AHPL - Eine hardware-beschreibende Sprache auf APL-Basis

von

Rolf Eckert

A 74/15

November 1974

Einleitung

Innerhalb einer zweisemestrigen Vorlesung über Rechnerorganisation führte Herr Prof. W. Giloi eine rechnerbeschreibende Sprache (AHPL - A Hardware Programming Language) ein, die zur Familie der Register-Transfer-Sprachen gehört und eine modifizierte Fassung von Iverson's APL (A Programming Language) ist. Angeregt dazu wurde er durch ein neu erschienenes Buch von Friedrich J. Hill und Gerald R. Peterson mit dem Titel

"Digital Systems
Hardware Organisation and Design" .

Die Autoren dieses Buches gehen davon aus, daß ein Digitalrechner einen "Hardware-Interpreter" oder ein "Hardware-Programm" für einen bestehenden Maschinenbefehlsvorrat darstellt und kommen deswegen zu dem Schluß, daß eine Programmiersprache das natürlichste Hilfsmittel ist, um einen Rechner zu beschreiben. Obwohl im obengenannten Buch sehr viele Fehler und keine geschlossene Darstellung der nur nach und nach eingeführten Notation enthalten ist, haben wir während des Vorlesungs- und Übungsbetriebes u.a. auch an der Resonanz bei den Studenten festgestellt, daß AHPL für den universitären Lehrbetrieb ein hervorragend geeignetes und schnell zu erlernendes Hilfsmittel ist, um dem Studenten den Zugang zu Hardware-Konfigurationen und Aktivitäten von Digitalrechnern zu erleichtern.

Da wir uns in der Forschungsgruppe Rechnerorganisation und Computer Graphics u.a. mit dem Entwurf eines neuartigen Rechnerkonzeptes (STARLET-Projekt) befassen, haben wir wegen der guten Erfahrung mit AHPL beschlossen, diese Sprache zur Strukturbeschreibung des STARLET-Rechners und als Simulation- und Designsprache zu verwenden.

Dieser Bericht ist eine Zusammenfassung der von uns mit AHPL gewonnenen Erfahrung und soll als gemeinsame Ausgangsbasis für mehrere Diplomarbeiten gelten, in denen zunächst untersucht werden soll, ob das bestehende AHPL so verändert werden kann, daß es eine echte Untermenge von APL\360 (die bekannteste Implementierung von Inverson's APL, die auf mehreren IBM Computersystemen und der Siemens 4004 vorhanden ist) darstellt.

Diese APL-Untermenge wollen wir dann als Simulationssprache auf Algorithmenebene benutzen. In nachfolgenden Arbeiten sollen dann spezielle Übersetzer für die Simulation auf Gatterebene und für den automatischen Entwurf von Rechnerkomponenten implementiert werden.

Im folgenden Bericht werden in Kapitel 2 die Grundelemente und die Eigenschaften der AHPL-Funktionen beschrieben. Für den analytischen Teil (Syntaxanalyse) der noch zu implementierenden Compiler wurde unter Punkt 2.1.3 eine Klassifizierung der AHPL-Funktionen eingeführt, die eine Übersetzung von AHPL-Ausdrücken erleichtern sollen. In Kapitel 3 werden die von Hill und Peterson vorgeschlagenen Transfer- und Branchstatements erläutert. Obwohl sich alle in diesem Kapitel erwähnten Statementtypen mit reinem APL\360 darstellen lassen, wurden bisher keine Änderungen vorgenommen. Das gleiche gilt für die in Kapitel 4 beschriebene Unterprogrammtechnik, die unnötigerweise sehr stark von der APL-Grammatik abweicht. In Kapitel 4 wird verbal die AHPL-Semantik beschrieben. Dieser Teil des Berichtes ist nur von Interesse für die spätere Implementierung eines Compilers für den automatischen Entwurf von Rechnerkomponenten, wobei es sinnvoll wäre, dem Entwerfer mittels Deklarationsstatements ein zusätzliches Mittel in die Hand zu geben, um die Bausteine seines zu entwerfenden Rechners präziser zu beschreiben.

AHPL - A Hardware Programming Language

1. ÜBERBLICK

2. SPRACHELEMENTE
   - 2.1 Primärelemente
     - 2.1.1 Konstanten
       - Flip-Flop Konstante
       - Register Konstante
       - Spezielle Vektoren
     - 2.1.2 Variable
       - Flip-Flop
       - Register
       - Memory
       - Adressierung durch Indizierung
     - 2.1.3 AHPL-Operatoren
       - Operatoren ohne strukturellen Effekt
       - Operatoren ohne Werteffekt
       - Operatoren mit Struktur- und Werteffekt
     - 2.1.4 AHPL-Unterprogrammaufrufe
       - AHPL subroutines
       - AHPL functions
     - 2.1.5 AHPL-Ausdrücke

3. STATEMENTTYPEN
   - 3.1 Controlstatements
     - Simple transferstatements
     - Alternative transferstatements
   - 3.2 Branchstatements
     - Unconditional branch
     - Computed gotø branch
     - Relation branch
     - Function branch
   - 3.3 Simple Simultanstatements
   - 3.4 HALT Statment

3.5 Simultane Kontroll-Programme

- DEAD END-Program
- CØNVERGE und DIVERGE-statements

3.6 Kommunikation zwischen mehreren autonomen Kontrolleinheiten

- Communication data register
- Communication lines

4. Übersetzung in Hardware

4.1 Grundelemente eines Controllers

- Pulse controller
- Delay element
- Asynchronous transfer
- Verzweigungen

4.2 Register Logik

- JAM - transfer
- Alternative transferstatement
- BUS - transfer
- Level controller

4.3 Unterprogrammtechnik

- Connection statement
- Bookkeeping statement

1. AHPL - A Hardware Programming Language

AHPL ist u.a. ein formales Mittel, um Vorgänge in einem Digitalrechner zu beschreiben. Ein Rechner besteht im wesentlichen aus 'steuernden' und 'gesteuerten' Einheiten. Die Aufgaben von AHPL bestehen also aus:

- Formale oder algorithmische Beschreibung von Steuerfunktionen (Leitwerkaufgaben)
- Algorithmische Beschreibung von Recheneinheiten (Addierer, Multiplizierer,... i.a. Rechenwerkelemente)

Dabei werden folgende Forderungen gestellt :

- Ein AHPL-Steuerprogramm muß sich gedanklich leicht in Hardware übersetzen lassen. Ein entsprechendes Programm, das diesen Übersetzungsvorgang unternimmt, nennen wir Hardwarecompiler. Die Ausgabedaten des Compilers stellen eine 'hardwired control-unit' dar.
- Gesteuerte Einheiten, in AHPL als Unterprogramme definiert, sollen in algorithmischer Form hardwaremäßig 'komplett' beschrieben werden; d.h. aus dem entsprechenden AHPL-U.P. sollen alle Leitungsverbindungen zwischen bestimmten Grundelementen (Gattern) ersichtlich sein.

Die Beschreibe-, Entwurfs und Simulationssprache AHPL ist stark an Iverson's APL (A Programming Language) angelehnt, da die APL-Primärelemente leicht als Grundhardwareelemente eines Digitalrechners interpretiert werden können. Die entsprechenden Analogien sind

APL - Typ		AHPL - Typ
Skalar	-	Flip-Flop
Vektor	-	Register
Matrix	-	Speicher

wobei nur APL-Typen von der Art Boolean betrachtet werden.

2. SPRACHELEMENTE VON AHPL

2.1 Primärelemente

2.1.1 Konstanten

Unterschieden wird zwischen Flip-Flop und Registerkonstanten :

<flipflop constant> ::=0|1

<register constant> ::=<boolean vector>|<encode vector>|
<full vector>|<unit vector>|
<suffix vector>|<prefix vector>

Ein boolean vector ergibt sich durch Catenation mindestens zweier Flip-Flop-Konstanten.

<boolean vector> ::=<flipflop constant>,<flipflop constant>|
<boolean vector>,<flipflop constant>

Für Registerkonstanten mit vielen Komponenten kann der APL-Encode-Operator ⊤ verwendet werden.
In APL gilt:

X ← j(n)⊤Z

Dabei ist X der n-Element Vector, der das Abbild der dezimalen Zahl Z im Zahlensystem zur Basis j darstellt. Da wir uns in AHPL auf das binäre Zahlensystem beschränken, kann die Angabe von j entfallen. Wird auf die Angabe von (n) verzichtet, so hat X die Mindestkomponentenzahl, die notwendig ist, um das dezimale Z darzustellen.

Beispiele:

MA ← 2(5)⊤13 }
MA ← (5)⊤13 } MA = 0,1,1,0,1

MA ← ⊤13 MA = 1,1,0,1

<encode vector> ::= $\{$<integer>$\}_{0}^{1}$ ⊤ <integer>
<integer> ::= <digit>|<integer><digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9

Die Angabe eines Vorzeichens für n und z ist verboten.

Um eine kompakte Schreibweise in AHPL-Programmen zu erzielen, wurden für oft benötigte Registerkonstanten spezielle Vektoren eingeführt. Diese sind :

<full vector> ::= ε(<integer>)

<unit vector> ::= $\varepsilon^{\langle integer\rangle}$(<integer>)

<suffix vector> ::= $\omega^{\langle integer\rangle}\{(\langle integer\rangle)\}_{0}^{1}$

<prefix vector> ::= $\alpha^{\langle integer\rangle}\{(\langle integer\rangle)\}_{0}^{1}$

Beispiele

	Name	Bedeutung
ε (n)	Full vector	Vektor mit n Komponenten, die alle den boolschen Wert 1 haben
ε^{i}(n)	Unit vector	Vektor mit n Komponenten i-te Komponente = 1 alle übrigen Komponenten = 0
α^{i}(n)	Prefix vector	Vektor mit n Komponenten, von denen die i linken Komponenten 1, der Rest 0 sind.
ω^{i}(n)	Suffix vector	Vektor mit n Komponenten, von denen die i rechten Komponenten 1, die Restkomponenten 0 sind.

ε^{2}(6) 0,0,1,0,0,0

ε (6) 1,1,1,1,1,1

$\bar{\varepsilon}$ (6) 0,0,0,0,0,0

α^{2}(6) 1,1,0,0,0,0

ω^{2}(6) 0,0,0,0,1,1

Beim Suffix- und Prefixvektor kann die Angabe der Komponentenzahl fehlen

d.h. $\alpha^{2} = \omega^{2} = \varepsilon(2) \mathrel{\hat{=}} 1,1$

2.1.2 Variable

AHPL ist deklarationsfrei, d.h. Flip-Flops, Register und Speicher können durch symbolische Namen gekennzeichnet werden, der Typ wird durch den Kontext spezifiziert.

Ein symbolischer Name besteht aus alphanumerischen Zeichen, von denen das erste ein Buchstabe sein muß.

<identifier> ::= <Buchstabe>$\{$<Buchstabe> | <Ziffer>$\}_0$

Durch einfache Indizierung kann ein einzelnes Bit eines Registers oder ein einzelnes Register eines Registerspeichers, durch doppelte Indizierung ein Bit eines Registerspeichers angesprochen werden. (siehe Bild 1)

<flipflop> ::= <flipflop indentifier>|<register><index>|
<memory><index><index>

<register> ::= <register identifier>

<memory register> ::= <memory identifier><index>|<memory identifier><index>

<memory> ::= <memory identifier>

<flipflop identifier> ::= <identifier>

<register identifier> ::= <identifier>

<memory identifier> ::= <identifier>

<index> ::= <integer>

Ein Bus ist ein weiteres Grundelement in AHPL, es besitzt ebenfalls Speichereigenschaften und kann mit einem symbolischen Namen gekennzeichnet werden, dessen letzte drei Zeichen den String BUS ergeben.

<BUS> ::= <bus identifier>
<bus identifier> ::= <identifier> BUS

Im Gegensatz zu einem Register darf ein Bus nicht indiziert werden.

Durch kombinierte Anwendung von Ausblend- und Reduktionsfunktion können zusammenhängende Teile eines Registers oder Busses angesprochen werden.

Beispiele:

ω^{12}/IR stellen die letzten 12 Bits eines Registers dar, dessen Name IR ist.

α^{6}/ω^{12}/IR Von den letzten 12 Bits werden die ersten 6 angesprochen.

ω^{12}/ABUS stellen die 12 rechtsbündigen Komponenten eines Busses dar, dessen Name ABUS lautet.

<register part> ::= $\{$<cut off vector>/$\}_{1}^{2}$<register>
<cut off vector> ::= <prefix vector>| <suffix vector>
<bus part> ::= $\{$<cut off vector>/ $\}_{1}^{2}$<bus>

Auf analoge Weise können mit Hilfe der im nächsten Kapitel eingeführten Funktionen (Row Compress und Column Compress) zusammenhängende Teile einer Matrix ausgeblendet werden.

Ein RAM-Zugriffsmechanismus (random access memory) wird in der AHPL-Notation durch den Decode Operator beschrieben. Ein Transfer vom oder in den Speicher kann nur über ein memory data (MD-Register) erfolgen. Die entsprechende Adresse im Speicher wird durch den Inhalt des memory adress register (MA-Register) bestimmt.

Durch den Ausdruck $M^{\perp MA}$ wird diejenige Zeile im Speicher M angesprochen, deren Adresse in codierter Form im MA-Register steht.

