Kapitel 1

Abwicklung sequentieller digitaler Prozesse

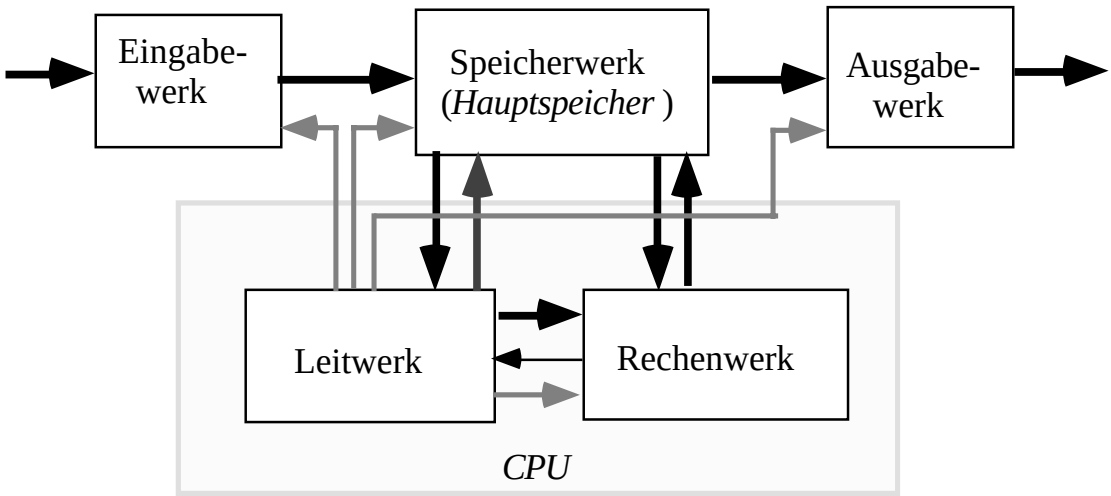
In diesem Kapitel steht die Abwicklung von Befehlen im Mittelpunkt der Überlegungen.

1.1. Von-Neumann und Wegener-Maschine

1.1.1. Von-Neumann-Maschine

Diese von J. von Neumann angegebene Architektur hat einen gemeinsamen Speicher für Programme und Daten (Princeton-Architektur).

Sie beschreibt die Mehrzahl aller Rechner.



Durchgezogene Linien beschreiben Daten- und Adressleitungen, schattierte Linien Steuerleitungen

Die Architektur ist gekennzeichnet durch folgende Eigenschaften:

a)*Es gibt fünf verschiedene Werke*

- ein Eingabewerk, akzeptiert Daten von außen (Kommandos und/oder Parameter)
- ein Ausgabewerk, gibt Daten auf geeigneten Ausgabemedien nach außen (Bildschirm, Drucker, auch Massenspeicher, ...)
- ein Speicherwerk für Daten und Programme (der Hauptspeicher des Rechners)
- ein Leitwerk, das Daten aus dem Speicher als Befehle interpretiert und die Steuerleitungen für alle anderen Werke aktiviert und für das Herbeiholen des nächsten Befehls sorgt
- ein Rechenwerk, das Daten aus dem Speicher als Zahlen und/oder Zeichen interpretiert und miteinander verrechnet

Leitwerk und Rechenwerk werden als Zentraleinheit - central processing unit, CPU - bezeichnet.

b) *Die Struktur ist problemunabhängig.*
Die Anpassung geschieht durch das Programm (= Abfolge der Daten, die als Befehle interpretiert werden).
Im Gegensatz dazu stehen Analogrechner und Spezialrechner, die in ihrer Struktur an eine Problemklasse angepaßt sind.

c) *Programme und Daten stehen in einem Speicher und sind dort ununterscheidbar*.
Das erlaubt das Rechnen mit Programmen: für einen Compiler ist der erzeugte Code die Ausgangsdaten seines Programms. Später werden sie als Programm interpretiert. Ein anderes Beispiel ist selbstmodifizierender Code.

Im Gegensatz dazu stehen
- tagged code: Bei jedem Datum im Speicher wird angegeben, ob es sich um einen Befehl, ein Datum und was für einen Typ handelt.
- Harvard-Architektur mit getrennten Speichern für Programme und Daten

d) *Jeder Speicherplatz hat seine eigene Adresse*
- bitparalleler, wortserieller Zugriff
- Granularität 8 Bit = 1 Byte aus historischen Gründen
- Zugriff meist wortweise: 4 - 8 Byte

e) *Befehle stehen an aufeinanderfolgenden Adressen*
- kompakte Programme in Blöcken mit einfacher Befehlsfortschaltung
- Gegensatz: Threaded Code: in jedem Befehl steht die Folgeadresse so daß eine Befehlsliste entsteht
  Beispiele: LISP oder horizontale Mikroprogrammierung

f) *Es gibt bedingte und unbedingte Sprungbefehle.*
- Bedingte Sprungbefehle waren die Erfindung von J. von Neumann.
- Sie geben dem Rechner die Mächtigkeit der Turingmaschine.
- Aber: Es gibt noch keine Unterprogramm-Sprünge im ursprünglichen Architekturentwurf.

g) *Es werden Dualzahlen verwendet.*
- Das hat technologische Gründe, da sich Speicher mit zwei verschiedenen Zuständen technisch leicht realisieren lassen (Flip-Flops, Magnetisierung in einem magnetischen Material, ...); bei Flash ROM´s benutzt man z:B. Mehrbitcodierung.
- Man nimmt den Rundungsfehler in Kauf beim Umrechnen vom Dezimalsystem ins Dualsystem bei Brüchen.

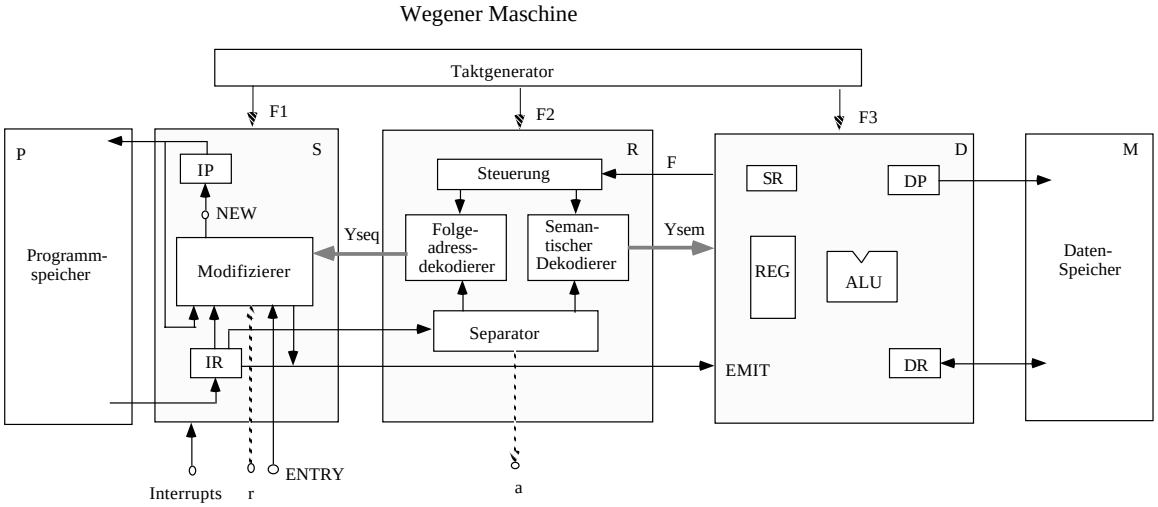
$$1/10 = 0.10_{10} < \quad 102/1024 = 0.000\ 11\ 00\ 11\ 00_2$$
$$> \quad 205/2048 = 0.000\ 11\ 00\ 11\ 01_2$$

Die von-Neumann-Architektur ist die einer universellen Maschine. Sie kann jedes Programm verarbeiten und hat die Mächtigkeit der Turing-Maschine.

Flaschenhals ist jedoch der Zugriff auf das Speicherwerk bei jedem Befehl (sog. von-Neumann bottle-neck).

1.1.2. Wegener-Maschine

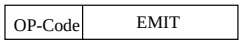
Das zweite Modell einer Maschine geht zurück auf P. Wegener: Es ist eine Harvard-Architektur mit getrenntem Programm- und Datenspeicher und soll hier zur Erklärung der Abarbeitung eines Befehls dienen.



P : Programmspeicher
S : Abwickler (engl. sequencer)
R : Entschlüssler (engl. recognizer)
D : Datenmanipulator oder Rechenwerk
M : Datenspeicher (engl. memory)

r : Anforderung (engl, request) oder Reset
a : Quittung (engl. acknowledge)
F : Bedingungen (engl. feedback) aus dem Datenmanipulator
Yseq : Steuerleitungen des Sequencers
Ysem : Steuerleitungen des semantischen Teils eines Befehls

IP : Befehlszeiger (engl. insruction pointer) oder Programmzähler (engl program counter, PC)
IR : Befehlsregister (engl. instruction register)
DP : (Daten)Adressregister (engl. data pointer)
DR : Datenregister
SR : Statusregister
REG . Registerfile
ALU : arithmetisch- logische Einheit



Befehlsformat

Der Programmspeicher wird adressiert mit dem Inhalt des Befehlszählers (instruction pointer, IP oder program counter, PC).

Der Befehl wird in ein Instruktionsregister geladen (instruction register, IR) und dort zwischengespeichert. IP und IR sind Register im Abwickler (sequencer), dem Schaltwerk ,

zuständig für die Befehlsfortschaltung. Die Berechnung der Folgeadresse NEW, die mit dem Takt Φ_1 in den Befehlszähler übernommen wird, geschieht im Modifizierer.

Aus dem Instruktionsregister wird ein Befehl im Entschlüssler (recognizer, R) umgesetzt in die Steuersignale Y_{seq} zum Ansteuern des Modifizierers und Y_{sem} für das Rechenwerk (data manipulator, D).

Im Rechenwerk wird die Verarbeitung der Daten durchgeführt. Sie werden in einer arithmetisch-logischen-Einheit (ALU) miteinander verknüpft.

Quellen und Senke können sein

- Register im Rechenwerk (mindestens 1, heute 32 und mehr)
- Datenspeicher (data memory, M)
  angesprochen mit einem Speicheradressregister (data pointer, DP)
  und Daten zwischengespeichert in einem Datenregister (data register, DR)
- direkt im Befehl übergebene Daten (immediate, EMIT) als Quelle von Daten.

Aus dem Rechenwerk laufen Rückmeldeleitungen (feedback, F) zum Entschlüssler zurück. Quelle ist die ALU, die den Ausgang der Rechnung in einem Statusregister (flag register oder status register, SR) festhält.

Das Statusregister hält Informationen von verschiedener Gültigkeitsdauer fest

- die o. g. Ergebnisse eines Befehls zur Auswahl der Folgeadresse bei bedingten Sprungbefehlen, Gültigkeitsdauer: ein Befehl

  Als Beispiel sei der Vergleich des Ergebnisses mit Null genannt: $>, \geq, =, \leq, <, \neq$, wobei dieses in den Bits P (positive), Z (zero), N (negative) und OV (overflow) festgehalten wird.
- langfristige Informationen über Modi des Rechners (User / Supervisormode, Trace / non Trace, Interrupts zugelassen / nicht zugelassen, Interruptmaske)

Von außen wird die Maschine angesteuert durch eine Reihe von Signalen

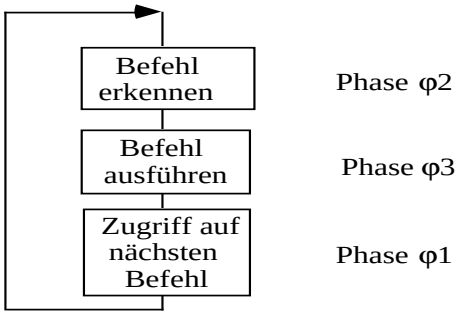
- Taktsignale Φ_1, Φ_2, Φ_3 für Abwickler, Entschlüssler und Rechenwerk aus einem Taktgenerator
- Rücksetzsignal (reset, r) zum Einstellen eines Startzustandes
- Anfangsadresse (ENTRY) für den Befehlszähler (häufig implizit als (0…0) angenommen)
- Interrupt-Signalleitungen zum Signalisieren eines externen asynchronen Ereignisses, auf das der Rechner reagieren können soll.

