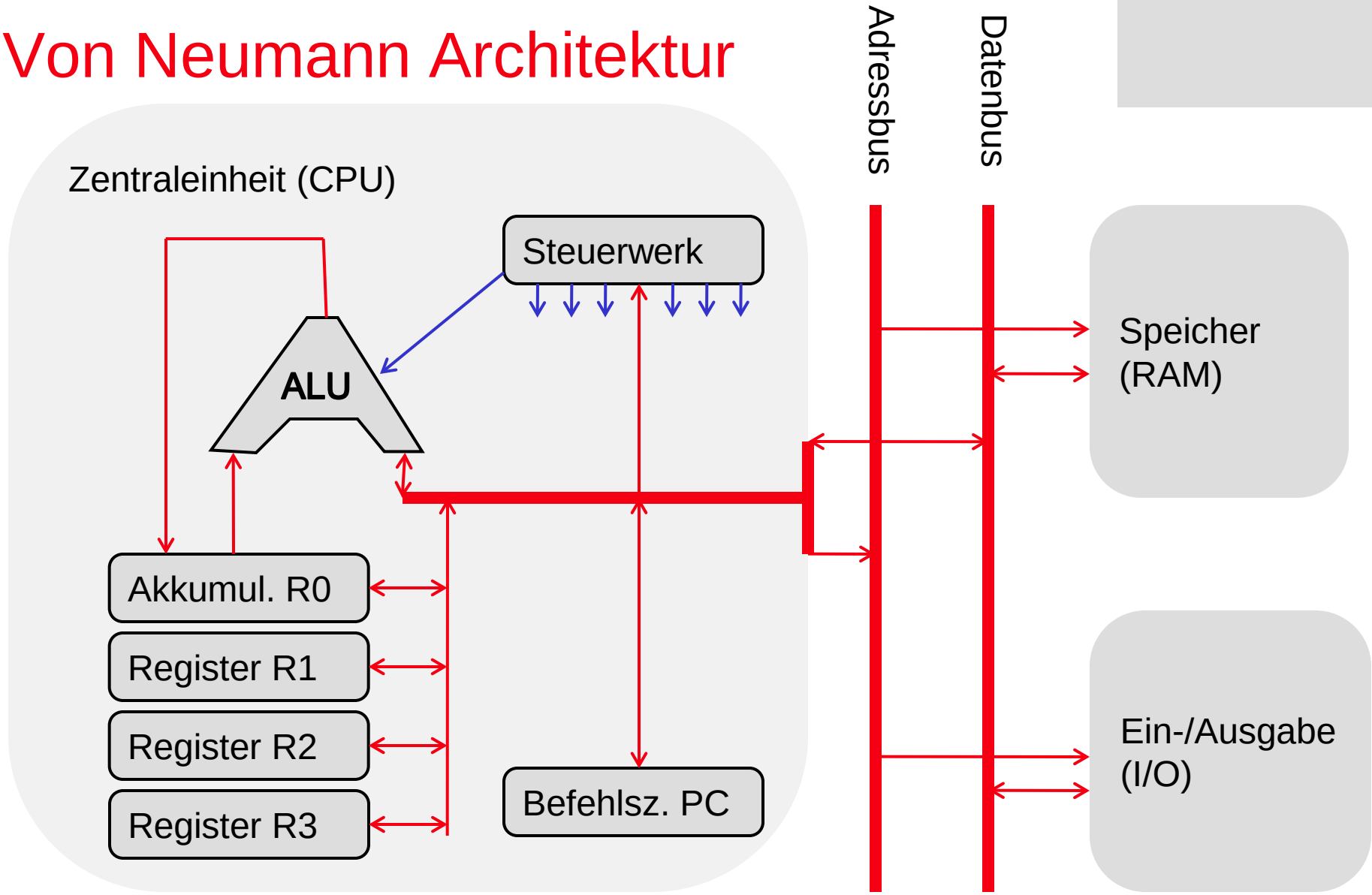
- Elektrisches Signal an Zentraleinheit (ein Ereignis ist passiert)
- Unterbricht aktuelles Programm
- Merkt sich Zustand (Register, aktuelle Anweisung)
- Führt spezielles Betriebssystemprogramm aus
- Überprüft alle Eingabegeräte auf neue Daten
- Trägt neue Daten in entsprechende Datenstrukturen ein (Gerätetreiber)
- Lädt alte Registerwerte
- Fährt mit unterbrochenem Programm fort
- So arbeiten die meisten Ein-/Ausgabegeräte (Tastatur, USB, Festplatte, Netzwerk, Kamera)

Von Neumann Architektur

Von Neumann Architektur

- **John von Neumann (1903-57, Princeton)**
- **“First Draft of a Report on the EDVAC”, 1945**
- **Programm und Daten in gemeinsamen Speicher**
- **Speicher ist fortlaufende Folge von Zahlen**
- **Auswahl einer Speicherzelle durch eine Zahl, sogenannte Adresse**
- **Komponenten**
  - Speicher hält große Datenmengen, angesprochen über Adresse
  - Ein-/Ausgabe verhält sich wie Speicher
  - ALU (Arithmetic Logical Unit) rechnet
  - Akkumulator hält einen Operanden und Ergebnis
  - Register halten weitere Zwischenergebnis
  - Programmzähler hält Adresse des nächsten Befehls

