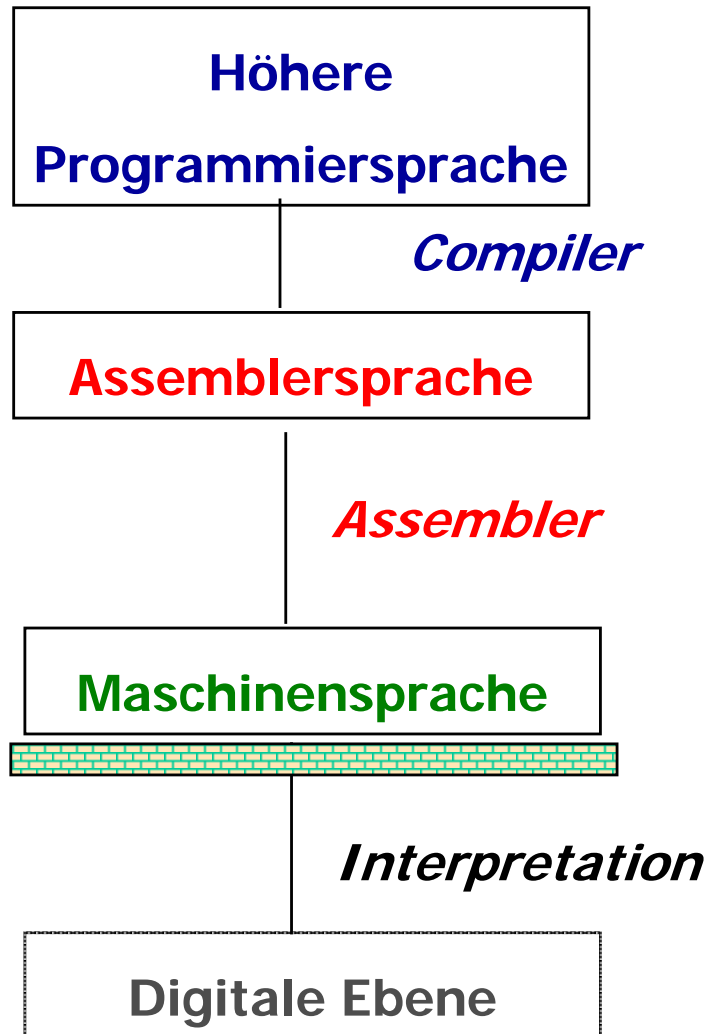

1. Übung

- ❑ **Mikroprogrammierung (2 Aufgaben)**
- ❑ **MIMA-Architektur**



Hierarchie



```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

```
lw    $15, 0($2)  
lw    $16, 4($2)  
sw    $16, 0($2)  
sw    $15, 4($2)
```

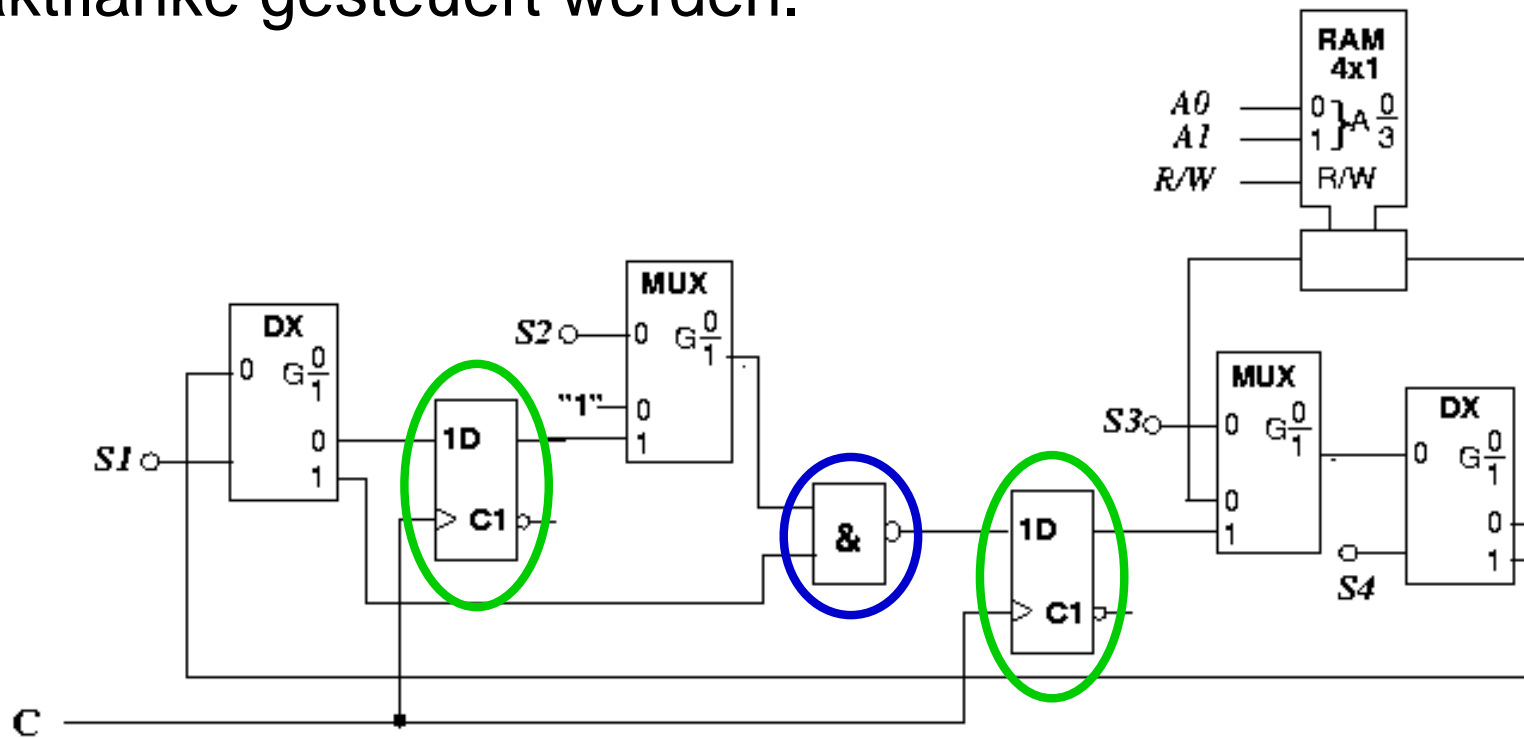
```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```

```
ALUOP[0:3] ← InstReg[9:11] & MASK
```



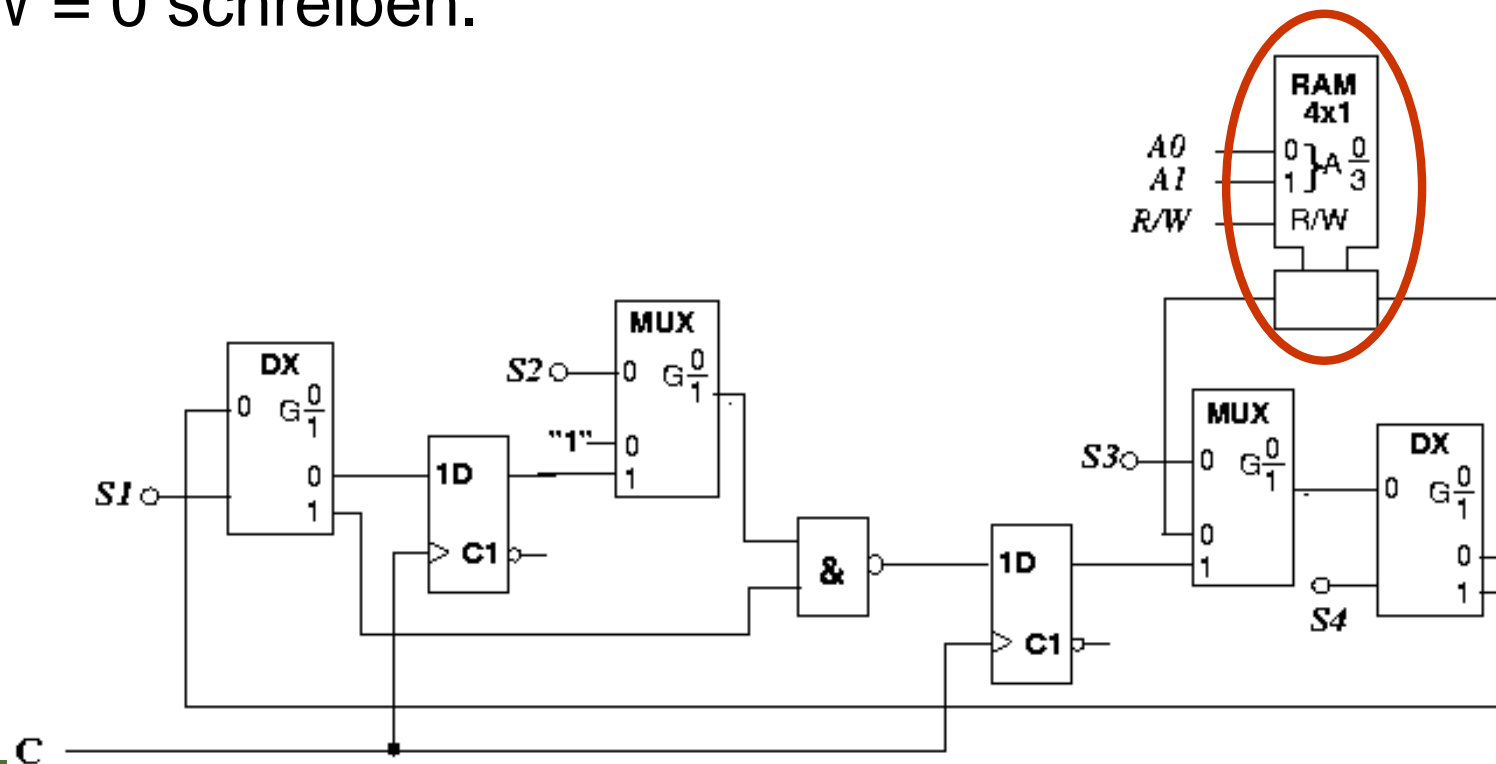
Aufgabe 1

Mit einem einfachen Rechenwerk soll die Funktion $x_1 \leftrightarrow x_2$ implementiert werden. Der Datenfluss enthält lediglich ein **NAND-Gatter** zur logischen Verknüpfung zweier Operanden. Die Daten stehen in **D-Flipflops**, die mit ansteigender Taktflanke gesteuert werden.