Nach außen hin gibt die Maschine ab

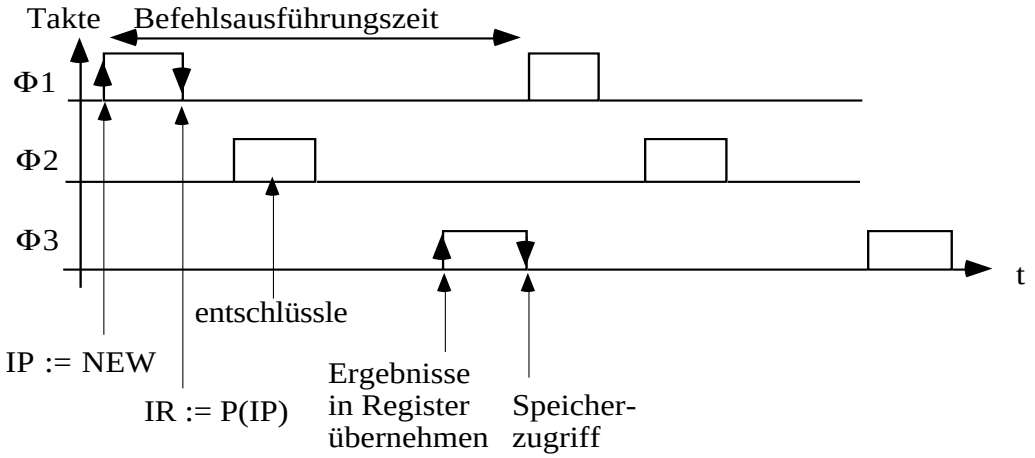
- Signale zum Ansteuern der Speicher (s. Synchronisation)
- eine Fertigmeldung (acknowledge, a) wenn ein HALT-Befehl erkannt wurde.

Das Abarbeiten eines Befehls geht in folgenden Schritten vor sich:

- **Weiterschalten zum nächsten Befehl und Programmspeicher adressieren**
- **Befehl holen**
- **Befehl entschlüsseln**
- **Daten manipulieren und Ergebnis in Register und Flags rückschreiben**



Die zeitliche Abarbeitung wird durch die Takte Φ_1, Φ_2 und Φ_3 gesteuert.



Dabei können $Ø_1$ und $Ø_3$ in mehrere Takte aufgeteilt werden:

Φ_{1a} : Lade NEW nach IP : IP := NEW

Φ_{1b} : Speichere P (IP) in IR ab : IR := P (IP)

Φ_{1b} kann fortfallen, wenn der Programmspeicher als ROM ausgeführt ist, denn dann bleibt P (IP) am Ausgang erhalten solange sich IP nicht ändert.

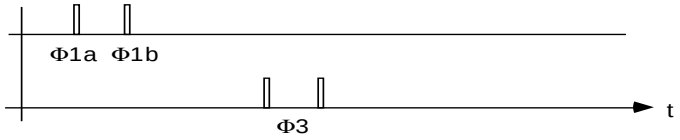
Φ_3 kann aufgeteilt werden in

Φ_{3a} : Abspeichern der Ergebnisse in den Registern
und/oder Laden des Datenspeicheradressregisters DP

Φ_{3b} : Übernahme von Daten in das Datenregister DR.

Die eigendliche Befehlsausführung liegt zeitlich zwischen den Takten Φ_2 und Φ_3

Der Takt $Ø_2$ kann fortfallen, wenn der Entschlüssler als Schaltnetz ausgeführt ist.



1.2. Eine primitive Maschine als Beispiel

Es soll eine Maschine vorgestellt werden, die einfach genug ist, um mit Mitteln aus der Digitalen Logik ohne Aufwand realisiert werden zu können.

Sie verarbeitet Programme einer Sprache

$L = \{ +, *, c, \Lambda \} \quad ; \quad c \in \{ 0, ..., 2^n - 1 \}$

mit 4 Befehlen: +: addiere, $*$: multipliziere, c: lade Konstante mit n-Bit Länge,
Λ: tue nichts.

Ein Befehl wird dargestellt durch n + 2 Bit:

opcode	Bedeutung	Emit
00	+	#
01	*	#
10	CONST	c
11	Λ	#

Die Maschine kennt keine Sprungbefehle.

Der Befehl Λ wird ausgeführt als

mit Φ_1 :

```
IP  := NEW
NEW .= if  Λ  then if  r  then  ENTRY
                               else  IP
                    else  (IP + 1)
```

Dabei beschreibt ":=" die Zuweisung an ein Register
".=" die Durchschaltung auf eine Leitung

Der Programmspeicher sei ein ROM (damit kann IR entfallen).

Das Programm ist eine Anweisungsfolge in Postfixnotation (auch UPN genannt, umgekehrte polnische Notation nach dem Mathematiker Lukaciewiez).

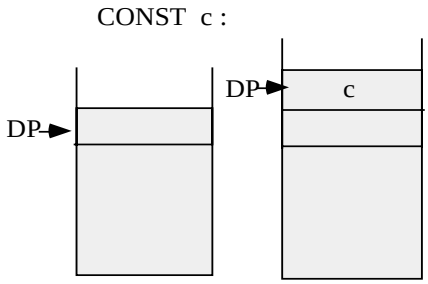
Beispiel: der Ausdruck $(3 * 4) + (5 * 6)$
wird geschrieben 3, 4, $*$, 5, 6, $*$, +, Λ und als Folge von Befehlen interpretiert.

Das legt die Abarbeitung über einen Stack nahe.

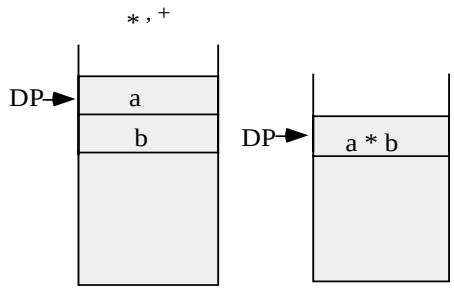
DP übernimmt hier die Rolle des Stackpointers und weist auf den obersten belegten Platz im Stack, realisiert im Speicher M.

Die folgende Abbildung zeigt die Realisierung der Operationen auf dem Stack.

Die Stackoperationen können seriell oder im Prinzip auch parallel ausgeführt werden. Wenn mit DP auch DP + 1 zur Verfügung steht, können der Zugriff auf den Speicher und die Zuweisung des inkrementierten Wertes an den Stackpointer parallel erfolgen:

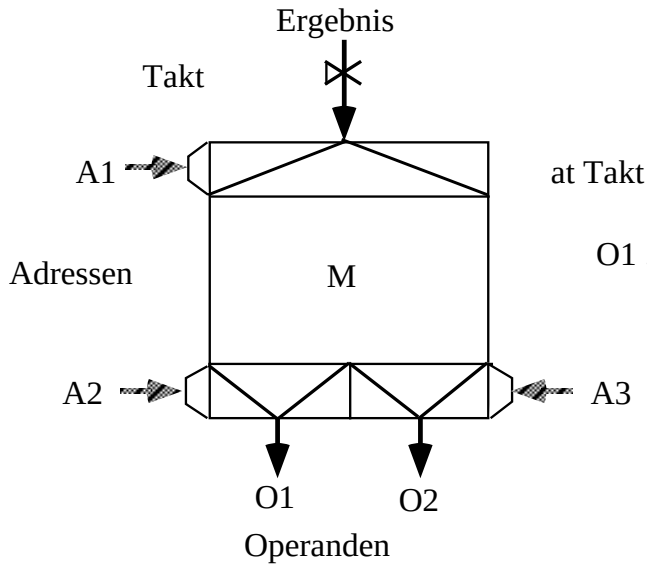



```
seriell :
   DP:= DP +1;
M(DP):= c
parallel :
M(DP + 1):= c ; DP:= DP + 1
```




```
seriell oder parallel :
M(DP - 1):= M(DP) {*,+} M(DP - 1);
      DP := DP - 1
```

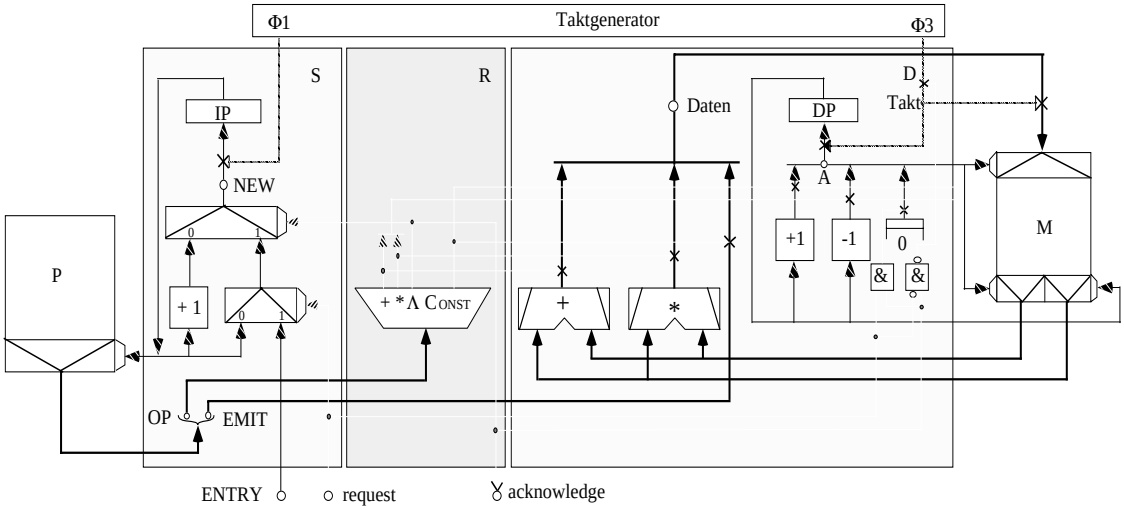
Hardwaremäßig könnte der Speicher als 3-Port-Speicher realisiert sein:



at Takt do M(A1) := Ergebnis

O1 .= M(A2) ; O2 .= M(A3)

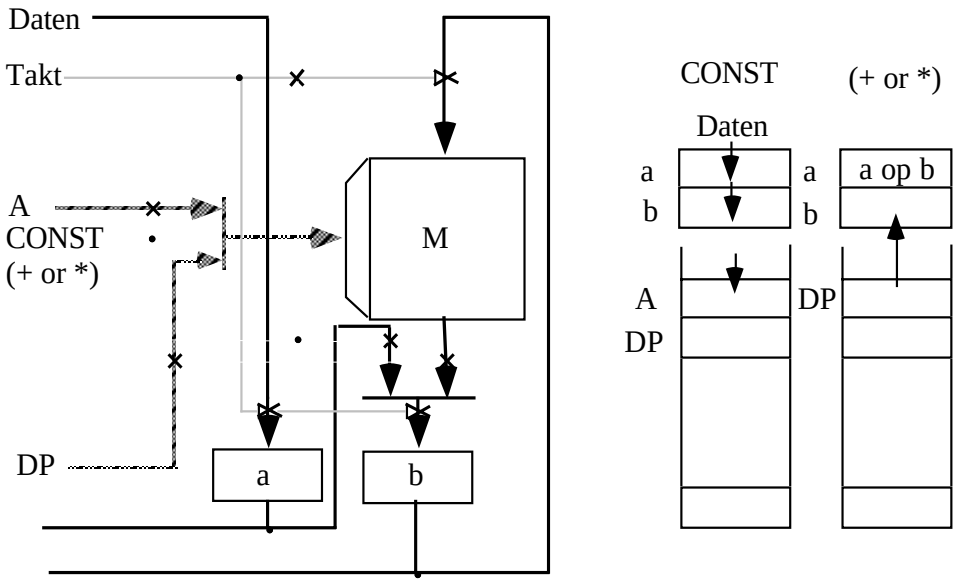
Die komplette Maschine zeigt das nächste Bild. Dabei ist
P der Programmspeicher, hier ein ROM, womit IR entfallen kann
M der Datenspeicher, IP und DP die beiden Register.