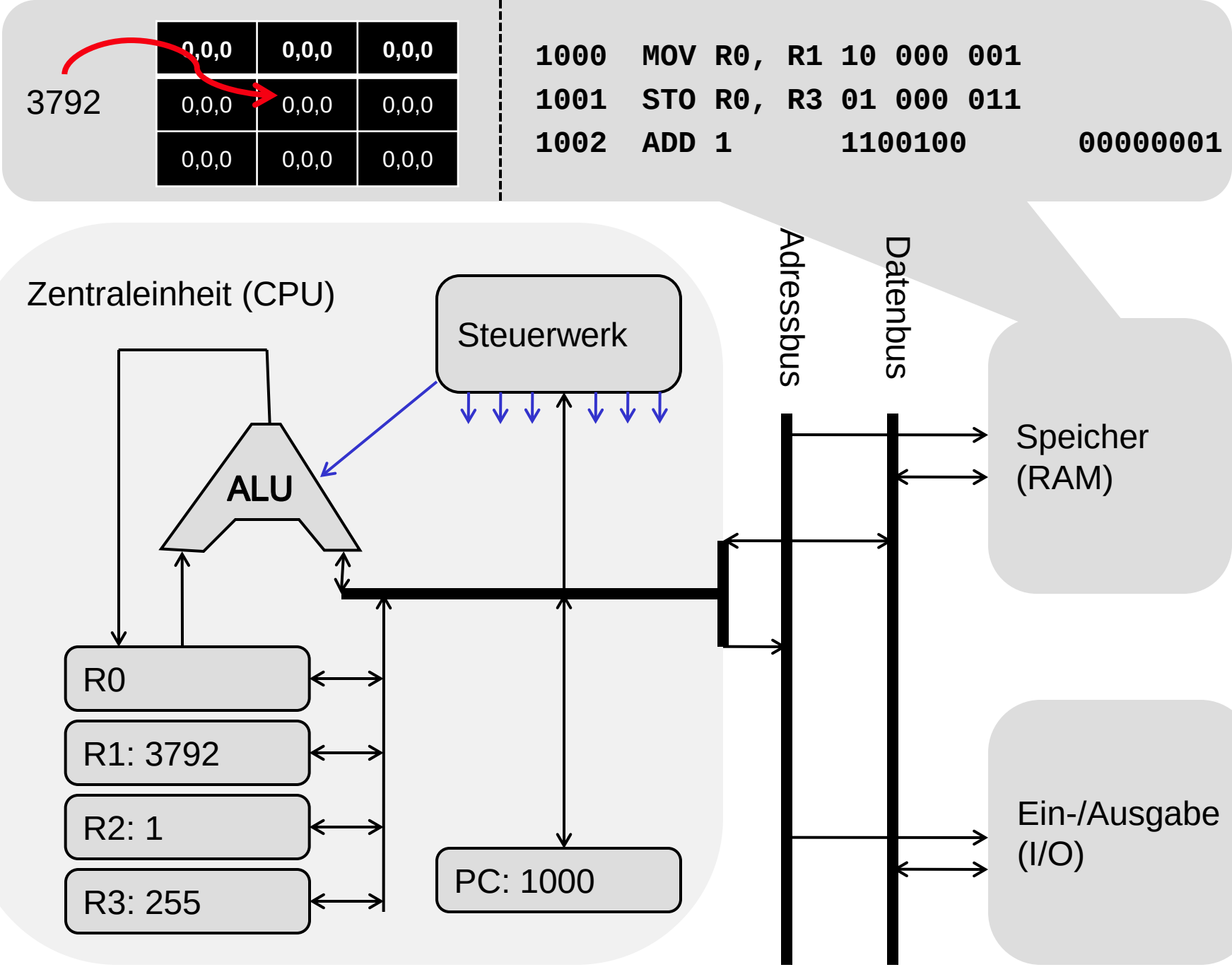
Von Neumann Architektur

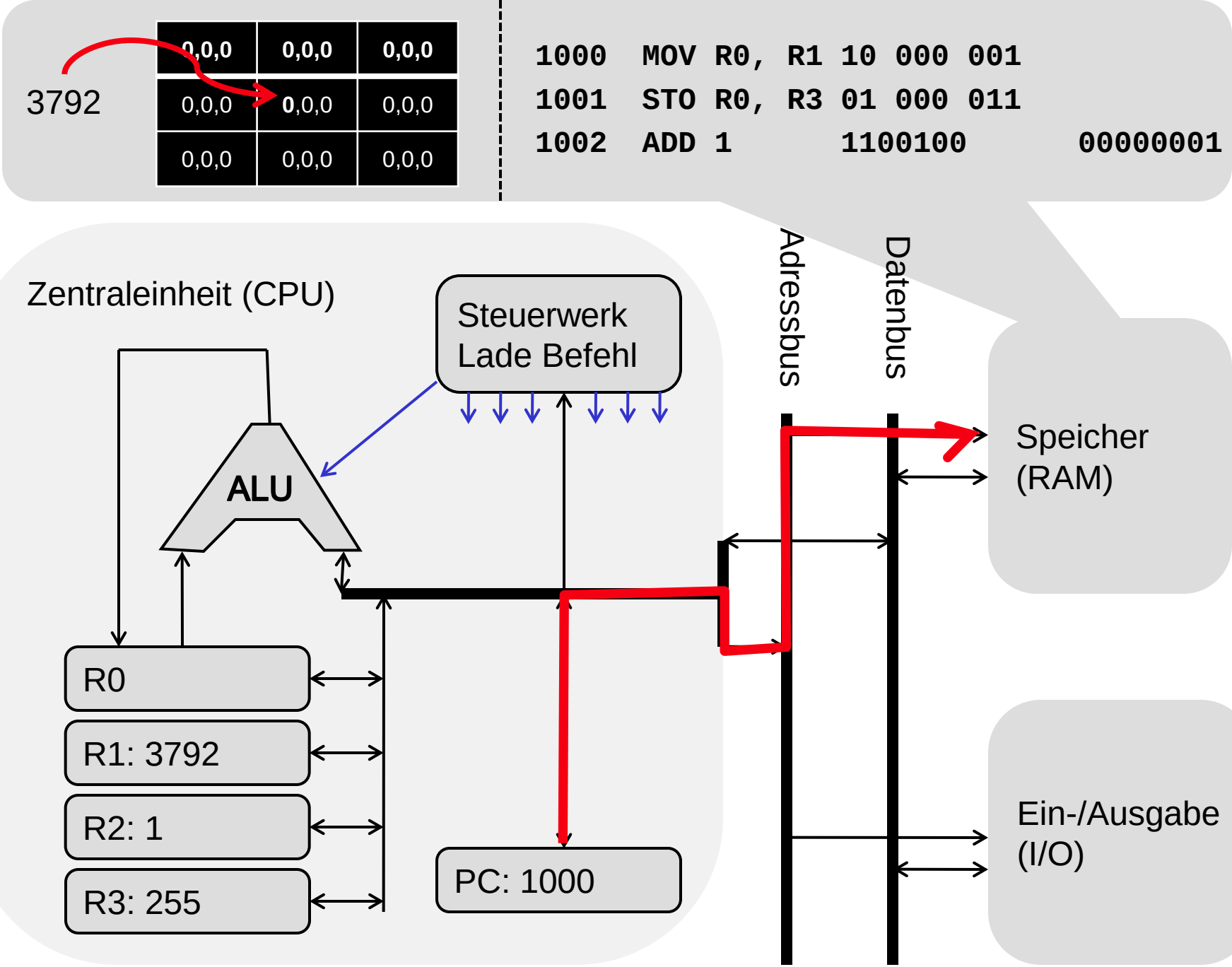


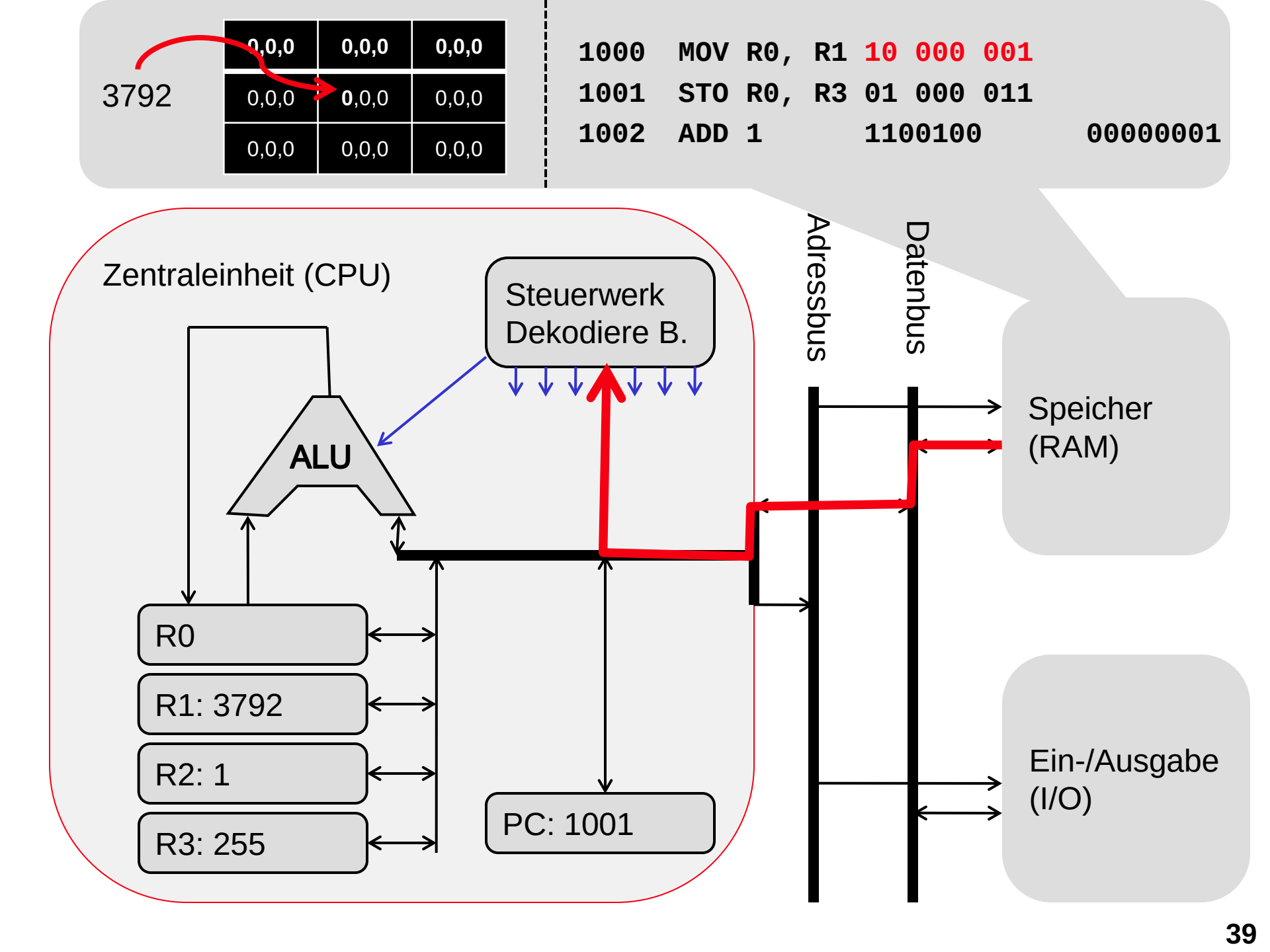
Von Neumann Architektur

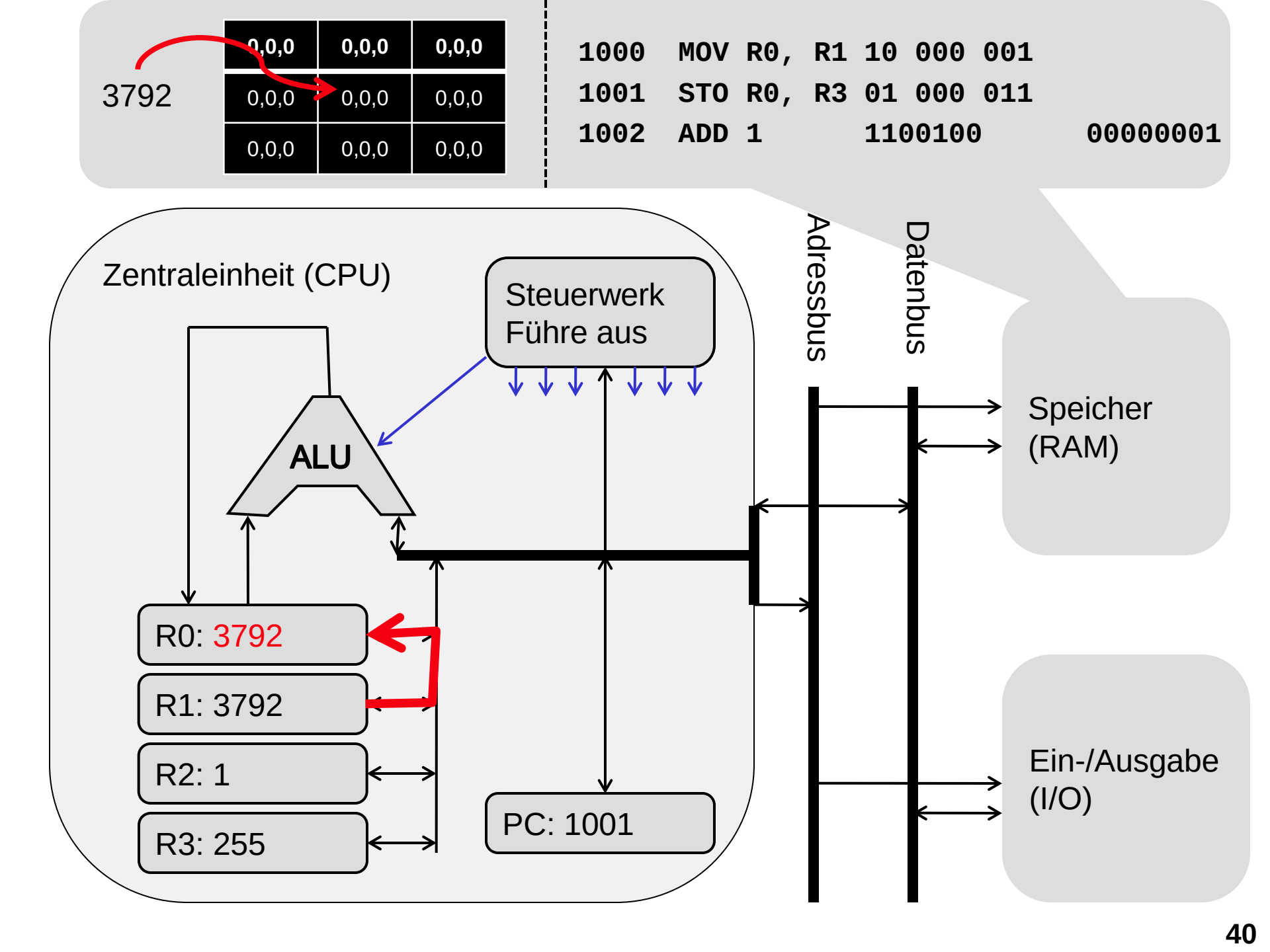
Übersetzung in Maschinensprache

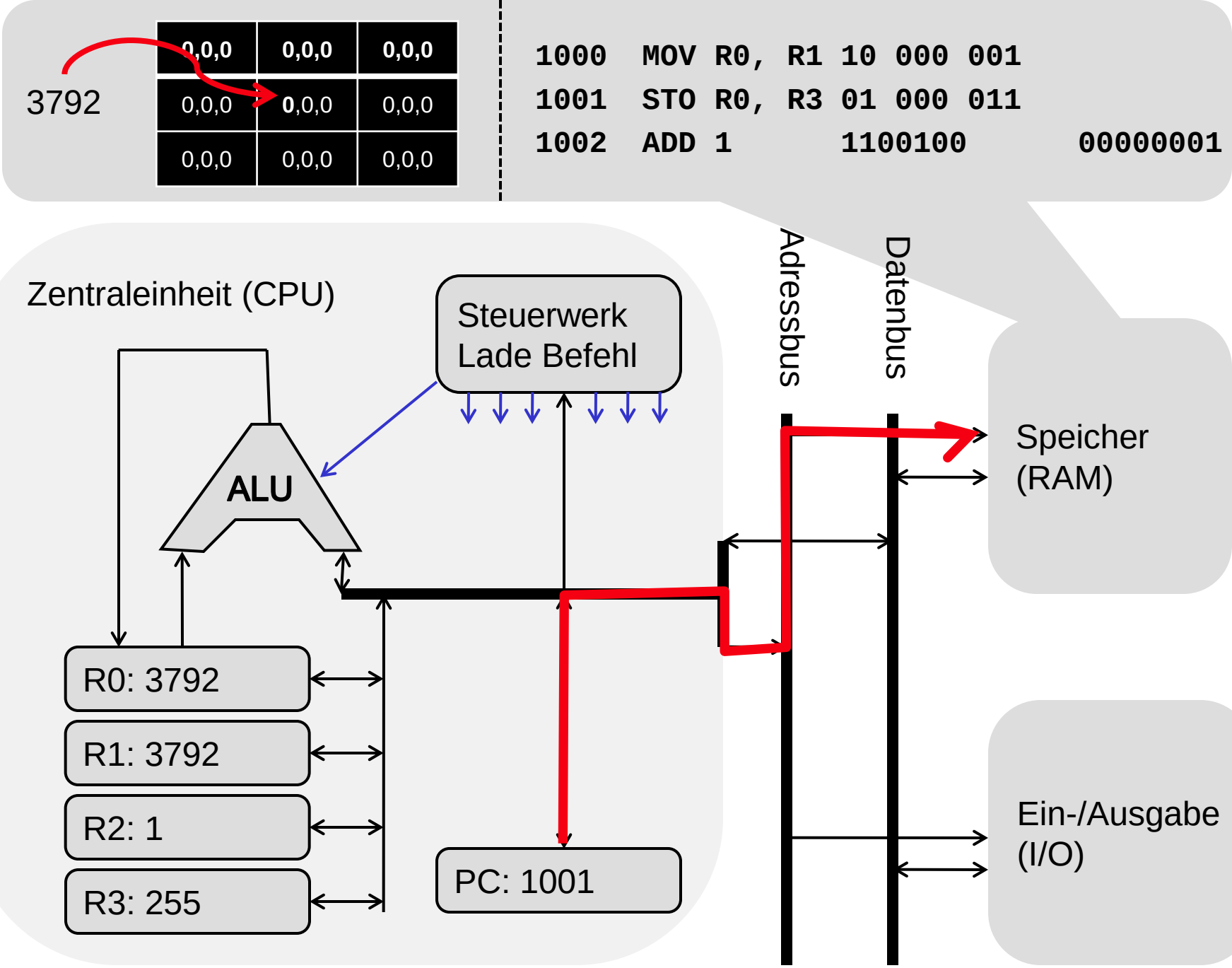
```
# R1=address, R2=X, R3=r, R4=g, R5=b, R6=xEnd
Loop:
  MOV R0, R1   # R0=R1                          10 000 001
  STO R0, R3   # [R0]=R3                        01 000 011
  ADD 1        # R0=R0+1                        1100100        00000001
  STO R0, R4   # [R0]=R4                        01 000 100
  ADD 1        # R0=R0+1                        1100100        00000001
  STO R0, R5   # [R0]=R5                        01 000 101
  ADD 1        # R0=R0+1                        1100100        00000001