Aufgabe 1

Die Variablen x_1 and x_2 stehen im **RAM** unter der Adresse **00** und **01**, das Ergebnis soll in **11** abgelegt werden. Die Speicherzelle mit der Adresse **10** ist frei verfügbar z. B. für Zwischenergebnisse. Dabei bedeutet $R/W = 1$ lesen und $R/W = 0$ schreiben.



Aufgabe 1.1

1. Geben Sie einen schaltalgebraischen Ausdruck für die Antivalenz, der nur NAND-Verknüpfungen enthält.

$$\begin{aligned}x_1 \nleftrightarrow x_2 &= \overline{\overline{\bar{x}_1 x_2} \vee x_1 \bar{x}_2} \\&= (\bar{x}_1 \wedge x_2) \wedge (x_1 \wedge \bar{x}_2)\end{aligned}$$

$$\begin{aligned}\bar{x}_1 &= x_1 \wedge 1 \\ \bar{x}_2 &= x_2 \wedge 1\end{aligned}$$



Aufgabe 1.2

2. Geben Sie die zur Steuerung des Datenflusses notwendigen Bitkombinationen (Belegungen der Steuervariablen) für die 2:1-MUX/DMUX an, die zur Berechnung der Antivalenz nötig sind.

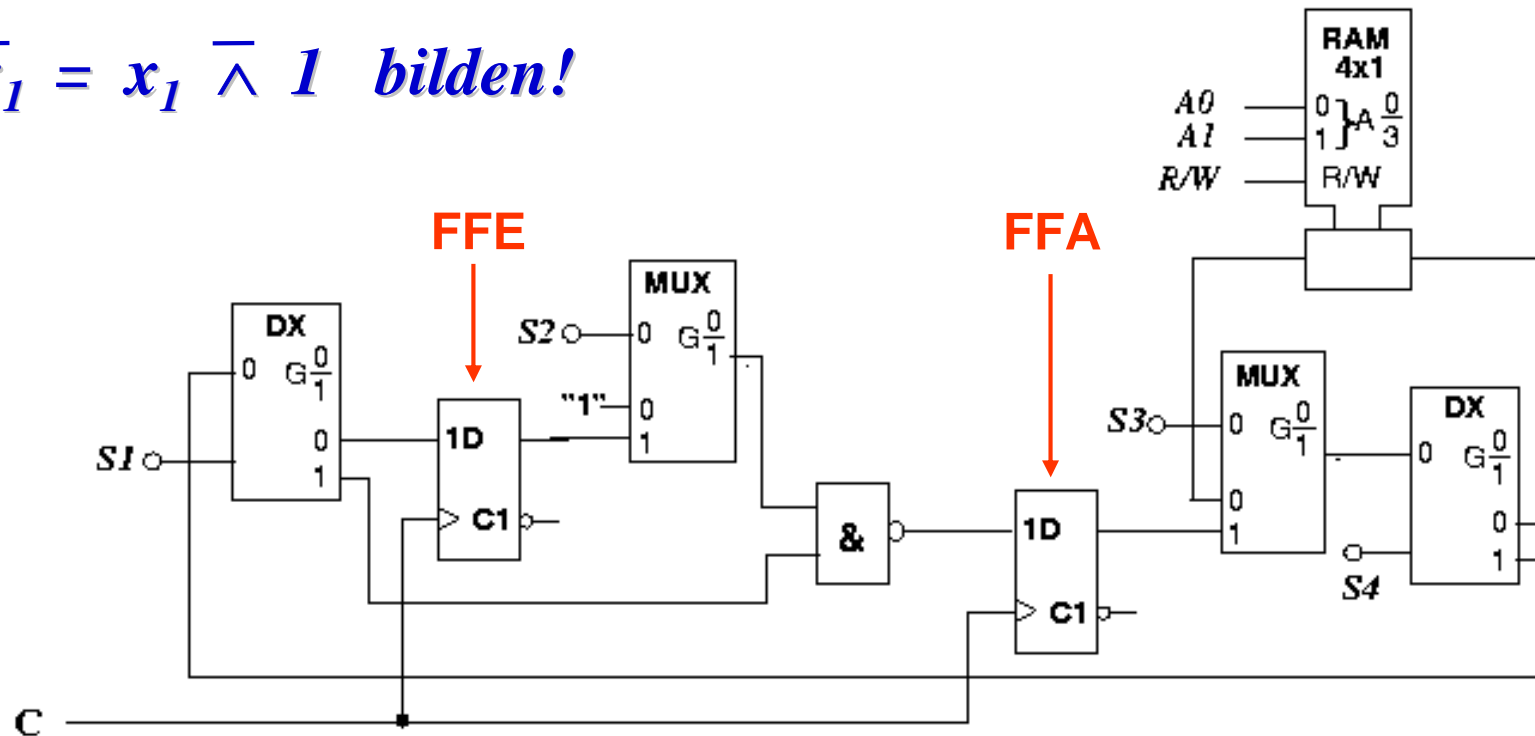
Tragen Sie diese Bitkombinationen zeilenweise in einer Tabelle ein, wobei jede Zeile einer Periode des Taktes entspricht.

Wird mit einer Zeile (=Bitkombination) der RAM-Baustein angesprochen, so ist die Adresse in der Tabelle einzutragen.



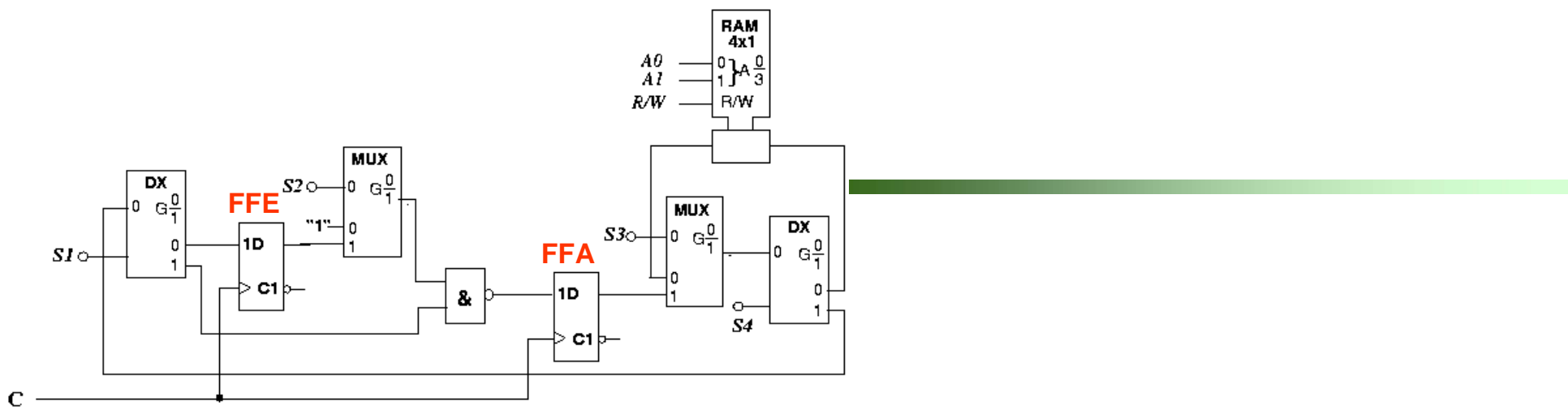
Aufgabe 1.2

$\bar{x}_1 = x_1 \bar{\wedge} 1$ bilden!