Primitive Maschine

```
OP.EMIT.= P(IP) ;
at F1 do IP:= NEW;
NEW.= if "L" then if r then ENTRY
                       else IP
             else IP + 1;

Daten.= if "+" then M(A) + M(DP)
                elseif "*" then M(A) * M(DP)
                elseif CONST then EMIT;

Takt .= if not ("L" & not r) then Φ3 ;
at not Takt do DP := A ;
at Takt do M(A) := Daten ;
A .= if ("+" or "* ")   then DP - 1
        elseif "CONST"  then DP + 1
        elseif ("L" & r)then 0 ;
```

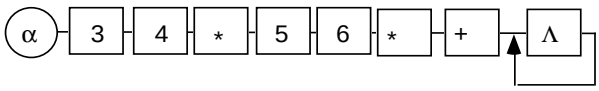
Der 3-Port-Speicher kann durch einen Stack ersetzt werden, dessen oberste Elemente zwei Register a und b sind.



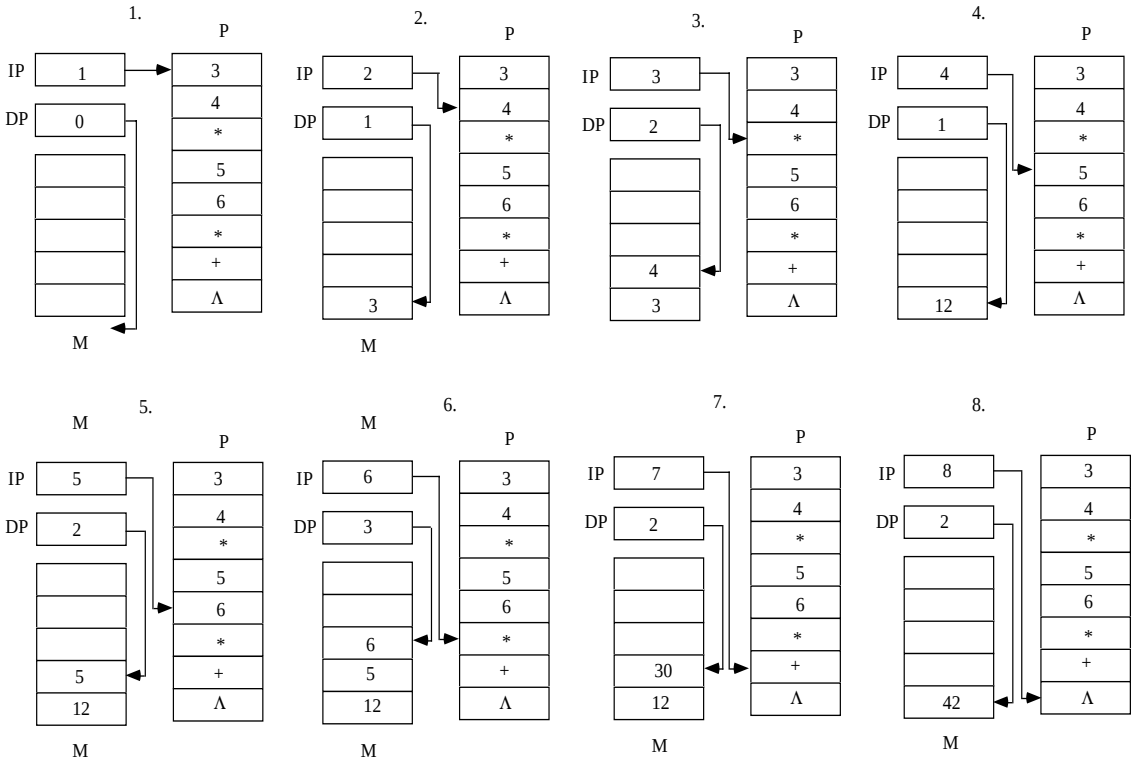

```
at Takt do
      a := Daten;
      if "CONST"    then   b := a;      M(A) := b
      if ("+"or"*") then   b := M(DP)
enddo
```

1.3. Zeitliche Struktur der Befehlsabarbeitung

Anhand eines Beispiels soll nun die Abarbeitung des Programms gezeigt werden.



Das Programm wird Daten so verändern:



Programmablauf in der Primitiven Maschine

Die Feinstruktur der Abarbeitung eines Befehls läuft in drei Phasen ab

Phase φ_1 : Lade IP mit NEW : IP := NEW
Adressiere P :
Hole P(IP) : IR := P(IP)

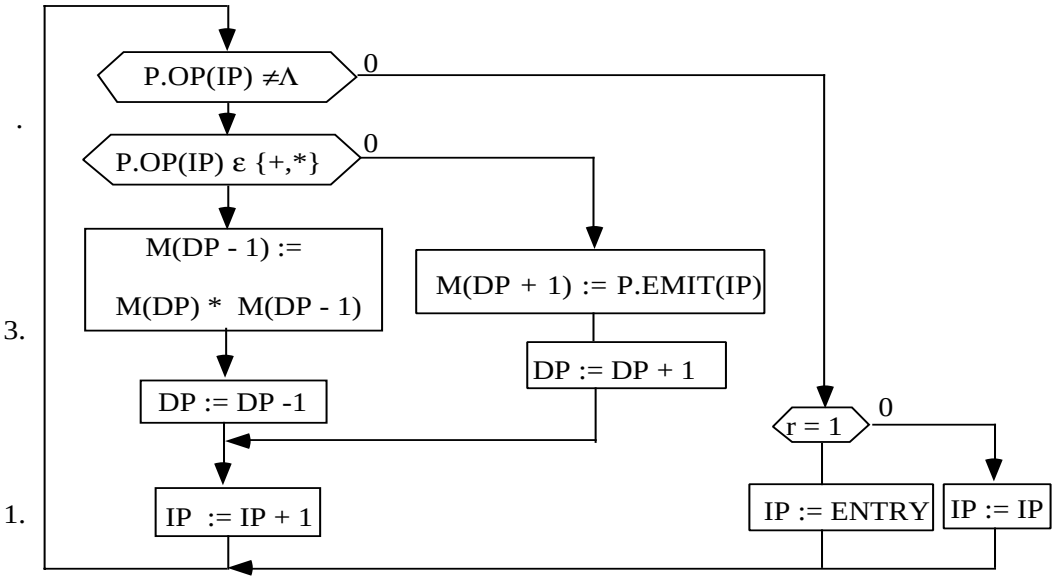
Die beiden Ladevorgänge können mit der Vorder- und Rückflanke eines Taktes Φ_1 durchgeführt werden oder durch zwei eigene Takte Φ_{1a} und Φ_{1b} .

Phase φ_2 : Entschlüssle den Befehl in einem Schaltnetz (dann braucht kein eigener Takt Φ_2 vorgesehen werden).

Phase φ_3 : Führe den Befehl aus und übernehme die Ergebnisse in die Register.

Dazu wird mindestens ein Takt Φ_3 gebraucht, häufig jedoch viele Takte, wenn die Ausführung ein Schaltwerk erfordert und man nicht mit einem Schaltnetz für die Datenverarbeitung auskommt.

Ein einzelner Befehl der Sprache wird nach folgendem Schema interpretiert:



1.4. Struktur eines Abwicklers

Der Abwickler (sequencer) ist der Teil der Maschine, der die Adresse des nächsten Befehls ermittelt.

Das kann sein:

- normale Befehlsfortschaltung
- Halt-Befehl
- asynchroner reset
- unbedingter Sprung
- bedingter Sprung
- Unterprogrammsprung
- Rücksprung aus dem Unterprogramm
- Interrupt

1.4.1. Normale Befehlsfortschaltung

Der neue Wert des Befehlszählers ist

IP := IR + c

mit einer Konstante c.

Bei byteweiser Adressierung und byteweisem Zugriff ist c = 1; bei byteweiser Adressierung und Wertzugriff mit 4 Byte ist c = 4 (das ist der Normalfall bei RISC-Rechnern).

1.4.2. Haltbefehl

Der neue Wert des Befehlszählers ist der alte, es sei denn ein "reset"-Signal liegt an:

IP := if r then ENTRY
else IP

1.4.3. Asynchroner Reset

Hier wird der Zeitpunkt von außen bestimmt:

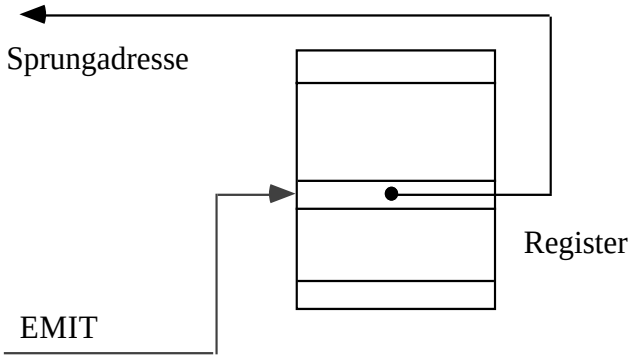
IP := if r then ENTRY else NEW

Mit dem Reset wird der bisherige Programmablauf unrestaurierbar unterbrochen.

z.B. als Notmaßnahme nach einem katastrophalen Fehler im Programm oder als Startbefehl

1.4.4. Unbedingte Sprünge

- IP := IP + 2 c
  Skip-Befehl; der nächste Befehl wird übersprungen;
- IP := EMIT
  absoluter Sprung; das Sprungziel wird direkt im Befehl angegeben.
  Diese Art von Sprungbefehlen bedingt, daß der EMIT-Teil des Befehls die volle Länge einer Adresse umfaßt (selten verwendet).
- IP := <EMIT>
  indirekter absoluter Sprung;
  EMIT weist auf die Adresse (häufig ein Register) in dem die Sprungadresse steht.
  Innerhalb des Befehls sind zwei Zugriffe nötig, wenn EMIT eine Speicheradresse bezeichnet, sonst braucht nur der Inhalt des in EMIT bezeichneten Registers nach IP kopiert zu werden (häufiger verwendet).
  Der mit absoluten Adressen des Befehlszählers erzeugte Code ist nicht verschiebbar, es sei denn, implizit wurde noch der Inhalt eines Segmentregisters (s. Adressierungen) addiert.



- IP := IP + EMIT
  relativer Sprung.
  EMIT braucht nicht die volle Länge der Adressen zu haben, da nur relativ adressiert wird.
- IP := IP + <EMIT>
  indirekter relativer Sprung.
  Meist in der Form, daß EMIT ein Register bezeichnet in dem die Sprungweite steht.

In beiden Fällen ist der erzeugte Code verschiebbar.

1.4.5. Bedingte Sprünge

Die nach IP geschriebene Folgeadresse hängt vom Wert eines Rückkopplungsbits F ab.

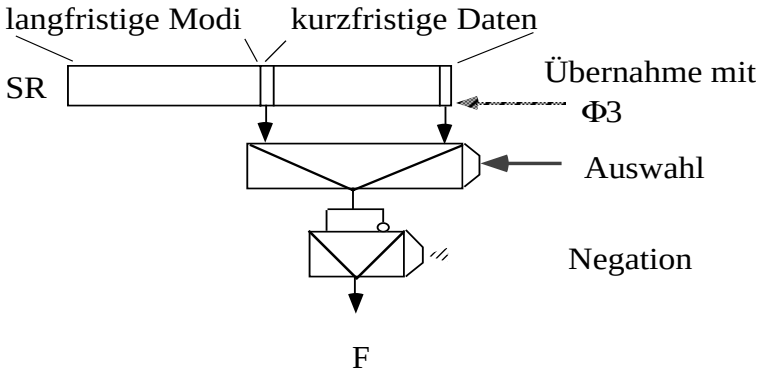
- IP := if F then (IP + 2 c) bedingter Skip
  else (IP + c)
- IP := if F then EMIT bedingter absoluter Sprung
  else (IP + c)
  Für EMIT gilt das gleiche wie bei dem absoluten Sprung.
- IP := if F then <EMIT> bedingter indirekter Sprung
  else (IP + c)
- IP := if F then (IP + EMIT) bedingter relativer Sprung
  else (IP + c)
- IP := if F then (IP + <EMIT>) bedingter indirekter relativer Sprung
  else (IP + c)

Die am häufigsten implementierten Typen sind bedingte absolute indirekte Sprünge über Register und bedingte relative Sprünge mit Sprungweite im EMIT-Teil.