  MOV R1, R0   # R1=R0                          10 001 000
  MOV R0, R2   # R0=R2                          10 000 010
  ADD 1        # R0=R0+1                        1100100        00000001
  MOV R2, R0   # R2=R0                          10 010 000
  CMP R6       # R0<=R6                         11010 110
  JLE Loop     # if R0<=R6 then goto Loop       11011 000      11101111
```

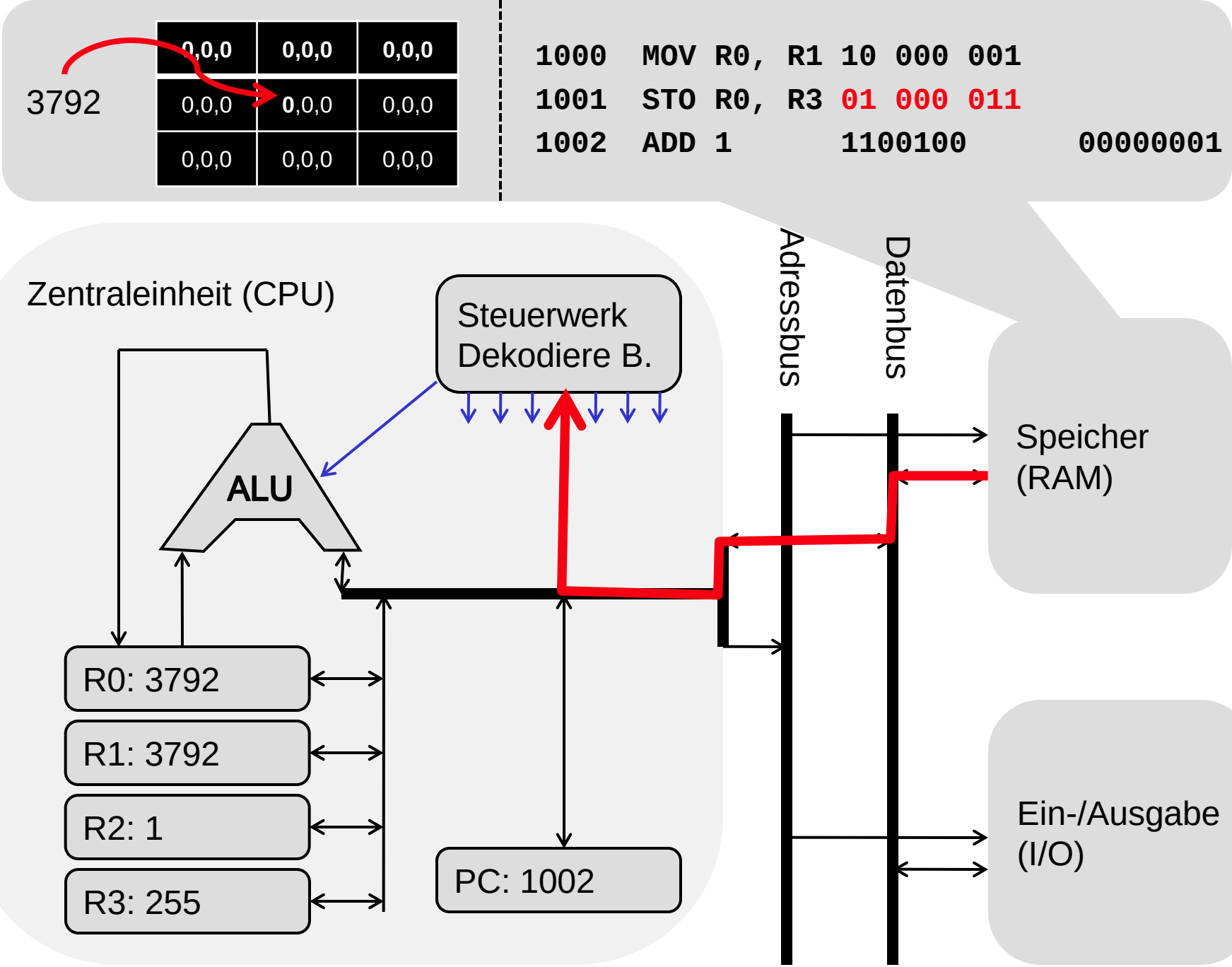

3792
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
ALU
R0
R1: 3792
R2: 1
R3: 255
PC: 1000
Adressbus
Datenbus
Speicher (RAM)
Ein-/Ausgabe (I/O)

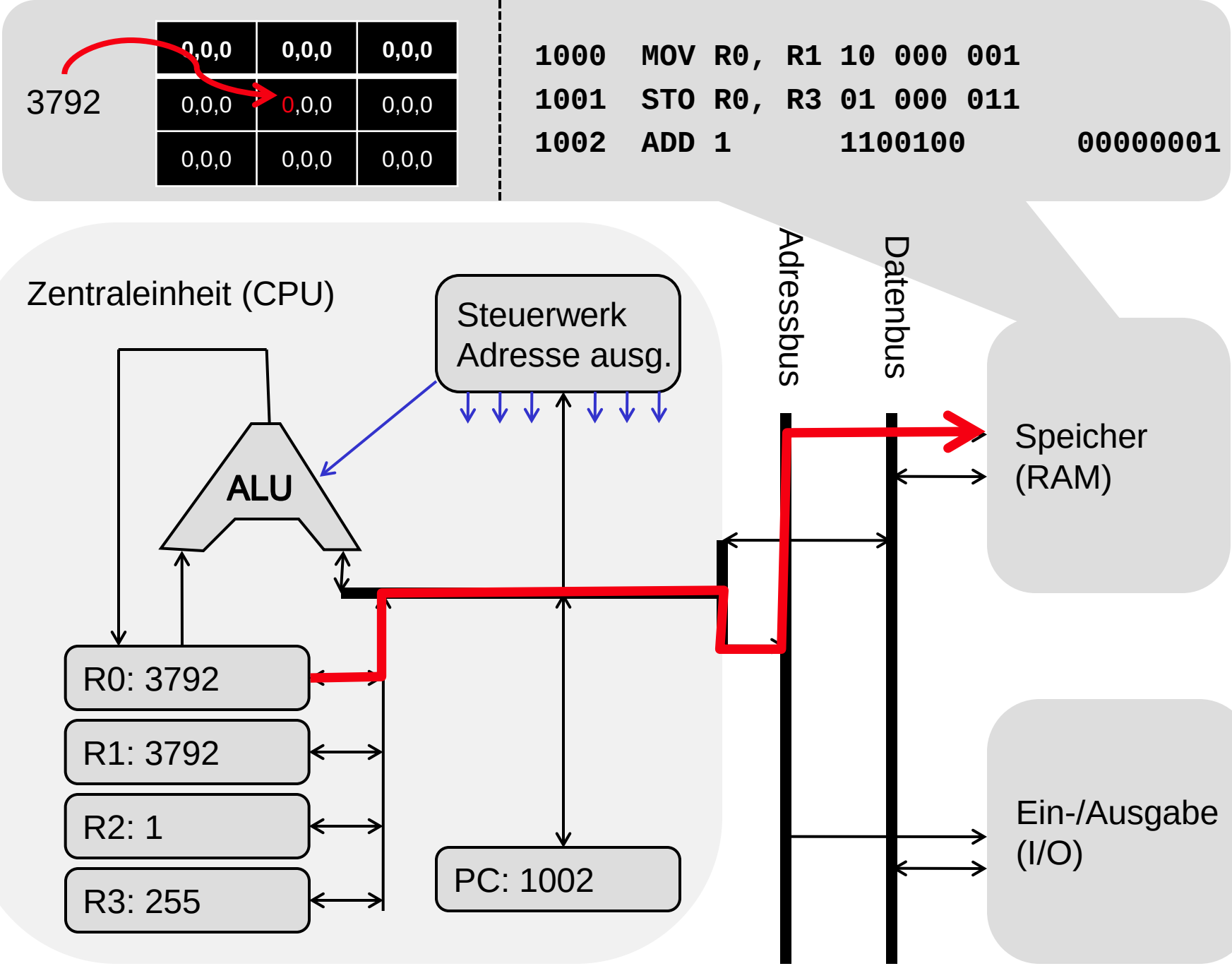

0,0,0 0,0,0 0,0,0
3792
0,0,0 0,0,0 0,0,0
0,0,0 0,0,0 0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Lade Befehl
ALU
R0
R1: 3792
R2: 1
R3: 255
PC: 1000
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)