REG_0	REG_1	REG_2	. . .	REG_{ρ_1-1}

REG ≙ Registername; ρ_1 ≙ Flipflop-Anzahl

Bild 1a Indizierung von Registern

$STCK^0_0$	$STCK^0_1$	$STCK^0_2$	$STCK^0_3$	...	$STCK^0_{\rho_2-1}$
$STCK^1_0$					
$STCK^2_0$					
.					
.					
$STCK^{\rho_1-1}_0$					$STCK^{\rho_1-1}_{\rho_2-1}$

STCK ≙ Name eines Registerspeichers

ρ_2 ≙ Flipflop-Anzahl je Register

ρ_1 ≙ Registeranzahl

$STCK^0$ ≙ 1-te Zeile der Speichermatrix

$STCK_0$ ≙ 1-te Spalte der Speichermatrix

$STCK^0_0$ ≙ 1-tes Bit der 1. Zeile

Bild 1b Indizierung von Registerspeichern

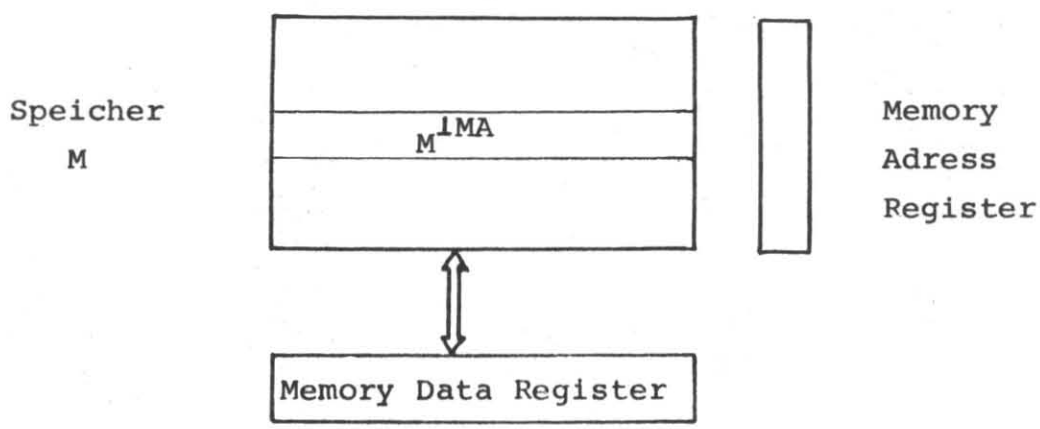


Bild 1c RAM - Zugriff

2.1.3 AHPL-Operatoren

Sämtliche AHPL-Operatoren lassen sich in 3 Gruppen einteilen :

- Operatoren ohne strukturelle Effekte
- Operatoren ohne Werteffekte
- Operatoren mit Wert- und Struktureffekten

Operatoren ohne strukturelle Effekte (Flipflop- oder Skalaroperatoren)

Skalaroperatoren haben keinen Einfluß auf die Struktur der Operanden und daher auch nicht auf die Menge der Operandenindizes.

Sie erzeugen ein Flipflop-Ergebnis, wenn die Operanden Flipflops sind; sie erzeugen Registerergebnisse, wenn sie durch Element-zu-Element-Beziehung auf Register angewandt werden.

Wenn bei dyadischen Flipflop-Operatoren ein Operand ein Skalar (Flipflop), der andere ein Register ist, wird das Flipflop mit jeder Komponente des Registers dyadisch verknüpft.

Wenn beide Operanden Register sind, müssen sie von gleicher Dimension sein.

Die einzelnen Operatoren dieser Gruppe sind :

Operation	Name	Bedeutung
$Z \leftarrow \bar{X}$	NØT (monadisch)	X | Z 1 | 0 0 | 1
$Z \leftarrow X \wedge Y$	AND	X Y | Z 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
$Z \leftarrow X \vee Y$	ØR	X Y | Z 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1

Operation	Name	Bedeutung
$Z \leftarrow X \oplus Y$	EXCLUSIVE ØR	X Y Z 0 0 0 0 1 1 1 0 1 1 1 0
$Z \leftarrow (X<Y)$	Relations-operator	Z=1, wenn X kleiner als Y, ansonsten Z=0
$Z \leftarrow (X \leq Y)$	Relations-operator	Z=1, wenn X kleiner oder gleich Y, ansonsten Z=0
$Z \leftarrow (X=Y)$	Relations-operator	Z=1, wenn X gleich Y, ansonsten Z=0
$Z \leftarrow (X \geq Y)$	Relations-operator	Z=1, wenn X größer oder gleich Y, ansonsten Z=0
$Z \leftarrow (X>Y)$	Relations-operator	Z=1, wenn X größer als Y, ansonsten Z=0
$Z \leftarrow (X \neq Y)$	Relations-operator	Z=1, wenn X ungleich Y, ansonsten Z=0

Beispiele:

W = 1
U = 1,0,0,1,0,1
V = 0,1,1,1,0,1
Z1 ← U∧V
Z1 = 0,0,0,1,0,1
Z2 ← U<V
Z2 = 0,1,1,0,0,0
Z3 ← W∨U
Z3 = 1,1,1,1,1,1

Operatoren ohne Werteffekt

Diese Operatoren wirken nur auf die Struktur der Operanden und lassen die Werte der einzelnen Indexpositionen unverändert, wenn man von der Tatsache absieht, daß durch einige Operatoren bestimmte Indexpositionen wegfallen oder hinzukommen. (Compress, Cut off und Catenate)

Operation	Name	Bedeutung
Z1 ← n⍴0	Reshape Vector Function	Z1 ← 5⍴0 Z1 = 0,0,0,0,0
N ← m n⍴X	Reshape Matrix Function	N ← 3 3⍴X 0 0 0 N = 1 0 1 1 0 0
Z2 ← ω^i/X Z3 ← ∝^j/X	Cut off (Ausblenden von Registerinhalten)	Z2 ← ω^5/X Z3 ← ∝^3/X Z2 = 1,0,1,0,0 Z3 = 0,0,0
Z4 ← k↑X	Left Rotate (Rundshift links um k Stellen)	Z4 ← 3↑X Z4 = 1,0,1,0,0,0,0,0
Z5 ← k↓X	Right Rotate (Rundshift rechts um k Stellen)	Z5 ← 3↓X Z5 = 1,0,0,0,0,0,1,0
M ← ⇑ N	Rotate up (Kreisshift aufwärts von Matrix N um eine Reihe)	M = 1 0 1 1 0 0 0 0 0
M ← ⇓ N	Rotate down (Kreisshift abwärts von Matrix N um eine Reihe)	M = 1 0 0 0 0 0 1 0 1
Z ← X,Y	Catenate	$Z = X_0, X_1 \ldots X_{(\rho X)-1}, Y_0, Y_1 \ldots Y_{(\rho Y)-1}$

X ≡ 0, 0, 0, 1, 0, 1, 0, 0

Operation	Name	Bedeutung
$Z \leftarrow U/X$	<u>Compress</u> Z ergibt sich aus X, indem alle X_i unterdrückt werden, für die $U_i=0$.	U = 1,0,1,0,1 X = 1,1,1,1,0 $Z \leftarrow U/X$ Z = 1,1,0
$A1 \leftarrow U1/M$	<u>Row Compress</u> (Spaltenselektion) Jede Zeile M^i der Matrix M wird entsprechend U1 komprimiert. $A^i = U1/M^i$	$M = \begin{matrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{matrix}$ U1 = 0,1,1,0 $A \leftarrow U1/M$ $A1 = \begin{matrix} 1 & 0 \\ 0 & 0 \\ 1 & 0 \end{matrix}$
$A2 \leftarrow U2//M$	<u>Column Compress</u> (Reihenselektion) Jede Spalte M_j der Matrix M^j wird entsprechend U komprimiert. $A2_j = U2/M_j$	U2 = 1,0,1 $A \leftarrow U2//M$ $A2 = \begin{matrix} 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{matrix}$

Funktionen mit Wert- und Struktureffekt

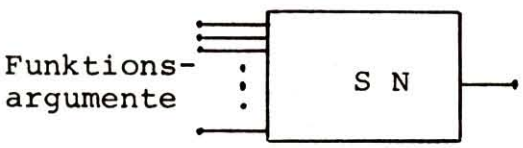
Operation	Name	Bedeutung
$Z \leftarrow k \uparrow_{0} X$	Left Shift Links Shift um k Stellen mit Nachziehen von Nullen	X = 1,1,1,0,0,0 Z1 ← $3 \uparrow_{0} X$ Z1 = 0,0,0,0,0,0
$Z \leftarrow k \downarrow_{0} X$	Right Shift Rechts Shift um k Stellen mit Nachziehen von Nullen	Z2 ← $3 \downarrow_{0} X$ Z2 = 0,0,0,1,1,1
$Z \leftarrow \odot /X$	Reduce Ein Vektor wird zu einem Skalar reduziert, indem von rechts nach links alle Vektorkomponenten nacheinander mit einer binären AHPL-Funktion verknüpft werden.	$\odot$ ≙ binäre AHPL-Funktion ohne strukturellen Effekt $Z = (X_0 \cdot (X_1 \cdot (X_2 \cdot (X_{(\rho X)-2} \cdot X_{(\rho X)-2}))))\ldots)$ X = 1,1,1,0,0 Z1 ← $\vee/X$ Z1 = 1 Z2 ← $\wedge/X$ Z2 = 0
$X \leftarrow \odot/M$	Row Reduce $X_i = \odot /M^i$ Jede Zeile M^i der Matrix M wird zu einem Skalar reduziert.	$M = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \end{matrix}$
$X \leftarrow \odot//M$	Column Reduce $X_i = \odot/M_i$ Jede Spalte M_i der Matrix M wird zu einem Skalar reduziert	X1 ← $\oplus/M$ X = 1,0,1,0 X2 ← $\oplus//M$ X2 = 0,0,0,0,0

2.1.4 AHPL Unterprogrammaufrufe

Es wird zwischen AHPL-Function und AHPL-Subroutine unterschieden. Solche Hardwareunterprogramme stellen eine Folge von Programmschritten dar, die ein logisches Netzwerk beschreiben. Ein Netzwerk (Schaltnetz) sei an dieser Stelle für uns eine 'black box', die über ein oder mehrere Eingänge und ein oder mehrere Ausgangsleitungen verfügt. Die Wirkungsweise des Schaltnetzes wird durch den Unterprogrammrumpf beschrieben. Der Function- oder Subroutine-Name gibt dem schwarzen Kasten einen Namen, der dann als Primärelement zur Bildung von AHPL-Ausdrücken verwendet werden kann.

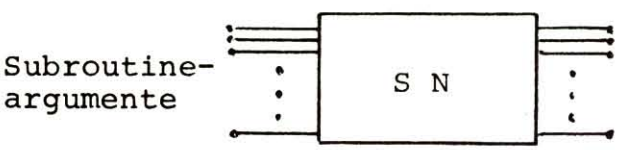
Eine Function wird definiert, wenn das Schaltnetz nur eine Ausgangsleitung, eine Subroutine, wenn das Schaltnetz einen Vektor, d.h. mehrere Ausgangsleitungen hat.

Schaltnetz mit skalarem Ausgang



Schaltnetz SN wird durch Funktionsunterpropramm beschrieben

Schaltnetz mit vektoriellem Ausgang



Schaltnetz wird durch Subroutine-Unterprogramm beschrieben

Der Funktionsname einer Subroutine ist formal identisch mit dem Aufbau eines Identifiers.

Der Funktionsname einer Funktion ist durch einen Identifier und durch den abschließenden Buchstaben F gegeben.

ADD ≙ Subroutinename eines Volladdierers

CARRYF ≙ Funktionname für ein Schaltnetz, das für ein Bit den Übertrag liefert.

Zum Aufruf eines Unterprogrammes gehört analog zu höheren Programmiersprachen die in die Klammern gesetzte Argumentliste.

Diese enthält die zu verknüpfenden Eingangsleitungen des Schaltnetzes.

Die Ausgangsleitungen werden durch den Funktionsaufruf beschrieben.

<subprogram reference> ::= <suboutine reference>|<function reference>

<subroutine reference> ::= <subroutine name>(<argument list>)

<function reference> ::= <function name>(<argument list>)

<subroutine name> ::= <identifier>

<function name> ::= <identifier>F

Argumente können sein :

- Konstanten
- Flipflops
- Register oder Registerteile
- Busse oder Busteile
- Unterprogrammaufrufe

Eine Argumentliste stellt die durch Kommata getrennten Eingangsgrößen des Schaltnetzes dar. Tritt in der Argumentliste ein Unterprogrammaufruf auf, so heißt das nur, daß ein Teil der Eingangsleitungen durch Leitungen gegeben sind, die Ausgangsleitungen anderer Schaltnetze darstellen.

<argument list> ::= <argument>|<argument list>,<argument>

<argument> ::= <flipflop constant>|<register constant>|
<flipflop>|<register>|<register part>|
<memory register>|<bus>|<bus part>|
<subprogram reference>

2.1.5 AHPL-Ausdrücke

Konstante, Variable, Operatoren, Unterprogrammaufrufe und Klammern können zu AHPL-Ausdrücken zusammengefaßt werden.

Es gelten folgende Bildungsgesetze :

a) Ist a eine Konstante, ein symbolischer Name oder ein einfach oder zweifach indizierter Name, so ist a ein Ausdruck.

b) Ist a eine Register- oder Busausblendung, so ist a ein Ausdruck.

c) Jeder Funktionsaufruf ist ein Ausdruck.

d) Ist a ein Ausdruck, so ist auch die Folge <monadische AHPL Funktion> a ein Ausdruck.

e) Sind a und b Ausdrücke, so ist die Folge

a <dyadische AHPL Funktion> b ein Ausdruck .

f) Ist a ein Ausdruck, so ist auch (a) ein Ausdruck.

<right side primary> ::= <argument>|<memory>|<memory part>|
(<right side expression>)

<memory part> ::= $\left\{\text{<cut off vector>}\left\{/|//\right\}_1^1\right\}_1^4$ <memory>

<monadischer Term> ::= <monadische AHPL Funktion>
<right side expression>|
$\overline{\text{<right side expression>}}$

<dyadischer Term> ::= <right side primary><dyad. AHPL Funktion>
<right side expression>

<right side expression>::= <right side primary>|<monad. Term>|
<dyadischer Term>

In AHPL gibt es genau wie in APL keine Vorrangbeziehungen zwischen den verschiedenen Operatoren. Ein Ausdruck wird prinzipiell von rechts nach links interpretiert und abgearbeitet. Durch Klammersetzungen kann diese Interpretation umgangen werden.

Beispiel: IR_1 und IR_2 seien 2 Register gleicher Länge.

$$\wedge/\alpha^6/\omega^{12}/\underbrace{IR_1 \wedge IR_2}_{\text{Vektor}}$$

12-Bit Vektor

6-Bit Vektor

1 Bit

Mit gedachten Klammern lautet obiger Ausdruck

$$(\wedge/(\alpha^6/(\omega^{12}/(IR_1 \wedge IR_2)))) \quad .$$

Da die AHPL-Funktionen auf verschiedene Strukturen wirken können (Skalar, Vektor, Matrix), empfiehlt es sich, folgende Notation einzuführen :

<α α function> ≙ monadische Funktion, die auf ein Skalar angewendet wieder ein skalares Ergebnis liefert.

