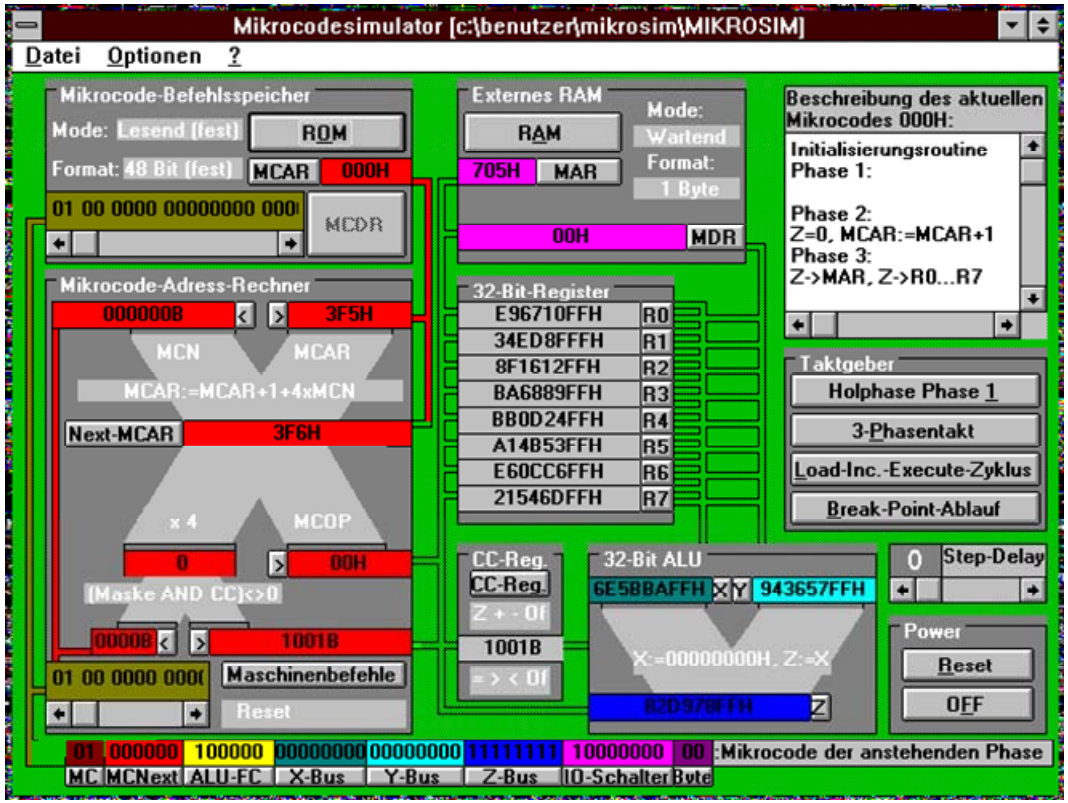
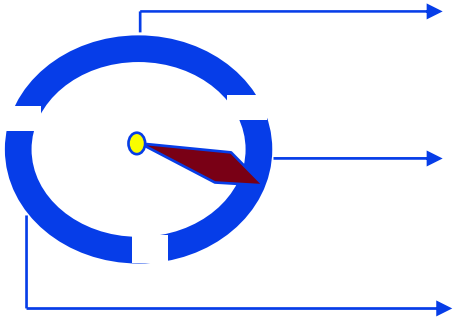
7. Aufbau und Arbeitsweise einer mikroprogrammierten CPU



Takte und Phasen
Register und Busse
ALU
Hauptspeicher
Mikrobefehle
ROM-Speicher
Adressberechnung
Mikroprogramme
Maschinenbefehle
Load-Increment-Execute Zyklus
Mikroprogrammierte CPU

Takte und Phasen

Durch die interne Uhr werden Takte geliefert. Jeder Takt wird in drei Phasen unterteilt. In jeder der Phasen liegt ein entsprechendes Signal P_1, P_2 oder P_3 auf 1 und alle anderen auf 0. Jeder Schalter ist nur in einer bestimmten Phase aktiv.



P1: Phase 1

P2: Phase 2

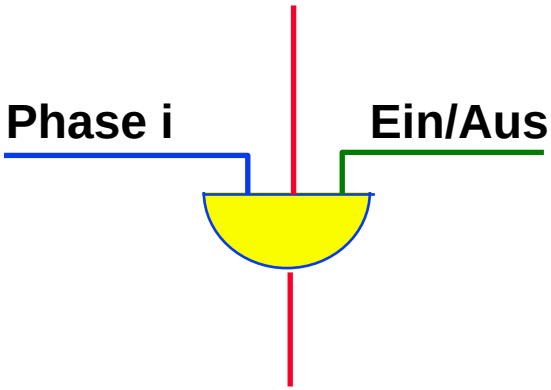
P3: Phase 3

Taktraten :

x MHz = x * 10^6 Takte/sec.

Taktdauer : 1/x * 10^{-6} sec

Bsp. 50 MHz ≅ 20 nS

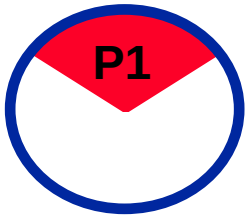


Ein Schalter, der nur in Phase i aktiv ist

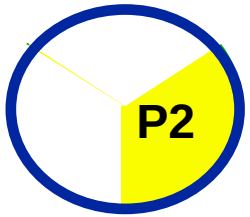
In Phase P_i ist P_i =1 und P_k = 0 für i <> k.

Die Bedeutung der Phasen

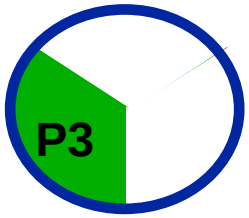
Die Ausführung eines Mikrobefehls soll jeweils einen Takt erfordern.
In jeder Phase wird ein bestimmter Teil eines Mikrobefehls erledigt.



Holphase
Argumente für CPU-Operationen werden bereitgestellt



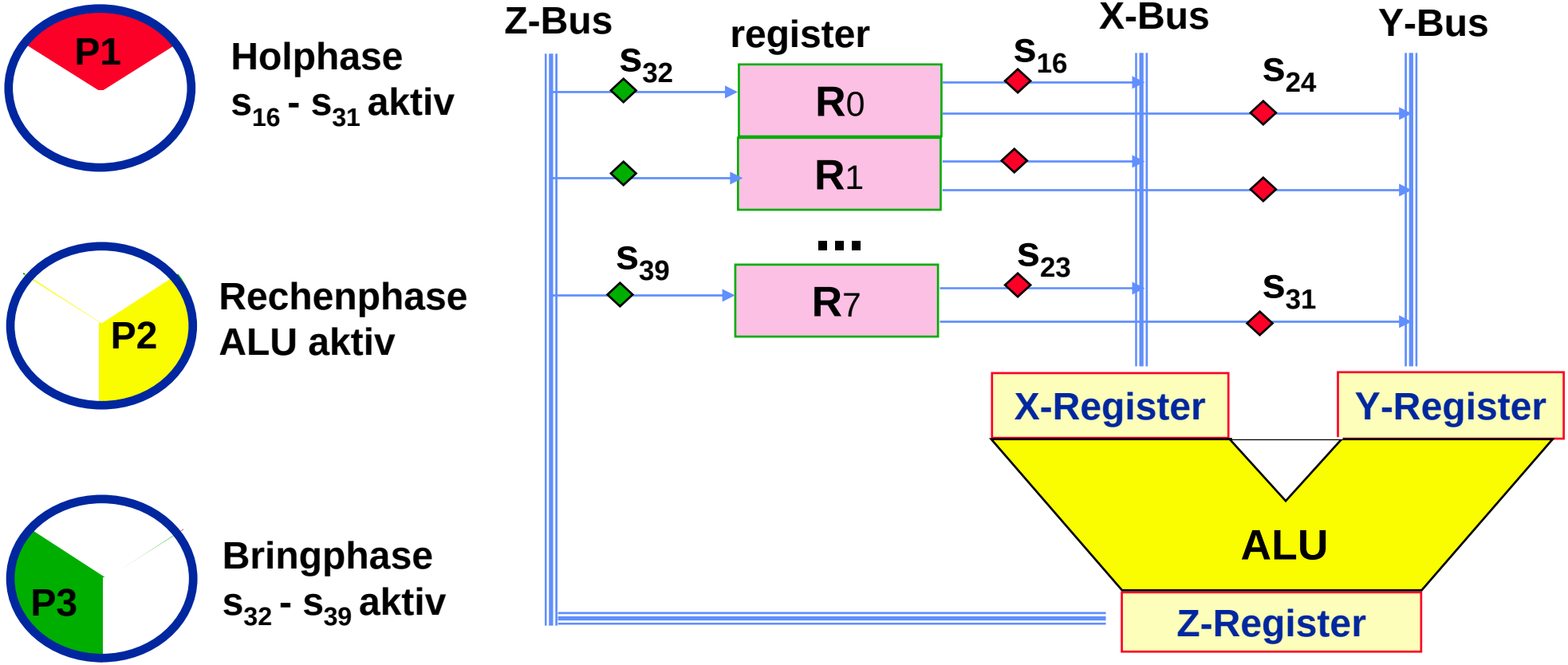
Rechenphase
CPU rechnet



Bringphase
Speichere Ergebnis der Berechnung

Aufbau einer CPU

Eine Gruppe von 32-Bit-Registern ist über Busse mit Registern X, Y und Z verbunden. Den Buszugang regeln Schalter, die nur in bestimmten Phasen aktiv sind. X und Y sind die Operanden- , Z das Ergebnisregister der ALU