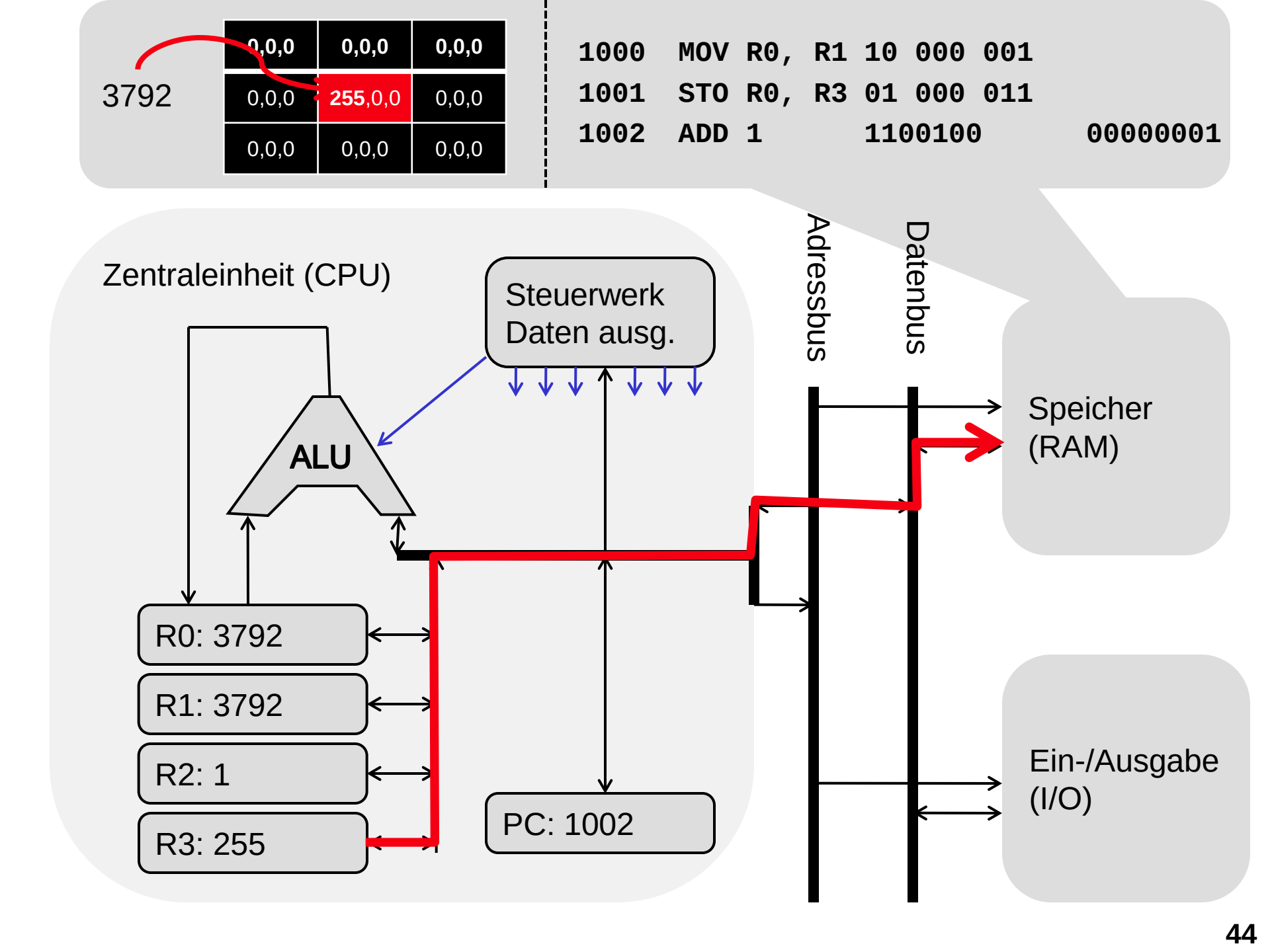

3792
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Dekodiere B.
ALU
R0
R1: 3792
R2: 1
R3: 255
PC: 1001
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)

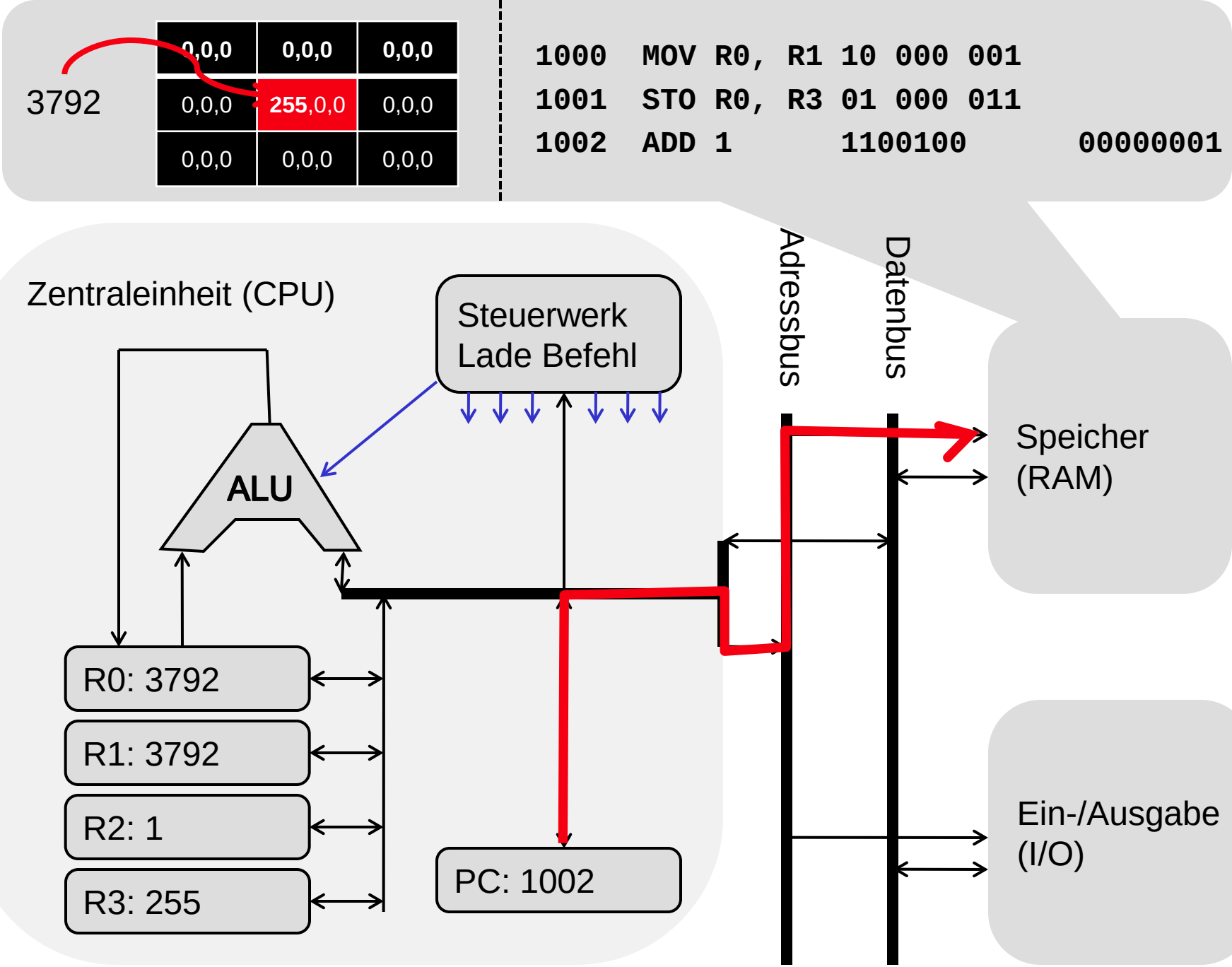

3792
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Führe aus
ALU
R0: 3792
R1: 3792
R2: 1
R3: 255
PC: 1001
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)

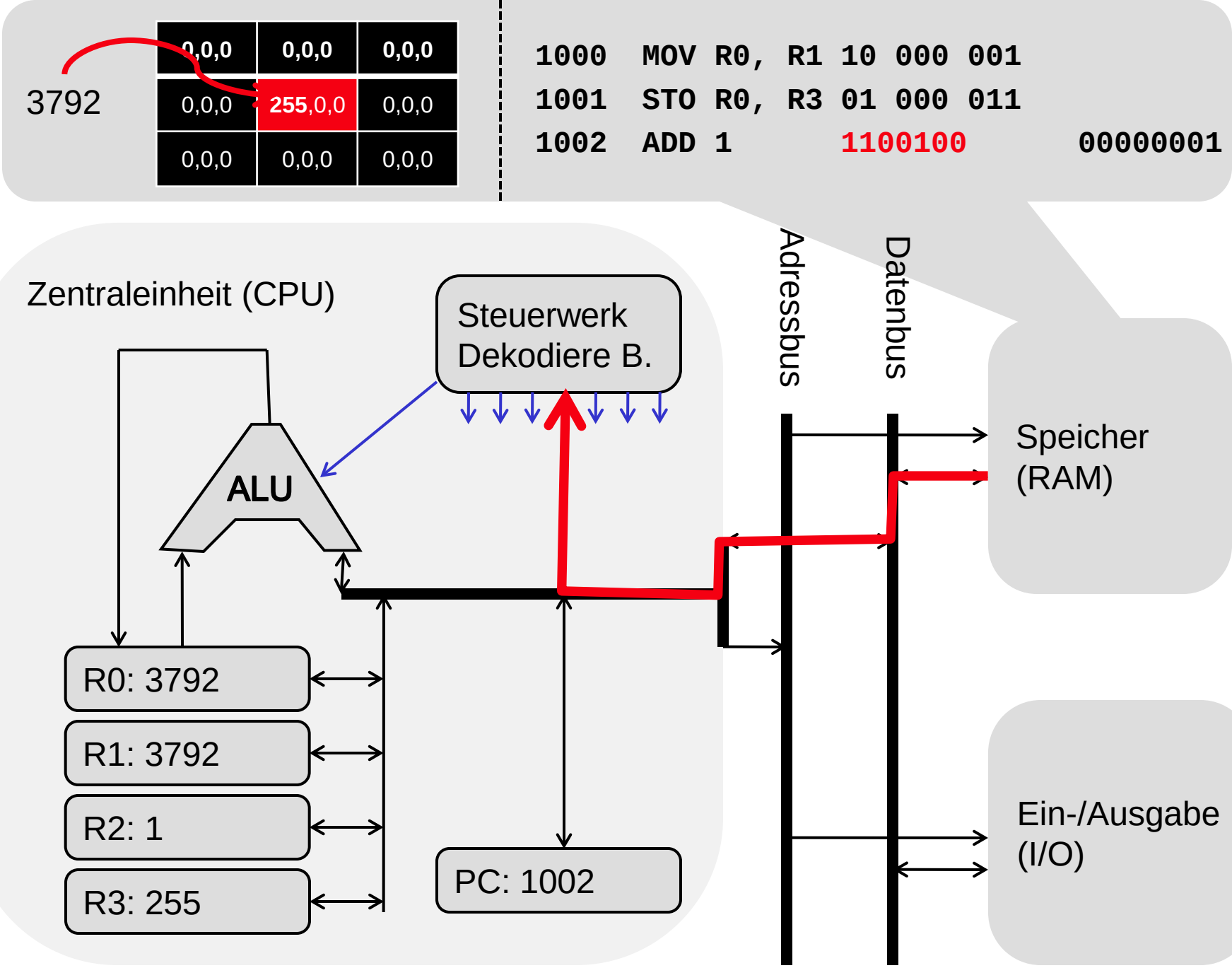

3792
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Lade Befehl
ALU
R0: 3792
R1: 3792
R2: 1
R3: 255
PC: 1001
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)