<α α α function> ≙ dyadische Funktion, die auf zwei Skalare angewendet wieder ein skalares Ergebnis liefert.

In der folgenden Schreibweise steht :

α für Skalar
β für Vektor
γ für Matrix

<scalar expression> ::= <scalar primary> |
<αα function><scalar expression>|
<scalar primary><ααα function><scalar expression>|
<αβ function><vector expression>

mit

<scalar primary> ::= <flipflop>|<flipflop constant>
<function reference>|(<scalar expression>)

<αβ function> ≙ monadische Funktion, die auf einen Vektor angewendet ein skalares Ergebnis liefert (z.B. Reduction)

desgleichen gilt :

<vector expression> ::= <vector primary>|<ββ function>
<vector expression>|<vector primary>
<βββ function><vector expression>|
<βγ function><matrix expression>|
<vector primary><ββγ operator>
<matrix expression>|
<scalar primary> <βαα function> <scalar expression>|
<scalar primary> <βαβ function> <vector expression>|
<vector primary> <ββα function> <scalar expression>

mit

<vector primary> ::= <register>|<memory register>|<register part>|
<bus>|<bus part>|<register constant>|
<subroutine reference>|(<vector expression>)

<ββ function> ≙ monadische Funktion, die auf einen Vektor angewendet wiederum ein Vektorergebnis liefert.
(z.B. Negation, Compression)

<βγ function> ≙ monadische Funktion, die auf eine Matrix angewendet ein Vektorergebnis liefert.
(z.B. RØw Reduction, Column Reduction)

desgleichen gilt :

<matrix expression> ::= <matrix primary>|
<γγ function><matrix expression>|
<vector primary><γβγ function>
<matrix expression>

mit

<matrix primary> ::= <memory>|<memory part>

Für die einzelnen meta-metasprachlichen Operatorsymbole gilt :

<αα function> ::= <boolean NØT>|
<cut off function>|<compress function>

<ααα function> ::= <boolean AND>|<boolean ØR>|
<exclusive ØR>|<relation operator>

<αβ function> ::= <reduce function>

<ββ function> ::= <boolean NØT>|<left rotate>|<right rotate>|
<left shift>|<right shift>

<βββ function> ::= <boolean AND>|<boolean ØR>
<exclusive ØR>|<relation function>|
<compress function>

<βγ function> := <row reduction>|<column reduction>

<ββγ function> := <row compression>|<column compression>

<βαα function> := <catenation>

<βαβ function> := <catenation>|<boolean AND>|<boolean ØR>|
<exclusive ØR>|<relation function>

<ββα function> := <catenation>|<boolean AND>|<boolean ØR>|
<exclusive ØR>|<relation function>

<γγ function> := <rotate up>|<rotate down>|<boolean NØT>

<γβγ function> := <catenate>|<row compression>|
<column compression>

3. Statementtypen

Vorgänge in einem Rechner, die durch AHPL beschrieben werden sollen, sind im wesentlichen unbedingte und bedingte Registertransfers. Es werden deswegen Transfers- und Branchstatements eingeführt.

Im Gegensatz zu herkömmlichen Programmiersprachen, deren Programmabläufe i.a. zeitlich rein sequentiell erfolgen, müssen in einer Hardwarebeschreibesprache Elemente enthalten sein, die zeitlich parallele Vorgänge in einem Rechner beschreiben.

Eingeführt werden deshalb in AHPL Sprachelemente, die eine simultane Abarbeitung einzelner Transfers (Simultanstatements) oder kompletter Transfersequenzen ermöglichen (Diverge- und Convergestatements).

3.1 Simple Transferstatements

Es hat folgende Form :

<simple transfer statement> := [<label>]<destination element> ← <right side expression>

Es bedeuten :

right side expression	≙	Rechtsseitige Quelle, die aus einem Hardwareelement (Flipflop, Register, usw.) oder aus einer Verknüpfung mehrerer Grundelemente entstanden ist.
destination element	≙	Name des Zielelementes, das den zu transferierenden Inhalt des rechtsseitigen Ausdrucks aufnehmen soll.
←	≙	Trennsymbol zwischen linker und rechter Seite
[<label>]	≙	Statementnummer; sie ermöglicht mit Hilfe von Branchstatements eine Änderung des sequentiellen Programmablaufes.

Bei einem Transferstatement muß gewährleistet sein, daß die Struktur und die Dimension auf Quell- und Zielseite identisch sind.

d.h. Skalar ← Skalarexpression

Vektor ← Vektorexpression

Matrix ← Matrixexpression

mit $(\rho \text{Vektor}) = \rho \text{Vektorexpression}$

$(\rho \text{Matrix} = \rho \text{Matrixexpression}$

Berücksichtigt man wieder die Struktur, so ergibt sich folgendes :

Ein skalares Zielelement kann ein skalarer Name (Flipflop), ein indizierter Vektorname, ein doppelt indizierter Speichername, ein durch Compression gewonnenes Flipflop eines Registers oder ein

Funktionsaufruf sein.

Ein vektorielles Zielelement kann ein Register oder Bus, ein zusammenhängender Register- oder Busteil, ein einfach indizierter Matrixname oder eine durch Reihen- oder Spaltencompression gewonnene Folge von Flipflop-Elementen sein. Ein durch Catenation entstandener Vektor und ein Subroutineaufruf sind ebenfalls auf der linken Seite zugelassen.

```
<scalar destination element> ::= <flipflop>|<vector selection element>|
                                 <function reference>

<vector selection element>   ::= <α-selection vector>/<register>

<α-selection vector>         ::= <subroutine reference>|
                                 <register constant>

<vector destination element> ::= <register>|<register part>|
                                 <memory register>|
                                 <bus>|<bus part>|
                                 <catenation vector>|
                                 <matrix selection element>|
                                 <subroutine reference>

<catenation term>            ::= <catenation primary>,
                                 <catenation primary>|
                                 <catenation term>,<catenation primary>

<catenation primary>         ::= <flipflop>|<register>|
                                 <memory register>|<register part>

<matrix selection element>   ::= <β-selection vector>{/|//}₁⁴<memory>

<β-selection vector>         ::= <register constant>|
                                 <subroutine reference>
```

Der α und β-Selection vector, der durch den Inhalt eines Registers, eines Registerteiles oder durch die Ausgangsleitungen einer Subroutine gegeben ist, darf nur eine einzige logische '1' enthalten, damit die Struktur für das Vector- und Matrixselectionselement erfüllt ist. Außerdem müssen die Komponentenanzahlen (Dimension) zwischen Selection vector und dem zu komprimierenden Vektor

bzw. Selection vector und der entsprechenden Kantenlänge der Matrix bei Zeilen- und Reihencompression identisch sein.

Eine Matrix als Zielelement kann ein kompletter Halbleiterspeicher, ein durch Zeilen- oder Reihencompression selektierter Halbleiterspeicherbereich sein, der ebenfalls durch einen Subroutineaufruf definiert werden kann.

<matrix destination element> ::= <memory>|<memory part>|
<subroutine reference>|
<γ-selection vector>$\{/|//\}_1^1$ <memory>

<γ-selection vector> ::= <subroutine reference>|
<register constant>

Damit die Strukturbedingung erfüllt wird, muß der γ-Selection vector mindestens mit zwei 1-Werten belegt sein.

Unter Berücksichtigung obiger Betrachtungen ergibt sich folgende Form für ein Simple Transferstatement :

<simple scalar transferstatement> ::= [<label>]<scalar destination element>
← <scalar expression>

<simple vector transferstatement> ::= [<label>]<vector destination element>
← <vector expression>

<simple matrix transferstatement> ::= [<label>]<matrix destination element>
← <matrix expression>

Alternative Transferstatement

Die Erweiterung des Simple Transferstatement zum Alternative - Transferstatement besteht darin, daß der zu transferierende Vektor aus einem rechtsseitigen Ausdruck bestimmt wird, der mehrere mögliche Transferelemente enthalten kann.

Die entsprechende Auswahl wird durch Bedingungen hergestellt, die entweder wahr oder nicht wahr sind, also Skalarausdrücke dargestellen. Dieser Statementtyp beschränkt sich auf vektorielle Transferelemente.

Die allgemeine Form lautet :

<alternative transfer statement> := <vector destination element> ←
<alternative transfer element>

alternative transfer statement := <conditional transfer element>|
<alternative transfer element>∨
<conditional transfer element>

conditional transfer element := (<vector expression>∧
<scalar expression<)|
(<scalar expression>∧
<vector expression>)

Das ØDER-Zeichen stellt das Trennsymbol zwischen den verschieden bedingten Transferalternativen dar. Das UND-Zeichen verbindet das zu transferierende Element mit einer skalaren Bedingung. Die verschiedenen Transferbedingungen müssen disjunkt sein.

Beispiel:

$$A \leftarrow ((\bar{b} \wedge c) \wedge B) \vee ((\bar{a} \wedge b \wedge c) \wedge (A \wedge B)) \vee ((a \wedge b \wedge \bar{c}) \wedge \bar{A})) \vee ((\bar{a} \wedge \bar{c}) \wedge \varepsilon(n)) \vee ((a \wedge b \wedge c) \wedge \overline{\varepsilon(n)})$$

a,b,c stellen die Kontrollvariablen dar.

B, $A \wedge B$, $\bar{A}$, $\varepsilon(n)$, $\overline{\varepsilon(n)}$ stellen die zu transferierenden Vektoren dar.

A ist der Name des Zielregisters mit n Komponenten.

a \ bc	00	01	11	10
0	$\varepsilon(n)$	B	$A \wedge B$	$\varepsilon(n)$
1	A	B	$\overline{\varepsilon(n)}$	$\bar{A}$

Obiges Bild gibt den Folgezustand von A an, je nachdem welchen Inhalt die Kontrollvariablen a,b,c haben.

- Der Vektor B wird transferiert, wenn $\bar{b} \wedge c$ wahr ist, unabhängig von Kontrollvariable a.

- Register A wird zu '1' gesetzt, wenn $\bar{a} \wedge \bar{c}$ wahr ist, unabhängig von b.

- Tritt der Fall $a \wedge \bar{b} \wedge \bar{c}$ als wahr auf, so bleibt der Inhalt von A unverändert, obwohl dies nicht explizit angegeben wurde. Dies stellt eine Abweichung von der APL-Semantik dar und sollte dahingehend geändert werden, daß auf der rechten Seite vollständige Ausdrücke stehen müssen, wenn die restlichen Kombinationen nicht in einem vorigen Transferstatement behandelt wurden. Ein APL-Interpreter würde für letztgenannte Kombination einen Nullvektor nach A transferieren.

- Ein komplettes Alternativ-Transferstatement enthält wieder eine in eckige Klammern gesetzte Statementnummer.

- Analog zum Simple Transferstatement gelten auch beim Alternative Transferstatement die Betrachtungen über die Struktur, d.h. linke und rechte Seite müssen gleiche Struktur (rank) und gleiche Dimension haben.

- Alternativen auf der Zielseite sollten generell ausgeschlossen sein, denn ein Ausdruck mit Werteffekt auf der linken Seite eines Statements ist in APL nicht vorhanden und würde in einer 'APL-like' Simulationssprache nur Verwirrung stiften.

Die einzigen Möglichkeiten in AHPL, ein variables Ziel zu definieren, sind im Indexing und durch den RAM-Speicherzugriff gegeben. In diesen Fällen wird aus mehreren Möglichkeiten ein bestimmtes Zielelement ausgewählt, das von bestimmten Indexparametern oder dem Inhalt des Memory-Adressregisters abhängig ist.

Die entsprechenden RAM-Lese- und Schreibestatements haben folgendes Aussehen :

<ram-write statement> ::= <memory>[<address vector>] ← <register>

<ram-read statement> ::= <register> ← <memory>[<address vector>]

<address vector> ::= <register>|<register part>

3.2 Branchstatements

Mit Hilfe von Branchstatements kann der rein sequentielle Ablauf eines Kontrollprogrammes unterbrochen werden.

Es wird zwischen bedingten und unbedingten Programmverzweigungen unterschieden.

Die allgemeine Form der unbedingten Verzweigung lautet :

<unconditional branch> ::= → (<label>)

<label> ::= <integer>

Der Pfeil nach rechts ist Erkennungssymbol für eine Verzweigung, die in Klammern gesetzte Statementnummer bestimmt das nächste auszuführende Statement (welches wiederum einen Branch darstellen kann).

Bedingte Verzweigungen können mit Hilfe von 3 Statementtypen definiert werden.

<conditional branch> ::= <computed goto branch>|<relation branch>
<function branch>

Ersteres hat die gleichen Eigenschaften wie das FØRTRAN-Computed-gøtø-statement.

(Diese Vermischung von Sprachelementen in AHPL, siehe auch Function und Subroutine Formalismen, sollte bei einer neuen Definition von AHPL vermieden werden, da die in APL enthaltenen Möglichkeiten ausreichen und in der Verzweigungstechnik teilweise sogar effektiver sind (Skipverzweigungen).

In der von Hill & Peterson (AHPL) definierten Fassung wird ein 'computed goto branch' durch eine Labelliste und einen Indexausdruck bestimmt. Der Indexausdruck, wird er berechnet, hat als Ergebnis einen Integerwert, der das entsprechende Verzweigungsziel aus der Labelliste auswählt.

<computed goto branch1> ::= → (<label list>)$_{\perp\text{<vector expression>}}$

<label list> ::= <label>,<label>|<label list>,<label>

Beispiel :

$$\rightarrow (20,14,16,18)_{\perp\alpha^2/IR}$$

$\perp\alpha^2/IR$ liefert als Ergebnis einen Integerwert, der zwischen 0 und 3 liegen kann, d.h. die ersten beiden Bits des Registers IR werden durch den DECØDE-Operator als Integerwert interpretiert.

Da in AHPL 'O origin' vereinbart wird, ist das Sprungziel durch das Statement mit der Nummer 20 bestimmt, wenn IR_0 und IR_1 beide den logischen Wert '0' haben.