Das Rückkopplungsbit F kommt aus dem Datenmanipulator. In einem Statusregister (SR) werden u. a. Vergleichsresultate aus der ALU mit Takt Φ_3 festgehalten: ob das Ergebnis der letzten Rechnung positiv war oder Null, ein Überlauf stattfand, etc.). Die Ergebnisse haben nur eine Gültigkeitsdauer von einem Befehl.

Das Statusregister enthält daneben langfristige Modi des Programms: ob sich das System

im Betriebssystem- oder Benutzermodus befindet, welche Interrupts erlaubt oder gesperrt sind (Interruptmasken), ob das System im Einzelschritt (trace)-Modus läuft oder im Normalbetrieb arbeitet u. a. Diese Modi sind gültig für viele aufeinanderfolgende Befehle.



Im OP-Code des Befehls "bedingter Sprung" sind enthalten die Auswahl des zu überprüfenden Bits und ob seine Negation überprüft werden soll:

z. B.: BP <Sprungziel> branch on positive; Sprung bei gesetztem P-Bit

BNZ <Sprungziel> branch on not zero; Sprung, wenn das Nullbit Z nicht gesetzt ist.

Meist werden 3 - 4 Bit des OP-Codes für die Auswahl benutzt. Sie aktivieren direkt die Steuerleitungen von Multiplexern am Ausgang des Statusregisters, mit denen F zum Entschlüssler durchgeschaltet wird.

1.4.6. Unterprogrammsprünge (call)

Von der Hardware des Abwicklers wird die Folgeadresse NEW gerettet und die Sprungadresse nach IP geladen:

Der Sprung in das Unterprogramm ist

absolut: IP := EMIT oder
indirekt: IP := <EMIT> oder
relativ: IP := IP + EMIT

wobei wiederum das bei den unbedingten Sprungbefehlen gesagte auch hier gilt.

Die Rückkehradresse NEW = (IP + c) kann gerettet werden

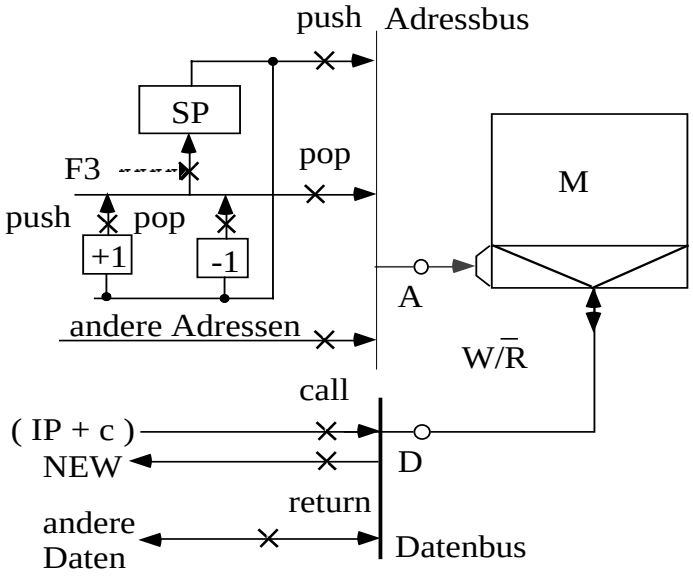
- in einem Register (blockiert dieses Register für andere Anwendungen)
- auf einem festen Speicherplatz (erlaubt wie das Retten in einem Register nur eine Schachtelungstiefe 1 der Unterprogrammsprünge)
- auf einem Stack (gebräuchliche Methode; sie erlaubt Schachtelungen bis zur Tiefe des Stacks).

Die Realisierung eines Rückkehradresstacks kann erfolgen

- durch einen speziellen Stack im Abwickler (in mikroprogrammierten Maschinen

älteren Datums oder einigen superskalaren Rechnern für x86-Code)
- durch einen Stack im Datenspeicher, dem aber ein eigenes ausgezeichnetes Register, der Stackpointer SP zugeordnet ist (u. a. in CISC-Maschinen vom 68.000-Typ, VAX-Rechnern, etc.).

Sind für den Stackpointer eigene Schaltnetze zum Inkrementieren und Dekrementieren vorgesehen, dann lassen sich die Operationen "push" und "pop" auf dem Stack einfach realisieren:




```
push:  M(SP) := datum ; (W/R̄ = 1)          parallel
       SP    := SP + 1                      ablaufend
       if SP = "upper limit" then "alarm UL"

pop :  datum := M(SP) ; (W/R̄ = 0)          parallel
       SP    := SP - 1                      ablaufend
       if SP = "lower limit" then "alarm LL"
```

Der Unterprogrammsprung besteht dann aus den parallel ablaufenden Operationen

push NEW to stack
IP := <Sprungziel>

1.4.7. Rücksprungbefehl (return)

Der alte Zustand vor dem Unterprogrammsprung wird wieder hergestellt:

IP := pop (stack)

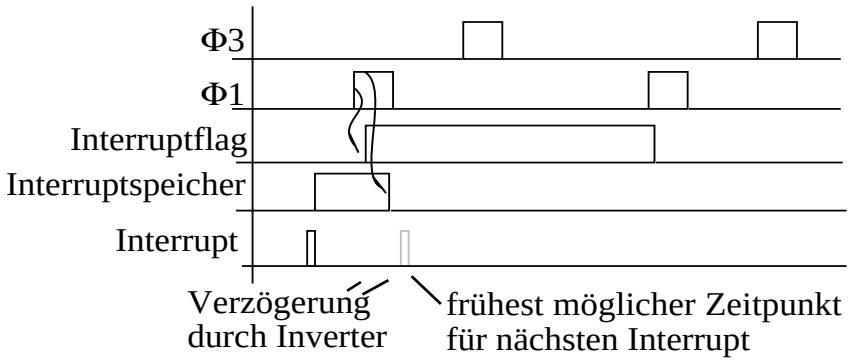
Das Programm läuft mit dem Befehl hinter dem Unterprogrammaufruf weiter.

1.4.8. Interrupts (asynchrone Ereignisse)

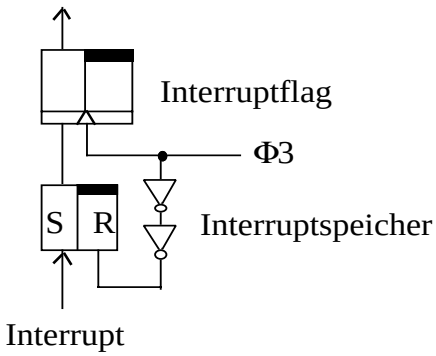
Ein asynchron auftretendes Signal (asynchron = unvorhersehbarer, maximal ungünstiger Zeitpunkt) soll das laufende Programm unterbrechen können. Ein dem Interrupt zugeordnetes Programm soll ablaufen und danach soll das unterbrochene Programm wieder weiterlaufen, ohne die Unterbrechung zu bemerken.

Vorgehensweise der Hardware:

- Durchschalten eines Interrupts, wenn er nicht maskiert ist und entsprechende Priorität (Dringlichkeit) hat.
- Zwischenspeichern des Interruptsignals bis zum Ende des laufenden Befehls.
- Wirksamwerden der Interruptanforderung mit Φ_3.
- Retten von NEW auf den Rückkehradresstack RAS.
- Retten des Statusregisters SR auf dem RAS.
- IP mit Anfangsadresse der Interruptroutine (Trapvektoradresse TVA) laden.
- Löschen der Interruptanforderung nach dem ersten Befehl.



Die Zwischenspeicherung des Interruptsignals erfolgt in zwei Stufen realisiert z. B. durch die folgende Schaltung.



Die bei einem Interrupt angesprungene Adresse ist entweder

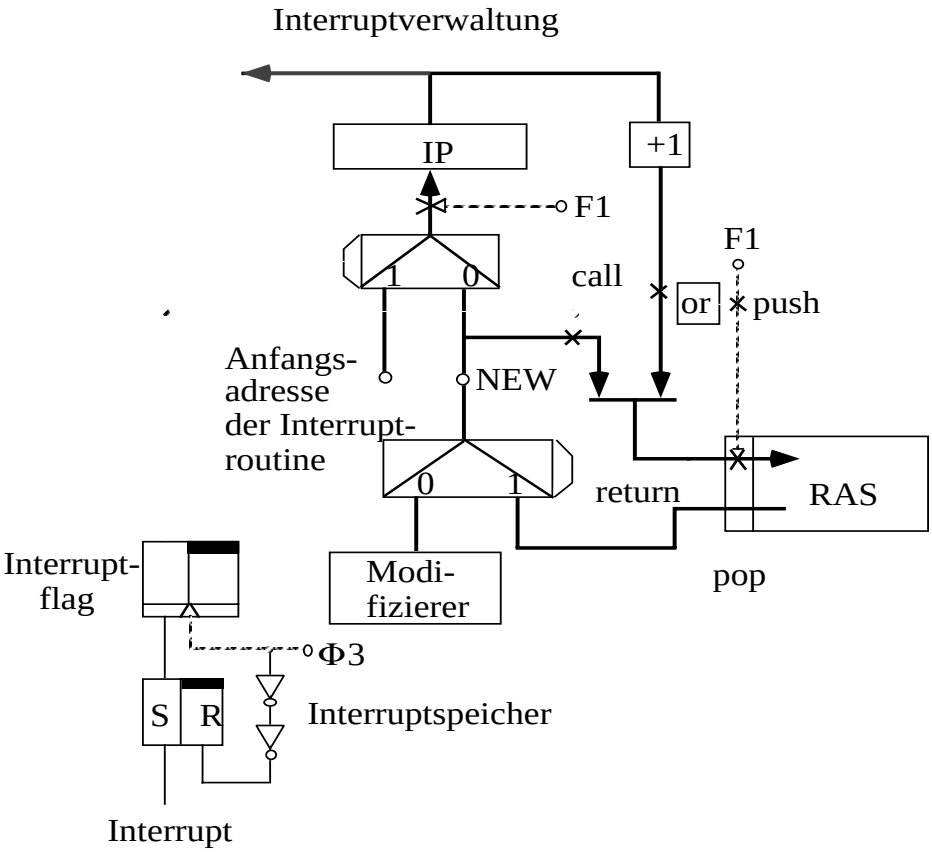
- eine feste Adresse im Betriebssystem
- eine Trapvektoradresse, bestimmt durch die Interruptnummer, die dem Interrupt zugeordnet ist, der das Interruptsignal gesetzt hat. Diese Nummer wirkt als Adresse in

einem Speicher, der die Trapvektoradresse ausliest.

Wann kann ein Interrupt das Interruptsignal setzen ?
Dazu muß er zwei Bedingungen erfüllen:

- Seine Priorität muß höher sein als der Rang des gerade laufenden Programms (= Priorität, die dem laufenden Programm zugeordnet ist).
- Seine Priorität muß höher sein als die aller anderen gerade anstehenden Interrupts.

Hier werde nur der Teil weiter betrachtet, nachdem ein Interrupt schon das Interruptsignal gesetzt hat.



Statusbits im Statusregister können zusätzlich

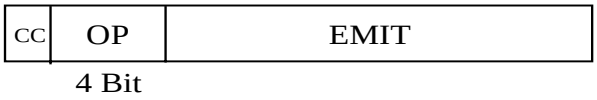
- alle Interrupts verbieten (Disable) oder zulassen (Enable) oder
- einzelne Interrupts über eine Interruptmaske sperren oder zulassen.

Häufig gibt es einen nicht maskierbaren Interrupt (non maskable Interrupt, NMI), der die höchste Priorität besitzt.

1.4.9. Ein konkretes Beispiel für einen Sequencer

Im Folgenden soll ein einfacher Sequencer dargestellt werden. Er beherrscht die o.g.

Sprungbefehle. Das Befehlsformat sei so aufgebaut:



Das Bit cc ist ein Umschaltbit und kennzeichnet

cc = 1 : Semantische Operationen mit IP := IP + 1
cc = 0 : Sprungoperationen mit NOP (no operation)
d. h. keinen Operationen im Rechenwerk

Die folgenden vier Bit sind der eigentliche Operationscode.