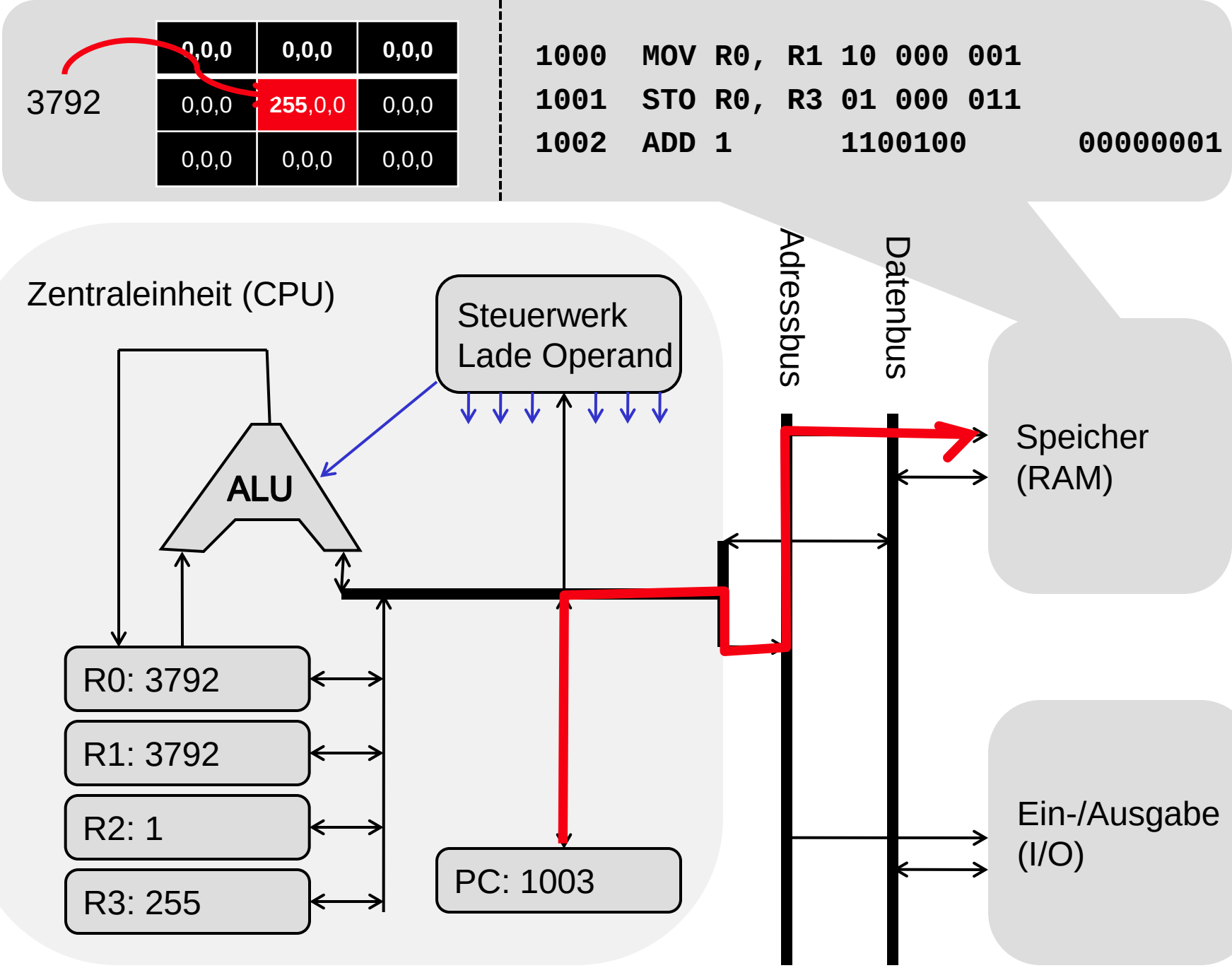

3792
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Dekodiere B.
ALU
R0: 3792
R1: 3792
R2: 1
R3: 255
PC: 1002
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)

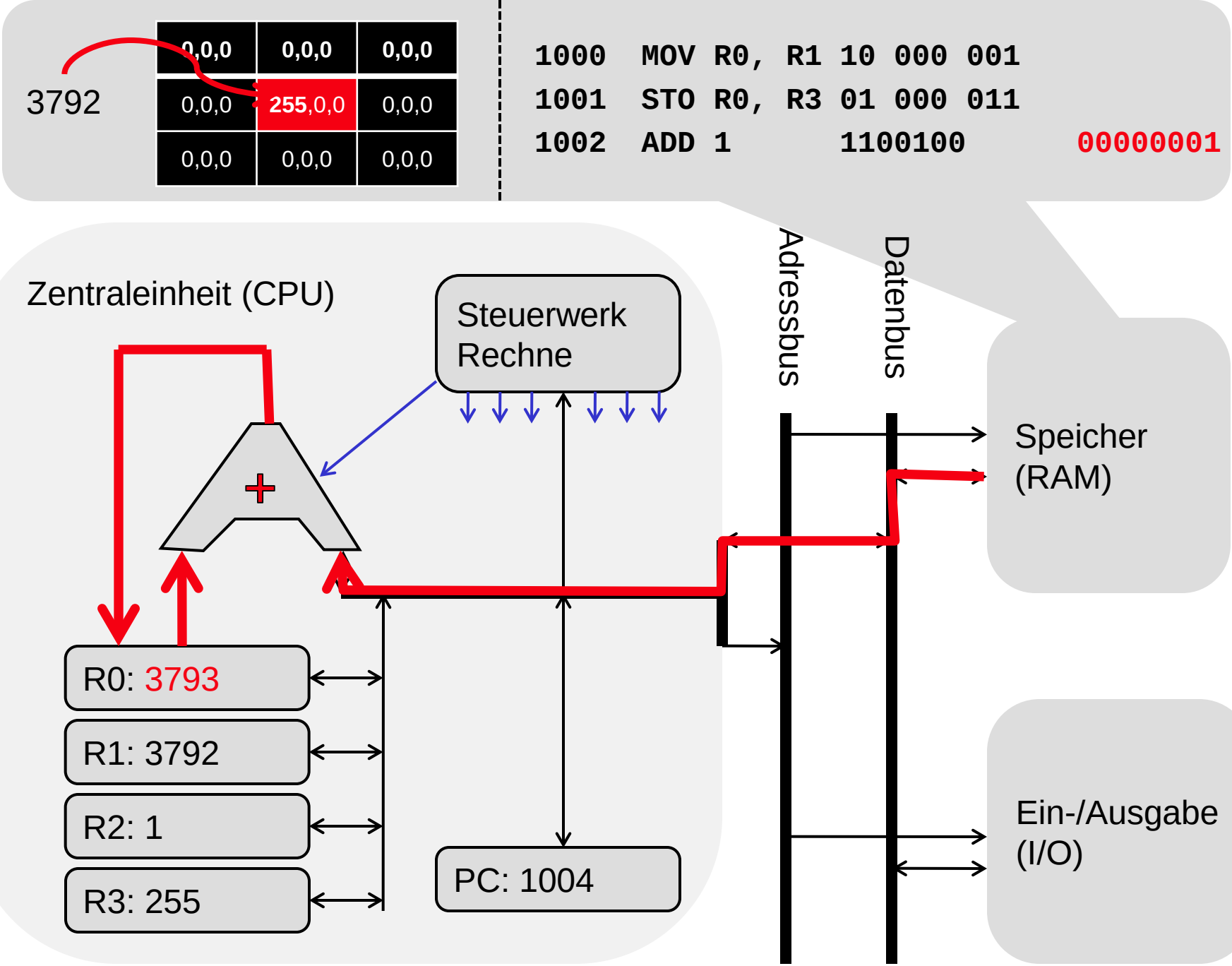

3792
0,0,0 0,0,0 0,0,0
0,0,0 0,0,0 0,0,0
0,0,0 0,0,0 0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Adresse ausg.
ALU
R0: 3792
R1: 3792
R2: 1
R3: 255
PC: 1002
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)


3792
0,0,0
0,0,0
0,0,0
0,0,0
255,0,0
0,0,0
0,0,0
0,0,0
0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Daten ausg.
ALU
R0: 3792
R1: 3792
R2: 1
R3: 255
PC: 1002
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)


3792
0,0,0 0,0,0 0,0,0
0,0,0 255,0,0 0,0,0
0,0,0 0,0,0 0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Lade Befehl
ALU
R0: 3792
R1: 3792
R2: 1
R3: 255
PC: 1002
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)


0,0,0 0,0,0 0,0,0
3792
0,0,0 255,0,0 0,0,0
0,0,0 0,0,0 0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Dekodiere B.
ALU
R0: 3792
R1: 3792
R2: 1
R3: 255
PC: 1002
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)


0,0,0 0,0,0 0,0,0
3792
0,0,0 255,0,0 0,0,0
0,0,0 0,0,0 0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Lade Operand
ALU
R0: 3792
R1: 3792
R2: 1
R3: 255
PC: 1003
Adressbus
Datenbus
Speicher
(RAM)
Ein-/Ausgabe
(I/O)


3792
0,0,0
0,0,0
0,0,0
0,0,0
255,0,0
0,0,0
0,0,0
0,0,0
0,0,0
1000 MOV R0, R1 10 000 001
1001 STO R0, R3 01 000 011
1002 ADD 1 1100100 00000001
Zentraleinheit (CPU)
Steuerwerk
Rechne
Adressbus
Datenbus
Speicher
(RAM)
R0: 3793
R1: 3792
R2: 1
R3: 255
PC: 1004
Ein-/Ausgabe
(I/O)

Von Neumann Architektur

Zusammenfassung

- **Mehr Elektronik für Logistik als fürs eigentliche Rechnen**
- **Im einfachsten Fall 1 Taktzyklus pro Phase**
  - 3-4 Taktzyklen pro Befehl
  - 100M – 4G Taktzyklen/s
- **Moderne Prozessoren: Phasen parallel (Pipelining)**
  - Mehrere Befehle laden
  - Vorweg dekodieren
  - Dekodierte Befehle zwischenspeichern
  - Unabhängige Befehle gleichzeitig ausführen
- **0.5-1 Taktzyklus pro Befehl (Pentium)**
- **Flaschenhals: Reaktionszeit des Speichers**