Ein Computed goto branch sollte nur Verwendung finden, wenn mindestens 3 Sprungziele zur Verfügung stehen. Nach Möglichkeit sollte es jedoch vermieden werden, da für eine solch einfache Aussage die relativ große Übersetzungsarbeit nicht gerechtfertigt ist. Anders dagegen, wenn ein skalarer Indexausdruck vorliegt. Dies ist praktisch 1:1 in Hardware übersetzbar.

<computed goto branch2> ::= (<label list>)$_{\text{<scalar expression>}}$

Beispiel:

$$\rightarrow (10,27)_{IR_0 \wedge IR_1}$$

Der Skalarausdruck $IR_0 \wedge IR_1$ kann den logischen Wert '0' oder '1' haben. Im ersteren Falle wird das Statement 10, im zweiten Fall das Statement 27 aktiviert.

Ein Relationbranch wird durch zwei durch Doppelpunkt getrennte Vergleichsoperanden, eine in Klammern gesetzte Operatorenliste, den Branchpfeil und eine abschließende in Klammern gesetzte Labelliste bestimmt.

Die Operatorenliste enthält die Vergleichsoperatoren, die auf die zu vergleichenden Operanden angewendet werden soll, d.h. die Länge der Labelliste wird durch die Länge der Operatorenliste bestimmt.

Je nachdem ob die zu vergleichenden Operanden skalaren oder vektoriellen Charakter haben, wird unterschieden zwischen :

a)<skalarer relation branch>::= <scalar operand list>,
(<scalar operator list>)
→(<label list>)

mit

<scalar operand list> ::= <scalar expression>:<scalar expression>
<scalar operator list> ::= <relation operator>,<relation operator>

Das Komma zwischen Operanden- und Operatorliste dient als Trennsymbol.

Beispiele:

IR_4 : 1,(=,≠) → (6,9)

IR_4 : MD_0 , (=,≠) → (5,10)

$(\overline{v/AM} \vee \overline{v/BM})$: 1 , (=,≠) → (5,10)

Die Vergleichsoperanden stellen Ausdrücke dar, die ein skalares Ergebnis haben; diese können gleich oder ungleich sein, eine Labelliste der Länge drei ist also nicht sinnvoll und wird verboten.

b) <vektorieller relation branch>::= <operand list>,(<relation list>) → (<label list>)

mit

<operand list> ::= ⊥<vector expression>;{⊥<vector expression>| <integer>}$_1^1$

<relation list> ::= <relation operator>{,<relation operator>}$_1^2$

Die Operatorliste muß mindestens 2 und darf höchstens drei Vergleichsoperatoren enthalten.

Beispiele:

$\perp(\alpha^4/IR)$: 7 , $(=,\neq) \rightarrow (10,15)$

$\perp(\alpha^4/IR)$: 7 , $(<,=,>) \rightarrow (10,15,20)$

$\perp(\alpha^3/IR)$: $\perp(\alpha^3/MD)$, $(<,=,>) \rightarrow (10,15,20)$

Ein Functionbranch ist die wichtigste und am häufigsten auftretende Verzweigung in AHPL, da diese am einfachsten in Hardware übersetzt werden kann.

Ein Functionbranch hat folgende Form :

<function branch> ::= → (<branch alternative>)

<branch alternative> ::= (<mixed expression>)+(<mixed expression>)| <branch alternative>+(<mixed expression>)

<mixed expression> ::= <scalar expression>×<label>| <label>×<scalar expression>

Ein Skalarausdruck kann den Wert 0 oder 1 haben; ein gemischter Ausdruck ist das Produkt aus einem boolean Wert und einem Integerwert (Statementnummer), ist also Null oder gleich der Statementnummer. Ein Functionbranch sollte so formuliert werden, daß genau eine Statementnummer durch die rechte Seite geliefert wird.

Beispiel:

$$\rightarrow ((\overline{IR_{14}} \times 55)+(IR_{14} \times 58))$$

d.h. Wenn $IR_{14} = 1$ goto Statement 58

Wenn $IR_{14} = 0$ goto Statement 55

Jedes Branchstatement muß genau wie jedes Transferstatement mit einer in eckige Klammern gesetzten Statementnummer versehen sein.

<branch statement> := [<label>]{<unconditional branch>|
<conditional branch>}*

3.3 Simple Simultanstatement

Will man Vorgänge in einem Digitalrechner und speziell Transfers beschreiben, so muß man unterscheiden zwischen synchronen und asynchronen Transfers. Ein synchroner Transfer ist ein Transfer, der zu einem bestimmten Zeitpunkt initiiert und in einer bestimmten Zeit durchgeführt wird. Start- und Endzeitpunkt des Transfers werden durch eine rechnerinterne 'clock',oder anders ausgedrückt durch den Takt des Rechners bestimmt. Synchrone Transfers sind z.B. alle Transfers der typischen Arbeitsregister wie Akkumulator, Memory data- und Memory addressregister, Indexregister und zusätzliche Registerbänke oder Hardwarestacks.

Da auf der rechten Seite eines Transferstatements komplizierte Ausdrücke stehen können und diese eine gewisse Zeit benötigen, um im Rechner zur Verfügung zu stehen, wird in AHPL folgende Notation eingeführt :

- Ein synchrones Transferstatement ohne weiteren Kommentar besagt, daß der Transfer innerhalb einer Clockperiode abgeschlossen ist.

- Ein synchroner Transfer, dessen Quellelement auf der rechten Seite erst berechnet werden muß, kann einen durch Komma getrennten Kommentar erhalten, aus dem die Anzahl der Clockperioden für obige Berechnung ersichtlich ist.

Beispiel:

1,AC ← ADD(MD,AC), 4 DELAYS

d.h. ADD ist eine Subroutine, die einen Summenvektor aus den beiden Operanden des MD-Registers und des Akkumulators liefert. Dieser so entstandene Vektor, dessen Komponentenanzahl um eine Stelle größer als die der Operanden sein kann, soll wiederum in den Akkumulator und ein Überlaufbit 1 transferiert werden.

Die Zeit, die dafür notwendig ist, wird durch den Kommentar, also 4 Delays, d.h. 4 Clockperioden, bestimmt.

- Ein asynchroner Transfer ist z.B. ein Transfer, der zwischen der CPU und anderen autonomen Teilen des Rechners, z.B. dem Speicher unabhängig vom Zentraltakt erfolgt. Solche Transfers werden durch bestimmte Initiierungsschritte und Ready-Signale gekennzeichnet.

  Der Kommentar WAIT ist für den Hardwarecompiler ein Zeichen, daß der auszuführende Transfer asynchron erfolgen soll.

Beispiel:

MD ← $M^{\perp MA}$, WAIT	Lesen aus Speicher
$M^{\perp MA}$ ← MD , WAIT	Schreiben in Speicher

In AHPL kann durch ein einfaches Simultantransferstatement der gleichzeitige Transfer mehrerer Quellelemente zu mehreren Zielelementen beschrieben werden.

Beispiel:

A ← INC(A) ; B ← C ; E ← D

d.h. innerhalb einer Clockperiode sollen die Registerinhalte von D und C nach E und B und gleichzeitig das A-Register incrementiert werden.

Formal anders formuliert, jedoch mit gleicher Wirkung ist auch folgende Notation möglich :

A , B , G ← IND(A) , C , D

Diese beiden Möglichkeiten bestehen in AHPL, wenn alle Transfers synchronen Charakter haben.

Ist eines dieser Statements vom asynchronen Typ, so besteht folgende Möglichkeit, dieses zu formulieren :

[1] A ← INC(A); B ← C; D ← F, NØ DELAY

[2] MD ← $M^{\perp MA}$, WAIT

[3] -----

d.h. zeitlich parallel werden die drei synchronen Transfers und die Initiierung des asynchronen Transfers durchgeführt. Anschließend wird gewartet, bis der asynchrone Transfer beendet ist, erst dann wird Statement [3] abgearbeitet.

Bemerkung: Die Anzahl der synchronen Transfers in Statement [1] ist beliebig.

Nicht erfaßt von dieser Schreibweise wird der Fall, wenn 2 asynchrone Transferstatements vorliegen. Dafür werden später besondere Sprachelemente eingeführt.

Ein einfaches Simultanstatement hat also folgende Form :

<simple simultanstatement> ::= <synchronous transfer statement> ;
<synchronous transfer statement>|
<simple simultanstatement>
<synchronous transfer statement>

mit

<synchronous transfer statement> ::= <simple transferstatement>|
<alternative transferstatement>

Die zweite angedeutete Möglichkeit **sollte**
vermieden werden, da der Catenationoperator schon auf der linken Seite eines einfachen oder Alternativtransferstatements zugelassen ist und so zu Mehrdeutigkeiten führen würde.

3.4 HALT Statement

Das HALT Statement in AHPL entspricht dem CALL EXIT in FØRTRAN, d.h. für den Compiler oder Interpreter ist dies ein Zeichen, daß das in AHPL geschriebene Steuerprogramm vollständig abgearbeitet ist.

Formal besteht das HALT Statement aus der in eckige Klammern gesetzten Statementnummer und dem Schlüsselwort HALT. Im Gegensatz zum FØRTRAN CALL EXIT kann das HALT Statement beliebig oft in einem Steuerprogramm auftreten.

<halt statement> ::= [<label>] HALT

3.5 Simultane Kontroll-Programme

Will man in AHPL simultane Vorgänge beschreiben, die jeweils asynchronen Charakter haben, so reichen die bisherigen Sprachelemente nicht aus, da die zeitliche Synchronisation der einzelnen Schritte nicht mehr gewährleistet ist. In diesem Falle ist es notwendig, Verzweigungs- und Konvergenzstatements einzuführen.

Ein Verzweigungsstatement wird durch das Schlüsselwort DIVERGE, ein Konvergenzstatement durch das Schlüsselwort CØNVERGE bestimmt.

Beispiel:

Simultan sollen folgende Zweige abgearbeitet werden :

a) Lesen aus einem Speicher (Speicheradresse stehe im Akkumulator)

b) Asynchrone Addition eines Indexregisters und des Akkumulators mit anschließendem Rechtsshift um eine Stelle.

```
[1]  PC ← ω^13/AC
[2]  DIVERGE(A3,B3)
[A3] MA ← PC
[A4] MD ← M^⊥MA, WAIT
[A5] → (6)
[B3] AC ← ADD(AC,IA), WAIT
[B4] AC ← ↓AC
[6]  CONVERGE(A4,B4)
[7]      .
 .       .
 .       .
 .       .
```

Im Statement 1 wird die 13-Bit-Adresse in ein Hilfsregister PC transferiert.

Statement 2 ist ein Verzweigungsstatement, gekennzeichnet durch das Schlüsselwort und eine Labelliste, deren Elemente die Start-Statementnummern der verschiedenen simultanen Programmsequenzen

angeben.

In [A3] wird das Speicheradressregister geladen, in [A4] erfolgt der asynchrone Speicherzugriff. In [6] ist die erste Verzweigungssequenz beendet, und es wird zum Konvergenzelement gesprungen.

[B3] ist der Startpunkt der zweiten Verzweigungssequenz und beschreibt die asynchrone Addition von Akkumulator AC und Indexregister IA.

Wenn beide asynchronen Sequenzen beendet sind, wird die Kontrolle dem Statement [7] übergeben. Wenn im obigen Beispiel bekannt ist, daß eine der beiden Sequenzen weniger Zeit benötigt als die andere, so kann erstere durch die Schlüsselwörter DEAD END gekennzeichnet werden und das Konvergenzelement entfallen.

Falls im obigen Beispiel der Speicherzugriff zeitlich aufwendiger als die asynchrone Addition ist, lautet das entsprechende AHPL-Programm :

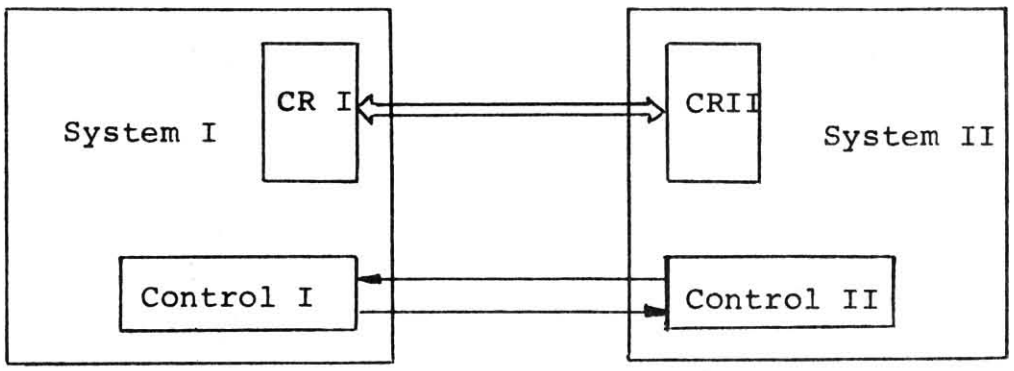
```
[1]  PC ← ω^13/AC
[2]  DIVERGE(A3,B3)
[A3] AC ← ADD(AC,IA), WAIT
[A4] AC ←↓AC, DEAD END
[B3] MA ← PC
[B4] MD ← M^⊥MA, WAIT
[6]       .
 .        .
 .        .
 .        .
```

Bei der Beschreibung simultaner Vorgänge müssen folgende Regeln beachtet werden :

a) Innerhalb einer Clockperiode darf ein einziges Register nicht das Ziel mehrerer Transfers darstellen.
b) Der Inhalt eines Registers darf nicht die Quelle eines Transfers darstellen, wenn es gleichzeitig das Ziel eines asynchronen Transfers darstellt.

3.6 Kommunikation zwischen mehreren autonomen Kontroll-Einheiten

Zur Beschreibung simultaner Vorgänge in einem Rechner ist es zweckmäßig, die einzelnen Sequenzen als selbständige Kontrolleinheiten aufzufassen und zu beschreiben. Die vorhandenen Grundelemente (Register, Flipflops, usw.) werden dann wie lokale Variable in einem Unterprogramm herkömmlicher Programmiersprachen bestimmten autonomen Einheiten zugeordnet. Um zu erreichen, daß zwischen diesen Einheiten Aktivitäten ausgetauscht werden können, werden Datenkommunikationsregister und Pulskommunikationslinien eingeführt.