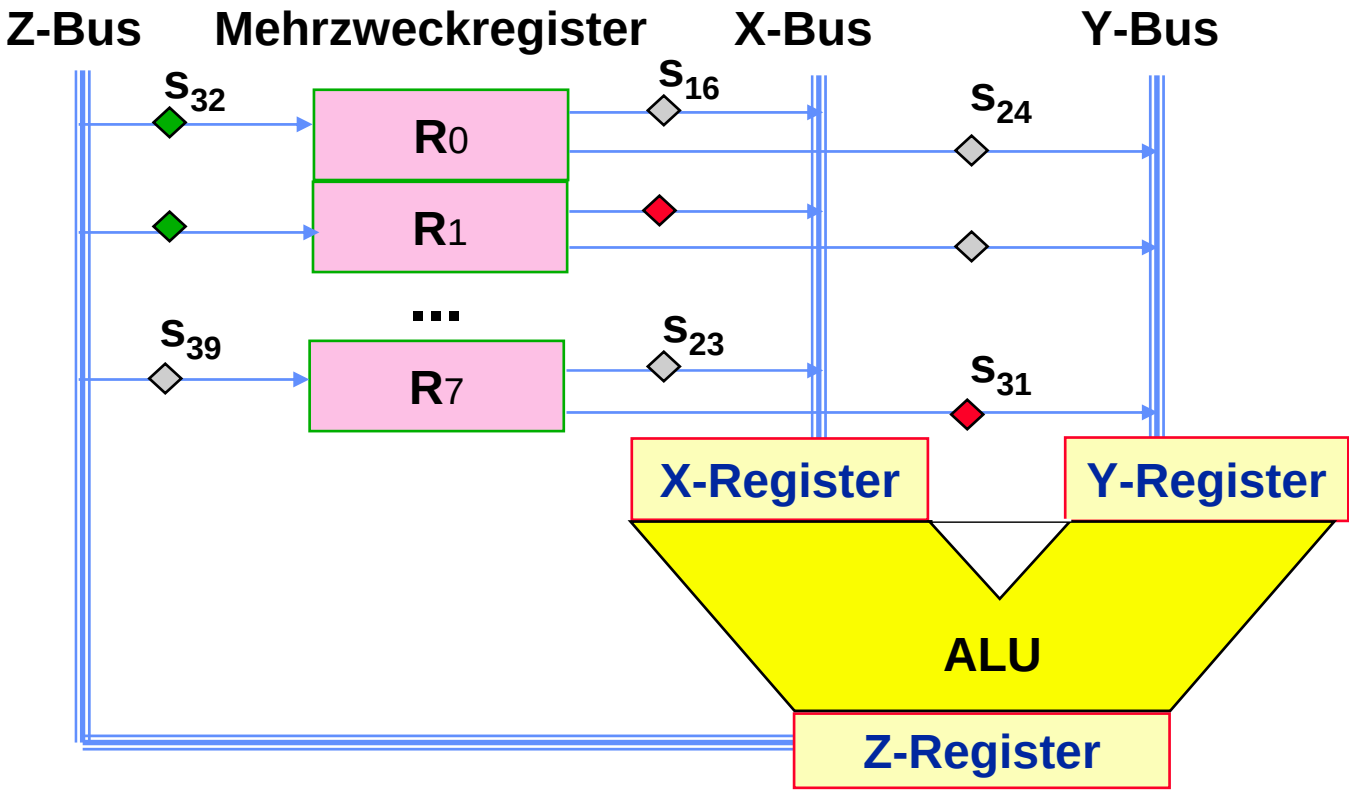


Mikrobefehle

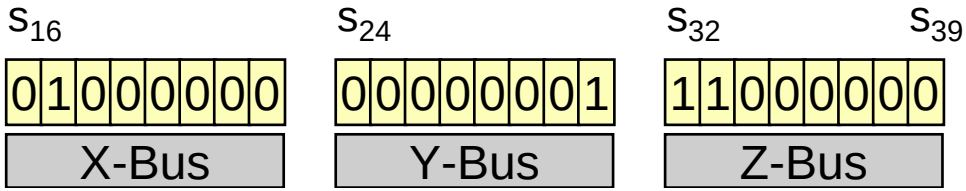
Die Stellung der Schalter für einen bestimmten Takt wird im *Maschinenwort* zusammengefaßt. Das Maschinenwort bleibt einen Takt lang gültig und steuert in diesem Takt die Arbeitsweise der CPU. Es wird daher als *Mikrobefehl* bezeichnet.

Schalter	Aktive Phase
S_{16} ... S_{31}	1
S_{32} ... S_{39}	3

In einem Takt passiert z.B.:
Phase 1 : $R_1 \rightarrow X$, $R_7 \rightarrow Y$
Phase 2 : *ALU rechnet*
Phase 3 : $Z \rightarrow R_0, R_1$



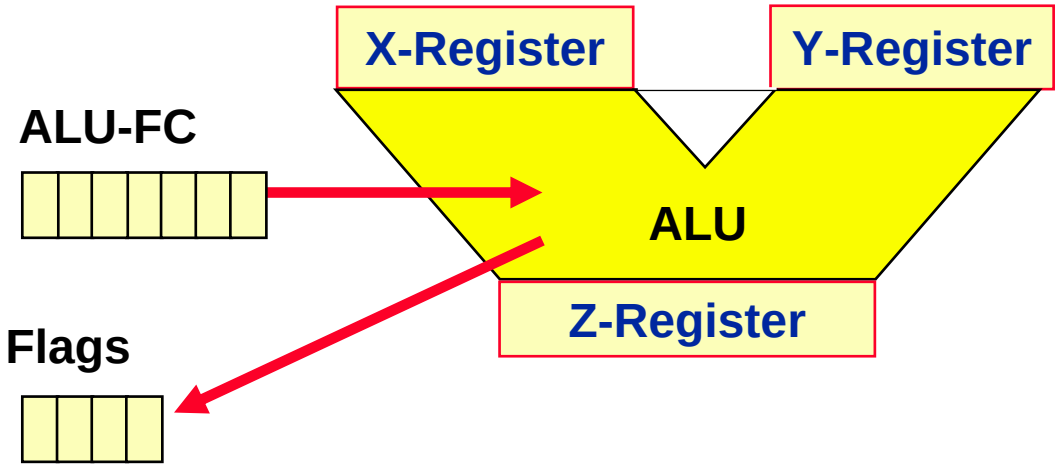
Maschinenwort



Funktionen der ALU

Die ALU unseres CPU-Modells kann 128 Funktionen ausführen. Dafür müssen die Operanden in den X- bzw. Y-Registern liegen.
Der Funktionswert kommt in das Z-Register.
Meist wird dabei auch das Flag-Register gesetzt. Die Vergleichsoperationen verändern nur das Flag-Register.

ALUFC	
0 – 18	Arithmetische Operationen
19 – 27	Logische Operationen
28 – 63	Konstante Operationen
64-127	wie 0-63 mit Kopie der ALU Flags in das CC-Register



Befehlsumfang der ALU (FC 0-63)

ALUFC	Operation
0	Z := Z (Wait)
1	Z := - Z
2	Z := X
3	Z := - X
4	Z := Y
5	Z := -Y
6	Z := Y, X <--> Y
7	Z := X, X <--> Y
8	Z := Y, Y --> X
9	Z := X+1
10	Z := X-1
11	Z := X+Y
12	Z := X-Y
13	Z := X MUL Y
14	Z := X DIV Y
15	Z := X MOD Y

ALUFC	Operation
16	Z := X SAL Y
17	Z := X SAR Y
18	Z := X CMP Y, arith.
19	Z := X AND Y
20	Z := X NAND Y
21	Z := X OR Y
22	Z := X NOR Y
23	Z := X XOR Y
24	Z := X NXOR Y
25	Z := X SLL Y
26	Z := X SLR Y
27	Z := X CMP Y, log.
28	X := 0
29	X := FFFFFFFFh
30	Y := 0
31	Y := FFFFFFFFh

Falls 31 < ALUFC < 47 :

Z, X := ALUFC - 32

Falls 48 < ALUFC < 63 :

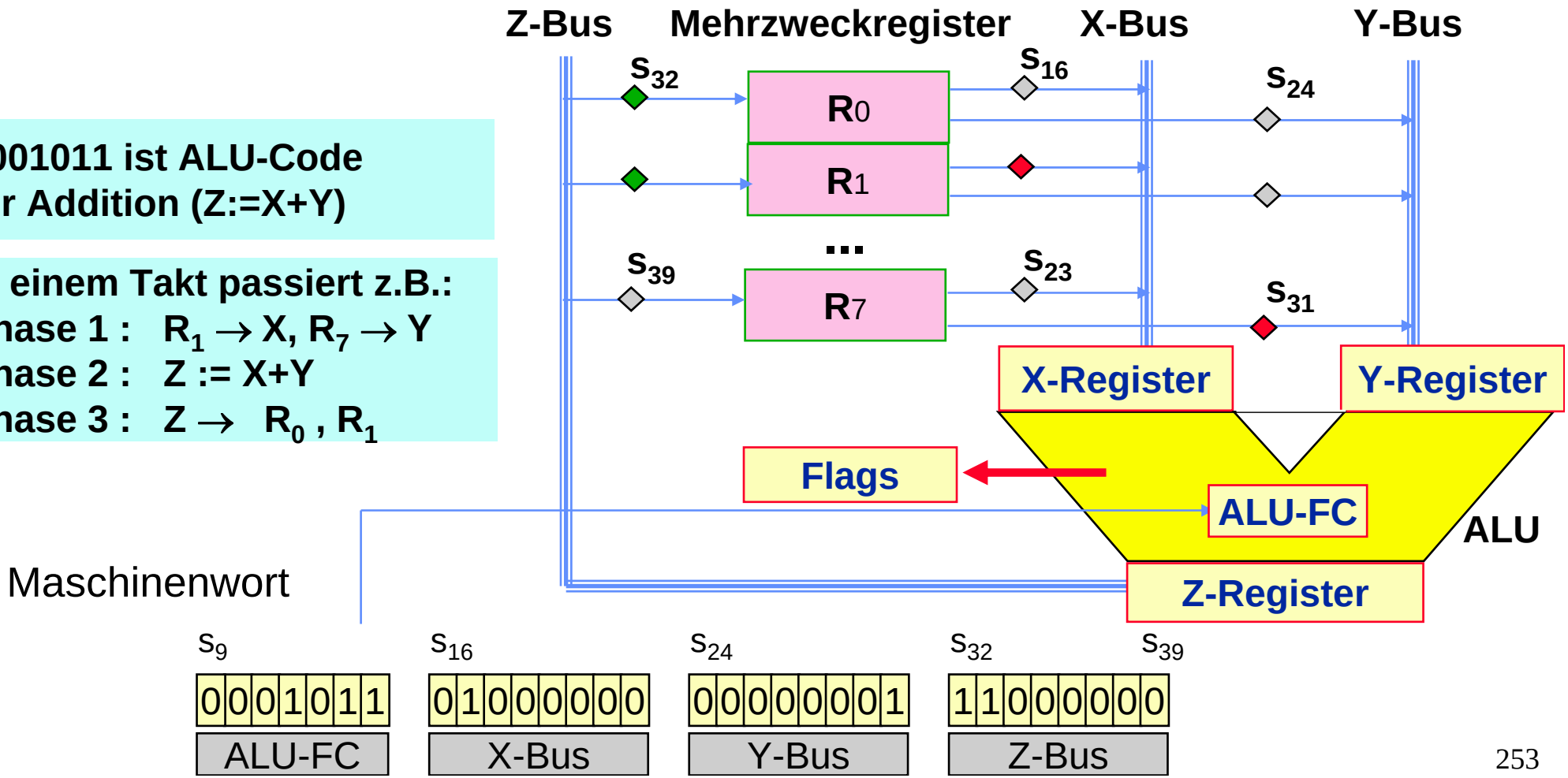
Z, Y := ALUFC - 48

Einbindung der ALU

Die Operation der ALU wird in ALUFC eingestellt. In Phase 2 berechnet die ALU aus den Inputs in X und Y ein Ergebnis, das dann in Z ausgegeben wird. ALU-FC wird ein Teil des Mikrobefehls.