Addierer

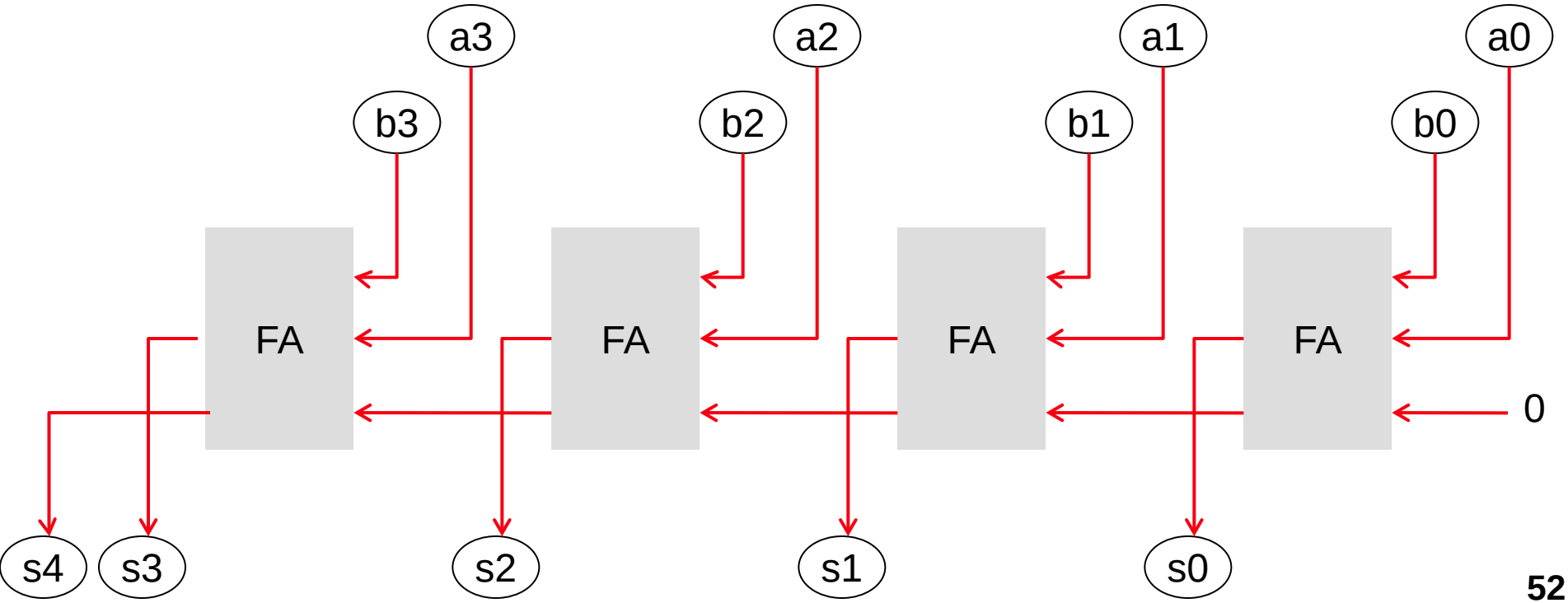
Addierer

Aufgabenstellung

- **Eingabe: 2 Binärzahlen (z.B. 4 Stellen) a,b**
- **Ausgabe: 1 Binärzahl s=a+b**
- **Realisiere die Addition mit Elektronik**
- **Frage an das Auditorium: Wie würdet Ihr die Aufgabe, zwei 4-stellige Binärzahlen zu addieren in Unteraufgaben zerlegen?**

Addierer

- **<u>Frage an das Auditorium</u>: Wie würdet Ihr die Aufgabe, zwei 4-stellige Binärzahlen zu addieren in Unteraufgaben zerlegen?**
- **Addiere *drei* 1-stellige Binärzahlen (a, b, Übertrag)**
- **„Volladdierer“ (Full-Adder, FA)**



Addierer

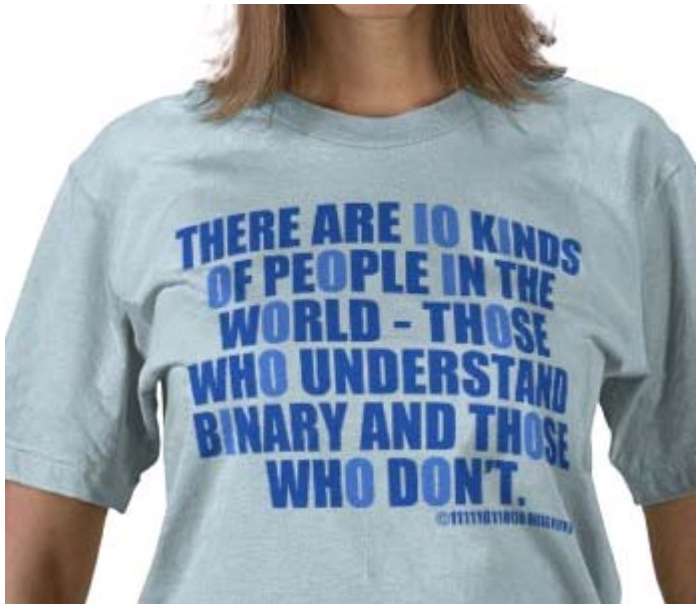
Halbaddierer

- **Halfadder, HA**
- **Addition von zwei Binärzahlen**
- **Im Prinzip ähnlich, wie Volladdierer, nur einfacher**
- **Realisierung durch Logikbausteine (Gatter)**

Addierer

Binärsystem: Rechnen durch Logik

- **Erfunden von G.W. Leibniz (1703)**
- **Stellenwert 2 (1, 2, 4, 8, 16, 32, …)**
- **Ziffer 0 oder 1**
- **kleinstmögliche Anzahl Ziffern**
- **kleines 1+1 und kleines 1*1 extrem einfach**
- **Rechnen mit Zahlen**
  - 0, 1, 2, 3, ….
  - plus (+), minus (-), mal (*), geteilt (/)
- **Logik**
  - Wahr und Falsch
  - und ($\wedge$), oder ($\vee$), nicht ($\neg$)
- **Rechnen im Binärsystem durch Logik**
  - 0 $\rightarrow$ Falsch, 1 $\rightarrow$ Wahr



+	0	1
0	00	01
1	01	10

*	0	1
0	0	0
1	0	1

Addierer

- **<u>Frage an das Auditorium</u>: Formuliert das binäre kleine 1*1 mit Worten der Logik! („a*b ist eins, wenn ….“)**

*	0	1
0	0	0
1	0	1

Addierer

- **Frage an das Auditorium: Formuliert das binäre kleine 1*1 mit Worten der Logik! („a*b ist eins, wenn ….“)**
- **a*b ist eins, wenn a eins ist und b eins ist.**
- **a*b = a und b**
- **Frage an das Auditorium: Formuliert das binäre kleine 1+1 mit Worten der Logik! („Die 2er Stelle (Übertrag) von a+b ist eins, wenn … Die 1er Stelle (Ergebnis) von a+b ist eins, wenn…. “)**

+	0	1
0	00	01
1	01	10

*	0	1
0	0	0
1	0	1

Addierer

- **Frage an das Auditorium: Formuliert das binäre kleine 1*1 mit Worten der Logik! („a*b ist eins, wenn ….“)**
- **a*b ist eins, wenn a eins ist und b eins ist.**
- **a*b = a und b**
- **Frage an das Auditorium: Formuliert das binäre kleine 1+1 mit Worten der Logik! („Die 2er Stelle (Übertrag) von a+b ist eins, wenn … Die 1er Stelle (Ergebnis) von a+b ist eins, wenn…. “)**
- **Die 2er Stelle (Übertrag) von a+b ist eins, wenn a eins ist und b eins ist.**
- **Die 1er Stelle (Ergebnis) von a+b ist eins, wenn a eins und b nicht eins oder b eins und a nicht eins ist.**
- **Die 1er Stelle (Ergebnis) von a+b ist eins, wenn a oder b aber nicht beide eins sind. (exklusives oder, xoder, engl. xor).**
- **a+b = (a und b, a xoder b)**

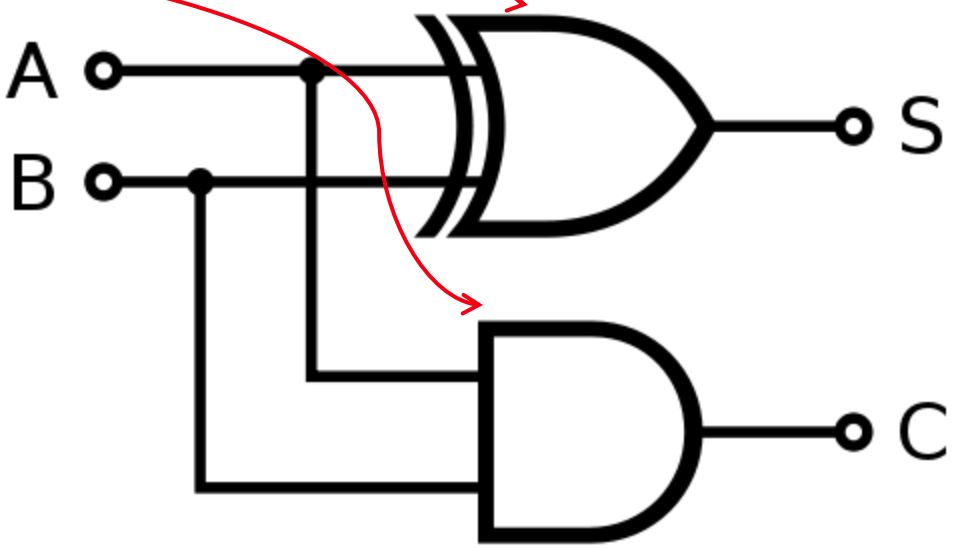
+	0	1
0	00	01
1	01	10

*	0	1
0	0	0
1	0	1

Addierer

Halbaddierer in Gatternotation

- **Oberer Gatter XOR**
- **Unteres Gatter UND**
- **Volladdierer aus 2 Halbaddierern und Oder-Gatter**

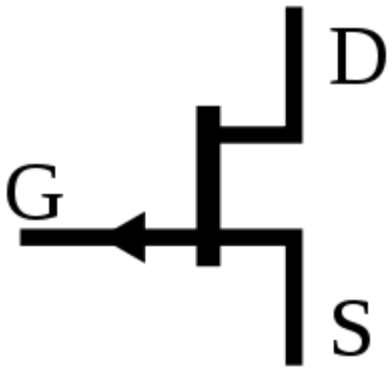


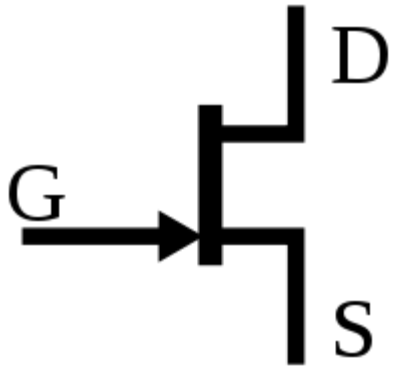
Gatter und Speicher

Gatter und Speicher

Feldeffekttransistor (FET)

- **Elektronischer Schalter**
- **Baustein für (nahezu) alle Computerelektronik**
- **Anschluss Gate (G) Schaltet Verbindung zwischen Source (S) und Drain (D)**
- **Spannung an G: S-D verbunden („leitet“)**
- **Keine Spannung an G: S-D getrennt („sperrt“)**
- **Zwei Typen, die auf positive bzw. negative Spannung reagieren**