OP	Befehl	Wirkung
0000	clear	IP := 0
0001	on r ENTRY	IP := if r then ENTRY else IP
0010	next	IP := IP + 1
0011	skip	IP := IP + 2
0100	halt	IP := IP
0101	back next	IP := IP - 1
0110	jump EMIT	IP := IP + EMIT
0111	goto EMIT	IP := EMIT
1011	skip on F	IP := if F then (IP + 2) else (IP + 1)
1110	jump EMIT on F	IP := if F then (IP + EMIT) else (IP + 1)
1111	call EMIT	RAS := push(IP + 1); IP := EMIT
1100	return	IP := pop(RAS)

Die Freiheit bei der Zuordnung der Codes zu den Befehlen wird zur Vereinfachung des Modifizierers ausgenutzt.

Zunächst wird die Durchschaltung auf NEW in Form eines Programms in einer Hardware-Beschreibungssprache gegeben. Hardware-Beschreibungssprachen beschreiben keine dynamischen Vorgänge wie eine (imperative) Programmiersprache, sondern eine statische Hardware.

Insbesondere läßt sich die case-Konstruktion abbilden als ein Multiplexer, gesteuert durch Bits des OP-Codes mit NEW als Ausgang. Das Symbol ".=" beschreibt die statische Durchschaltung auf die benannte Leitung NEW.

NEW wird mit Φ_1 nur dann nach IP übernommen, wenn kein Interrupt vorliegt. Sonst wird die Trapvektoradresse IADR übernommen und NEW (und die Flags) auf einen Rückkehradresstack gelegt.

```
Beschreibung  von NEW:
NEW.=
case OP of
  (0000  : 0)
  (0001  : if r then ENTRY else IP)
  (0010  : IP + 1)
  (0011  : IP + 2)
  (0100  : IP)
  (0101  : IP - 1)
  (0110  : IP + EMIT)
  (0111  : EMIT)
  (1011  : if F then (IP + 2) else (IP + 1))
  (1100  : pop(RAS))
  (1110  : if F then (IP + EMIT) else ( IP + 1 ))
  (1111  : EMIT)
end-case
at  F1 do
  if (I = 1) then    IP:= TVA(INT)  {* Interrupt I ist erkannt *}
                     RAS:= push(NEW)
             else  IP := NEW
                   if (OP = 1111) then RAS:= push(IP + 1)
                   end-if
   end-if
end-at
```

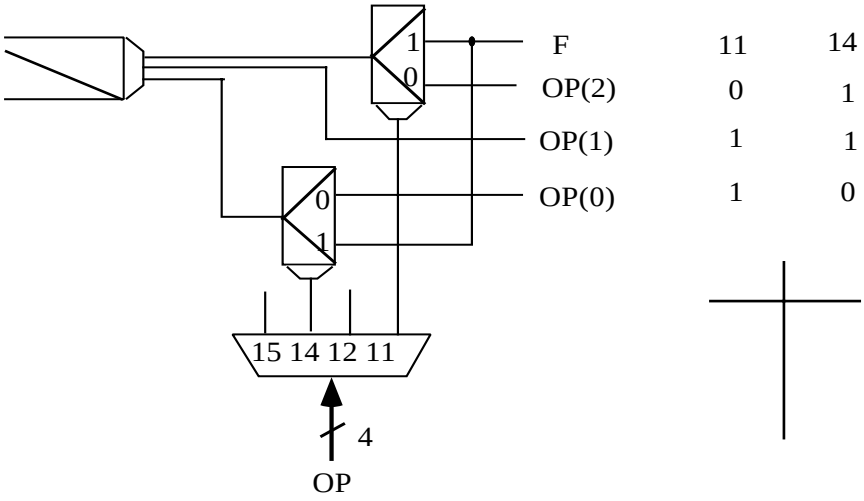
Der Multiplexer für die Durchschaltung der Folgeadresse auf die Leitung NEW kann vereinfacht werden.

Ein Sonderfall, der RETURN-Befehl, wird abgetrennt: in einem eigenen Multiplexer wird in diesem Fall der Wert für NEW vom Rückkehradresstack geholt.

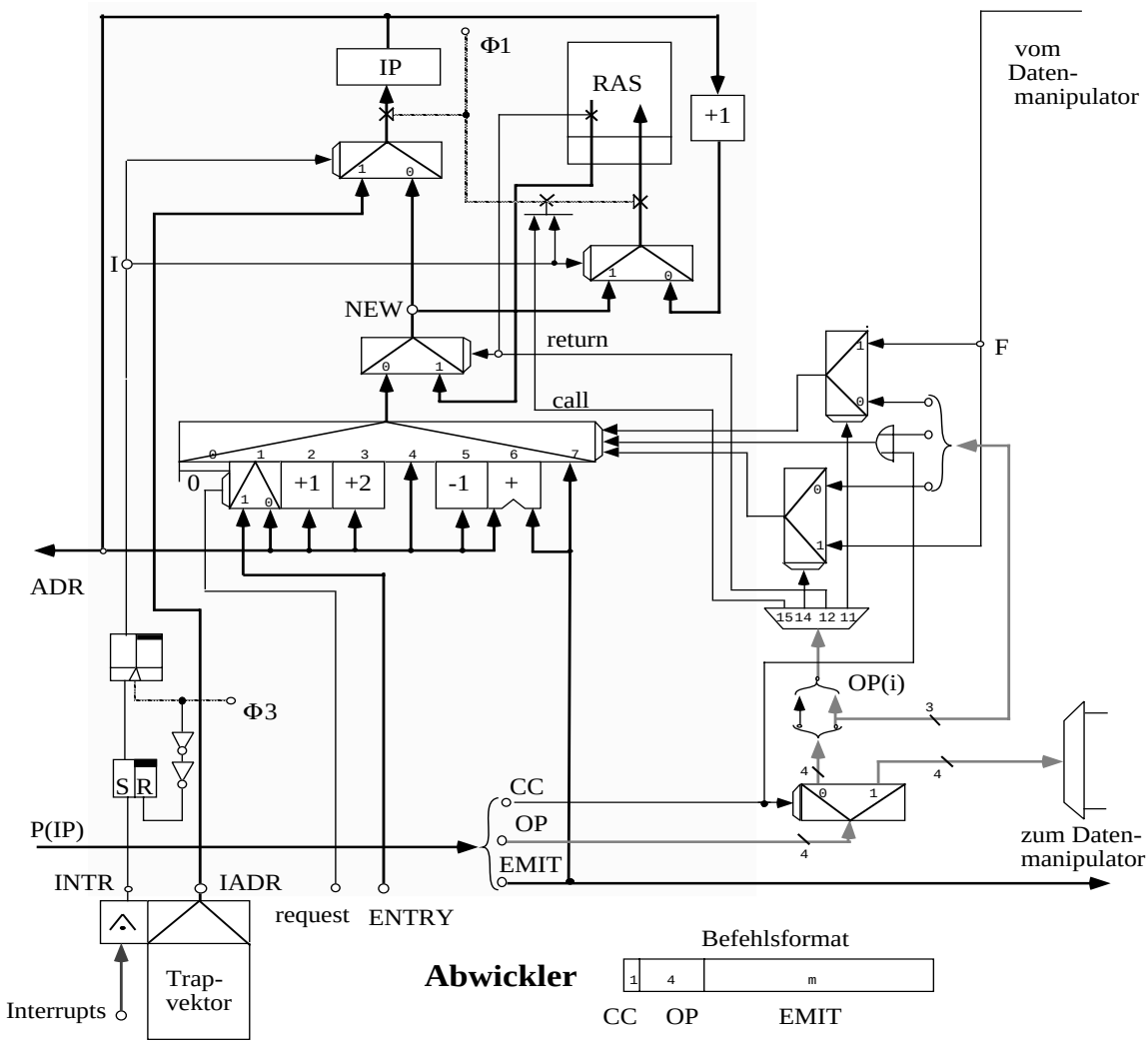
Für alle anderen Befehle reicht ein 8-fach Multiplexer, indem man den Code geeignet wählt:
die Befehle "next", "skip", "skip on F", "jump EMIT" und "jump EMIT on F" sind so codiert, daß sich "skip" und "skip on F" und "jump EMIT" und "jump EMIT on F" nur in jeweils einem Bit unterscheiden und "next" und "skip" bzw. "jump emit" auch nur in einem Bit unterscheiden.

next:	0 0 1 0	next:	0 0 1 0
skip:	0 0 1 1	jump EMIT:	0 1 1 0
skip on F:	1 0 1 1	jump EMIT on F:	1 1 1 0

Diese Bits kann man dann zur Ansteuerung von zwei Multiplexern verwenden, die F oder OP(0) bzw. OP(2) durchschalten. Dabei bezeichnet OP(i) das i-te Bit des Operationscodes.



Der ganze Abwickler erhält dann eine Gestalt wie in der folgenden Abbildung.



1.5. Eine einfache Maschine

Die hier vorgestellte Maschine soll etwas komplexer sein als die primitive Maschine, und sie soll einfache Recheoperationen beherrschen.

Sie hat einen einfachen Abwickler mit

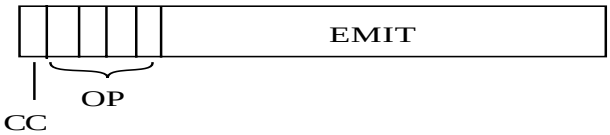
Sprungbefehlen (unbedingt, bedingt)
Unterprogrammsprüngen (call, return)
Startmöglichkeit (ENTRY als Startadresse des Programms)
normale Fortschaltung (IP := IP + 1)

und einen einfachen Datenmanipulator mit vier Registern

Register		Operationen
Adressregister	DP	± 1, laden
Datenregister	DR	laden, (speichern)
Rechenregister	ACC	+, ∗, laden
Flagregister	F	laden

und einen Speicher M(DP), der geladen und gelesen werden kann.

Befehle haben das Format

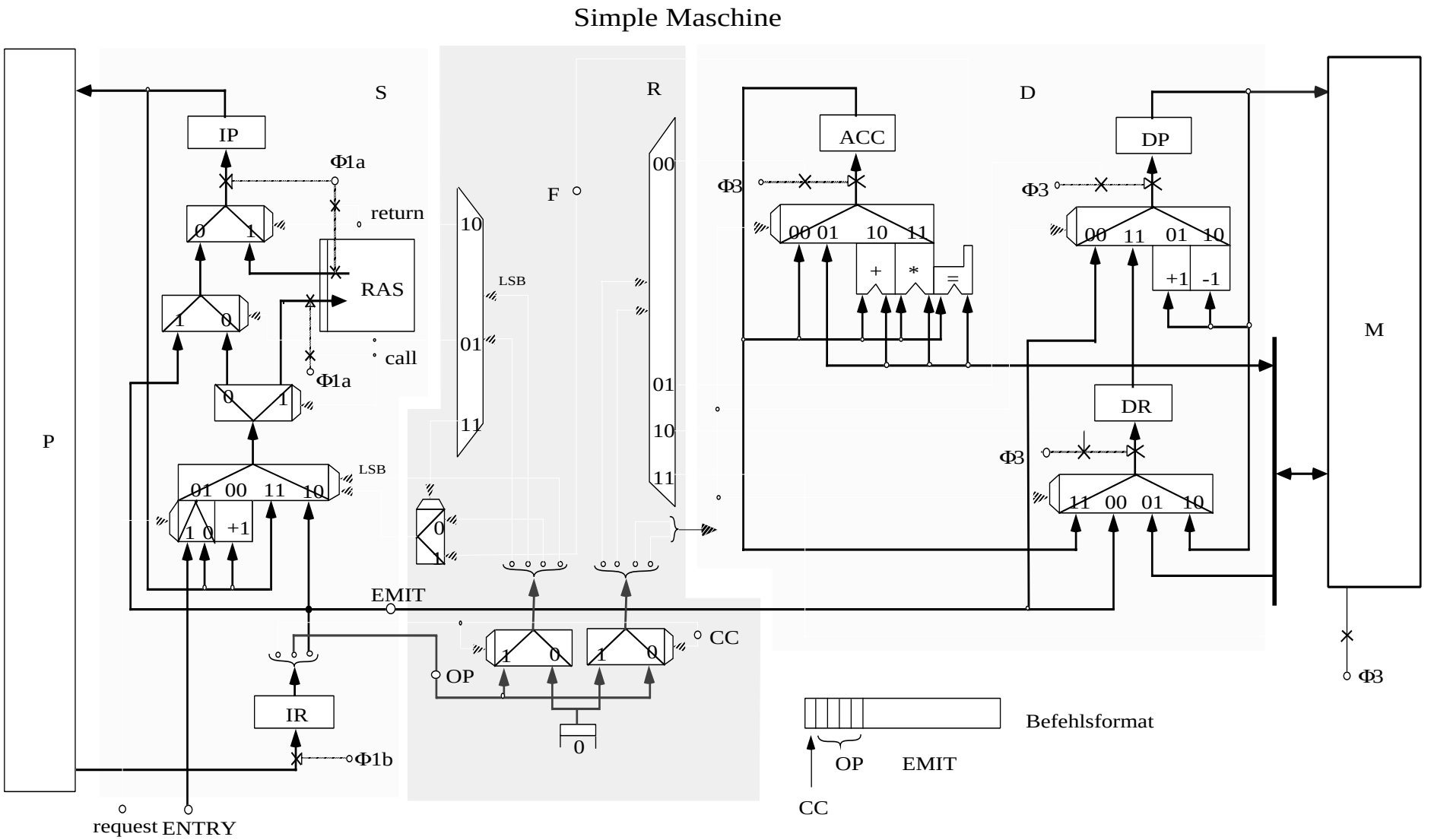


Es gibt insgesamt 20 Befehle.