Die Kommunikationsregister CRI und CRII können in beiden Kontrollsequenzen Control I und Control II verwendet werden (globale Variablen).

Pulskommunikationslinien werden vorgesehen, um Kontrollpulse von System zu System zu senden oder zu empfangen.

Kommunikationspulse zwischen autonomen Kontrolleinheiten werden benötigt, um folgende Aktivitäten auszulösen :

a) ein System solange zu unterbrechen, bis ein entsprechendes Readysignal den Haltekreis wieder auflöst.

b) Senden eines Signales, das ein unterbrochenes System wieder initiiert.

Beispiel a

```
    .
    .
    .
[5]  A ← B
[6]  WAIT FØR Y-PULSE
[7]
    .
    .
    .
```

Im Statement [6] wird die Kontrollsequenz unterbrochen, nachdem der Transfer im Statement [5] ausgeführt wurde. Trifft ein Puls auf der Puls-Kommunikationslinie Y ein, wird die Kontrolle Statement [7] übergeben.

Beispiel b

```
    .
    .
    .
[4]  LINE Y ← PULSE
[5]         .
            .
```

Im Statement [4] wird ein Puls auf die globale Pulskommunikationslinie LINE Y gegeben. Danach wird die Kontrolle Statement [5] übergeben, während der Puls auf der Y-Linie in einem anderen System einen Haltekreis auflösen kann.

Beispiel:

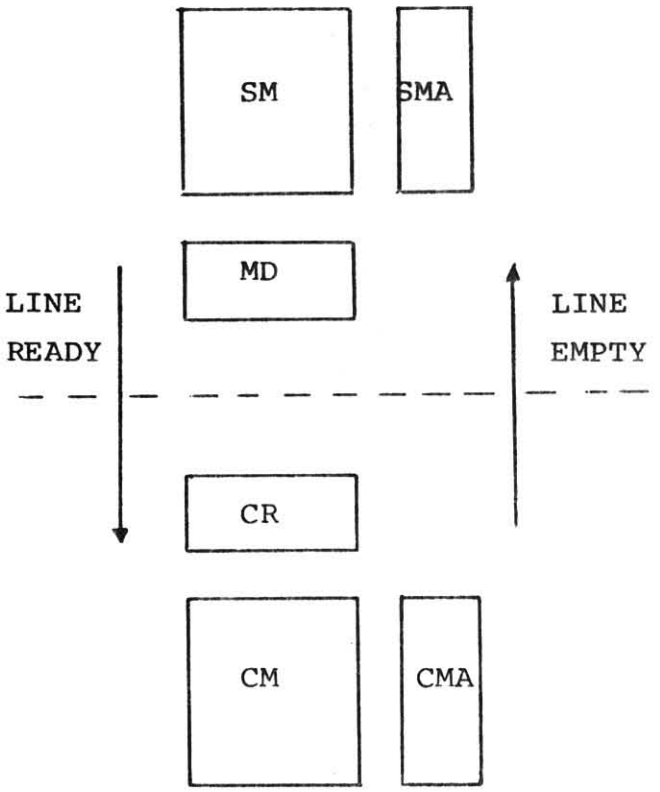
Gesucht ist die AHPL-Kontrollsequenz, die einen Transfer einer Serie von Vektoren aus einem Halbleiterspeicher SM eines autonomen Systems in einen langsameren Massenspeicher CM eines anderen autonomen Systems bewirkt. Ein Vektor des Halbleiterspeichers kann innerhalb einer Clockperiode gelesen werden.

System 1

[1] MD ← SM$^{\perp SMA}$
[2] LINE READY ← PULSE
[3] SMA ← INC(SMA)
[4] WAIT FØR EMPTY-PULSE
[5] → (1)

System 2

[1] WAIT FØR READY-PULSE
[2] CR ← MD
[3] CM$^{\perp CMA}$ ← CR, WAIT
[4] LINE EMPTY ← PULSE
[5] CMA ← INC(CMA)
[6] → (1)



Im sendenden System 1 ist das Tor zum Halbleiterspeicher durch das Memory Data Register MD und das Semiconductor Memory Address-register SMA gegeben. Im empfangenden System 2 stellt das Kommunikationsregister CR mit dem Addressregister CMA das Tor zum Massenspeicher dar.

Ein Puls auf der Linie READY sagt dem empfangenden System, daß ein Vektor im MD-Register zur Verfügung steht.

Ein Puls auf der Linie EMPTY sagt dem sendenden System, daß das Kommunikationsregister auf der empfangenden Seite frei ist. Während ein Vektor von MD nach CR transferiert wird, wird simultan die Adresse im SMA inkrementiert; ebenso wird die Adresse im CMA inkrementiert, während simultan aus dem Halbleiterspeicher ein Vektor nach MD gebracht wird.

Formal lassen sich die zuletzt eingeführten AHPL-Statements wie folgt beschreiben :

```
<diverge statement>   ::=  [<label>] DIVERGE (<branch label list>)
<converge statement>  ::=  [<label>] CØNVERGE (<branch label list>)
<branch label list>   ::=  <branch label>,<branch label>|
                           <branch label list>,<branch label>

     <branch label>   ::=  <letter><digit>|
                           <branch label><digit>
```

Um simultane Kontrollsequenzen auch optisch gegenüber einer sequentiellen Kontrollfolge hervorzuheben, werden die Statementnummern jeweils eines Kontrollzweiges mit einem führenden Buchstaben versehen.

Das 'control disable' statement wird durch die Schlüsselwörter WAIT FØR PULSE, das 'control enable' statement durch die Schlüsselwörter LINE PULSE gekennzeichnet.

```
<control disable>     ::=  [<label>] WAIT FØR  <line identifier>
                           PULSE

<control enable>      ::=  [<label>] LINE  <line identifier>
                           ← PULSE

<line identifier>     ::=  <identifier>
```

Damit sind alle in einem AHPL-Kontrollprogramm möglichen Statements beschrieben und es ergibt sich :

```
<control statement>   ::=  <simple transfer statement>|
                           <alternative transfer statement>|
                           <ram read statement>|
                           <ram write statement>|
                           <unconditional branch statement>|
                           <conditional branch statement>|
                           <halt statement>|
                           <diverge statement>|
                           <converge statement>|
                           <control enable>|
                           <control disable>|
<control program>     ::=  <control statement><control statement>|
                           <control program><control statement>
```

4. Übersetzung in Hardware

Dieses Kapitel betrifft AHPL nur als Entwurfssprache. Es wird hier die Hardware-Übersetzung der in den bisherigen Kapiteln eingeführten Statements vorgestellt, so wie sie der Hardware-Compiler "generiert". Dabei entspricht i.a. ein AHPL-Statement einer genau feststehenden Hardware, d.h., eine 1-1-Übersetzung ist gegeben. Auf Abweichungen davon wird im folgenden an den entsprechenden Stellen genauer eingegangen werden.

4.1 Grundelement eines Controllers

Aus einem Control-Programm geht nicht nur der Aufbau des Controlers hervor, sondern auch die Konfiguration der durch diesen Controler gesteuerten Flipflops, Register, usw. Obwohl also Controler und Register-Logik parallel generiert werden, soll hier der Übersicht wegen zunächst der Controler behandelt werden, danach die Register-Logik.

Die Funktion eines Controlers besteht darin, zur richtigen Zeit auf den richtigen Leitungen Pulse oder Levels in eine arithmetische Einheit und in alle an ihn angeschlossenen Einheiten eines Computers zu leiten.

Als Grundelement dazu dient das Delay-Element. Dieses Element liefert eine "1" am Ausgang, wenn es einen Maschinentakt zuvor am Eingang eine "1" erhalten hat. Dabei muß unterschieden werden zwischen Puls - eine "1", die nur für die Dauer eines Maschinentaktes existent ist - und Level - eine "1", die bereits während der vorangegangenen Pause des Maschinentaktes existiert.

Entsprechend unterscheidet man zwischen Pulse-Controler und Level-Controler. Im folgenden wird zunächst der Pulse-Controler veranschaulicht und danach auf die Hardware-Veränderungen bei Benutzung eines Level-Controlers eingegangen.

Pulse-Controler

Bild 2a zeigt den Aufbau eines Pulse-Delay-Elementes. Für dieses Delay-Element wird im folgenden das schematische Symbol nach Bild 2b benutzt.

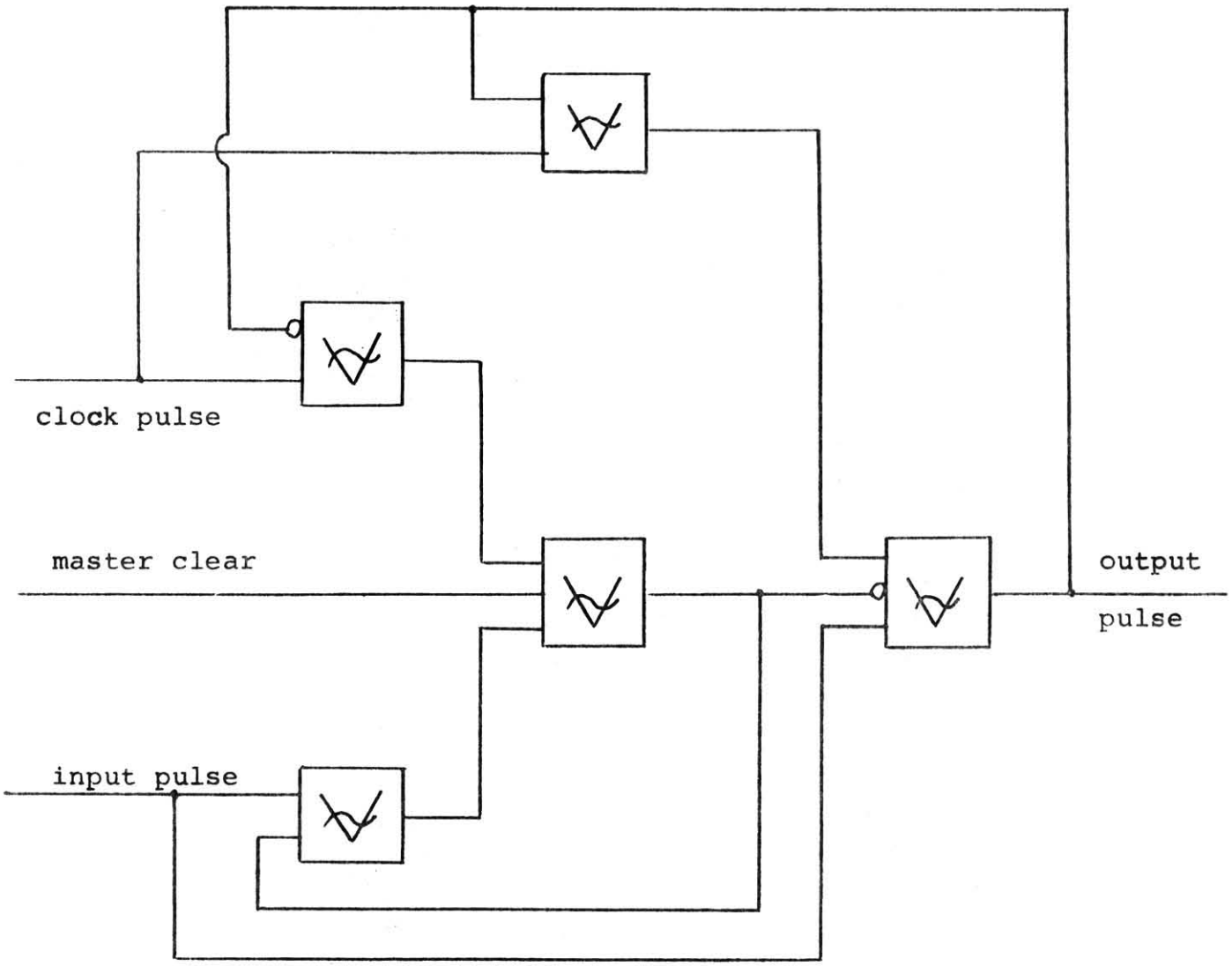


Bild 2a

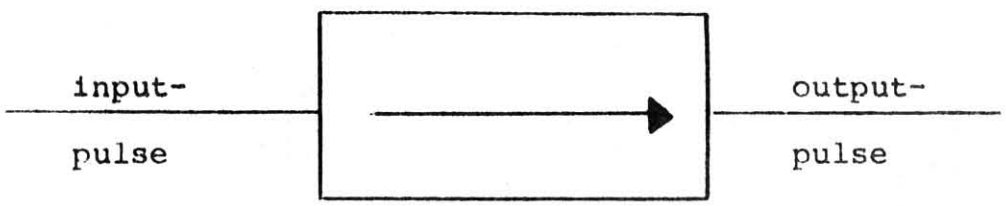


Bild 2b

Bemerkung: Bei Bildern werden Gatter durch Quadrate mit dem entsprechenden Funktionszeichen dargestellt. Negationen, die außer den implizit in den Gattern vorhandenen notwendig sind, werden im allgemeinen als Punkte an den Gattereingängen dargestellt.

Es wurde in Kapitel 3.3 zwischen synchronen und asynchronen Transfers unterschieden. Bei synchronen Transfers wurde dabei eine nicht-negative Integerzahl im Kommentarteil gefordert, welche angibt, wieviel Delay-Elemente notwendig sind, um einen Transfer durchzuführen. Ein Statement ohne Kommentarteil wie

A ← B

d.h. innerhalb einer clock-Periode kann dieser Transfer durchgeführt werden, bewirkt folgende Controler-Übersetzung (Bild 3): Von einer Leitung, die einen Puls führt, zweigt eine Leitung ab, die den Transfer A ← B ausführt. Diese pulsführende Leitung bildet den Eingang eines Delay-Elementes, an dessen Ausgang der Puls also nach einem Takt wieder erscheint.