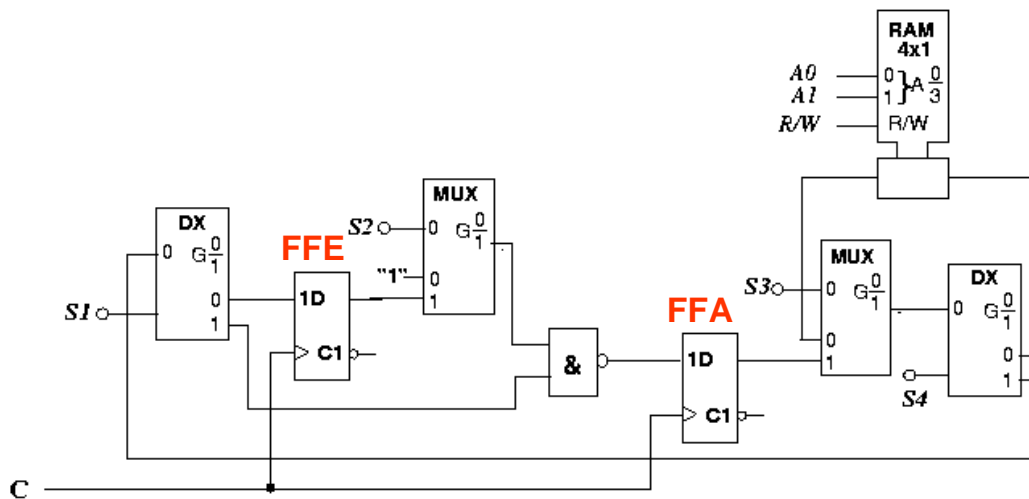
Takt	S1	S2	S3	S4	A1	A0	R/W	Kommentar
	1	0	0	1	0	0	1	\bar{x}_1 bilden und in Flipflop FFA speichern





Takt	S1	S2	S3	S4	A1	A0	R/W	Kommentar
1.	1	0	0	1	0	0	1	$\overline{x_1} \rightarrow \text{FFA}$





$$x_1 \leftrightarrow x_2 = (\bar{x}_1 \wedge x_2) \wedge (x_1 \wedge \bar{x}_2)$$

Takt	S1	S2	S3	S4	A1	A0	R/W	Kommentar
1.	1	0	0	1	0	0	1	$\bar{x}_1 \rightarrow \text{FFA}$
2.	0	-	1	1	-	-	1	$\text{FFA} \rightarrow \text{FFE}$
3.	1	1	0	1	0	1	1	$\bar{x}_1 \wedge x_2 \rightarrow \text{FFA}$
4.	-	-	1	0	1	0	0	$\text{FFA} \rightarrow \text{RAM (adr. 10)}$
5.	1	0	0	1	0	1	1	$\bar{x}_2 \rightarrow \text{FFA}$
6.	0	-	1	1	-	-	1	$\text{FFA} \rightarrow \text{FFE}$
7.	1	1	0	1	0	0	1	$x_1 \wedge \bar{x}_2 \rightarrow \text{FFA}$
8.	0	-	1	1	-	-	1	$\text{FFA} \rightarrow \text{FFE}$
9.	1	1	0	1	1	0	1	$(\langle \text{Adr. 10} \rangle \wedge \bar{\text{FFE}}) \rightarrow \text{FFA}$
10.	-	-	1	0	1	1	0	$\text{FFA} \rightarrow \text{RAM (adr. 11)}$

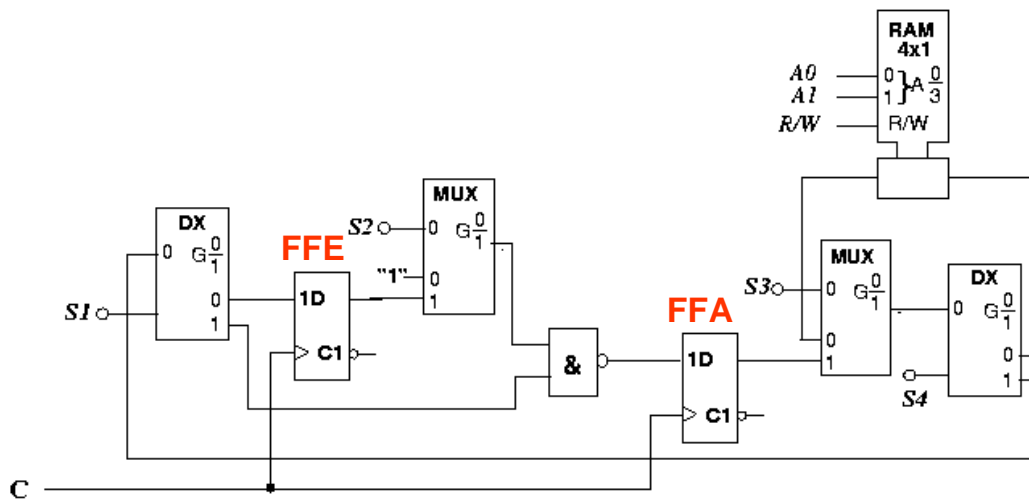


Aufgabe 1.3

3. Entwerfen Sie eine „Programmiersprache“, d. h. Befehle in lesbarem Quellcode, wobei ein Befehl einer Zeile der Tabelle (= Objektcode) entspricht. Jeder Befehl soll die auszuführende Operation und ggf. die RAM-Adresse (Hauptspeicheradresse) enthalten.

Befehlsformat: **Operation [RAM-Adresse]**





Takt	S1	S2	S3	S4	A1	A0	R/W	Kommentar
1.	1	0	0	1	0	0	1	$\bar{x}_1 \rightarrow \text{FFA}$
2.	0	-	1	1	-	-	1	$\text{FFA} \rightarrow \text{FFE}$
3.	1	1	0	1	0	1	1	$\bar{x}_1 \bar{\wedge} x_2 \rightarrow \text{FFA}$
4.	-	-	1	0	1	0	0	$\text{FFA} \rightarrow \text{RAM (adr. 10)}$
5.	1	0	0	1	0	1	1	$\bar{x}_2 \rightarrow \text{FFA}$
6.	0	-	1	1	-	-	1	$\text{FFA} \rightarrow \text{FFE}$
7.	1	1	0	1	0	0	1	$x_1 \bar{\wedge} \bar{x}_2 \rightarrow \text{FFA}$
8.	0	-	1	1	-	-	1	$\text{FFA} \rightarrow \text{FFE}$
9.	1	1	0	1	1	0	1	$(\langle \text{Adr. 10} \rangle \bar{\wedge} \text{FFE}) \rightarrow \text{FFA}$
10.	-	-	1	0	1	1	0	$\text{FFA} \rightarrow \text{RAM (adr. 11)}$



Aufgabe 1.3

Es werden vier verschiedene Befehle benötigt, z. B.

➤ **NOT [adresse]**

Das Bit an der Adresse **adresse** holen und invertieren.
Das Ergebnis steht in FFA

➤ **NAND[adresse]**

Das Bit an der Adresse **adresse** holen und mit dem Bit in FFE durch die NAND-Funktion verknüpfen.
Das Ergebnis steht in FFA

➤ **TAE**

Transferiert den Inhalt von FFA nach FFE

➤ **STA[adresse]**

Speichert den Inhalt von FFA an der Adresse **adresse**



Aufgabe 1.3

Quellcode	Kommentar
NOT[00]	$\bar{x}_1 \rightarrow \text{FFA}$
TAE	$\text{FFA} \rightarrow \text{FFE}$
NAND[01]	$\bar{x}_1 \bar{\wedge} x_2 \rightarrow \text{FFA}$
STA [10]	$\text{FFA} \rightarrow \text{RAM (adresse 10)}$
NOT[01]	$\bar{x}_2 \rightarrow \text{FFA}$
TAE	$\text{FFA} \rightarrow \text{FFE}$
NAND[00]	$x_1 \bar{\wedge} \bar{x}_2 \rightarrow \text{FFA}$
TAE	$\text{FFA} \rightarrow \text{FFE}$
NAND[10]	$(\langle \text{adr. 10} \rangle \bar{\wedge} \text{FFE}) \rightarrow \text{FFA}$
STA[11]	$\text{FFA} \rightarrow \text{RAM (adr. 11)}$

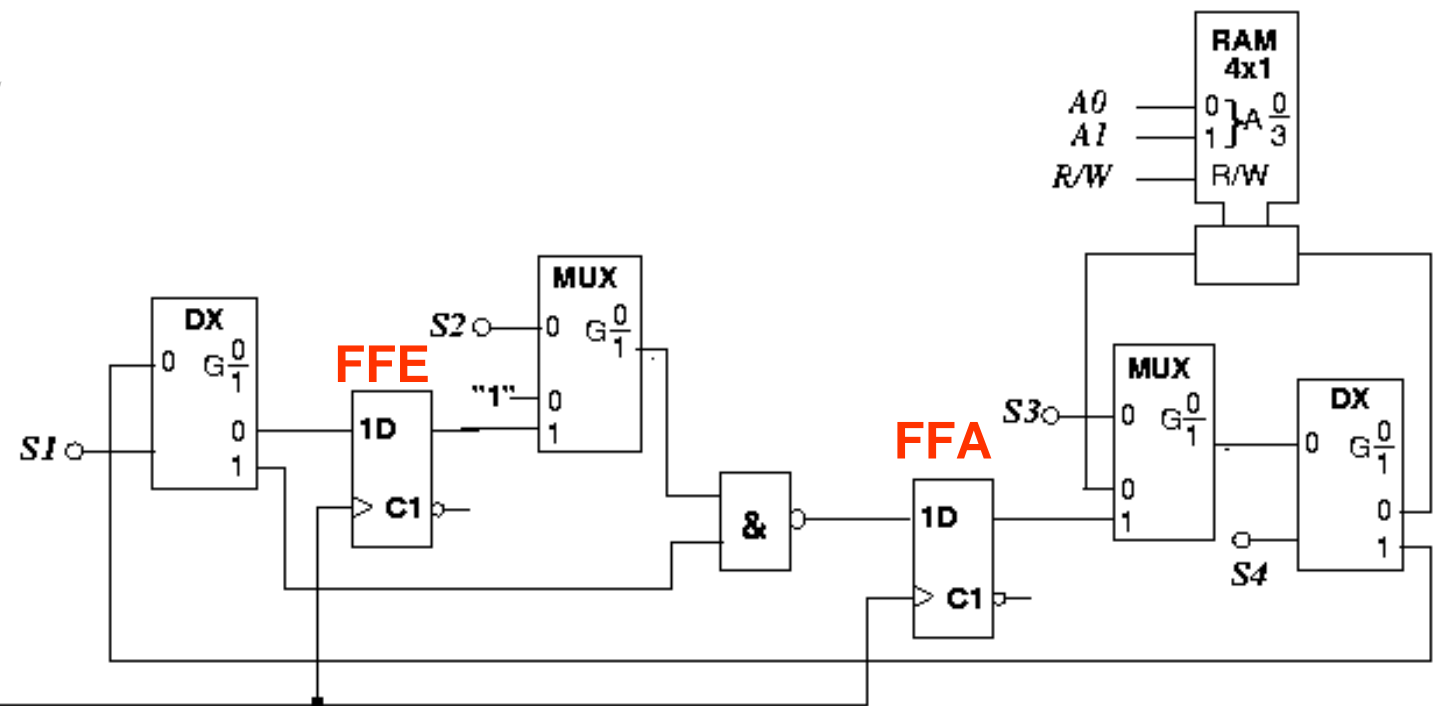


Aufgabe 1.4

4. Geben Sie ein „Programm“ zur Berechnung der Äquivalenzfunktion $x_1 \leftrightarrow x_2$ an.

$$\begin{aligned}
 x_1 \leftrightarrow x_2 &= \overline{\overline{x_1} \overline{x_2} \vee x_1 x_2} \\
 &= (\overline{x_1} \wedge \overline{x_2}) \wedge (x_1 \wedge x_2)
 \end{aligned}$$