CC	op-Code	Operation	Aktion
a) Abwicklungsanweisungen (Im Rechenwerk NOP)			
1	00 00	next	IP := IP + 1
1	00 10	goto EMIT	IP := EMIT
1	01 00	call EMIT	IP := EMIT; PUSH(IP + 1)
1	10 xx	return	IP := pop(RAS)
1	00 01	end	IP := if r then ENTRY else IP
1	00 11	halt	IP := IP
1	11 00	bedingter Sprung	IP := if F then EMIT else (IP + 1)
b) Semantische Anweisungen (im Abwickler IP := IP +1)			
Akkumulator ACC (Kennung 00; Ziel der Aktion ist ACC)			
0	00 00	NOP	ACC := ACC
0	00 01	load	ACC := DR
0	00 10	add DR	ACC := ACC + DR
0	00 11	mul DR	ACC := ACC * DR
Adressregister DP (Kennung 01; Ziel der Aktion ist DP)			
0	01 00		DP := EMIT
0	01 01		DP := DP + 1
0	01 10		DP := DP - 1
0	01 11		DP := DR
Datenregister DR (Kennung 10; Ziel der Aktion DR)			
0	10 00		DR := EMIT
0	10 01		DR := M(DP)
0	10 10		DR := DP
0	10 11		DR := ACC
Abspeichern im Speicher (Kennung 11; Ziel der Aktion ist M(DP))			
0	11 xx		M(DP) := DR

Die Codierung der semantischen Operationen ist so gewählt, daß die ersten beiden Bits des OP Code das Ziel des Befehls angeben (00 = Akkumulator ACC, 01 = Adressregister DP, 10 = Datenregister DR, 11 = Speicher M). Das vereinfacht die Dekodierung der Befehle und entspricht der Praxis, im Befehl die Register extra zu benennen. Gegebenenfalls führt man extra Bits mit, um die Dekodierung zu vereinfachen.

Die nachfolgende Abbildung stellt die einfache Maschine dar. Sie ist beschränkt auf die R–T–Ebene, wenn man die Operationen + und * sich durch Schaltnetze realisiert denkt.

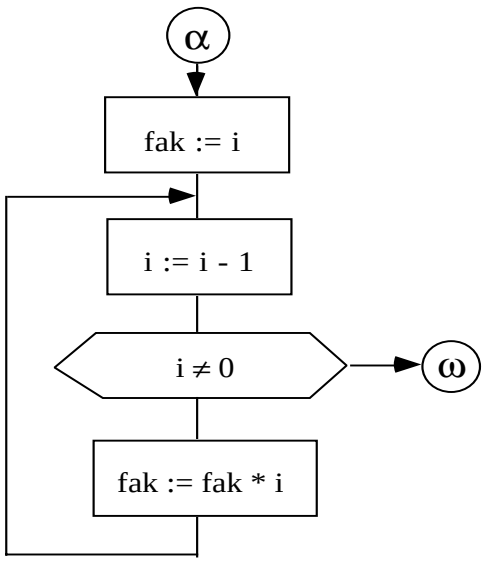

Simple Maschine
P
S
R
D
M
IP
Φ1a
return
RAS
call
Φ1a
LSB
01 00 11 10
1 0 +1
F
10
01
11
LSB
00
01
10
11
ACC
Φ3
00 01 10 11
+ * =
DP
Φ3
00 11 01 10
+1 -1
DR
Φ3
11 00 01 10
EMIT
CC
OP
IR
Φ1b
request
ENTRY
Befehlsformat
OP
EMIT
CC
Φ3

1.6. Ein Beispielprogramm

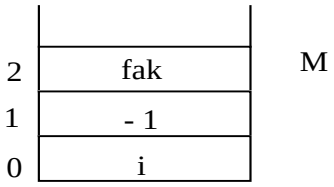
Mit der einfachen Maschine soll die Fakultät von i berechnet werden.

```
FUNCTION fac(i: interger): integer

  VAR fak: integer ;

  BEGIN
      fak := 1 ;
      i := i -1;
      WHILE  i ≠ 0 DO
         BEGIN
             fak := fak * i ;
             i := i - 1 ;
         END ;
      fac := fak ;
  END.
```

Die Organisation der Rechnung folgt dem Flußdigramm



Speicherorganisation



das seinerseits die elementaren Operationen von Pascal in Form von Unterprogrammen realisieren wird:

FIM1 (finde i minus eins) für i := i - 1
und AMI (Akkumulator multipliziert mit i) für fak := fak ∗ i

Ferner enthält das Flußdiagramm einen unbedingten Sprung und berechnet den gesuchten Wert direkt in fak.

Neben den Variablen i und fak soll auch die Konstante -1 im Speicher gehalten werden. Die Speicherorganisation ist oben ebenfalls gezeigt.

Das Programm zur Berechnung der Fakultät lautet dann im Assemblercode der einfachen Maschine:

```
 Initialisierung :  i:= 3

0  10 00  1… 1      DR:= -1
0  01 00  0… 1      DP:= 1            M(1) <-- -1
0  11 00  0… 0      M(DP):= DR

0  10 00  0…11      DR:= 3
0  01 00  0… 0      DP:= 0            M(0) <-- 3
0  11 00  0… 0      M(DP):= DR        (Anfangswert  n)

0  01 00  0…10      DP:= 2
0  11 00  0… 0      M(DP):= DR        M(2) <-- (Anfangswert
                                                 von fak )
```

```
  Unterprogramm FIM1 : i:= i - 1

0  01 00  0… 0      DP := 0
0  10 01  0… 0      DR := M(DP)         ACC  <-- i
0  00 01  0… 0      ACC := DR

0  01 00  0… 1      DP := 1
0  01 01  0… 0      DR := M(DP)         ACC  <-- i - 1
0  00 10  0… 0      ACC := ACC + DR

0  10 11  0… 0      DR := ACC
0  01 00  0… 0      DP := 0             M(0) <-- i - 1
0  11 00  0… 0      M(DP) := DR

1  10 00  0… 0      return
```

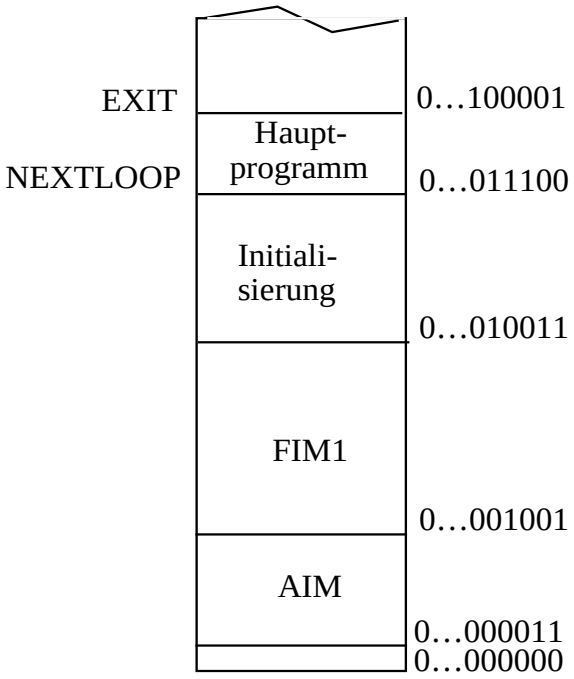
```
  Unterprogramm AIM : fak:= fak * i

0  01 00  0…10     DP := 2
0  10 01  0… 0     DR := M(DP)
0  00 11  0… 0     ACC := ACC * DR     M(2) <-- M(2) * ACC
0  10 11  0… 0     DR := ACC
0  11 00  0… 0     M(DP) := DR

1  10 00  0… 0     return
```

Bei der Umwandlung des Assemblercodes in Maschinencode müssen die Adressen im Speicher insbesondere für die Unterprograsprünge festgelegt werden; hier in Form absoluter Adressen. Jede Änderung im Programm schlägt auf diese Adressen durch. Diese absoluten Adressen stehen hier im Hauptprogramm.

Hauptprogramm - Schleife
(liegt unmittelbar hinter der Initialisierung)

```
1  01 00  0…001001      call FIM1
0  10 00  0…000000      DR := 0
1  11 00  0…100001      if ( ACC = DR ) goto EXIT
1  01 00  0…000011      call AIM
1  00 10  0…011100      goto NEXTLOOP
```

Die hier gewählte Abbildung des Programms in den Speicher zeigt das folgende Bild:



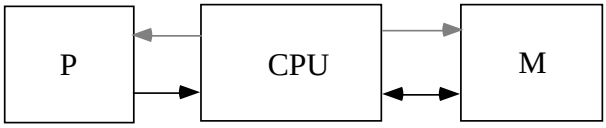
1.7. Princeton- und Harvard-Architektur

Die bislang vorgestellte Wegener-Maschine realisiert eine *Harvard-Architektur,* gekennzeichnet durch getrennten Programm- und Datenspeicher.

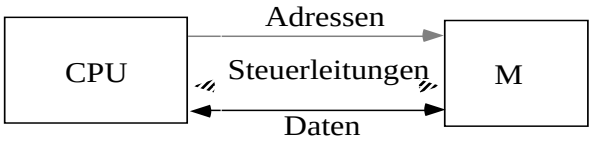
Diese Architektur ist günstig

- bei ROM als Programmspeicher

- bei schnellen Speichern (caches) an Bord des Prozessors um Programm- und Datenzugriffe zu entkoppeln.



Die *Princeton-Architektur* hingegen ist gekennzeichnet durch den gemeinsamen Speicher für Daten und Programme.



Sie ist günstig

- bei teurem Speicher

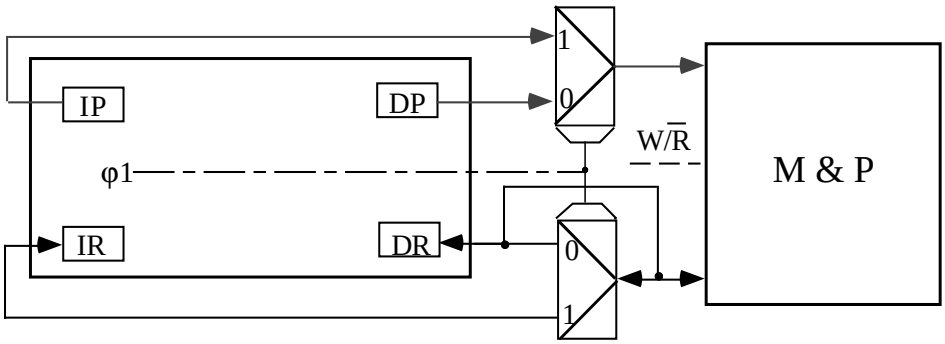
- bei Speicher außerhalb des Prozessors wegen der begrenzten Anzahl von Pins für Daten, Adressen und Steuerleitungen.