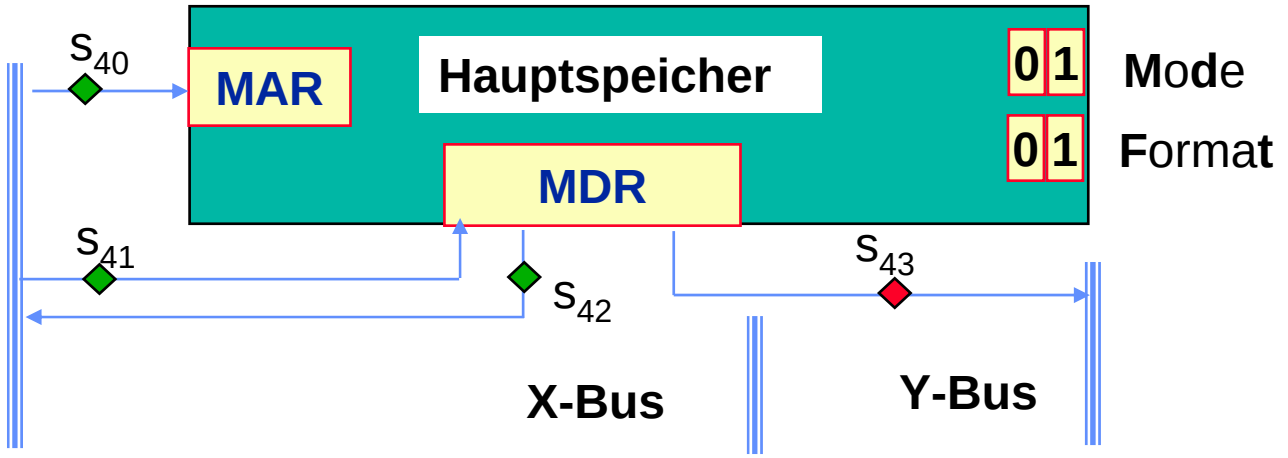
0001011 ist ALU-Code für Addition (Z:=X+Y)

In einem Takt passiert z.B.:
Phase 1 : $R_1 \rightarrow X$, $R_7 \rightarrow Y$
Phase 2 : Z := X+Y
Phase 3 : $Z \rightarrow R_0, R_1$



Der Hauptspeicher

Schalter	Aktive Phase
s_{43}	1
s_{40} ... s_{42}	3



Der Arbeitsmodus des Hauptspeichers wird in den Mode Schaltern eingestellt:

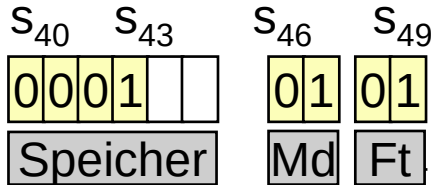
lesen = 01, schreiben = 10, warten sonst.

Die Format Schalter geben an, ob auf 1, 2 oder 4 Bytes zugegriffen wird.

In MAR (Memory Address Register) wird die Adresse eingestellt.

Über MDR (Memory Data Register) gelangen die Daten von bzw. zum Speicher.

Eine typische Leseoperation



CPU mit RAM

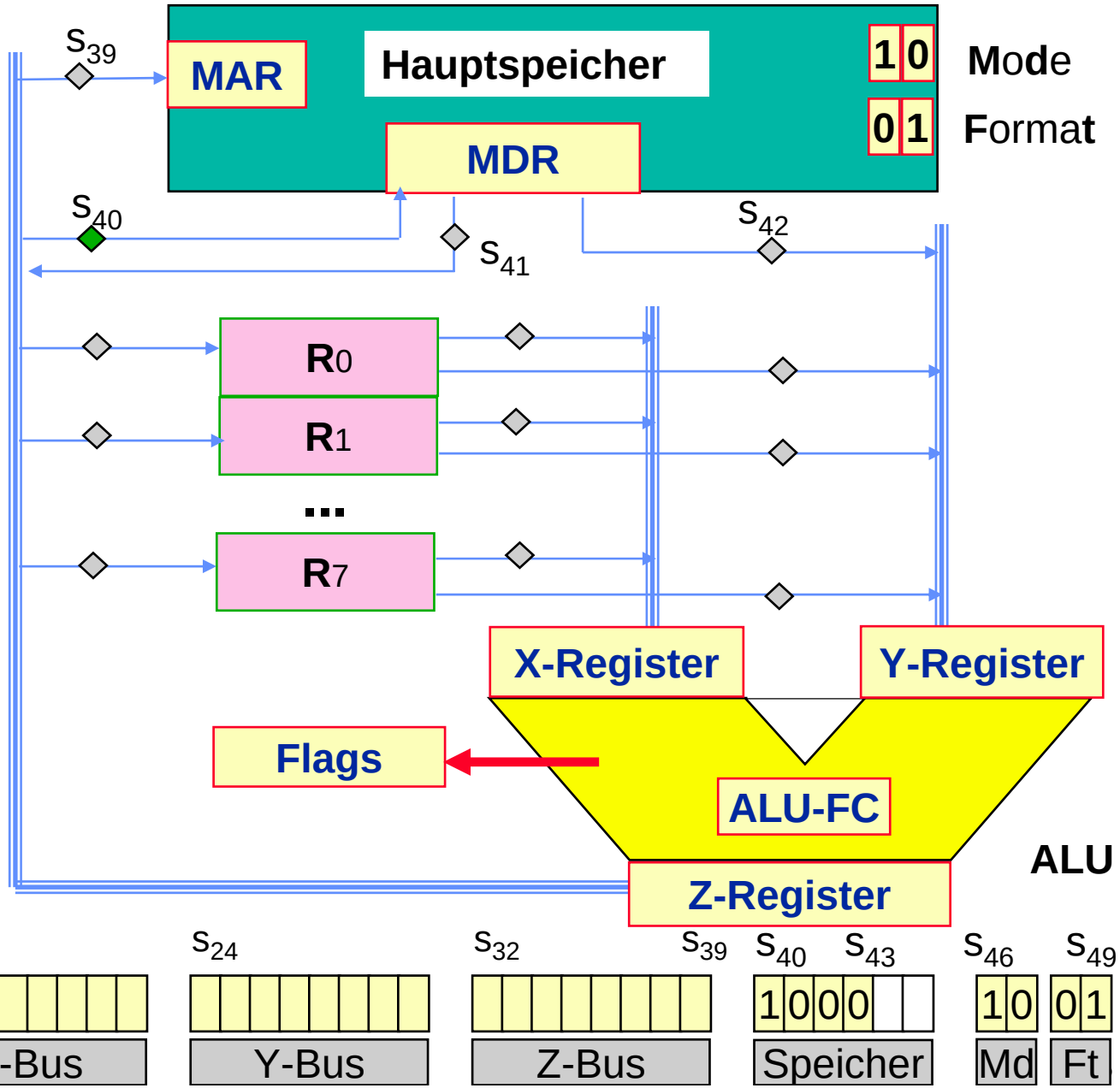
Zum Schreiben sind zwei Takte notwendig:

Phase 3 : Z → MDR

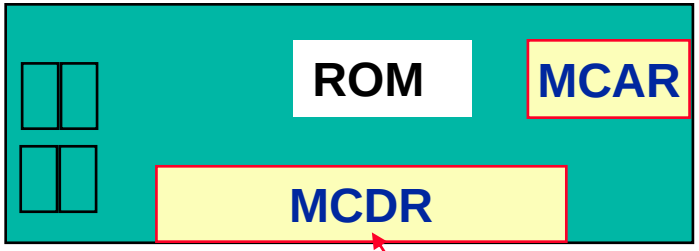
Phase 1 :