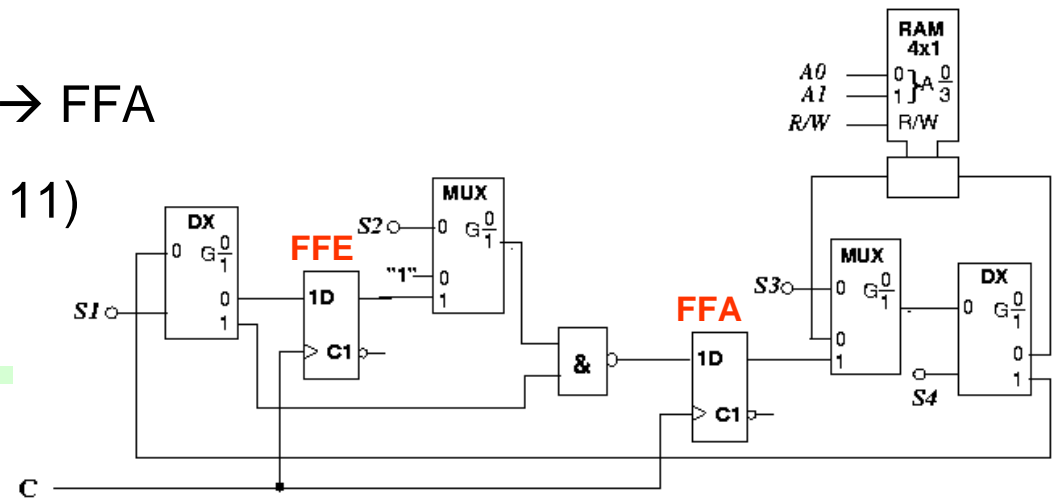
$$\begin{aligned}
 \overline{x_1} &= x_1 \wedge 1 \\
 \overline{x_2} &= x_2 \wedge 1
 \end{aligned}$$



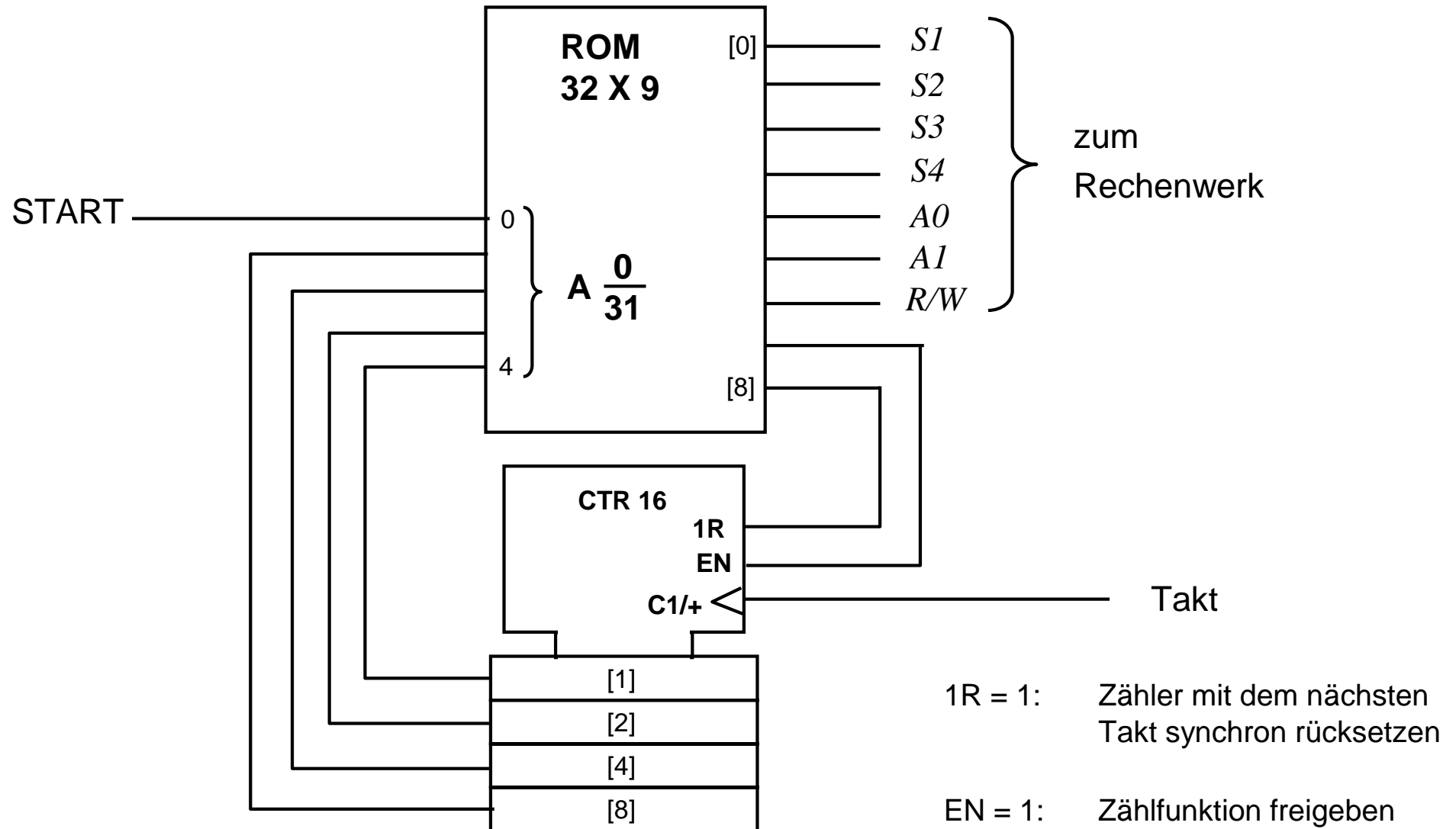
Aufgabe 1.4

NOT[00]	$\overline{x}_1 \rightarrow \text{FFA}$
STA [10]	FFA \rightarrow adresse 10
NOT[01]	$\overline{x}_2 \rightarrow \text{FFA}$
TAE	FFA \rightarrow FFE
NAND[10]	$\overline{x}_1 \wedge \overline{x}_2 \rightarrow \text{FFA}$
STA [10]	FFA \rightarrow adresse 10
LOAD[00]	$x_1 \rightarrow \text{FFE}$
NAND [01]	$x_1 \wedge x_2 \rightarrow \text{FFA}$
TAE	FFA \rightarrow FFE
NAND[10]	($\langle \text{adr. 10} \rangle \wedge \text{FFE}$) \rightarrow FFA
STA [11]	FFA \rightarrow RAM (adr. 11)

- **LOAD[adresse]**
Holt den Inhalt der Adresse **adresse** ins FFE



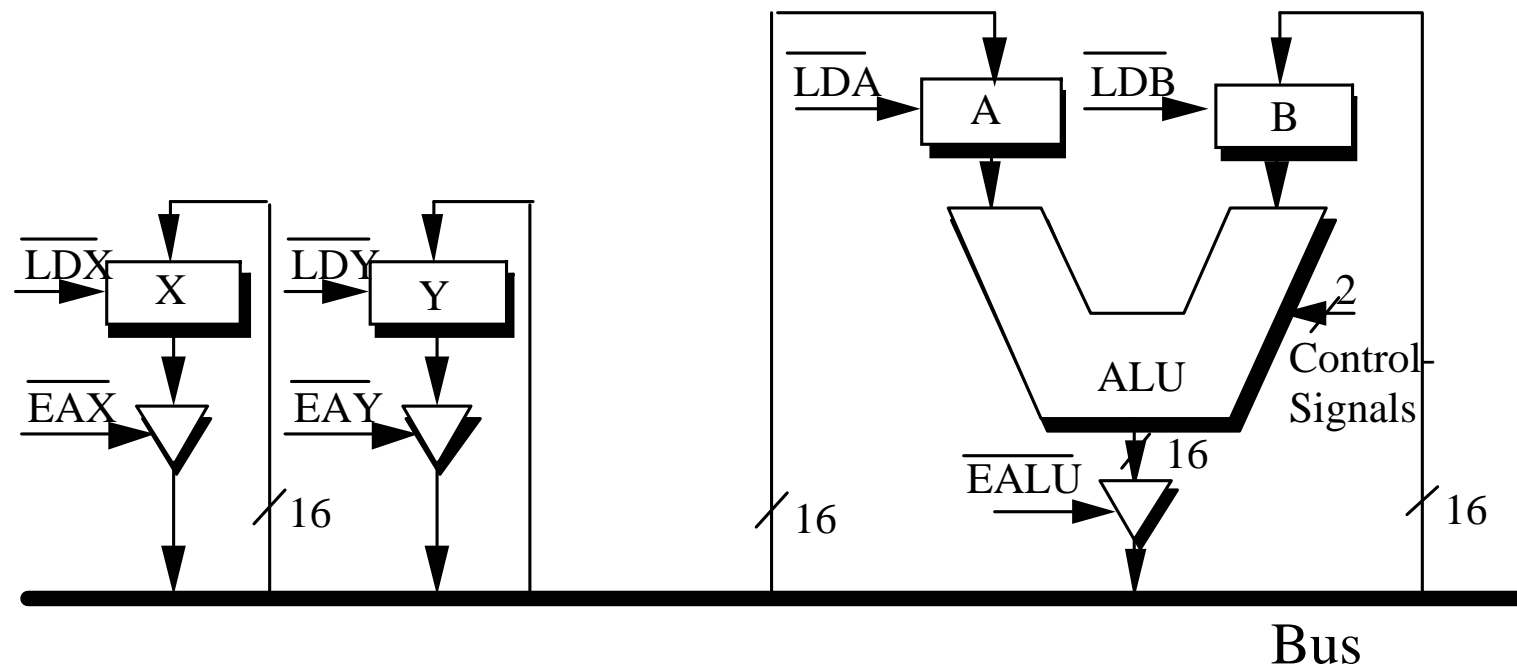
Steuerwerk



Aufgabe 2

Gegeben:

Eine mikroprogrammierbare Schaltung besteht aus einem Bus, 4 Registern, einer ALU und einigen Tristate-Treibern



Aufgabe 2

Gesucht:

Mikroprogramme für die folgenden Operationen 1-5 durch Programmierung eines Datenpfades:

		Controlsignale	Operation
		C_1 C_0	
1.	$x = x + y$	0 0	A + B
2.	$x = x - y$	0 1	A - B
3.	$x = x \text{ and } y$	1 0	A und B
4.	$x = x \text{ or } y$	1 1	A oder B
5.	$y = x$ (move x to y)		



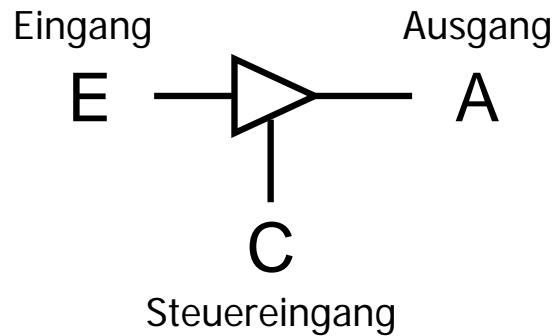
Tri-State-Treiber

Gatter, die neben den Pegelzuständen **H** (high) und **L** (low) einen **dritten hochohmigen Zustand** besitzen.

- In diesem Zustand ist der Ausgang hochohmig gegen Betriebsspannungen beider Polaritäten.
- Diese Gatter ermöglichen es, mehrere Gatterausgänge auf eine gemeinsame Leitung zusammenzuschalten (Bussystem)
- Sie dienen auch durch Ausgangstreiber zur elektrischen Anpassung der Prozessorsignale an die Signalspezifikationen, die von anderen Systemkomponenten verlangt werden.

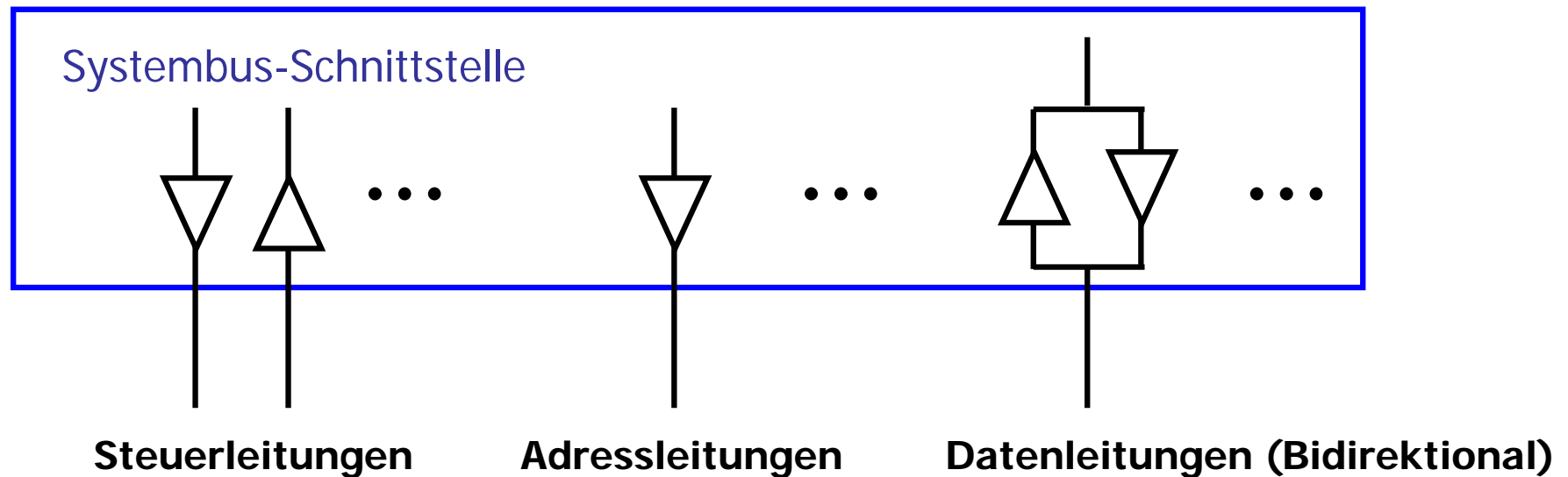


Tri-State-Treiber

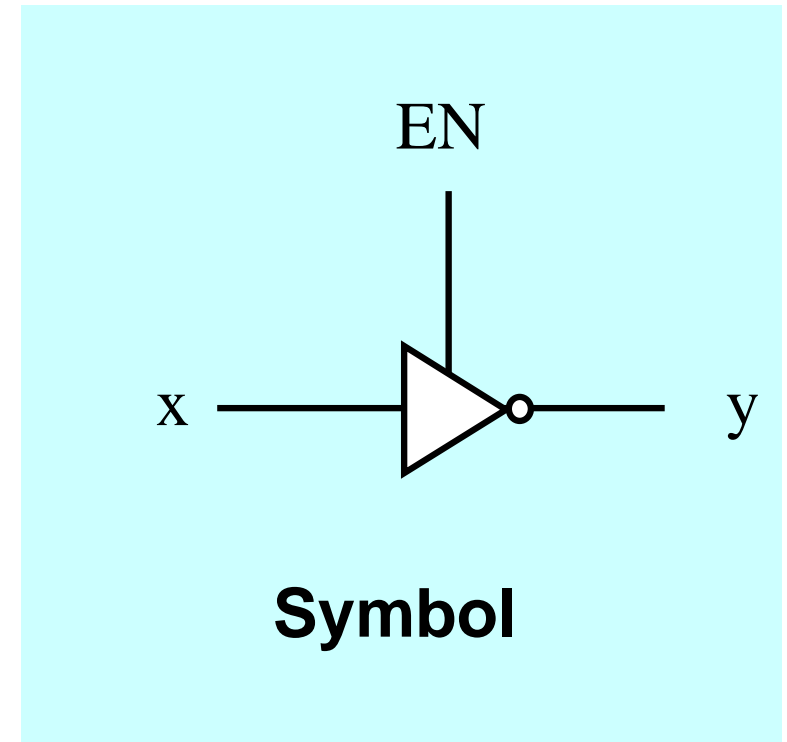
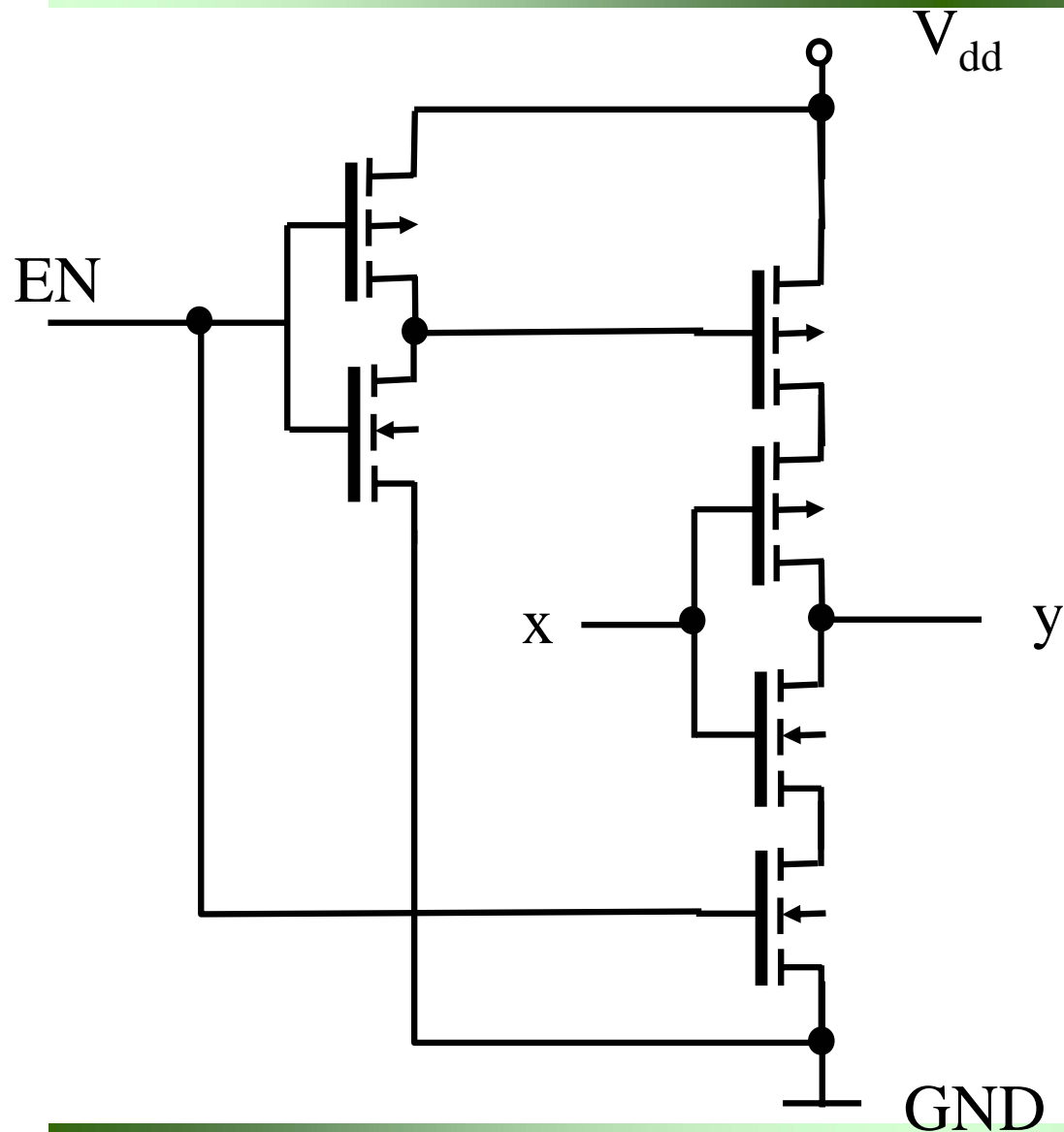