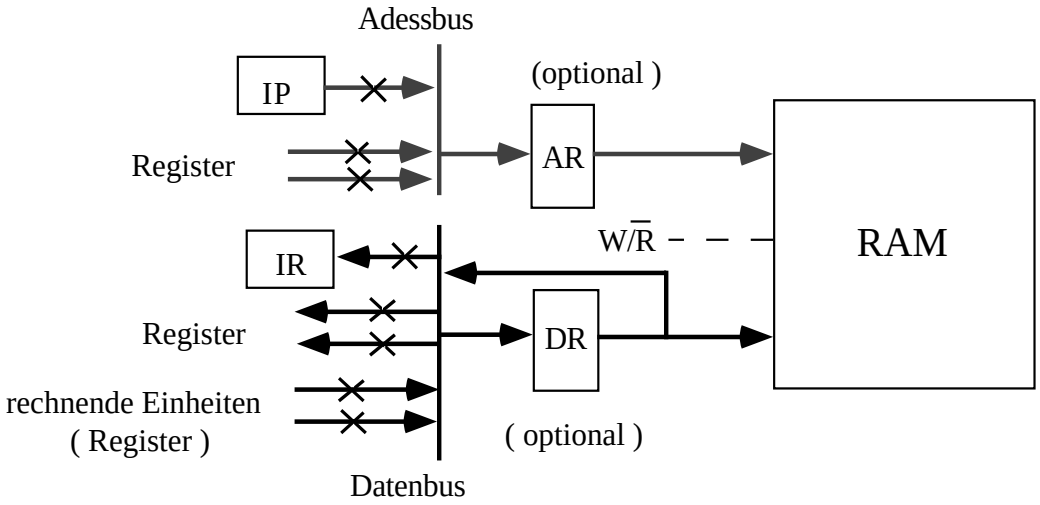
Man findet sie bei µ-Controllern und bei kleinen Maschinen.

Ausgehend von einer Wegener-Maschine könnte sie realisiert sein nach dem untenstehenden Schema, wobei φ_1 die Umschaltung zwischen IP und DP bzw. IR und DR realisiert.

Faktisch wird man statt der Multiplexer zwei Busse finden: einen Adressbus, auf dem von Registern und dem Befehlszähler Adressen geschrieben werden, und einen Datenbus, an dem die Register und das Instruktionsregister hängen.

Optional liegen vor dem Speicher Pufferregister AR und DR, welche günstig sind, wenn die Zugriffszeiten auf dem Speicher langsam sind im Vergleich zum Zugriff in der CPU.





1.8. Beschreibungsebenen und -modelle bei digitalen Prozessen

Es werden vier Beschreibungsebenen unterschieden:

1. Programmebene: sie sagt, **was** getan werden soll, ohne über mögliche Parallelarbeit Aussagen zu machen

2. Register-Transfer-Ebene: sagt, **womit**, d. h. mit welcher Hardware das Problem abgearbeitet wird, gibt die Hardwarebeschreibung an

3. Steuersignalebene: sagt, **wann** und in welcher Reihenfolge das Problem abgearbeitet wird; Zeitverlaufsbeschreibung

4. Befehlsinterpretationsebene: sagt, **wie** im einzelnen die Befehle in Form eines Maschinenprogramms umgesetzt werden.

Der Zusammenhang zwischen den Beschreibungsebenen soll anhand eines Beispiels gezeigt werden.

Es soll ein Multiplizierwerk realisiert werden, welches eine 16-Bit-Multiplikation durch wiederholte Anwendung von Addition und bitweises Schieben realisiert.

Eingabe sind drei Werte x, y und z; die Ausgabe $x * y + z$ soll in drei Registern OV, PU und PL zur Verfügung stehen.

1.8.1. Programmebene

Auf dieser Ebene wird die Multiplikation als Programm beschrieben, hier in Form einer Prozedur in einer Pascal-ähnlichen Sprache.

OV : PU : PL bezeichnet die Verkettung (Konkatenation) der Variablen OV, PU und PL zu einer 33 Bit langen Variablen und *shr* die hier als elementar vorausgesetzte Operation des bitweise Schiebens nach rechts, wobei links Null nachgezogen wird. Das Ergebnis der

Rechnung steht am Ende in den Registern OV:PU:PL = x * y + z

```
PROCEDURE  multiply(x,y,z:  integer);


   var PL,PU,M: integer ;{*16 Bit*}
             k: 0…15;
            OV: boolean; {*Überlaufanzeige*}
            PL: array (0…15 ) of boolean;
                       {*Doppeldefinition von PL*}

    begin

          OV:PU:= z;
          M:= x;
          PL:= y;                          } {*Initialisierung*}
          k:= kmax {≠ 0 oder 16 };

           repeat

                if  (PL(0) = 1)  then OV:PU := M + PU;
                OV:PU:PL := shr ( OV:PU:PL );
                k := k - 1;

            until  k = 0;

        end
```

1.8.2. Register-Transfer-Ebene (oder R-T-Ebene)

Hier wird die Hardware beschrieben, die den Algorithmus abarbeiten soll, wobei zunächst nur die eigentliche datenverarbeitende Einheit beschrieben wird.

Die Beschreibung geschieht in Form einer Hardware-Beschreibungssprache (hardware description language -HDL).

Sie kennt Register, Eingänge, benannte Leitungsbündel (terminal) und Durchschaltungen ".=". Die Konkatenation von Registern (Zusammenfassung in einer Schaltung) wird beschrieben durch ":"

Die Steuersignale für den datenverarbeitenden Teil sind Takte, bei deren Auftreten Zuweisungen ":=" an Register durchgeführt werden.

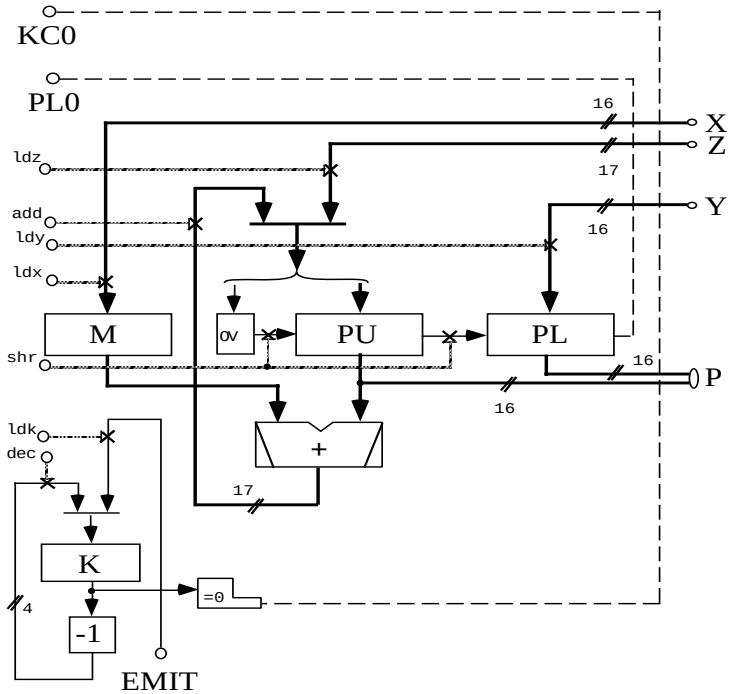
Die R-T-Ebene trifft keine Aussagen über zeitliche Reihenfolgen; sie ist eine statische Beschreibung von Hardware.

```
register  M(15:0), OV, PU(15:0), PL(15:0), K(3:0);
input     x(15:0), y(15:0), z(16:0), KMAX(3:0);
terminal P(31:0)  .= PU:PL;
         PL0      .= PL(0)
         KC0      .= ( K = 0 );

at ldz do  OV:PU  := z   endat;
at add do OV:PU   := M + PU  endat;
at ldy do  PL     := y  endat;
at ldx do  M      := x  endat;
at shr do OV:PU:PL:= shr(OV:PU:PL)  endat;
at ldk do  K      := KMAX  endat;
at dec do  K      := K - 1  endat;
```

Dabei seien ldx, ldy, usw. die o.g. Steuersignale, auf deren Erzeugung auf der RT-Ebene jedopch nicht eingegangen wird.

Im Vergleich zur Programmebene werde die Variable zder Einfachheit halber um 1 Bit erweitert.

Zu einer R-T-Beschreibung gibt es ein Blockdiagramm der Hardware, wobei die R-T-Be-



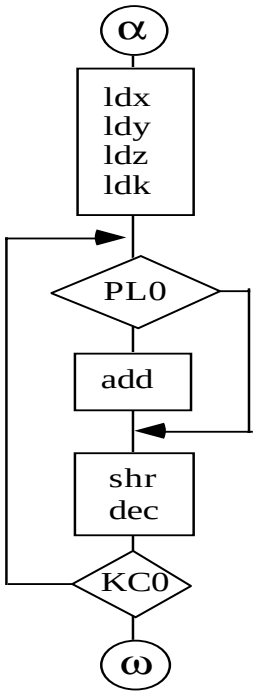
schreibung nur die Topologie festlegt, aber keine Aussage über die geometrische Anordnung macht. Umgekehrt läßt sich ein Blockdiagramm eindeutig in eine R-T-Beschreibung umsetzen. Da es eine statische Beschreibung ist, spielt die Reihenfolge der Terme keine Rolle.

Eingänge sind von außen eingegebene Werte X, Y Z, und EMIT und dieTaktleitungen ldz, ldy, ldx, add, ldk,, dec und shr.

Nach außen werden zur Verfügung gestellt die Bitleitungen PL0 (LSB des Registers PL = 1 ==> PL0 =1) und KC0 (Zähler K = 0 ==> KC0 = 1).

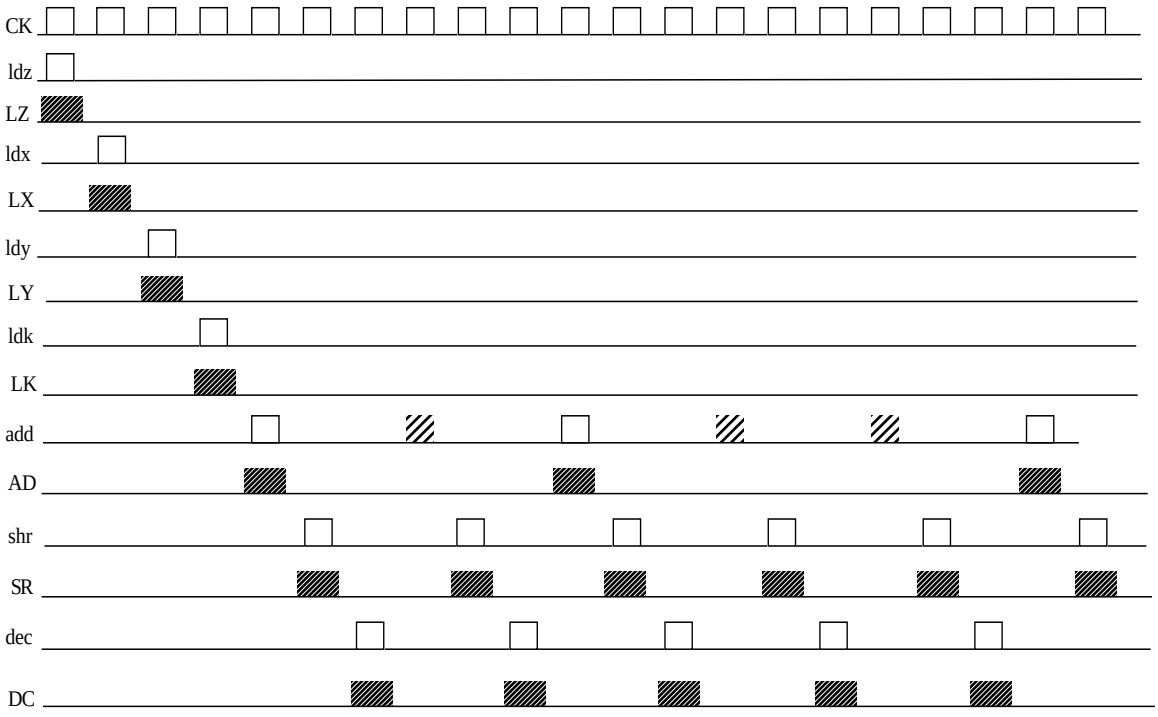
1.8.3. Steuersignalebene

Die Reihenfolge der Steuersignale an die datenverarbeitende Hardware wird hier festgelegt und folgt aus dem Algorithmus.