Phase 2 : Adresse berechnen

Phase 3 : Z → MAR

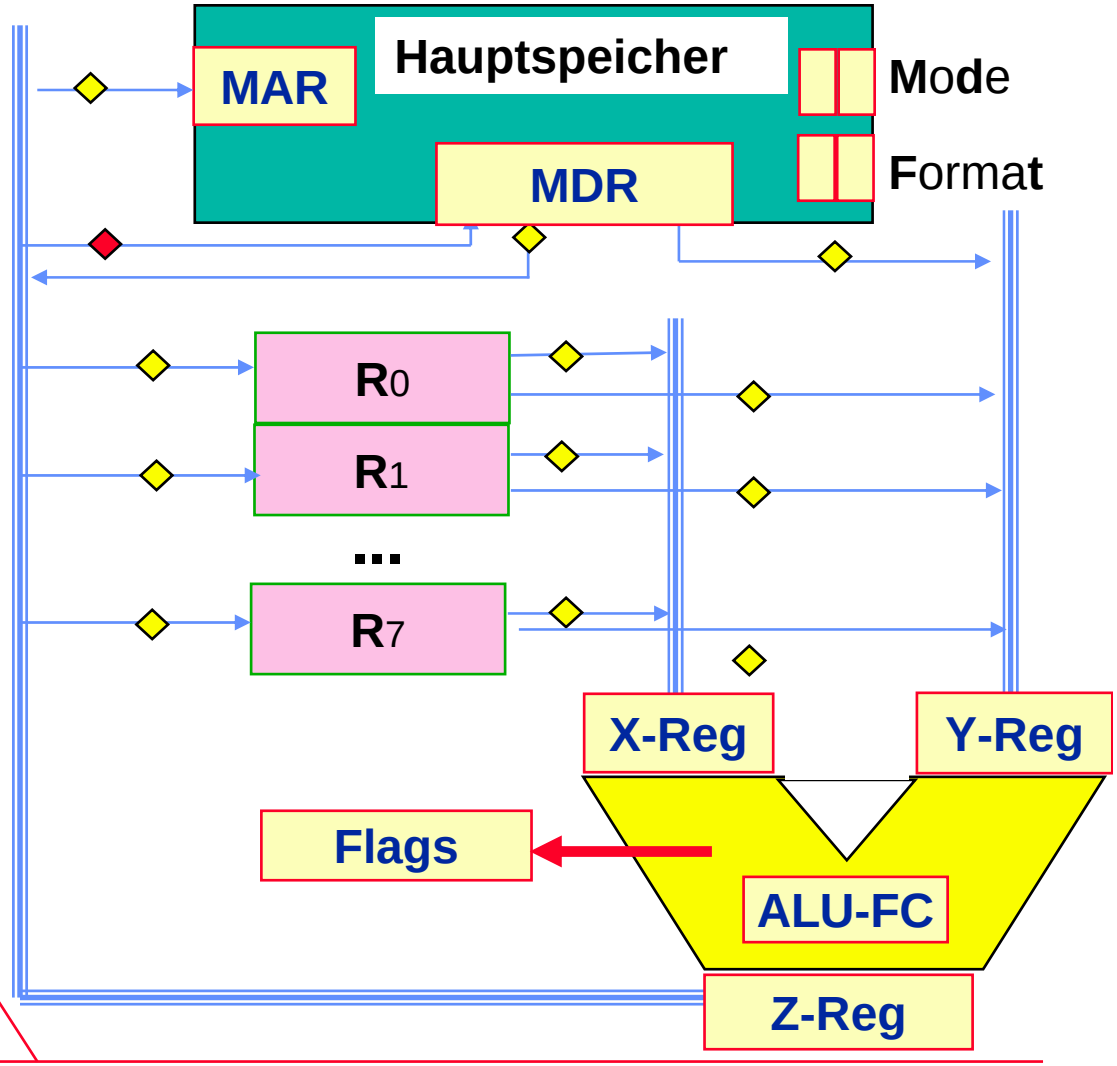


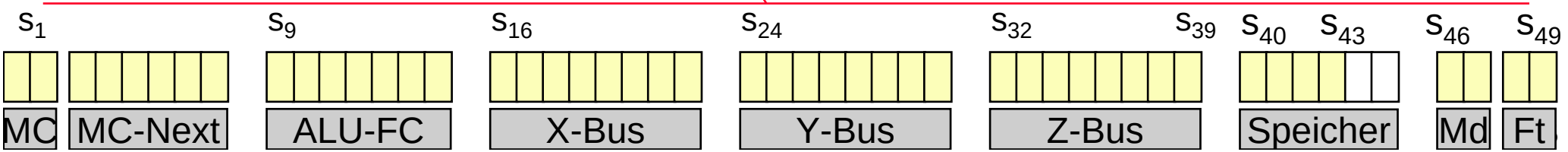
Das ROM



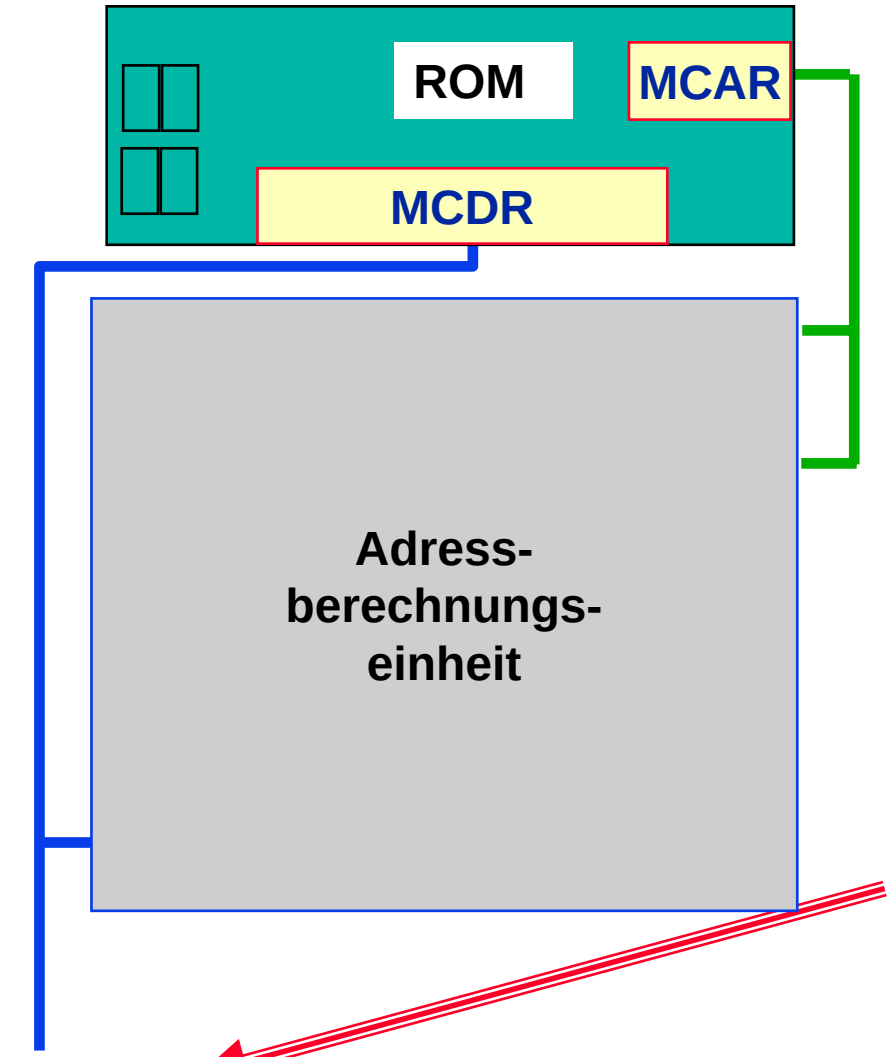
ROM = Read Only Memory
Ein Speicher, in dem die Mikrobefehle stehen.
MC = micro code

MCAR ist 10 Bit breit
1024 Adressen
MCDR ist 49 Bit breit.
ROM-Größe ca. 6 KByte
Es gilt immer :
Mode = lesen
Format = 49 Bit





Adressberechnung



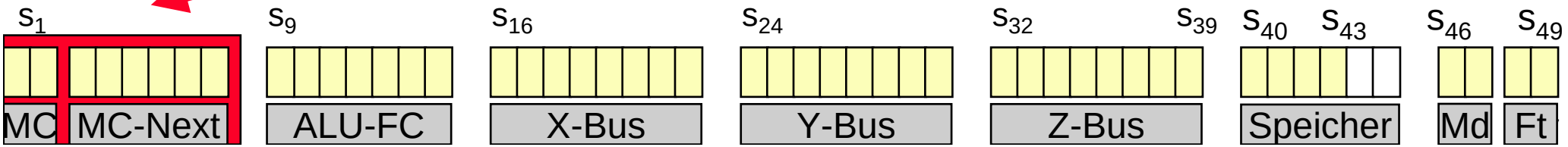
Während die CPU den gegenwärtigen Befehl noch bearbeitet, berechnet eine Adressberechnungseinheit daraus die Adresse des nächsten Befehls im ROM. Diese Adresse wird im MCAR abgelegt und der gefundene nächste Befehl liegt danach im MCDR vor.

Der erste Teil des Mikrobefehles (1 Byte) gibt an, *wie* der nächste Befehl zu ermitteln ist.