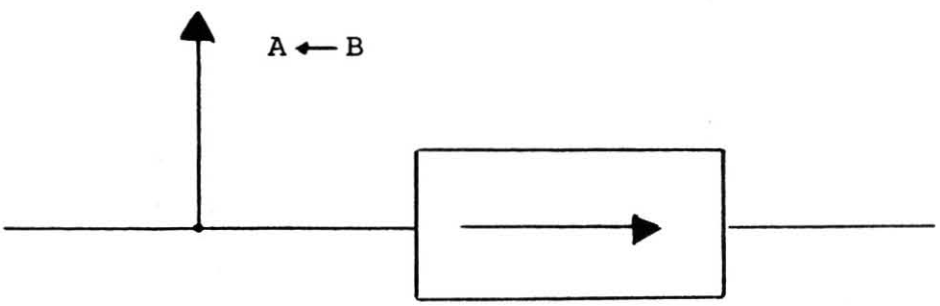


Bild 3

Synchrone Transfers, die mehrere Takte dauern, erfordern entsprechend mehrere Delay-Elemente im Controler.

A ← B, 3 DELAYS

z.B. hat die Übersetzung nach Bild 4.

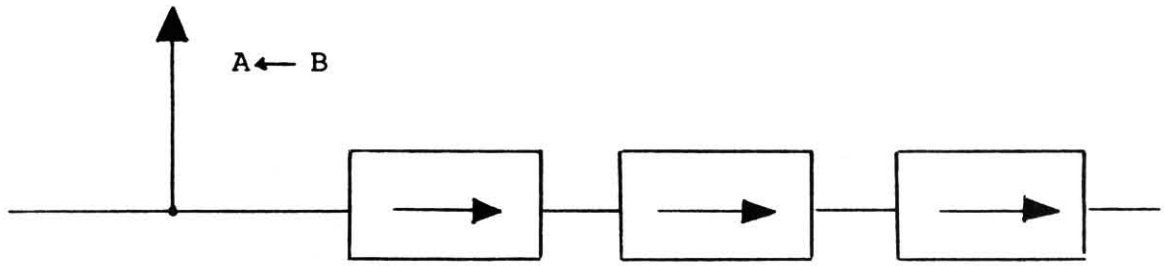


Bild 4

Dabei führt die abzweigende Leitung wieder den Transfer aus; anschließend wird der ausführende Puls um drei Takte verzögert, bevor ein weiterer Transfer ausgeführt werden kann.

Ein synchroner Transfer mit dem Kommentarteil NO DELAY hat dementsprechend die Übersetzung nach Bild 5. Er besteht also lediglich aus einem Leitungsstück und der abgehenden Leitung, die den Transfer ausführt.

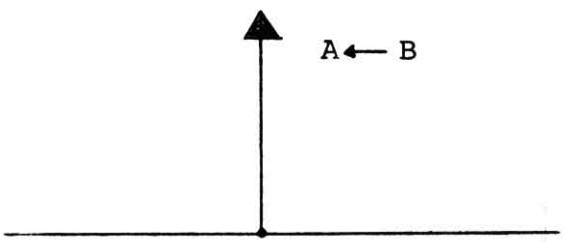


Bild 5

Eine derartige Schreibweise hat nur einen Sinn im Zusammenhang mit weiteren Statements. Die Statementfolge

```
A ← B, NO DELAY
C ← D
```

entspricht dem Controler-Ausschnitt gemäß Bild 6. Wie auch im folgenden ist hier an den abgehenden Leitungen notiert, welchen Transfer sie ausführen.

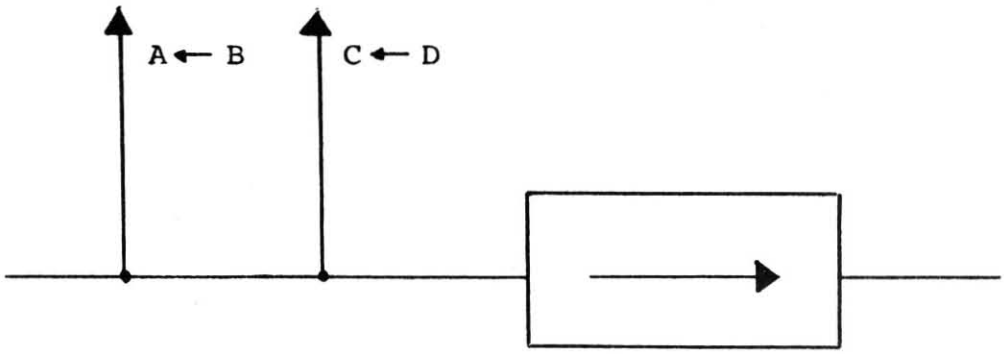


Bild 6

Die Notwendigkeit eines synchronen Transfers ohne Delay-Element wird allerdings erst einsichtig in Verbindung mit Branch-Statements (siehe unten).

Der Controler-Ausschnitt aus obigem Beispiel hätte auch erreicht werden können durch ein Simple Simultanstatement :

A ← B ; C ← D

Der Unterschied zwischen einfachen Transferstatements und einfachen Simultanstatements liegt für den Controler allein darin, daß beim letzteren statt einer einzigen, mehrere Leitungen vom gleichen Leitungsstück wegführen und einen Transfer bewirken.

Beim asynchronen Transfer leitet der Controler genau wie beim synchronen einen Initiierungs-Puls zu dem entsprechenden asynchronen Schaltnetz. Nach unbestimmter Zeit erhält er von dieser Einheit einen sogenannten Completion Pulse als Zeichen für die Beendigung des Transfers. Dieser Completion Pulse erfolgt nicht taktgleich mit der Clock des Controlers und muß daher synchronisiert werden. Ein solches Synchronisierelement zeigt Bild 7a; im folgenden wird es wie Bild 7b vereinfacht dargestellt.

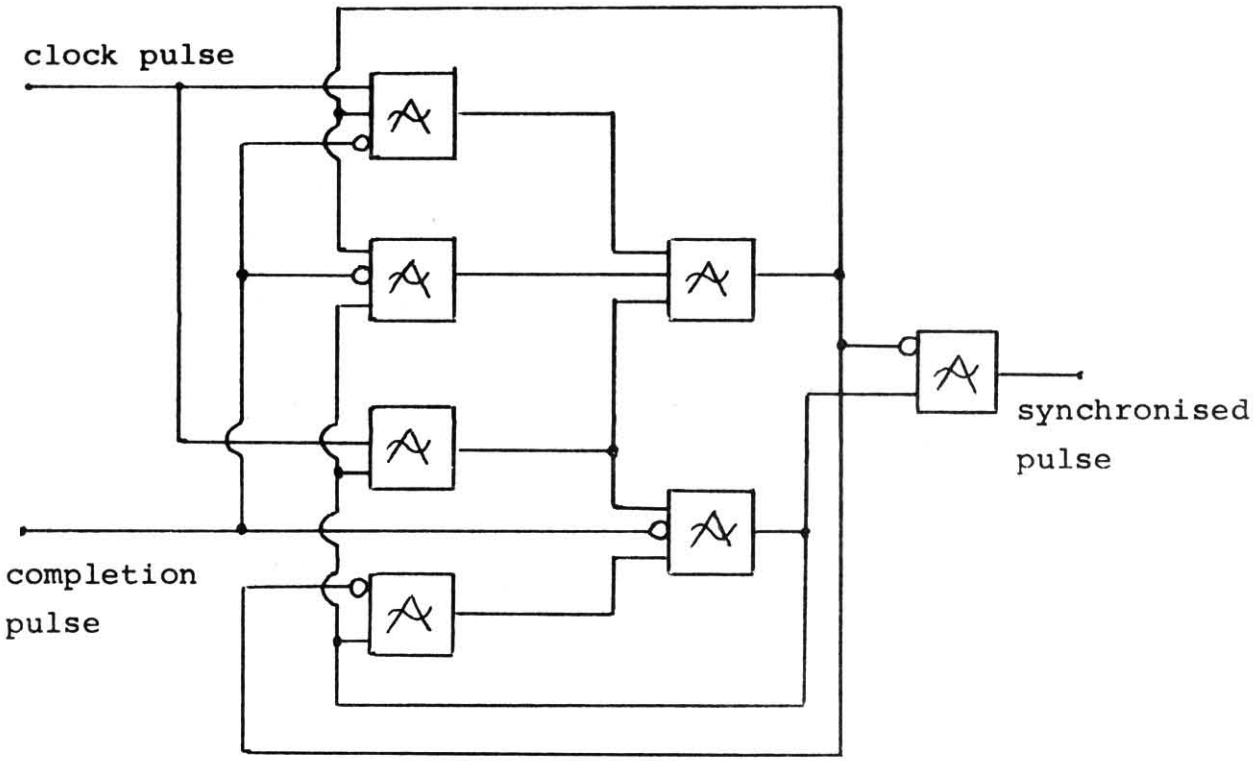


Bild 7a

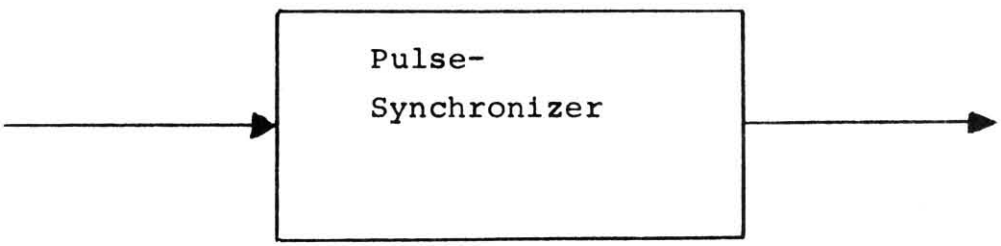


Bild 7b

Das Lesen eines Kernspeicher wortes, dessen Adresse im Register MA steht, in das Register MD mit Hilfe des Kernspeicher-lese-Statements

$$MD \leftarrow M^{\perp MA}, \text{ WAIT}$$

ist z.B. ein typischer asynchroner Transfer.

Der Controler-Ausschnitt könnte also aussehen, wie Bild 8a zeigt. Der Ausgang des Synchronisierelementes, der in jedem Falle synchron mit der Controler-Clock ist, wird dabei zur Steuerung des nächsten Statements verwendet.

Dies ist aber nur dann der Fall, wenn der betreffende asynchrone Transfer - hier das Kernspeicher-lese-Statement - nur von einer einzigen Stelle des Controlers angesprochen wird und folglich nur an eine einzige Stelle einen Completion Pulse leitet. Dies trifft jedoch im allgemeinen - vor allem beim Kernspeicherlesen und -schreiben - nicht zu; der Completion Pulse wird vielmehr gleichzeitig an mehrere Stellen im Controler geleitet werden.

Die Hardware-Übersetzung des obigen Beispiels muß dann aussehen wie Bild 8b.

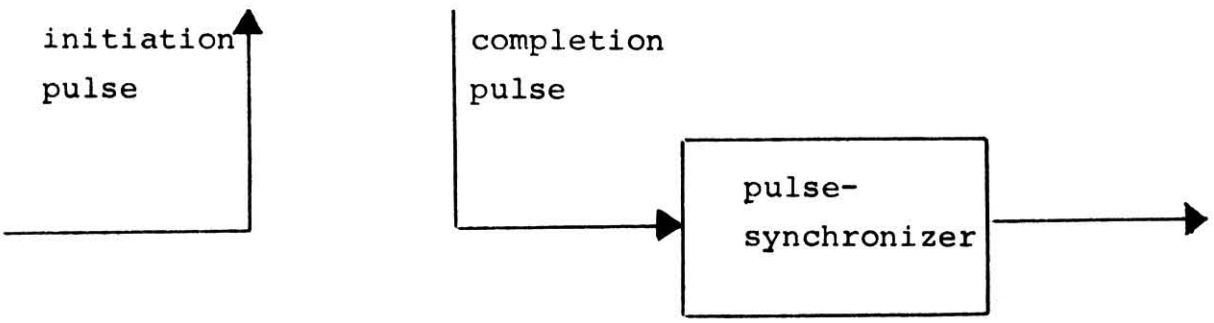


Bild 8a

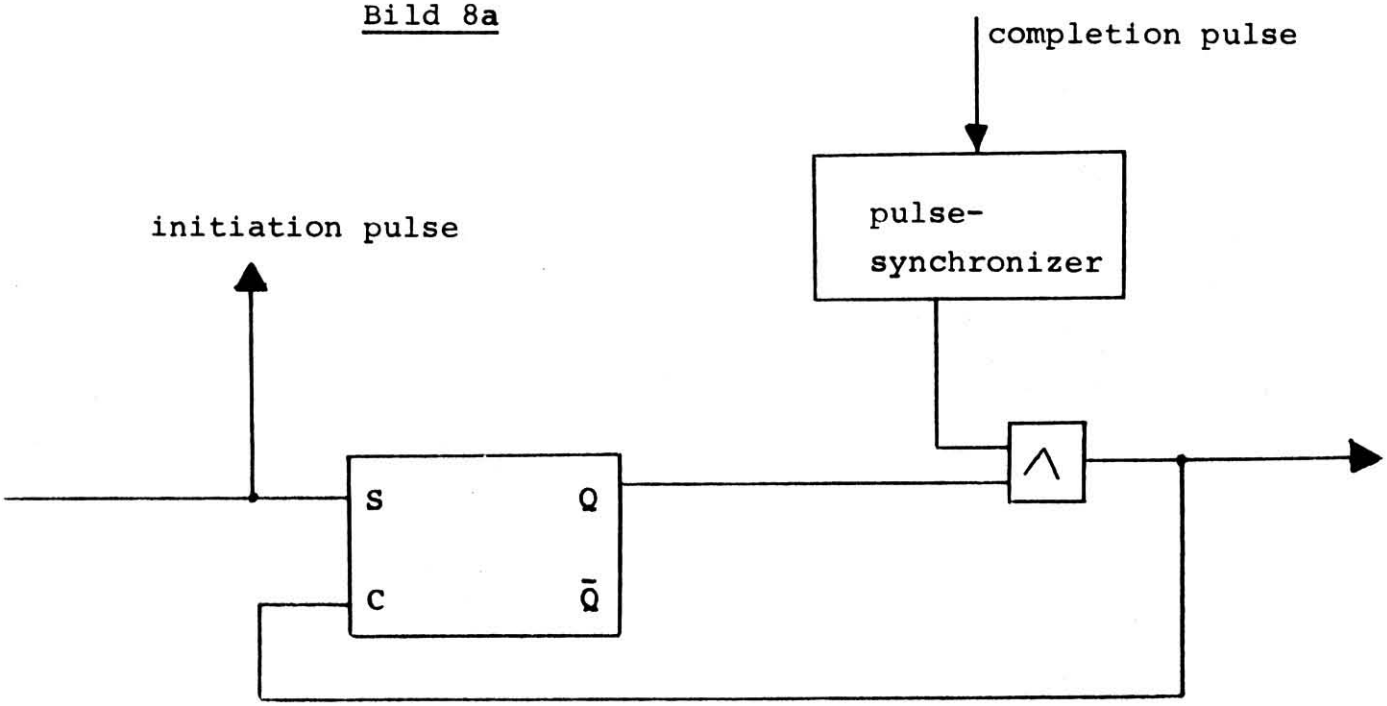


Bild 8b

Wir haben hier also einen ersten Fall, in dem die Hardware-Übersetzung aus einem Statement nicht eindeutig zu entnehmen ist, sondern nur durch ein Überschauen des gesamten Control-Programmes und durch Abzählen ermöglicht wird.