Dieses Diagramm nach Art eines Flußdiagramms läßt sich umsetzen in ein Bild der tatsächlich erzeugten Taktsignale, wenn man für einen konkreten Satz von Eingangsvariablen x, y und z den Algorithmus ablaufen läßt.

Die Steuersignale lx, ly, … werden als Taktsignale in das Multiplizierwerk gegeben. Ein Steuerwerk erzeugt Signale LX, LY,…, die mit einem Takt CK zu den Signalen ldx, ldy,… synchronisiert werden, die dann die Dauer der Taktsignale haben.



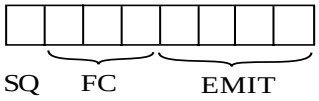
1.8.4. Befehlsinterpretationsebene

Hier muß das Maschinenprogramm angegeben werden, das die Taktfolge generiert.

Dazu müssen drei Dinge vorab festgelegt werden

- das Befehlsformat
- der Befehlssatz
- die Realisierung der nötigen Hardware für den Abwickler und den Entschlüssler.

Das Befehlsformat kommt hier mit Befehlen von 1 Byte Länge aus. Ein Bit unterscheidet semantische und Sprungbefehle, 3 Bit sind Funktionscode.

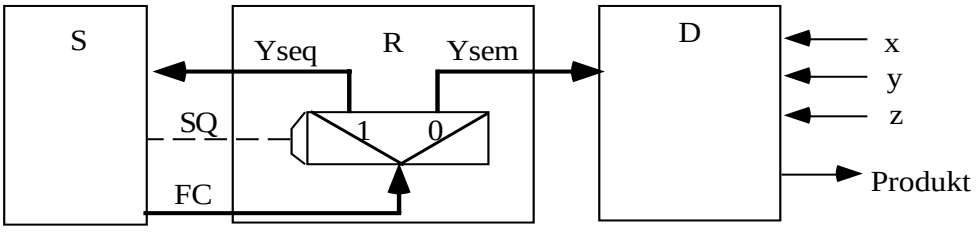


In der Konsequenz stehen für EMIT nur 4 Bit zur Verfügung und damit ist die Sprungweite auf 16 begrenzt, d.h. bei absoluten Sprüngen zugleich auch die maximale Programmlänge und der adressierbare Raum des Speichers.

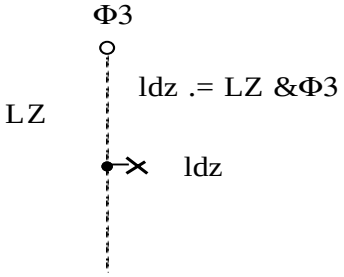
Den Befehlssatz der Multipliziermaschine zeigt folgende Tabelle:

SQ	FC	Name	Aktion	
0 (seman-tische Befehle)	0	NOP	./.	
	1	DC	K := K - 1	
	2	LK	K := EMIT	
	3	SR	OV:PU:PL := shr(OV:PU:PL)	
	4	LX	M := x	IP:= IP + 1
	5	LY	PL := y	
	6	AD	OV:PU := M + PU	
	7	LZ	OV:PU := z	
1 (Sprung-befehle)	000	NOP	IP := IP + 1	
	001	JMP	IP := EMIT	
	010	BNKC0	IP := if not KCO then EMIT else IP + 1	
	011	BNPL0	IP := if not PLO then EMIT else IP + 1	
	101	SMUL	IP := if MUL then 0001 else EMIT	

Der gleiche Funktionscode FC wird je nach Bit SQ verschieden interpretiert und bildet Yseq bzw. Ysem.



Ysem wird aus FC direkt gebildet bei SQ = 0 und die Takte für den datenverarbeitenden Teil daraus mit dem Übernahmetakt Φ_3 nach dem folgenden Bild erzeugt: (am Beispiel LZ -> ldz gezeigt)



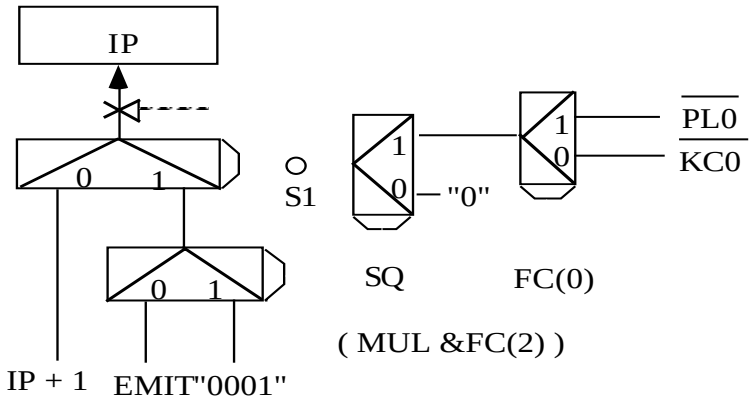
Die Bildung von Yseq ist etwas komplizierter. Nach IP werden mit dem Takt Φ_1 wahlweise durchgeschaltet:

-	IP + 1	bei	SQ = 0
		oder	(SQ = 1 und FC = 000)
		oder	(SQ = 1 und FC = 010 und KCO)
		oder	(SQ = 1 und FC = 011 und PLO)
-	EMIT	bei	(SQ = 1 und FC = 001)
		oder	(SQ = 1 und FC = 010 und ¬KCO)
		oder	(SQ = 1 und FC = 011 und ¬PLO)
		oder	(SQ = 1 und FC = 101 und ¬MUL)
-	"0001"	bei	(SQ = 1 und FC = 101 und MUL)

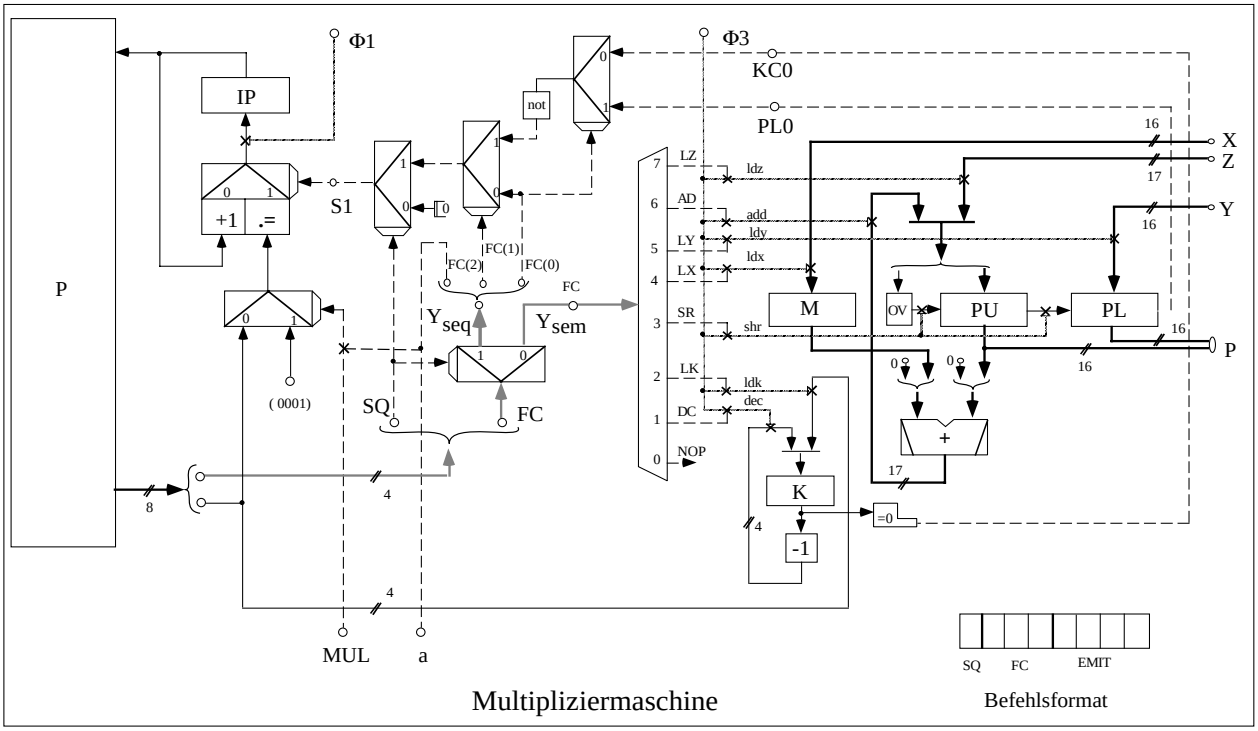
Das Signal MUL ist ein externes Startsignal, mit dem die Startadresse "0001" geladen wird; SQ = 1 und FC = 100 kennzeichnet einen "Halt"-Befehl.

Eine R-T-Beschreibung des Sequencers und der Ansteuerung aus dem Entschlüssler zeigen die nächsten Bilder:

```
at Φ1 do IP := if S1 then
                        if (MUL and FC(2)) then " 0001"
                                           else  EMIT;
                        else(IP + 1);
endat;

terminal  S1 .= if SQ  then
                     if FC(0) then  ¬PL0
                              else  ¬KC0;
                   else  "0".
```



Damit ist die Hardware der Multipliziermaschine vollständig beschrieben. Der Programmspeicher ist als ROM ausgeführt und es gibt infolgedessen auch kein Instruktionsregister.



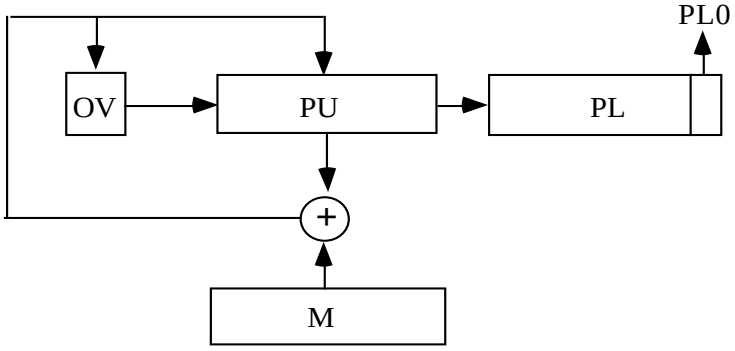
Bleibt nun noch die Umsetzung des Programms "multiply" auf höherer Ebene in ein Maschinenprogramm.

Die Schritte kann man sich recht gut klarmachen an dem Datenflußbild:
Mit dem Laden der Anfangswerte x -> M ; y -> PL und z -> PU steht nach ld(n) = 16 Schritten das Ergebnis in der Konkatenation von OV, PU und PL:

x ∗ y + z -> OV : PU : PL .

Das Datenflußbild und das Maschinenprogramm zeigt das nächste Bild.

Programm der Multipliziermaschine



Anfangswerte:

x --> M

y --> PL

z --> PU

Ergebnis:

x * y + z -->

OV:PU:PL

Adresse		Befehl	Aktion	
WAIT	0000	1101 0000	SMUL	if MUL goto ANF else WAIT
ANF	0001	0111 xxxx	LZ	OV:PU := Z
	0010	0100 xxxx	LX	M := X
	0011	0101 xxxx	LY	PL := Y
	0100	0010 0000	LK	K := 0 (= K := 16)
	0101	1011 0111	BNPL0	if not (PLO) goto L1
LOOP	0110	0111 xxxx	AD	OV:PU := M + PU
	0111	0011 xxxx	SR	OV:PU:PL := shr(OV:PU:PL)
L1	1000	0001 xxxx	DC	K := K - 1
	1001	1010 0101	BNKC0	if not (KCO) goto LOOP
	1010	1001 0000	JMP	goto WAIT

Es hat eine Länge von 11 Byte, paßt also in den Speicher hinein.