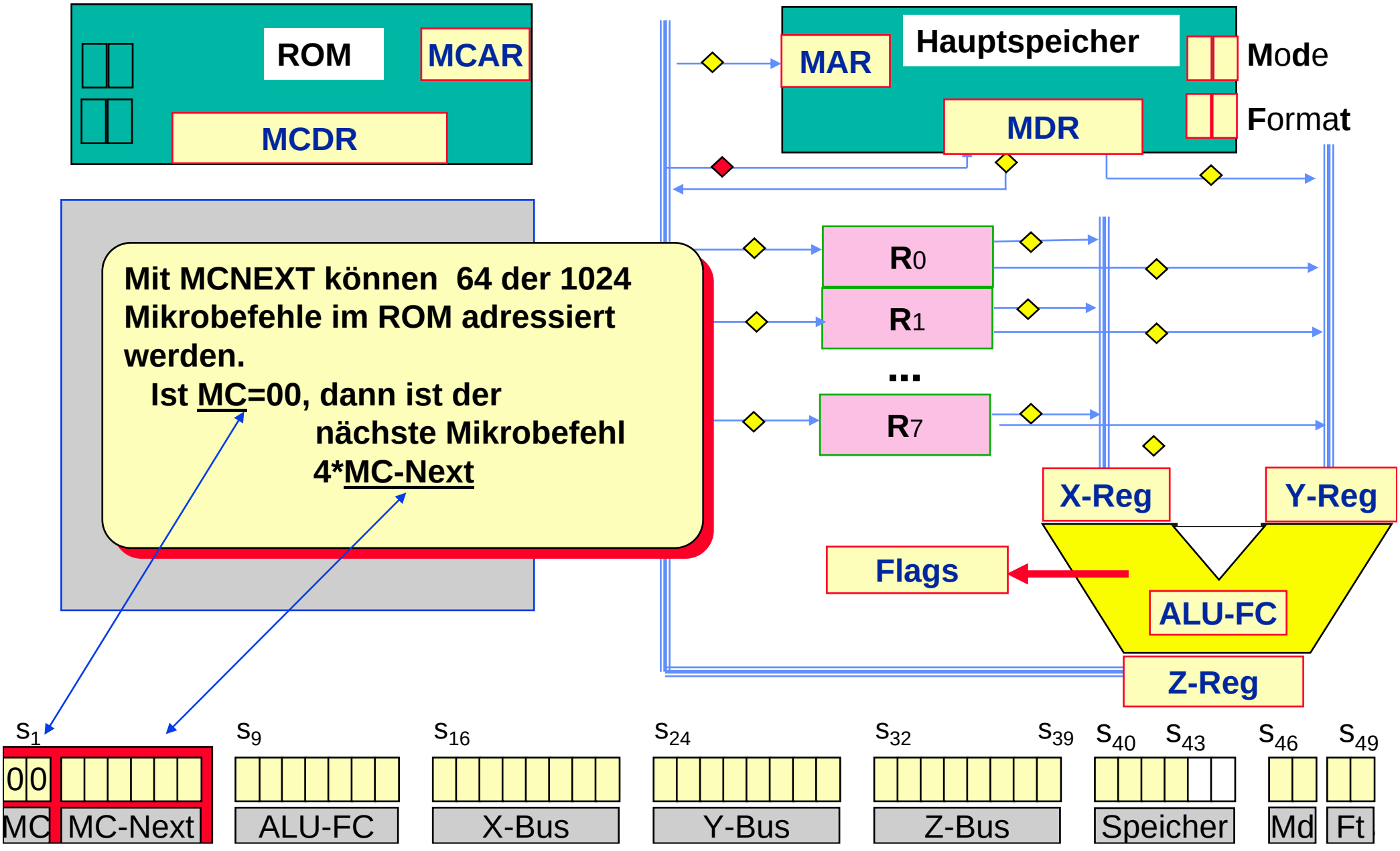
00 : MCAR := 4 * MC-Next

01 : MCAR := MCAR + 1 + 4*MCNext

10 : MCAR := MCAR + 1 - 4*MCNext

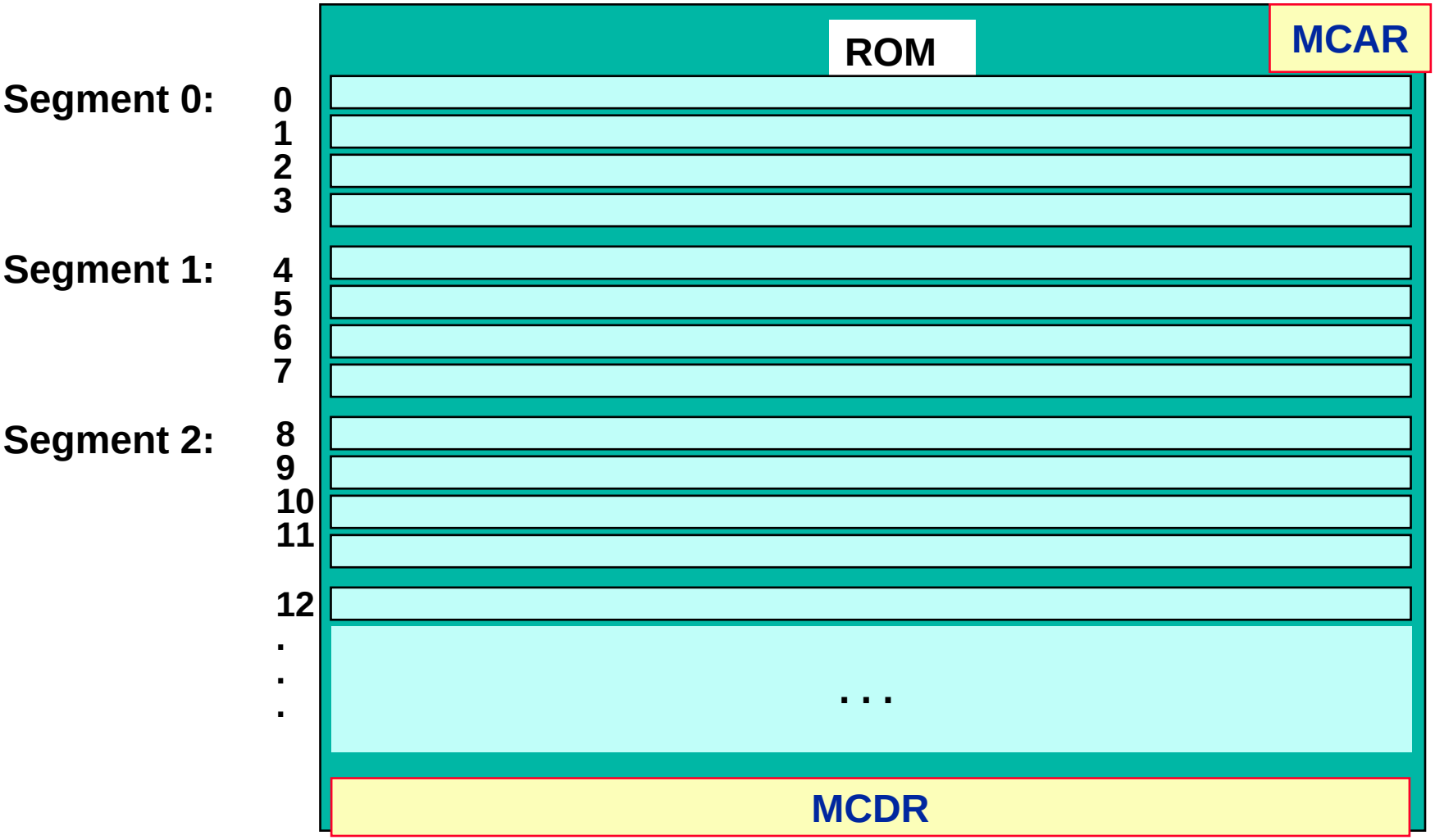


Absolute Adressierung



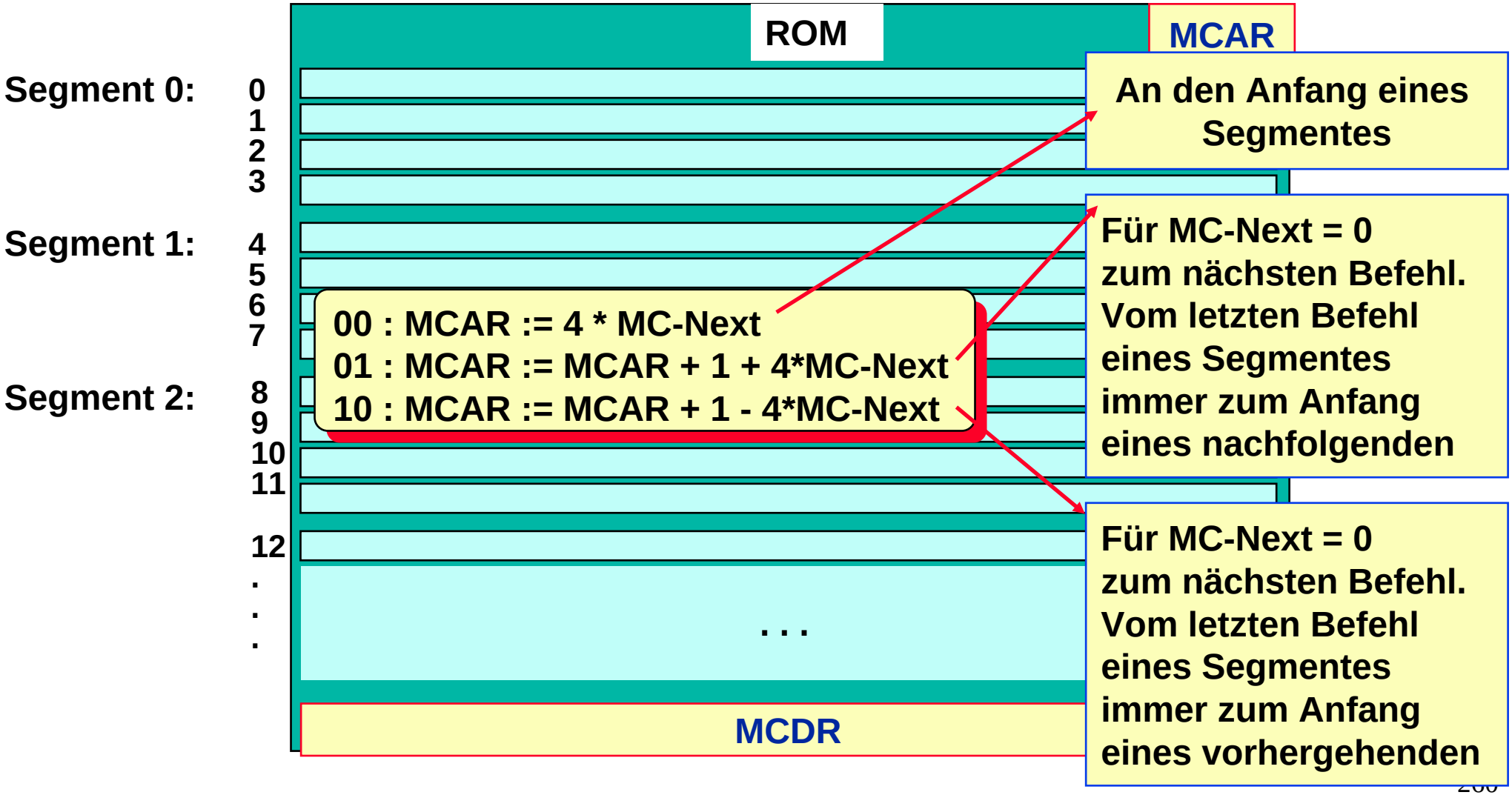
Segmente

Je 4 ROM-Adressen werden zu einem Segment zusammengefasst.
Absolute Sprünge sind nur an den Segmentanfang möglich.



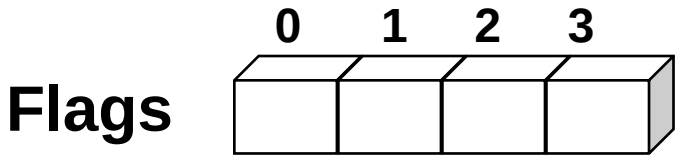
Einfache Sprünge

Für MC <> 11 wird die Adressberechnung folgendermaßen durchgeführt:



Das Flag-Register

Im Flagregister hat jedes einzelne Bit eine eigenständige Bedeutung. Viele Operationen oder Vergleichsoperationen zeigen das Vorliegen besonderer Eigenschaften durch das Setzen bestimmter Bits an.



Eine Vergleichsoperation

X < Y

setzt	falls X - Y
Bit 0 :	=0
Bit 1 :	>0
Bit 2 :	<0

Eine arithmetische Operation

Z := X ○ Y

setzt	falls Ergebnis
Bit 0 :	= 0
Bit 1 :	> 0
Bit 2 :	< 0
Bit 3 :	Overflow

Bedingte Sprünge

Bedingte Sprungbefehle beziehen sich immer nur auf den Inhalt der Flags. Mikrocode-Sprungbefehle besitzen eine “Maske”, d.h. ein Bitmuster.

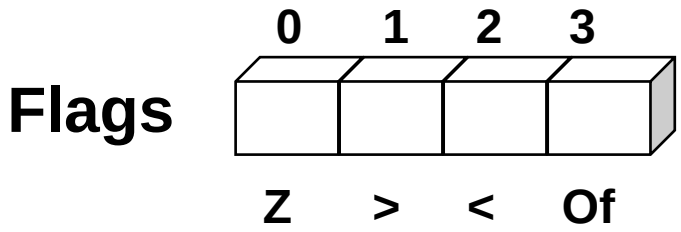
Diese Maske wird durch ein AND mit dem Flag-Register verknüpft.

Ist das Resultat <> 0000 , so wird der Sprung ausgeführt.