Funktionstabelle:

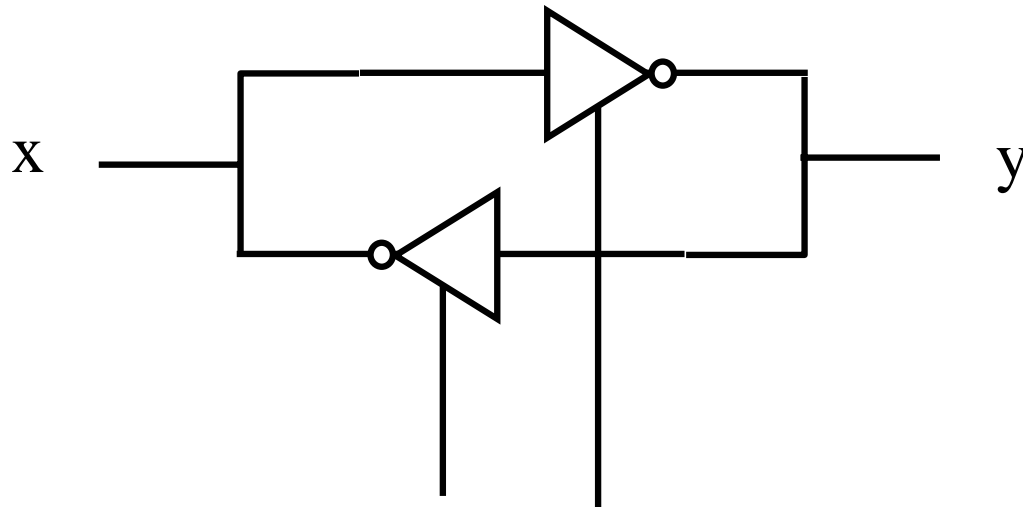
C	E	A
H	L	L
H	H	H
L	-	Z hochohmig



CMOS Tri-state-Inverter



Bidirektionale Tristate-Gatter



Funktionstabelle:

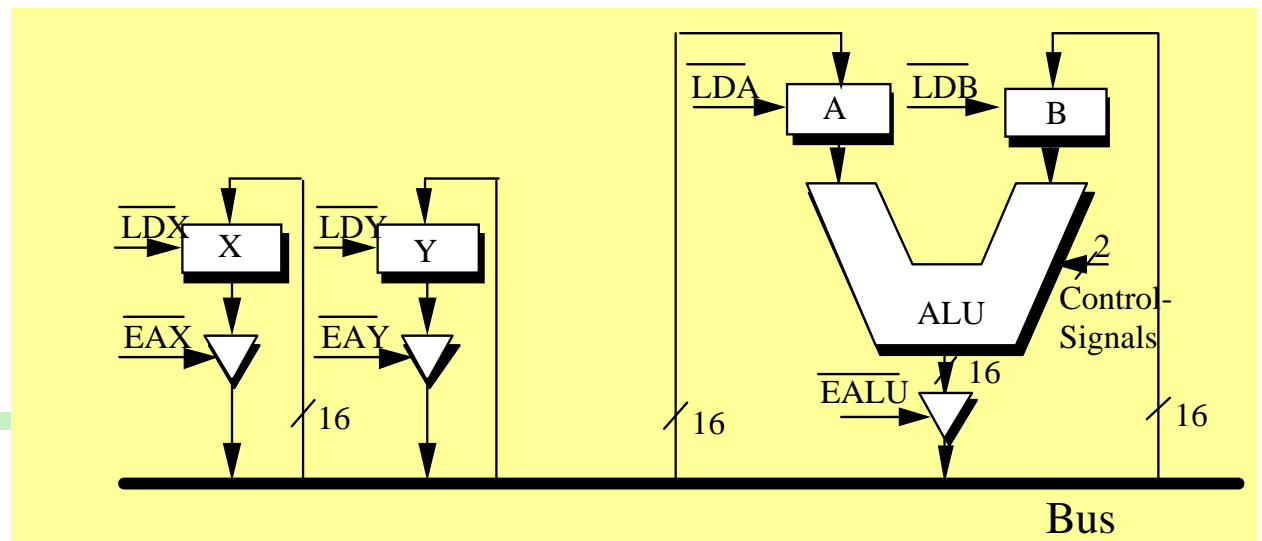
C	DIR	
L	L	$x \rightarrow y$
L	H	$y \rightarrow x$
H	-	Z hochohmig



Lösung 2

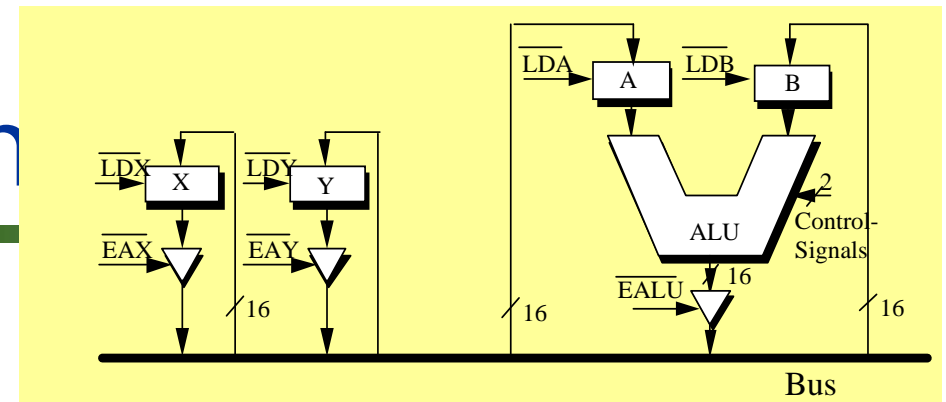
Mikroprogramme für die Operationen 1-4:

- X auf dem Bus legen. Warten bis die Daten stabil anliegen.
- X ins Register A laden.
- Y auf dem Bus legen. Warten bis die Daten stabil anliegen.
- Y ins Register B laden.
- Ergebnis auf den Bus. Warten bis es stabil anliegt.
- Ergebnis ins Register X laden.



Lösung

\overline{EAX}	0	0	1	1	1	1
\overline{LDX}	1	1	1	1	1	0
\overline{EAY}	1	1	0	0	1	1
\overline{LDY}	1	1	1	1	1	1
\overline{LDA}	1	0	1	1	1	1
\overline{LDB}	1	1	1	0	1	1
\overline{EALU}	1	1	1	1	0	0
C_1	-	-	-	0	0	0
C_0	-	-	-	0	0	0



$$x = x + y$$

$$0 \quad 1 \quad 1$$

$$x = x - y$$

$$x = x \text{ and } y$$

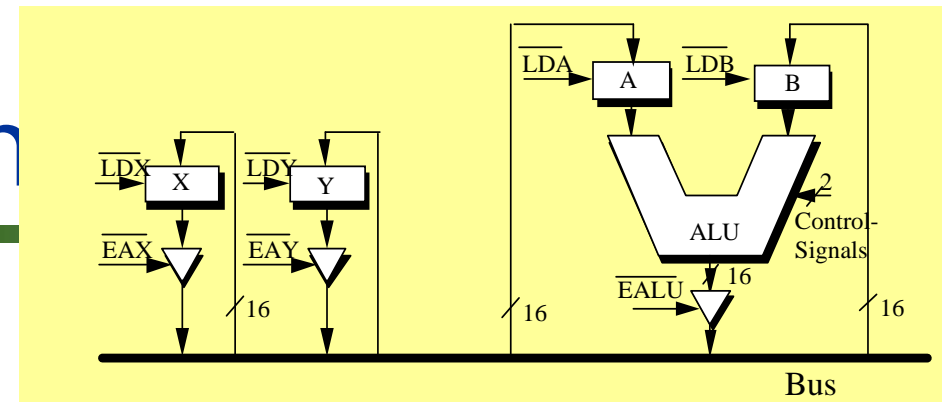
$$1 \quad 0 \quad 1$$

$$x = x \text{ or } y$$



Lösung

Mikroprogramm für die Operation 5:

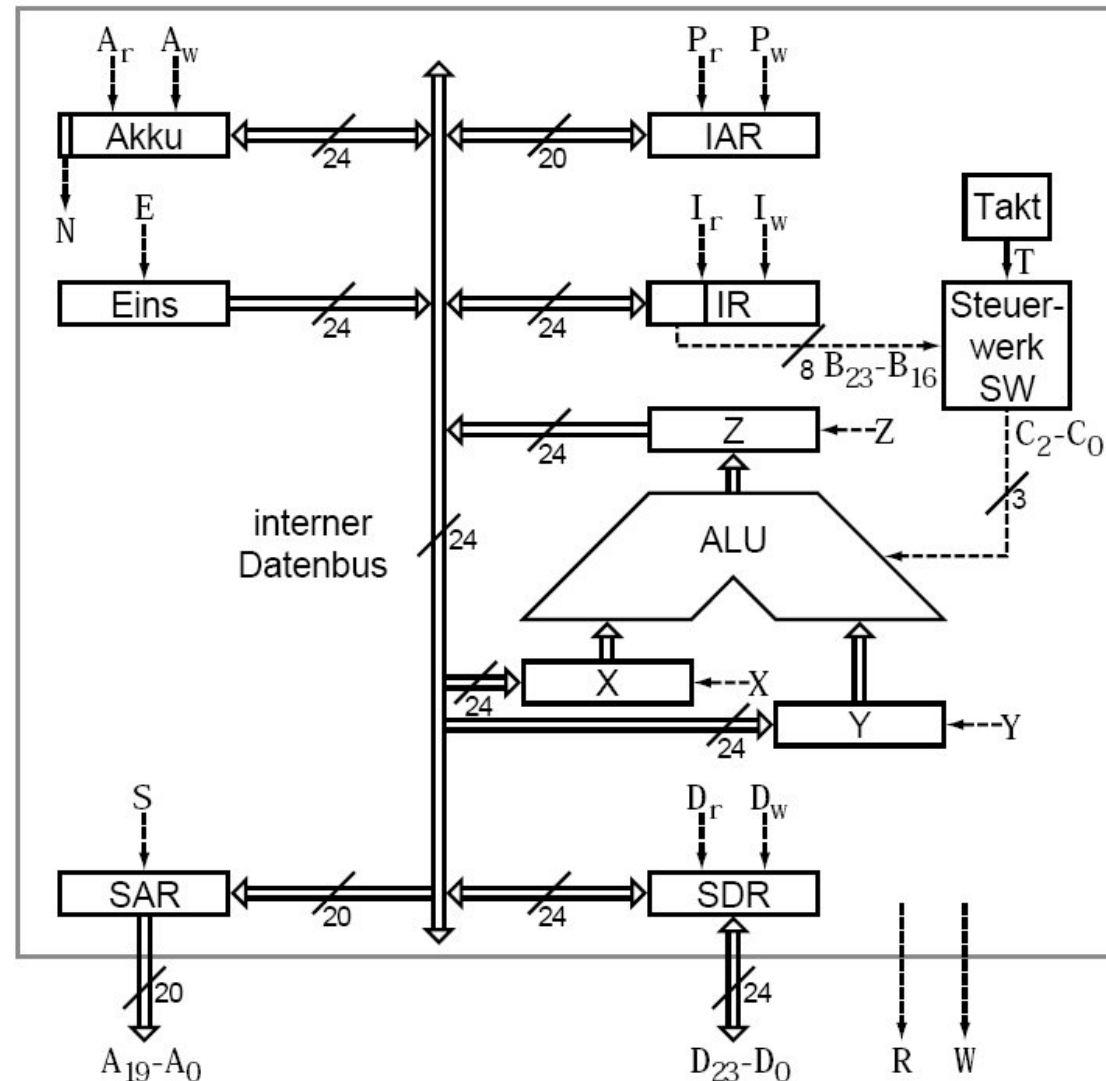


- X auf dem Bus legen. Warten bis die Daten stabil anliegen.
- Y laden.

\overline{EAX}	0	0
\overline{LDX}	1	1
\overline{EAY}	1	1
\overline{LDY}	1	0
\overline{LDA}	1	1
\overline{LDB}	1	1
\overline{EALU}	1	1

Mima-Architektur (Siehe Übungsblatt 2)

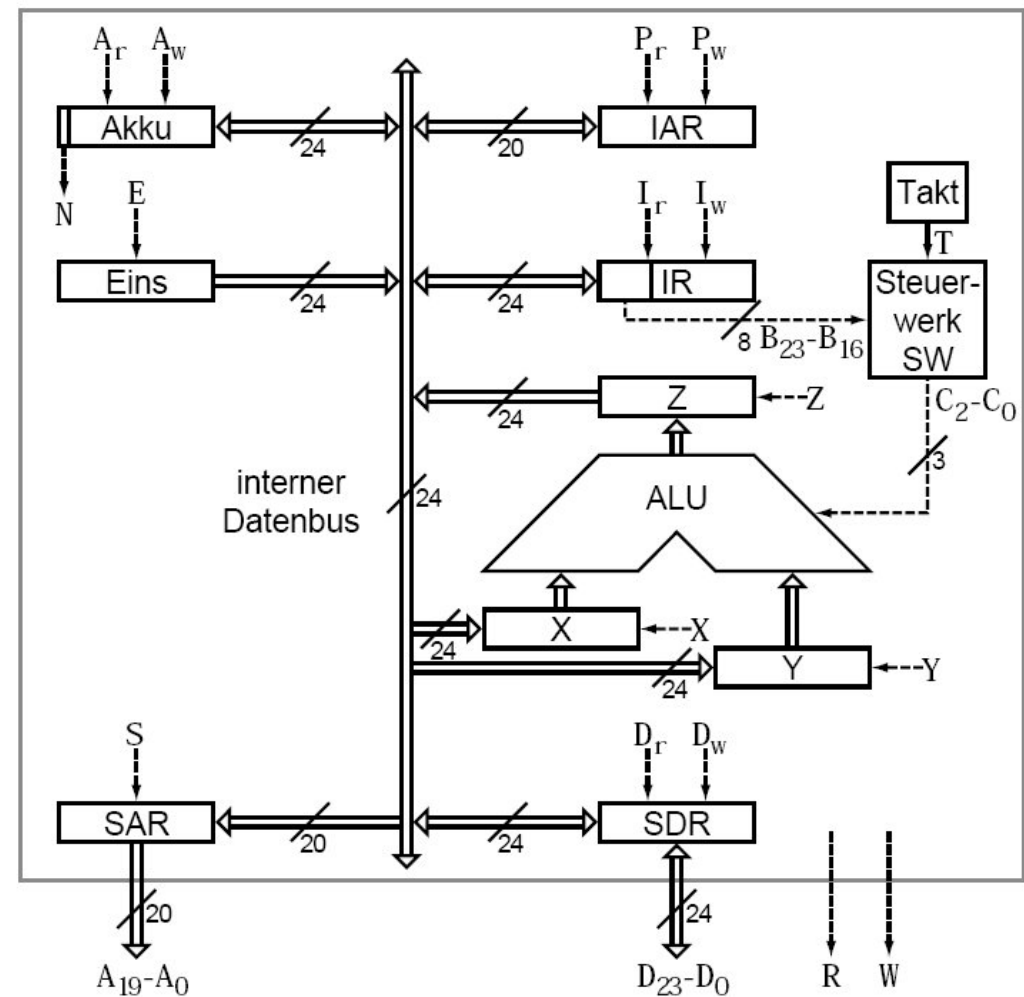
- Mikroprogrammierte Minimalmaschine (von-Neumann-Prinzip)
- SW mit 10 Meldesignale, 18 Steuersignale und Mikroprogrammspeicher für maximal 256 Mikrobefehle
- Befehlsabarbeitung:
 - Lese-Phase
 - Dekodierphase
 - Ausführungsphase
- 3 Taktzyklen für Lese- und Schreibzugriffe