Ein analoger Fall dazu taucht bei Branchestatements auf, die ein Verzweigen innerhalb des Controlers zulassen. Dabei wird - gleichgültig ob bedingt oder unbedingt - zu einer Stelle im Controler gesprungen, die im allgemeinen - aber nicht notwendigerweise - zumindest auch über den direkten Vorgänger des entsprechenden Statements erreicht werden kann. An der Zielstelle muß also in der Control-Leitung ein OR-Gatter vorhanden sein und zwar mit einer solchen Anzahl von Eingängen, wie das entsprechende Gatter mögliche Vorgänger haben kann. Diese Zahl aber ist ebenfalls nur durch Abzählen im gesamten Control-Programm herauszufinden.

Eine unbedingte Verzweigung wie

⟶ (LABEL)

stellt also lediglich eine Leitungsverbindung her vom Ausgang des letzten Statements bis zu einem freien Eingang eines OR-Gatters vor dem Statement, das die Marke LABEL trägt (Bild 9).

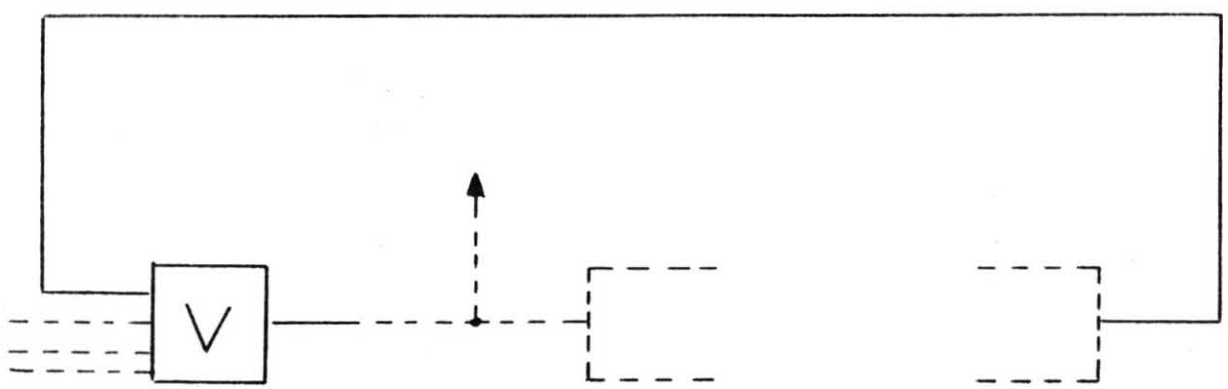


Bild 9

Eine bedingte Verzweigung wie zum Beispiel

$$\longrightarrow (L1 \times \bar{A}) + (L2 \times (A \wedge B)) \quad ,$$

die nicht vollständig ist, muß natürlich hardware-mäßig vervollständigt werden. Es wird also übersetzt :

$$\longrightarrow (L1 \times \bar{A}) + (L2 \times (A \wedge B)) + (NEXT \times (A \wedge \bar{B}))$$

Dabei bezeichnet "NEXT" das nächste Statement, das der nicht vollständigen Verzweigung folgt.

In der Hardware stellen die Leitungen von den Flipflops A und B auf die bezeichneten Gatter diese Verzweigung dar. Die Ausgänge dieser Gatter liegen dann zusammen mit dem Ausgang des vorangegangenen Statements auf AND-Gattern. Die Ausgänge dieser Gatter wiederum führen vor die entsprechenden Ziel-Statements der Verzweigung (Bild 10a).

Allgemein ist eine solche Übersetzung nicht gerade optimal und kann vom Hardware-Compiler minimisiert werden. Obiges Beispiel hat dann das Aussehen von Bild 10b.

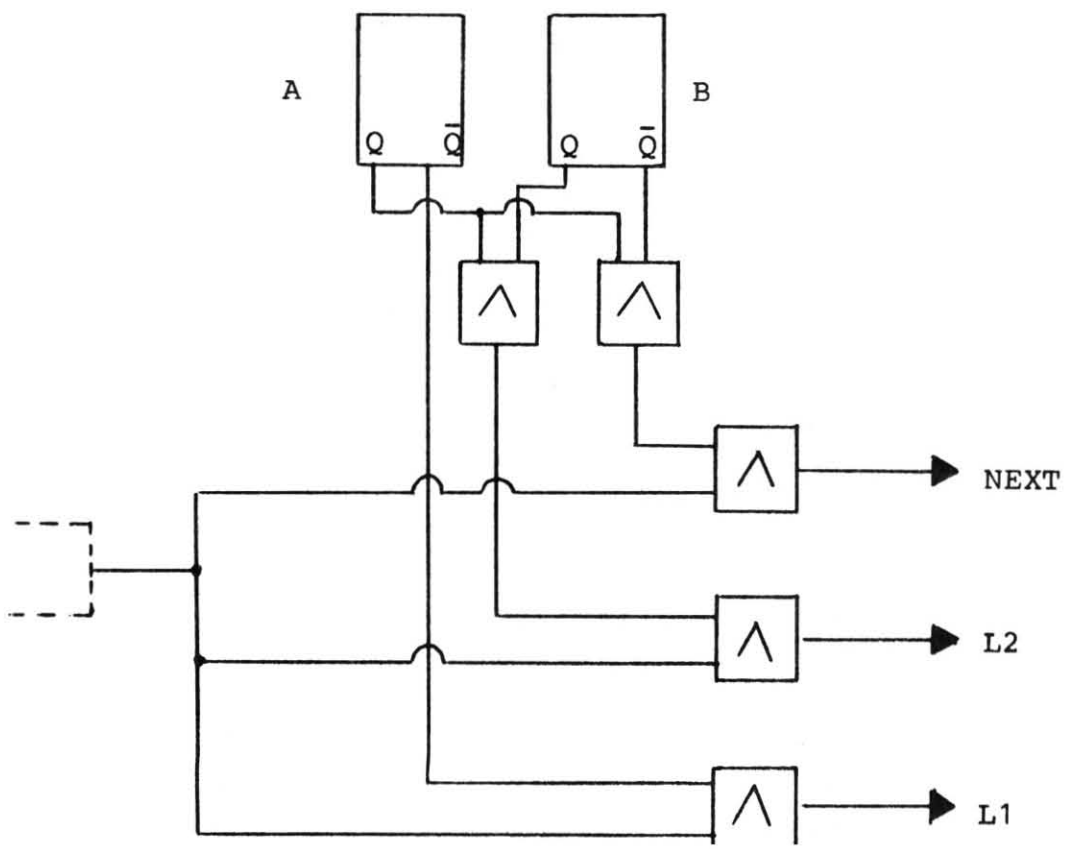


Bild 10a

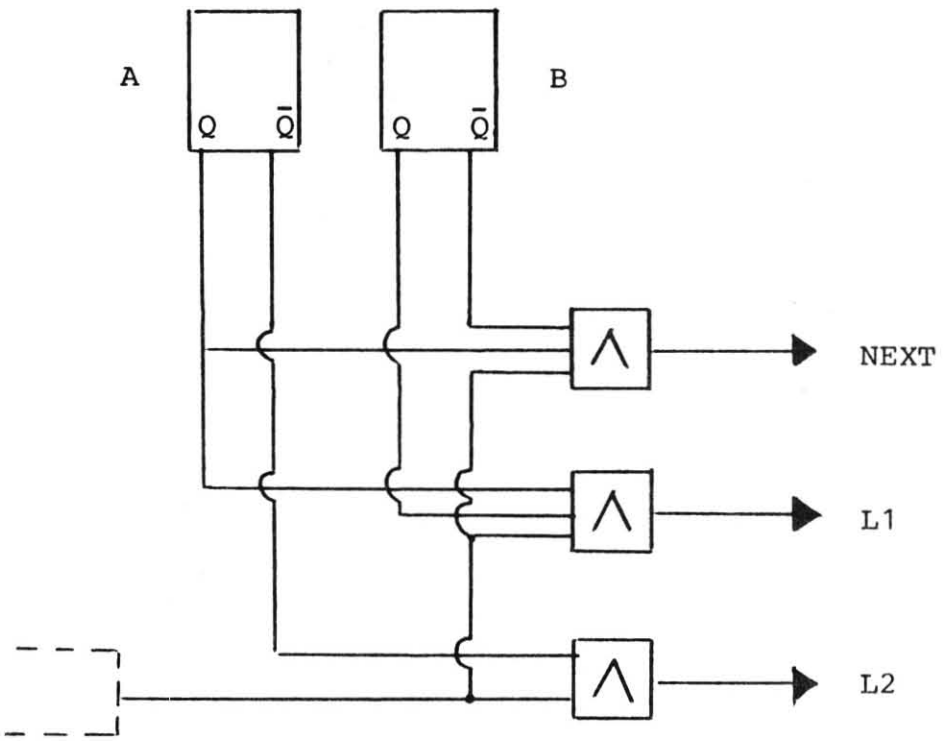


Bild 10b

Das HALT Statement ist einem Sprung in eine nicht definierte Zeile gleichzusetzen. Für die Hardware bedeutet dies, daß ein Controler "ausläuft", d.h. daß ein Puls an eine Stelle gelangt, an der er während der Pausen der Clock nicht mehr in einem Delay-Element gespeichert wird und somit verschwindet. Normalerweise bringt dies dann die Inaktivität eines Controlers mit sich.

Es gibt jedoch auch die Möglichkeit, daß mit Hilfe eines Diverge-Statements mehrere Pulse in einem Controler Aktivitäten auflösen. In diesem Falle verschwindet durch ein HALT Statement natürlich nur ein Puls von mehreren möglichen.

Ein solches Diverge-Statement hat folgendes Aussehen :

DIVERGE (L1, L2, ..., L_n)

L1 .... L_n sind dabei Marken, was einer unbedingten Mehrfachverzweigung innerhalb des Controlers entspricht. Die abgehenden Leitungen stellen also keine Initiierungspulse, durch die ein Transfer bewirkt wird, dar (Bild 11).

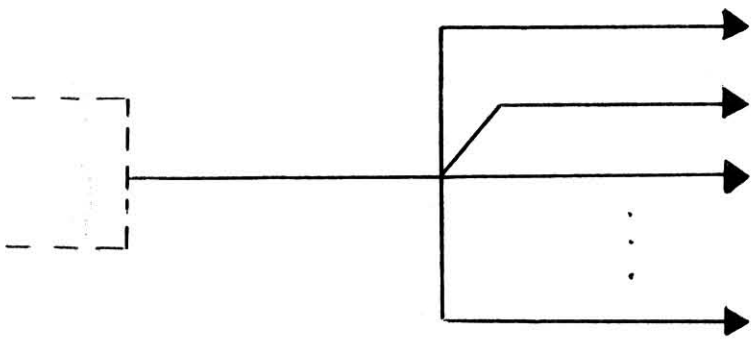


Bild 11

Entsprechend dem Diverge-Statement gibt es ein Converge-Statement, das mehrere Pulse innerhalb eines Controlers wieder zu einem vereinigt. Die Anzahl der zu vereinigenden Pulse muß dabei nicht mit der Anzahl der bei einem Diverge verzweigten Pulse übereinstimmen, sondern sie kann sowohl größer als auch kleiner sein; es ist nämlich durchaus möglich, daß ein Ast durch ein weiteres Diverge-Statement weiter aufgesplittet worden ist, oder durch HALT Statements Pulse zum "Auslaufen" gebracht worden sind.

Natürlich können aus elektrischen wie aus logischen Gründen die Leitungen nicht einfach vereinigt werden; durch Laufzeitunterschiede der einzelnen Äste würden dadurch mehrere hintereinanderlaufende Pulse erzeugt. Um dies zu vermeiden, bedient man sich eines speziellen Converge-Elementes, das m Flipflops entspricht,

deren Q-Ausgänge auf einem m-Input AND-Gatter liegen.

Das Statement

CONVERGE (M1, ... , Mm)

soll hier jedoch nur als Blockschaltbild nach Bild 12 dargestellt werden.

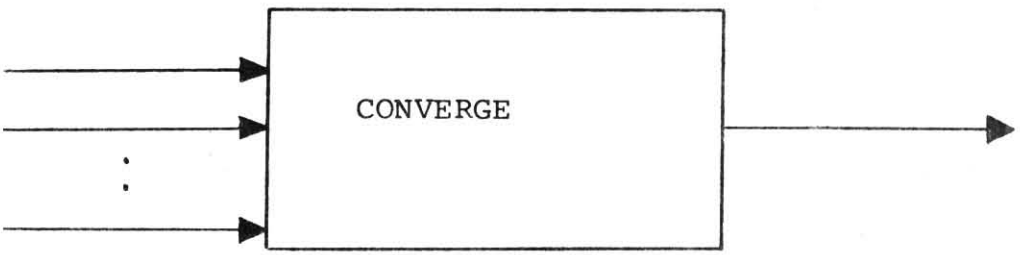


Bild 12

Treten in einem AHPL-Programm Pulskommunikationslinien auf zwischen mehreren autonomen Kontrolleinheiten, die über verschiedene Taktfrequenzen verfügen, so müssen die von einem in das andere System gesendeten Pulse im empfangenden System synchronisiert werden. Dies erfolgt in analoger Weise wie bei der Synchronisation asynchroner Transfers. In Bild 13 stellt Kontrolleinheit I das sendende, Kontrolleinheit II das empfangende System dar. Die globale Pulskommunikationslinie habe den symbolischen Namen X.