Ein Sprung mit der Maske

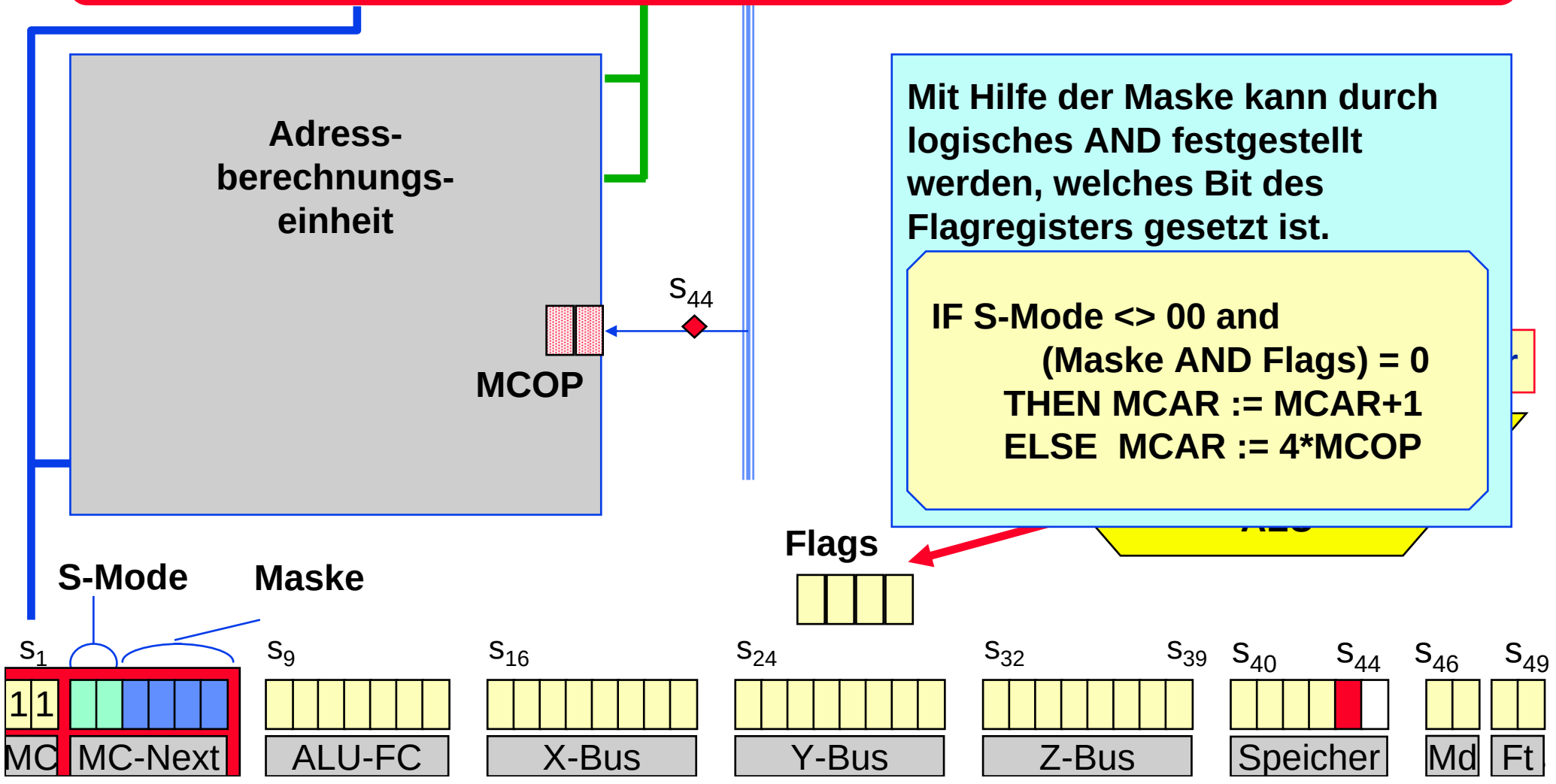
0 1 1 0

wird genau dann ausgeführt, wenn Bit 1 oder Bit 2 des Flag-Registers eine 1 enthält, also wenn das Ergebnis der letzten Operation <> 0 war, oder das Ergebnis des letzten Vergleiches “ungleich” ergab.



Berechnete Sprünge

Berechnete Sprünge sind solche, deren Sprungziel dynamisch berechnet werden kann. Dafür ist der Modus MC = 11 vorgesehen. Ein Register MCOP kann dazu über den Z-Bus geladen werden. Der Schalter s_{44} öffnet diesen Datenweg. MCNext wird aufgespalten in S-Mode und Maske.

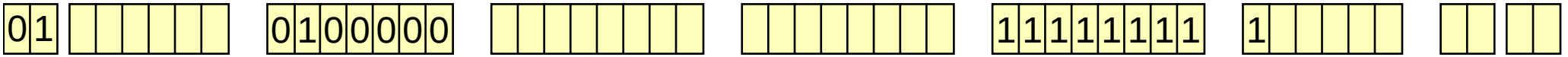


Ein Mikroprogramm

Das folgende Mikroprogramm berechnet die Summe aller Zahlen von 1 bis N, wobei N anfangs als 1 Byte Größe an der Adresse 00h liegt.
Das Ergebnis soll in 01h gespeichert werden.

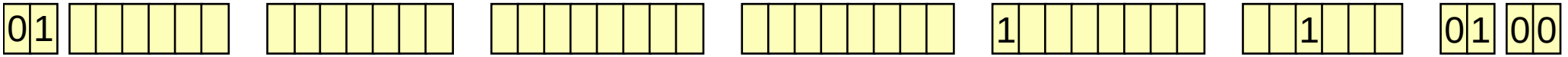
Initialisieren

P 2: Z := 0
P 3: Z --Z-Bus -> R_0 - R_7, MAR



MOV R_0 , [00h]

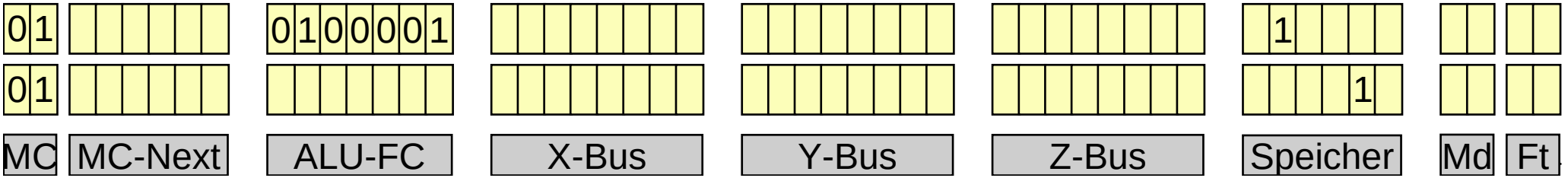
P 1: Mode = 01 (Read)
P 3: MDR --Z-Bus -> R_0



MCOP := 1

P 2: Z := 1
P 3: Z --> MDR

P 1: MDR --> MCOP



Ein Mikroprogramm (Fortsetzung)

Add R_2, R_0

P 1: R_0-- X-Bus--> X, R_2 --Y-Bus-> Y
P 2: Z := R_0+R_2
P 3: Z --Z-Bus-> R_2

DEC R_0
JNZ 4

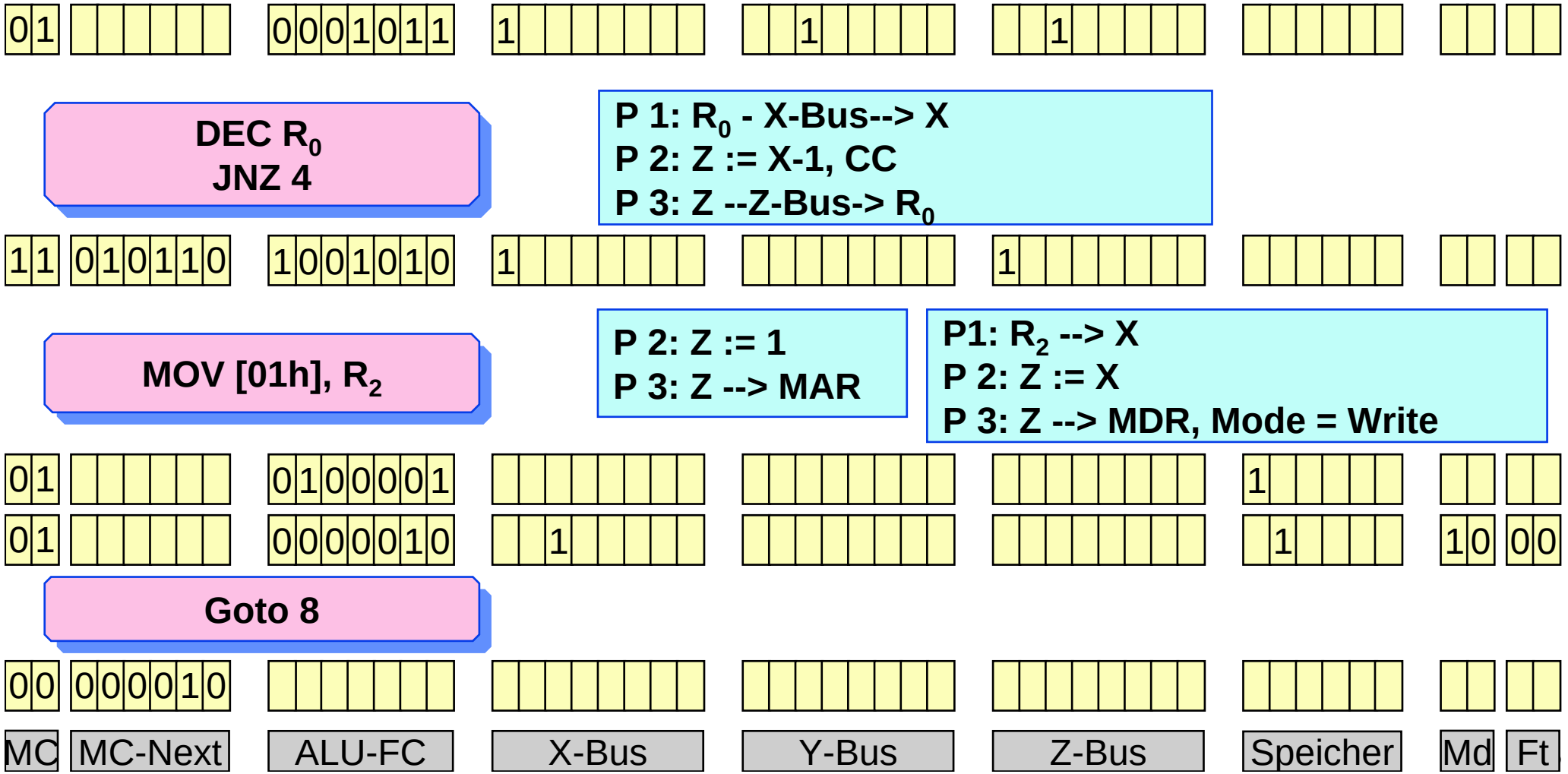
P 1: R_0 - X-Bus--> X
P 2: Z := X-1, CC
P 3: Z --Z-Bus-> R_0

MOV [01h], R_2

P 2: Z := 1
P 3: Z --> MAR

P1: R_2 --> X
P 2: Z := X
P 3: Z --> MDR, Mode = Write

Goto 8



Maschinensprache

Mikrocode-Programmierung ist für praktische Zwecke zu umständlich. Eine abstraktere Sicht der CPU zeigt nur noch Register, Speicher, Operationen, und Sprungbefehle, verbirgt aber Schalter, Takte und Phasen, Sprungmodi und Masken.

Ein typischer Maschinenbefehl hat die Form :

$Op\ R_1, R_2$

dabei ist Op eine Operation, R1 und R2 sind Register. Die Bedeutung ist :

$R_1 := R_1\ Op\ R2$

Beispiele :

```
Add  R1, R2
Mov  R1, R2
And  R1, R2
```

Bedingte Sprünge bestehen aus einer Vergleichsoperation mit anschließender Sprunganweisung, die auf dem Ergebnis des Vergleiches basiert

$Op\ R_1, R_2$
CondJmp Ziel

Jede Operation, die die Flags der ALU beeinflusst, kann als Vergleichsoperation dienen. Die Sprungbedingung ergibt sich aus den Flags.

Beispiel :

```
Sub R1, R2
JLE  127
```

Aufgaben der Register

Die Operanden von Maschinenbefehlen sind Register oder Speicherplätze. Es dürfen aber nie beide Operanden Speicherplätze sein.
Die Register werden für bestimmte Zwecke reserviert, etwa als Programmzähler, als Zeiger auf den Stack oder den Datenbereich.
Einige Register behält man als Rechenregister. Diese heißen Allzweckregister oder Akkumulatoren. Die Maschinensprache-Befehle sind meist nur mit Allzweckregistern durchführbar.