Mima-Architektur

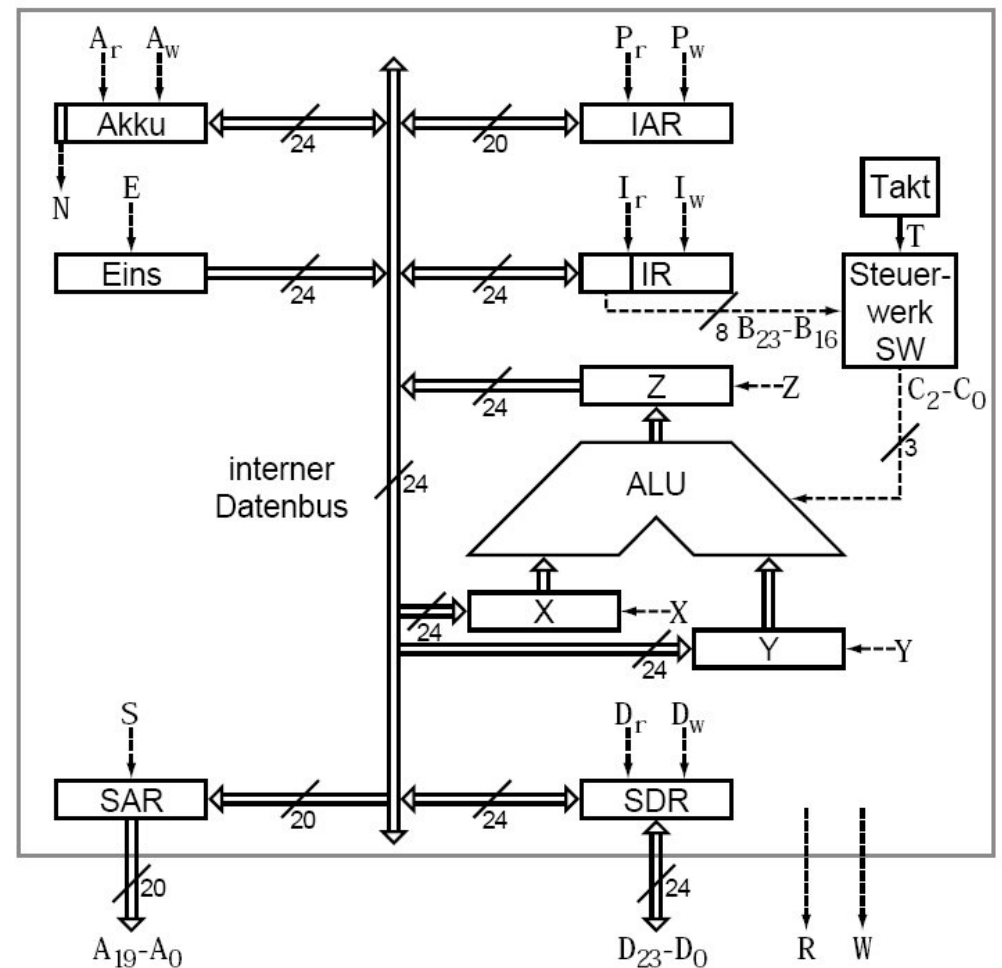
- Mikroprogramm für die Lese phase besteht aus 5 Mikrobefehlen:

1.Takt: $IAR \rightarrow SAR$; $IAR \rightarrow X$; $R=1$
2.Takt: $Eins \rightarrow Y$; $R=1$
3.Takt: ALU auf Addieren; $R=1$
4.Takt: $Z \rightarrow IAR$
5.Takt: $SDR \rightarrow IR$

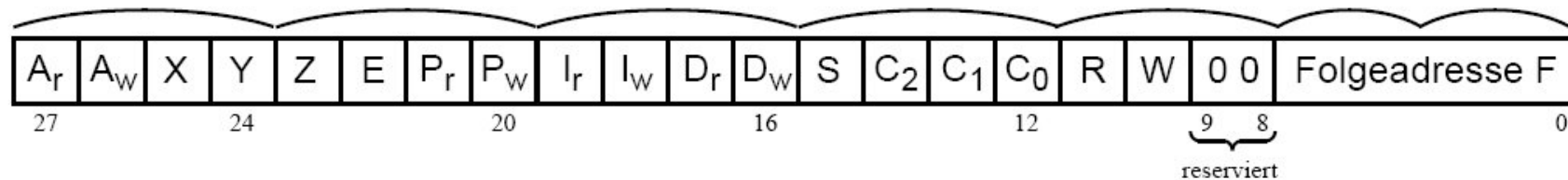


Mima-Architektur

1. Takt: $IAR \rightarrow SAR; IAR \rightarrow X; R = 1$
2. Takt: $Eins \rightarrow Y; R = 1$
3. Takt: ALU auf Addieren; $R = 1$
4. Takt: $Z \rightarrow IAR$
5. Takt: $SDR \rightarrow IR$



➤ Mikrobefehlsformat



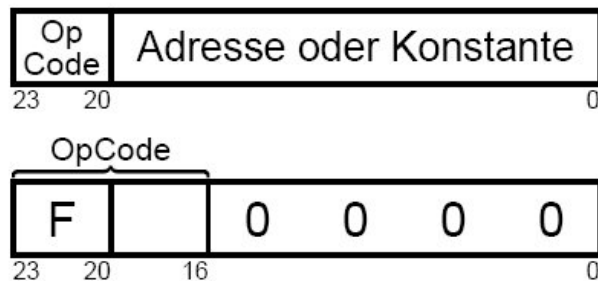
Beispiel: 0x77:

7	0	0	0	0	7	9
---	---	---	---	---	---	---

$A_w = X = Y = 1$ (Akku \rightarrow X; Akku \rightarrow Y)
Adresse des nächsten Befehls ist 0x79



Befehlsformate, ALU-Operationen, ...



$C_2C_1C_0$	ALU Operation
0 0 0	tue nichts (d.h. $Z \rightarrow Z$)
0 0 1	$X + Y \rightarrow Z$
0 1 0	rotiere X nach rechts $\rightarrow Z$
0 1 1	$X \text{ AND } Y \rightarrow Z$
1 0 0	$X \text{ OR } Y \rightarrow Z$
1 0 1	$X \text{ XOR } Y \rightarrow Z$
1 1 0	Eins-Komplement von X $\rightarrow Z$
1 1 1	falls $X = Y$, $-1 \rightarrow Z$, sonst $0 \rightarrow Z$

OpCode	Mnemonik	Beschreibung
0	LDC c	$c \rightarrow \text{Akku}$
1	LDV a	$\langle a \rangle \rightarrow \text{Akku}$
2	STV a	$\text{Akku} \rightarrow \langle a \rangle$
3	ADD a	$\text{Akku} + \langle a \rangle \rightarrow \text{Akku}$
4	AND a	$\text{Akku AND } \langle a \rangle \rightarrow \text{Akku}$
5	OR a	$\text{Akku OR } \langle a \rangle \rightarrow \text{Akku}$
6	XOR a	$\text{Akku XOR } \langle a \rangle \rightarrow \text{Akku}$
7	EQL a	falls $\text{Akku} = \langle a \rangle$: $-1 \rightarrow \text{Akku}$ sonst: $0 \rightarrow \text{Akku}$
8	JMP a	$a \rightarrow \text{IAR}$
9	JMN a	falls $\text{Akku} < 0$: $a \rightarrow \text{IAR}$
F0	HALT	stoppt die MIMA
F1	NOT	bilde Eins-Komplement von Akku $\rightarrow \text{Akku}$
F2	RAR	rotiere Akku eins nach rechts $\rightarrow \text{Akku}$



Beispiel

Mikroprogramm für:

OR a Akku OR <a> → Akku

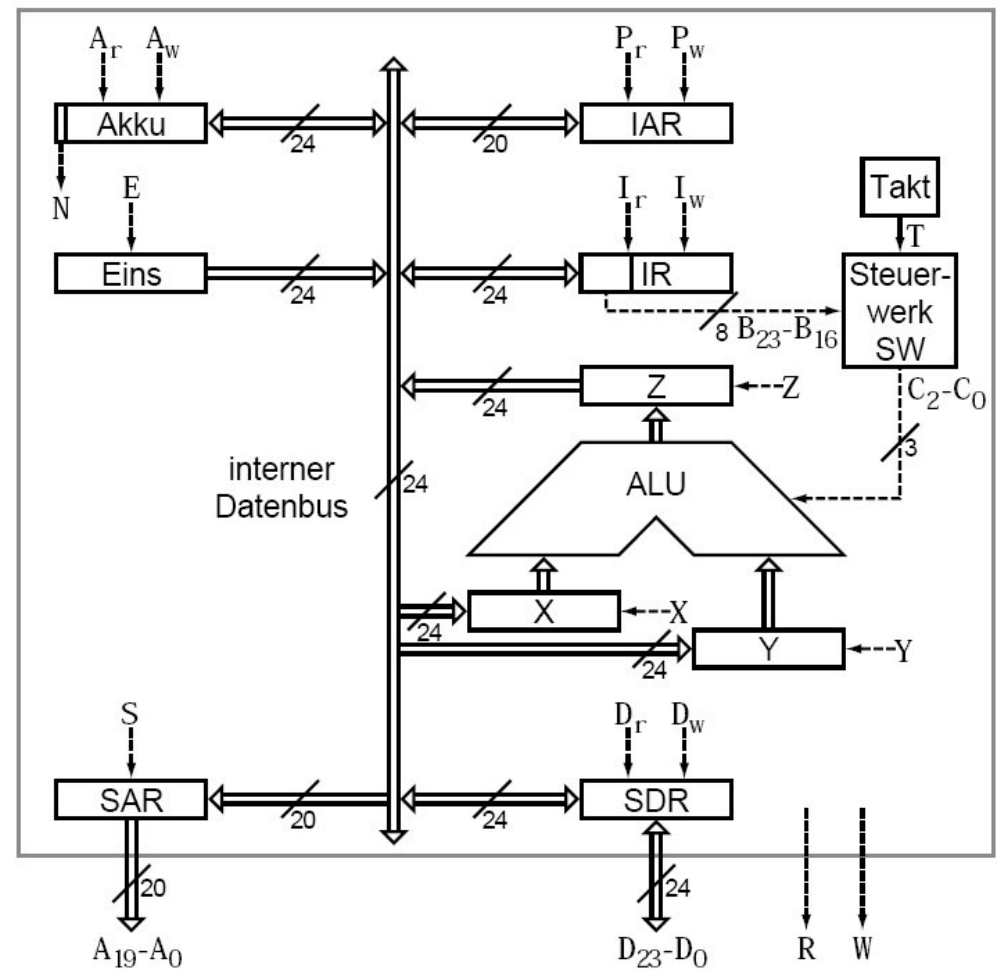
1.Takt: IAR → SAR; IAR → X; R = 1

2.Takt: Eins → Y; R = 1

3.Takt: ALU auf Addieren; R = 1

4.Takt: Z → IAR

5.Takt: SDR → IR



Mima-Architektur

- JAVA-Simulation der MIMA, Beispielprogramme und ein c-Interpreter auf der TI-Homepage:

<http://i61www.ira.uka.de/users/asfour/TI/Mima/>