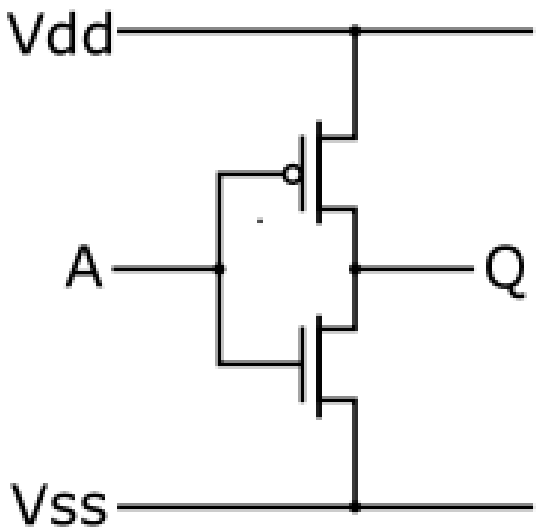




Gatter und Speicher

Nicht-Gatter

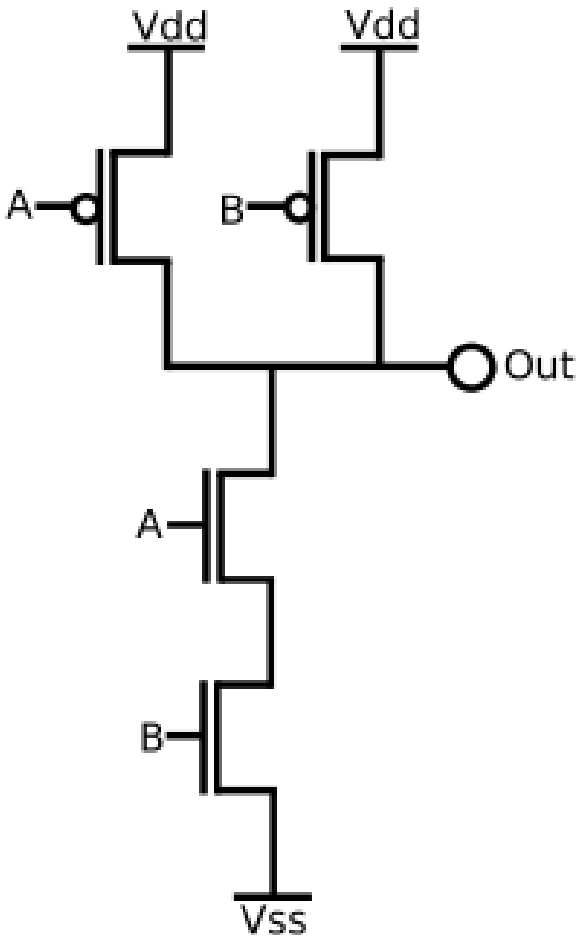
- Zwei Komplementäre FET verschiedenen Typs
- Ist A=0 (Vss) leitet oberer FET, unterer sperrt, Q=1
- Ist A=1 (Vdd) sperrt oberer FET, unterer leitet, Q=0
- Durch Komplementärprinzip, ist „Nicht“ natürlich Funktion



Gatter und Speicher

Nicht-Und (NAND) Gatter

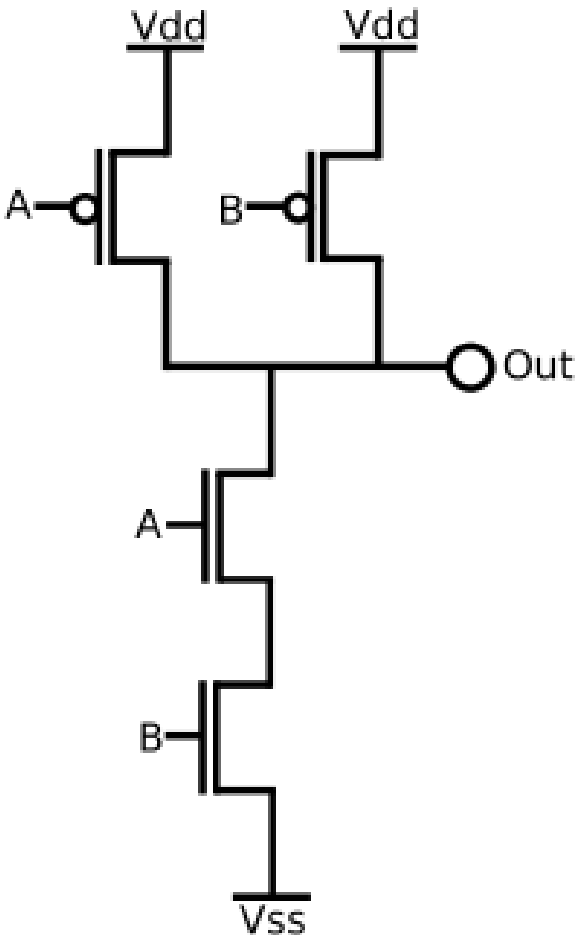
- **Invertiertes Und**
  - Wenn A=1 und B=1, Out=0
  - Wenn A=0 oder B=0, Out=1
- **Wenn A=0 (Vss), leitet A-FET oben, A-FET unten sperrt, Out=1 (Vdd)**
- **Wenn B=0 (Vss), leitet B-FET oben, B-FET unten sperrt, Out=1 (Vdd)**
- **Wenn A=B=1 (Vdd), sperren A/B-FET oben, A/B-FET unten leiten, Out=0 (Vss)**



Gatter und Speicher

Nicht-Oder (NOR) Gatter

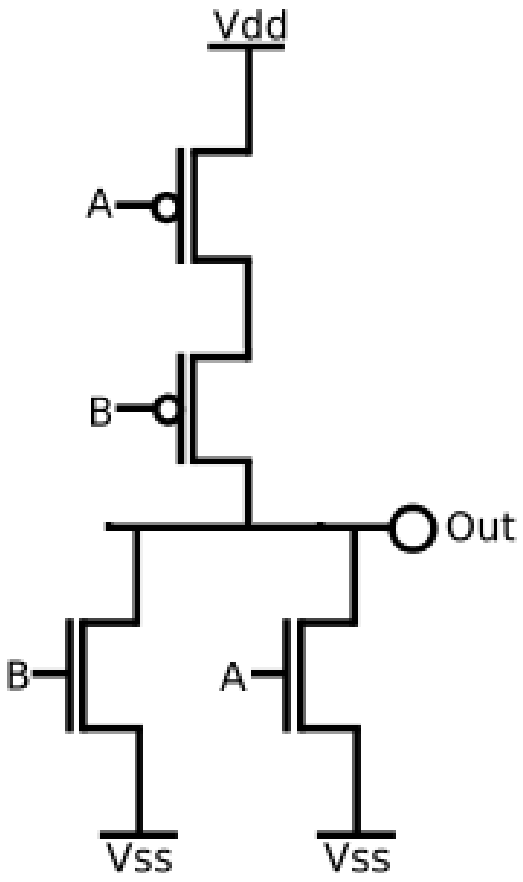
- **Invertiertes Oder**
- **Wenn A=1 oder B=1, Out=0**
- **Wenn A=0 und B=0, Out=1**
- **Frage an das Auditorium: Wie kann man aus dem NAND-Gatter rechts ein NOR-Gatter bauen?**

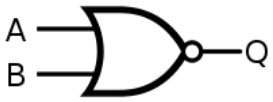


Gatter und Speicher

Nicht-Oder (NOR) Gatter

- **Invertiertes Oder**
- **Wenn A=1 oder B=1, Out=0**
- **Wenn A=0 und B=0, Out=1**
- **Frage an das Auditorium: Wie kann man aus dem NAND-Gatter rechts ein NOR-Gatter bauen?**
- **Oben FET seriell, unten parallel.**

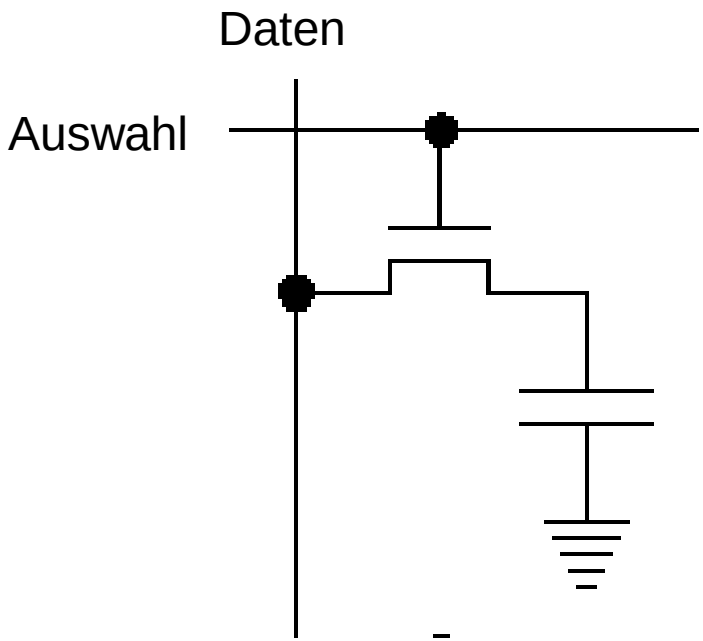




Gatter und Speicher

Speicherzelle

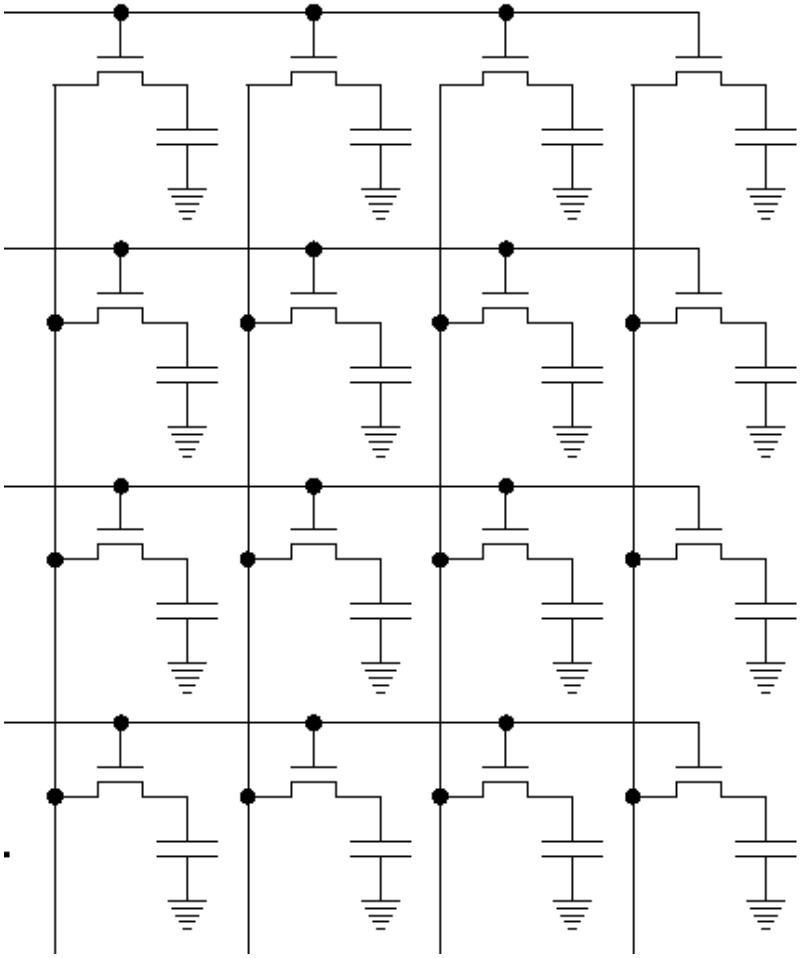
- **Aufgabe:**
  - Ein Bit (Binärstelle) Information aufbewahren
  - Kontrolliert lesen/schreiben
- **Kondensator (r.u.) ist ein „Behälter“ für Spannung**
  - Keine Spannung → 0
  - Spannung vorhanden→ 1
- **FET steuert Zugriff**
  - Auswahl=1: an Datenleitung lesen
  - Auswahl=1: über Datenleitung schreiben
  - Auswahl=0: Speicherzelle passiv, behält Daten