R_0	Prog. Counter PC
R_1	Akkumulator A
R_2	Akkumulator B
R_3	Loop Index I
R_4	Hilfsregister Aux
R_5	I/O Port
R_6	Data Pointer
R_7	Stack Pointer

Erlaubt :

```
Mov A, B
Add A, [021h]
AND B,A
```

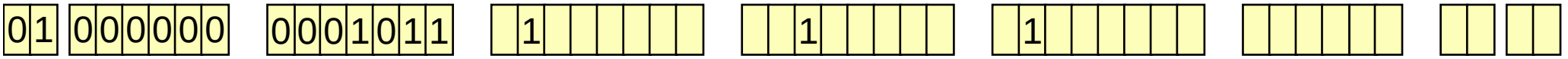
Nicht erlaubt :

```
Add PC, A
Mov I,  [0FFh]
```

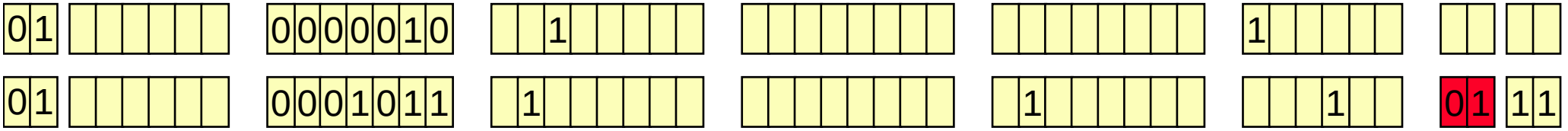
Mikroprogrammierte Maschinenbefehle

Jeden Maschinensprachebefehl können wir durch ein kurzes Mikroprogramm implementieren.

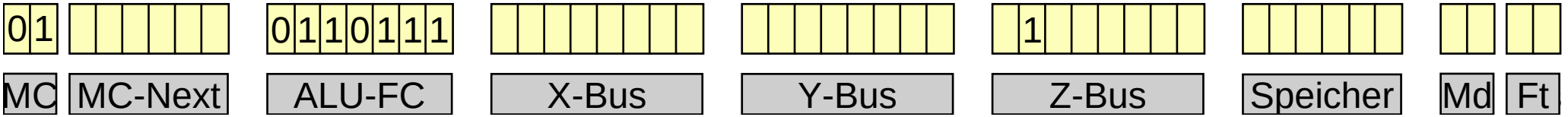
ADD A, B



ADD A, [B]



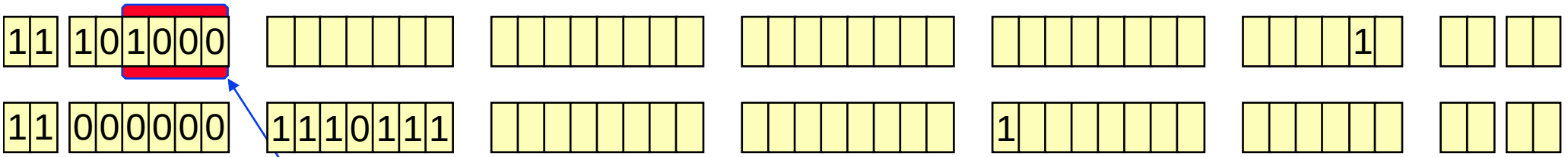
Mov A, 7



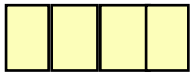
Sprünge

Falls das Ergebnis der vorigen Operation = 0,
führe den nächsten Maschinenbefehl aus.
MCOP sei entsprechend gesetzt.
Sonst setze PC (R0) auf 7 (Sprung!).

JNZ 7

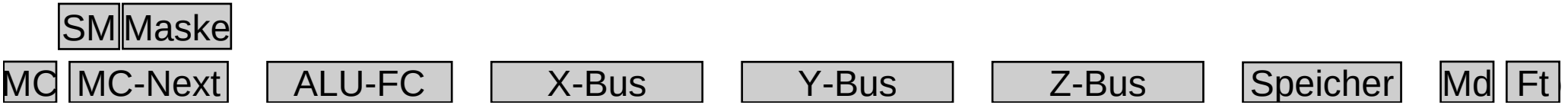


Flags



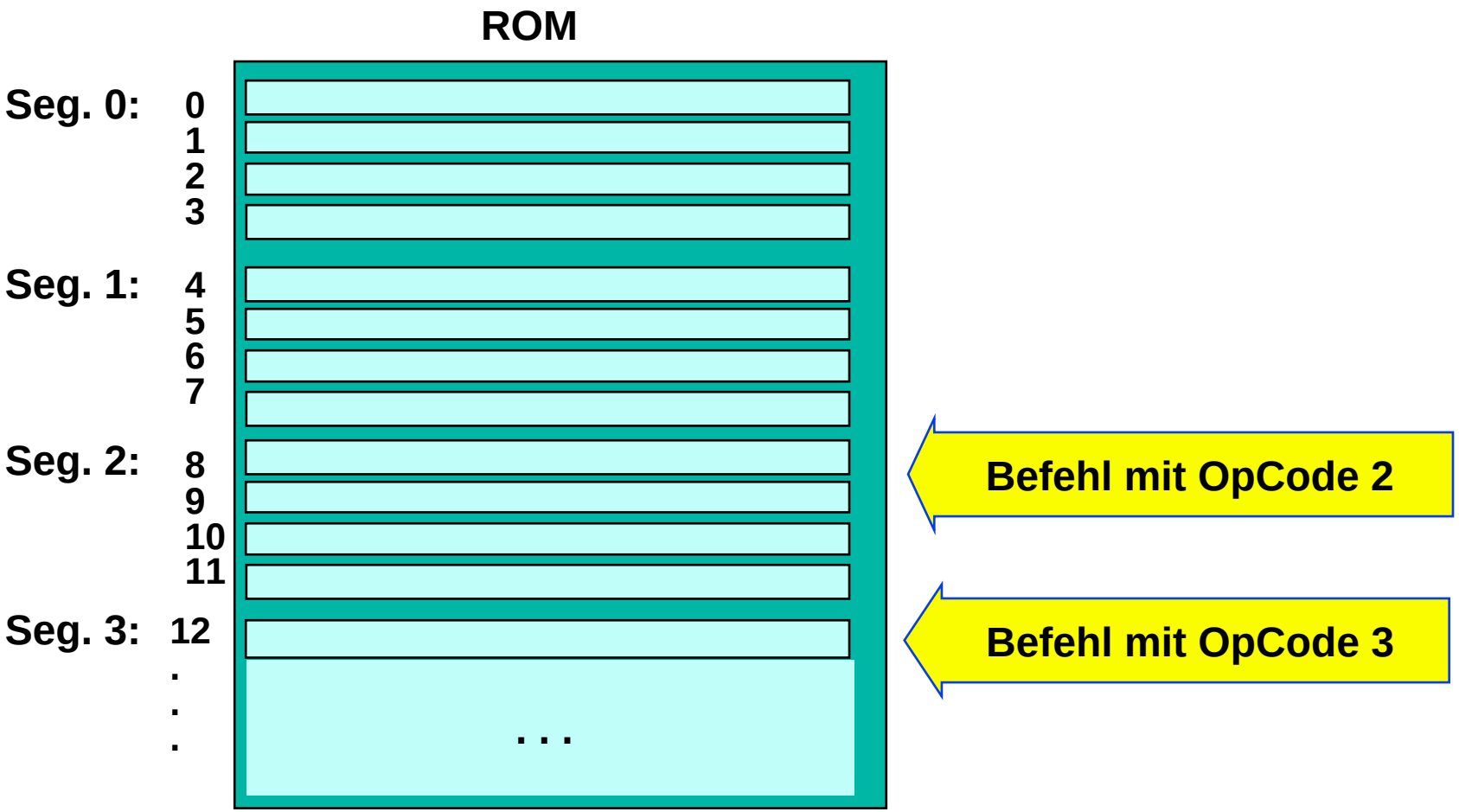
Z > < Of

1000 AND Flags <> 0
<==>
Z = 1



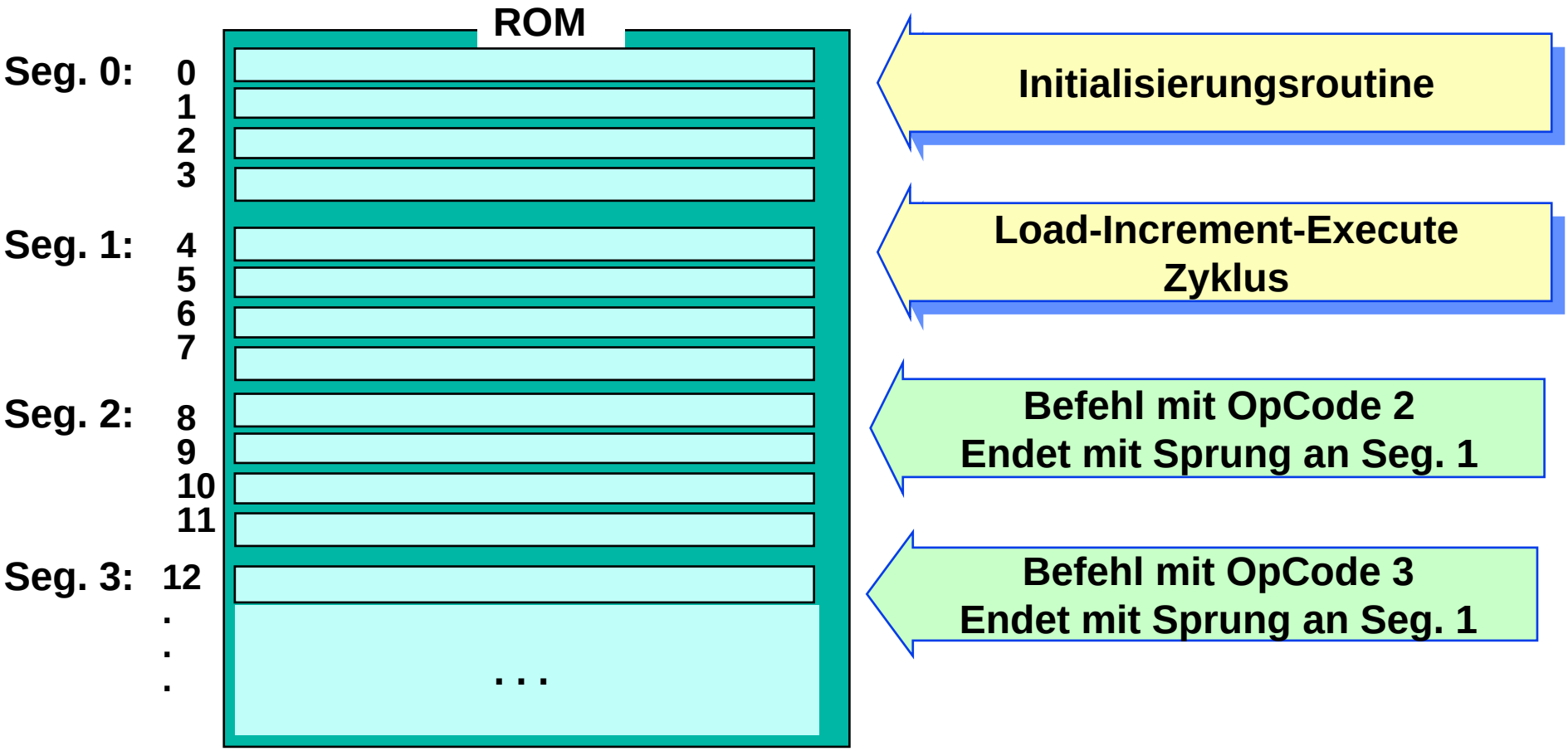
Der Maschinensprache-Interpreter

Jeder Maschinensprachebefehl erhält eine Nummer, den OpCode. Er wird als kurze Mikrocoderoutine im ROM abgelegt. Die Routine beginnt immer am Anfang eines Segmentes. Der OpCode ist identisch mit der Segmentnummer.



Der Mikrocode im ROM

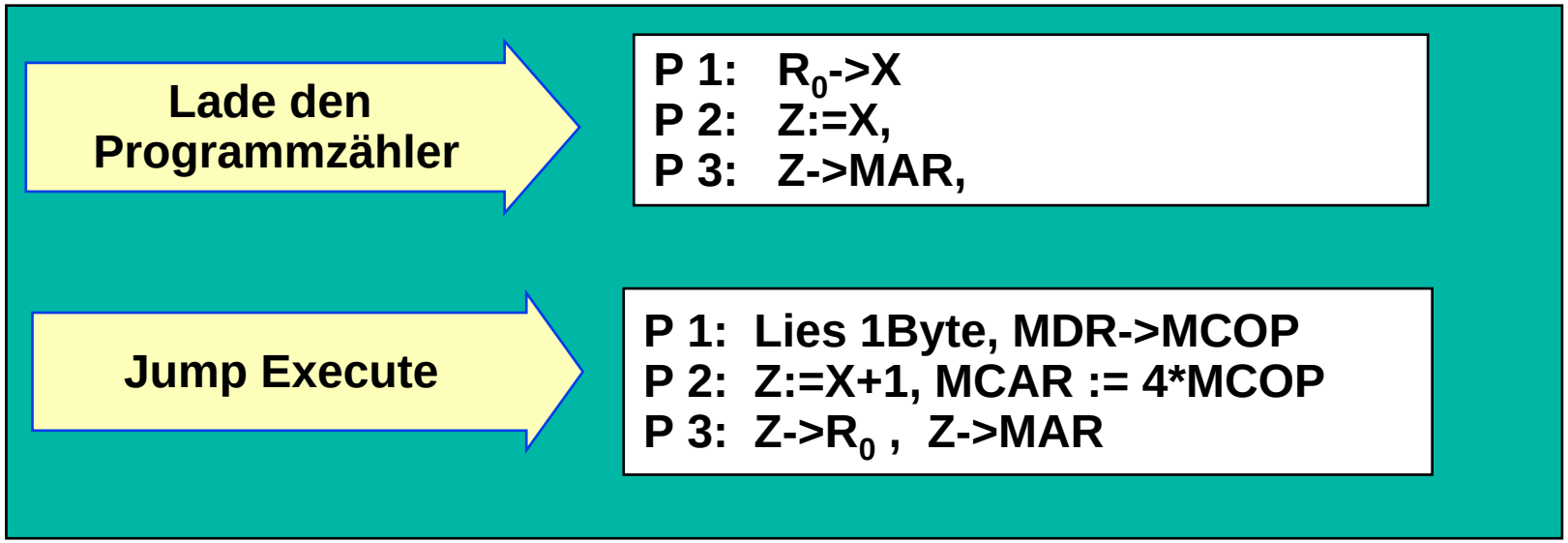
Segment 0 enthält Initialisierungsbefehle.
Segment 1 enthält einen Load-Increment-Execute-Zyklus. Dies ist ein Interpreter für Maschinensprache.
Jeder Maschinensprachebefehl endet mit einem Sprung an den Anfang von Segment 1.



Load-Increment-Execute-Zyklus

Der Load-Increment-Execute-Zyklus besteht aus 2 Mikrobefehlen.

ROM

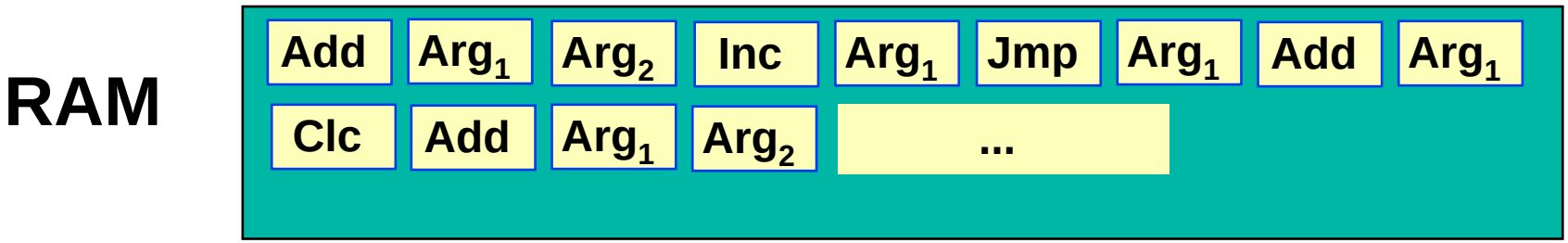


Jeder Opcode, der Argumente benötigt, kann davon ausgehen, daß MAR auf das erste benötigte Argument zeigt.

* **Die obigen Befehle dürfen keine Flags verändern. Entweder müssen also die Flags gerettet (PUSHF) und später wiederhergestellt werden (POPF) oder es sind die Varianten der Befehle Z:=X und Z:= X+1 zu benutzen, welche die Flags nicht beeinflussen.**

Maschinencode im RAM

Während das Programm im ROM fest vorgegeben und nicht veränderbar ist, kann im RAM ein Programm in Form von Maschinenbefehlen zusammen mit ihren Argumenten liegen.
Der Load-Increment-Execute-Zyklus führt dieses Programm aus.



Jeder Opcode, der Argumente benötigt, kann davon ausgehen, dass MAR auf das erste benötigte Argument zeigt.
Die Implementierung jedes Opcodes muss garantieren, dass hinterher PC auf dem Opcode des nächsten Befehls steht.

Entwicklung eines Maschinenspracheprogrammes

1. **Pascal-Program**

```
{ Gauss'sche Summe:
  Berechne die Summe
  aller Zahlen von 1 bis N }

VAR N,sum : Integer;

BEGIN
  sum := 0 ;
  WHILE n > 0 DO
     Begin
       sum := sum+n ;
       n := n-1
     End
END
```

Entwicklung eines Maschinenspracheprogrammes

2. **Ersetze Programmstrukturen durch Sprünge (Linearisiere das Programm)**

```
VAR   n,sum : Integer;
LABEL loop,ende;

BEGIN
        sum := 0 ;
 loop: IF n = 0 Goto ende ;
        sum := sum+n ;
        n := n-1 ;
        Goto loop
 ende:
END
```

Entwicklung eines Maschinenspracheprogrammes

3. **Wähle Register oder Speicherplätze für die Variablen, benutze Assemblerbefehle**

```
; Registerbelegung:
;      sum --> A
;      n   --> B
;
       MOV A,0
 loop:CMP B,0
       JE ende
       ADD A,B
       DEC B
       JMP loop
 ende:STOP
```

Entwicklung eines Maschinenspracheprogrammes

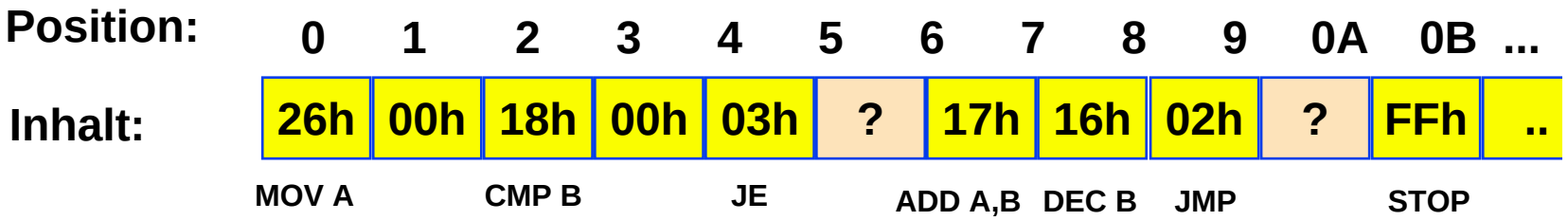
4. **Implementiere die benötigten Befehle im Simulator. Hier sind es:**

Befehl	Länge	OpCode*)
JMP <adr>	2	02h
JE <adr>	2	03h
DEC B	1	16h
ADD A,B	1	17h
CMP B,<num>	2	18h
MOV A,<num>	2	26h
STOP	1	FFh

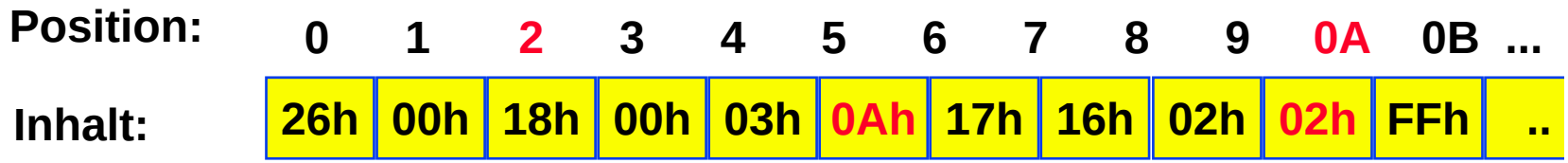
*) Der OpCode kann frei gewählt werden. Die obigen Zahlen sind aus der Beispieldatei `Summe.rom`.

Entwicklung eines Maschinenspracheprogrammes

5.a **Übertrage das Programm in das RAM. Im ersten Durchlauf bleiben Sprungadressen noch offen.**



5.b **Trage die richtigen Sprungadressen nach. Dies nennt man "Backpatching".**



Ausführung des Maschinenspracheprogrammes

1. Lade Datei in ROM
2. Implementiere "Load-Increment-Execute" Zyklus *)
3. Implementiere die benötigten Maschinenbefehle *)
4. Schreibe Maschinenprogramm in das RAM *)
5. Drücke "Reset"
6. Schreibe das gewünschte N in Register B (=R_2)
7. Führe das Programm aus, entweder
   - phasenweise
   - im Drei-Phasen-Takt
   - in L-I-E Schritten
   - vollständig (Break-Point-Ablauf)

*) Schritte 2.,3.,4. können gespart werden, wenn die fertige Datei "Summe" geladen wird.