Gatter und Speicher

Speicher

- **Sehr viele (1GB=8Mrd. Zellen) Speicherzellen**
- **Systematisch über Adresse ansteuerbar**
- **Matrix von Speicherzellen**
  - Auswahlleitungen zeilenweise verbunden
  - Datenleitungen spaltenweise verbunden
- **Aufgrund von Adresse**
  - Richtige Auswahlleitung auf 1
  - Richtige Datenleitung lesen/schreiben



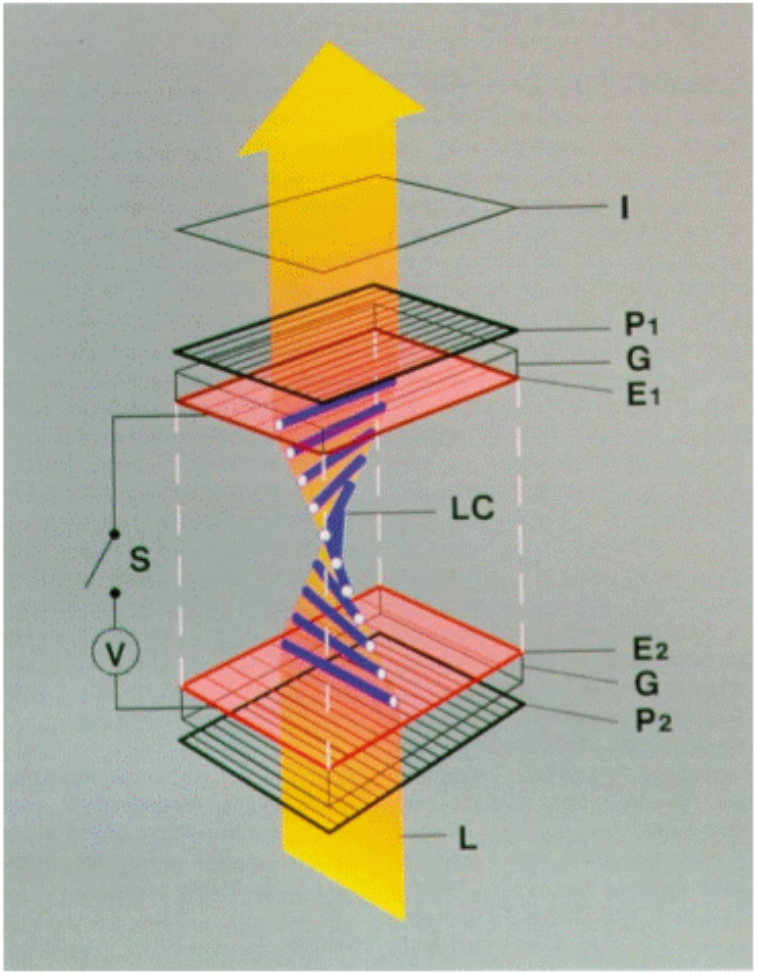
Grafik (LCD)

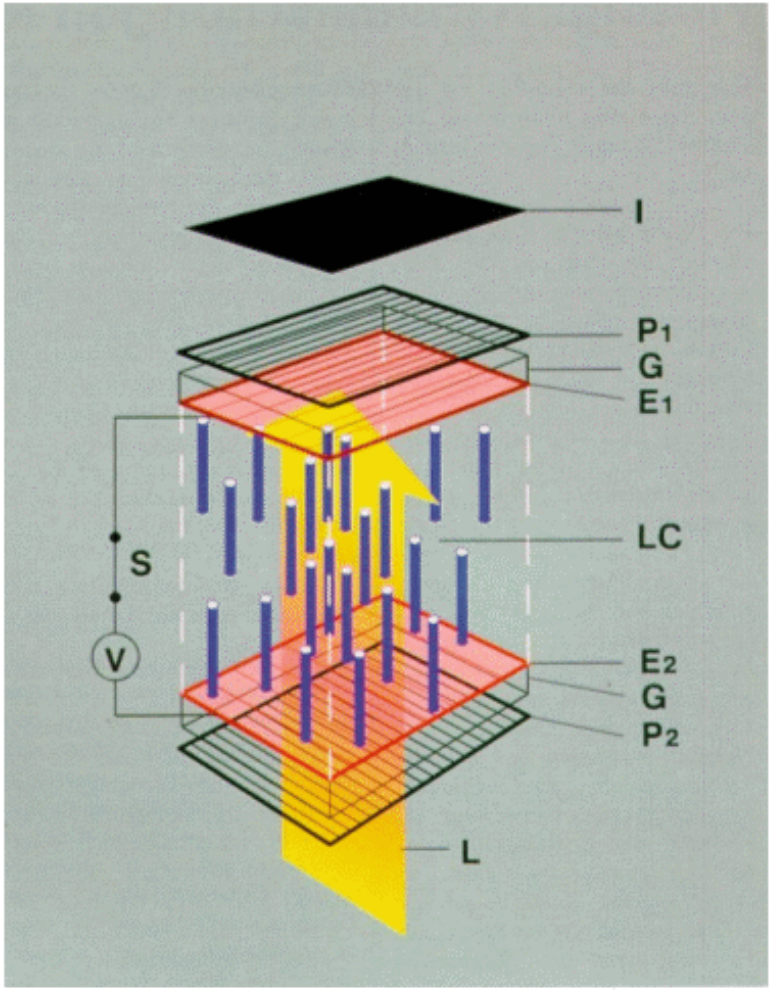
Grafik (LCD)

Funktionsprinzip

- Licht kann polarisiert werden (d.h. eine Richtung bekommen)
- Polarisationsfilter lassen nur Licht einer Richtung durch
- Flüssigkristalle drehen die Polarisationsrichtung
- Aber nicht, wenn Spannung anliegt

Grafik (LCD)

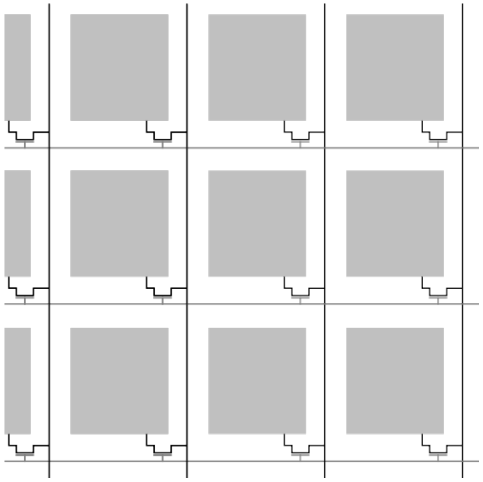


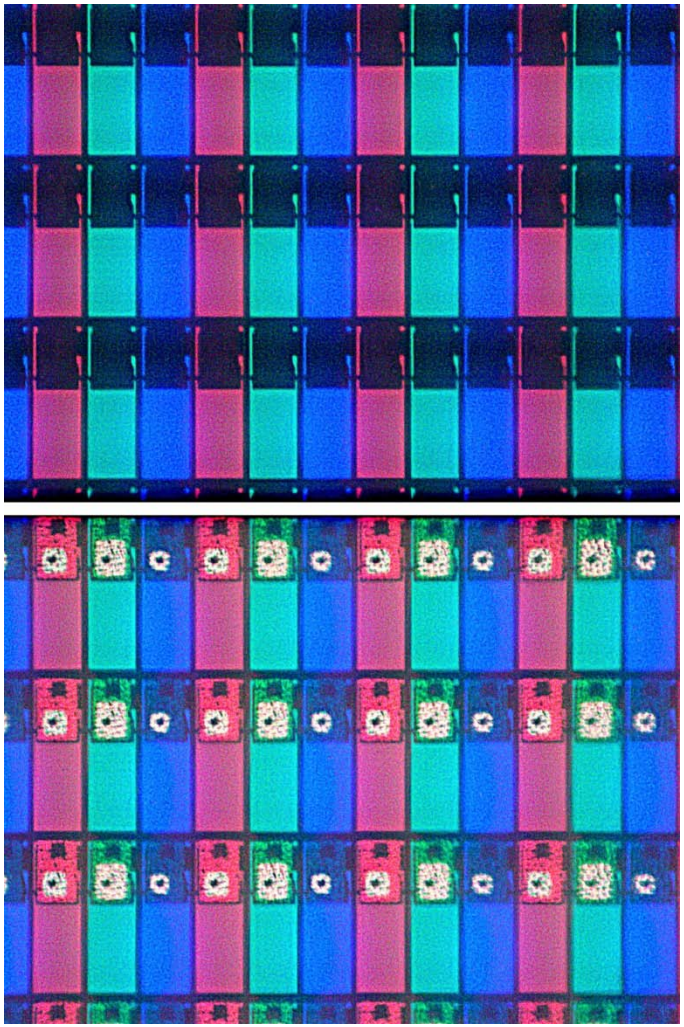


Grafik (LCD)

Anzeige (Matrix von Pixeln, Farbe)

- **Nebeneinander Pixel in Rot, Grün, Blau**
- **Matrixanordnung**
  - Auswahlleitung (Zeilen) wählt Zeile aus
  - Datenleitung (Spalten) liefert Spannung für ausgewählte Zeile
  - Ein FET (auf Glas) pro Pixel





Zusammenfassung

Die verschiedenen Ebenen der Ausführung eines Programmes

- **Programm wird in Maschinensprache übersetzt**
  - einzelne Befehle, wie LOD, STO, ADD
- **Phasen der Befehlsausführung (von Neumann Architektur)**
  - Befehl aus Speicher laden
  - Befehl dekodieren
  - Befehl ausführen
- **Steuerwerk steuert Logistik des Informationsflusses im Prozessor**
- **Addierer (und ähnliche Rechenbausteine)**
  - Aus 1 Bit Volladdierer
  - Aus XOR und AND Gatter
- **Gatter**
  - Aus Parallel- und Seriellschaltung von Feldeffekttransistoren
- **Matrixkonzept mit zeilenweiser Auswahlleitung für Ansteuerung**