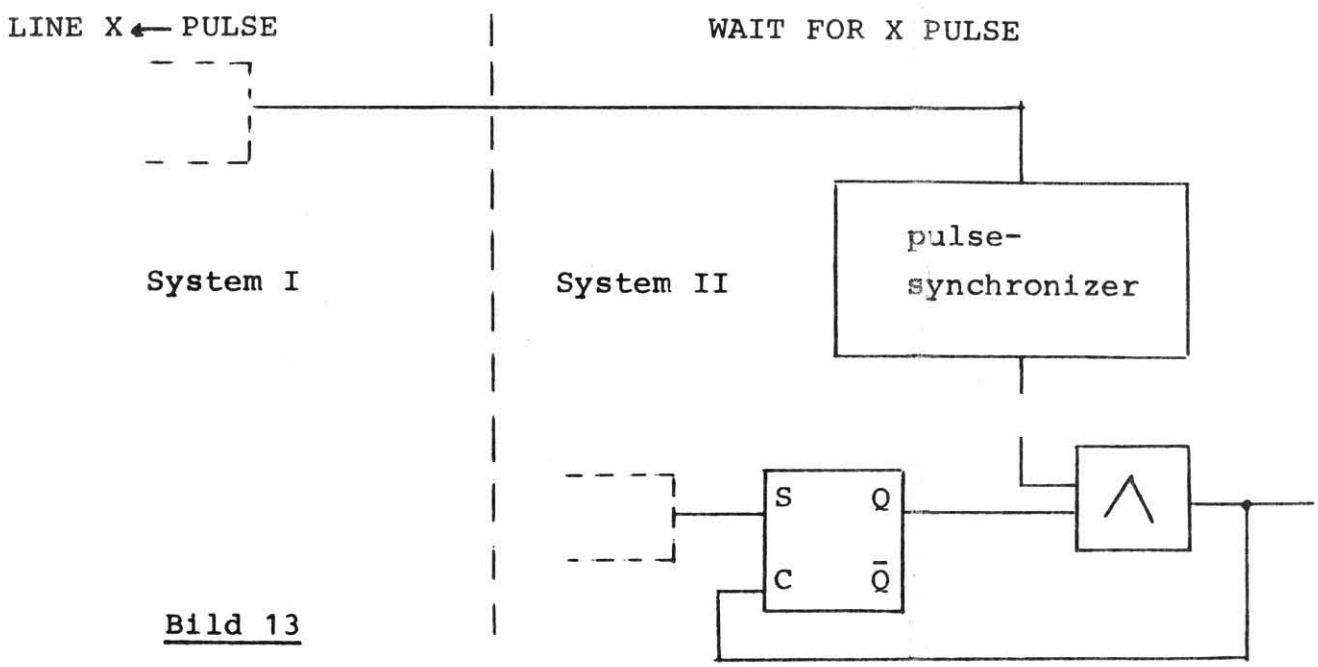


Bild 13

4.2 Register Logik

Den folgenden Register-Logik-Betrachtungen liegt zunächst ein Pulse-Controler zugrunde. Die Flipflops, die bei einem solchen Pulse-Controler benutzt werden, sind ungetaktete SC-Flipflops.

Ein Datentransfer zwischen zwei Registern wird als sogenannter "Jam-Transfer" ausgelegt. Dabei liegen die Q-Ausgänge des Quellregisters über AND-Gattern auf den S-Eingängen des Zielregisters, die $\bar{Q}$-Ausgänge auf den C-Eingängen. Den zweiten Eingang aller AND-Gatter belegt eine Pulsleitung, die aus dem Controler kommt. Auf dieser Leitung wird ein Puls geführt, wenn der entsprechende Transfer ausgeführt werden soll.

Bild 14 zeigt den Transfer

BR ← AR

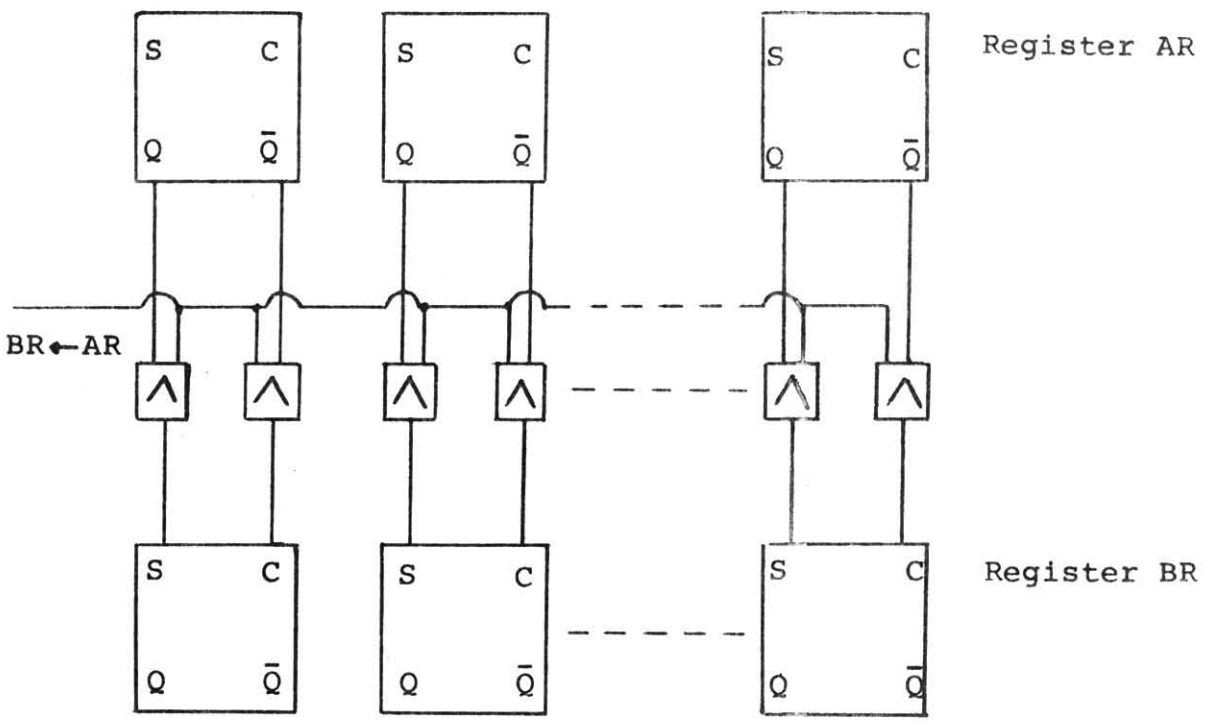


Bild 14

Soll BR auch Ziel anderer Transfers sein, so müssen vor den Eingängen von BR zusätzlich OR-Gatter vorhanden sein, und zwar mit so vielen Eingängen, wie BR Ziel von Transfers aus verschiedenen Registern sein kann (Bild 15). Die Anzahl dieser Transfers ist wiederum nur durch Auszählen im gesamten Control-Programm zu erhalten.

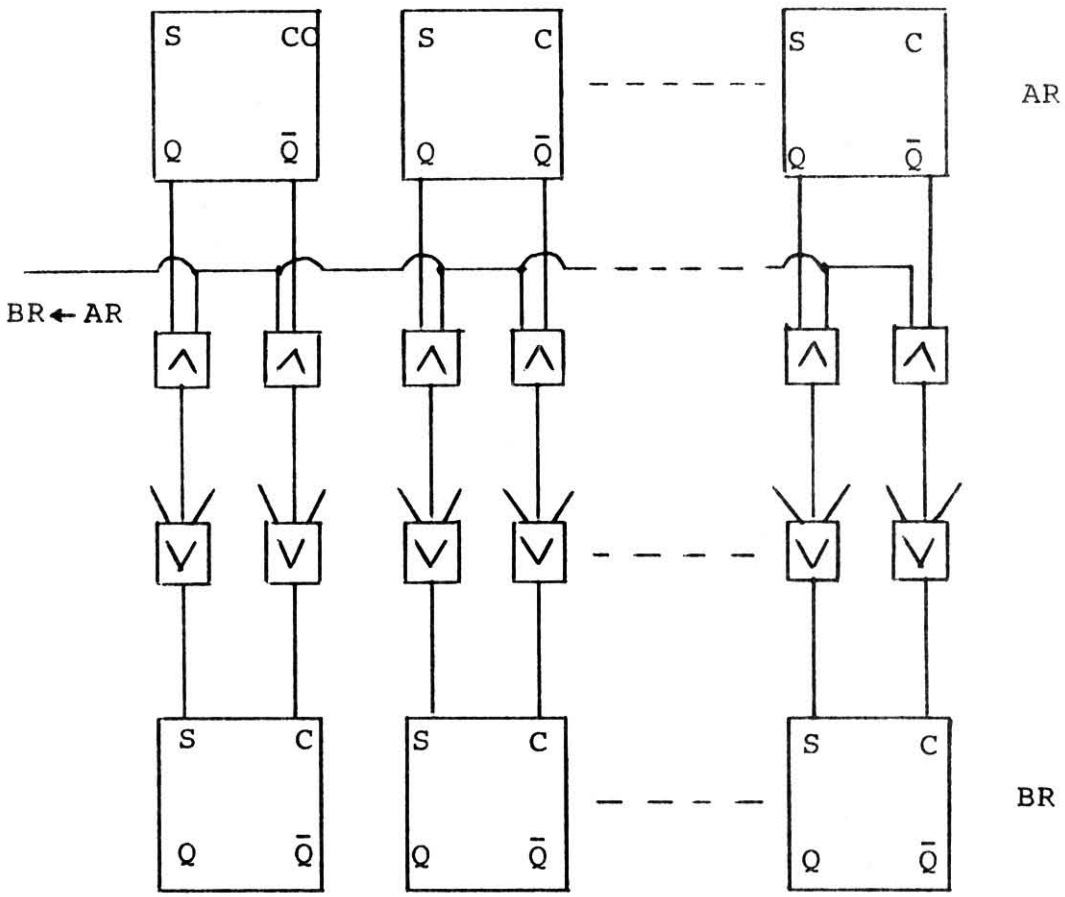


Bild 15

Wie alle Flipflops des BR-Registers mit allen Flipflops des AR-Registers beim Transfer

BR ← AR

belegt wurden, so können nun auch ausgewählte Flipflops von AR auf

die Eingänge von ausgewählten Flipflops aus BR gelegt werden. Eine solche Auswahl ist einmal möglich durch Indizierung. Dabei wird je ein Flipflop des Quell- und Zielregisters ausgewählt.

Beispiel:

$$BR_1 \leftarrow AR_2$$

Die Hardware-Übersetzung zeigt Bild 16.

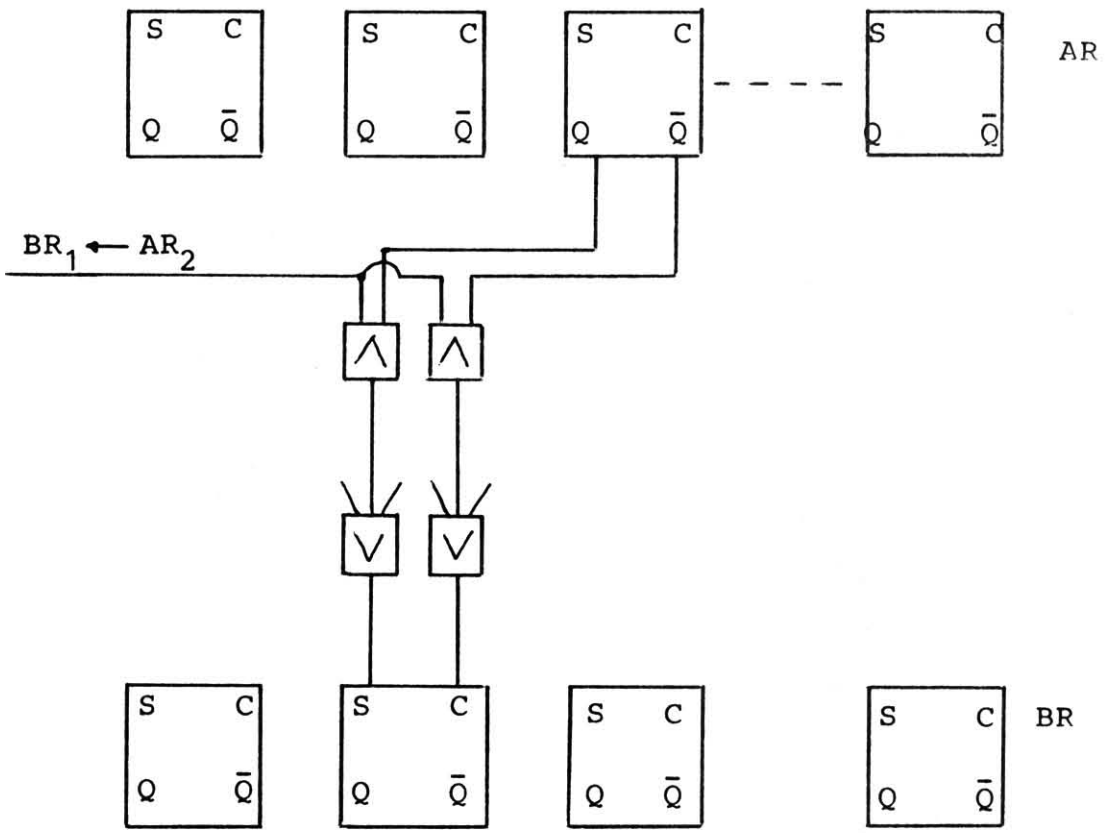


Bild 16

Mehrere einzelne Flipflops können durch ein Simple Simultanstatement natürlich auch parallel transferiert werden :

$$BR_o \leftarrow AR_1 \quad ; \quad BR_1 \leftarrow AR_2 \quad ; \quad BR_2 \leftarrow AR_3$$

Dafür wurde unter Primärelemente aber bereits eine abkürzende Schreibweise angegeben. Mit Hilfe der Ausblendoperatoren hat obiger Transfer dann folgendes Aussehen :

$$\alpha^3/BR \leftarrow \omega^3/\alpha^4/AR$$

Die Übersetzung zeigt Bild 17.

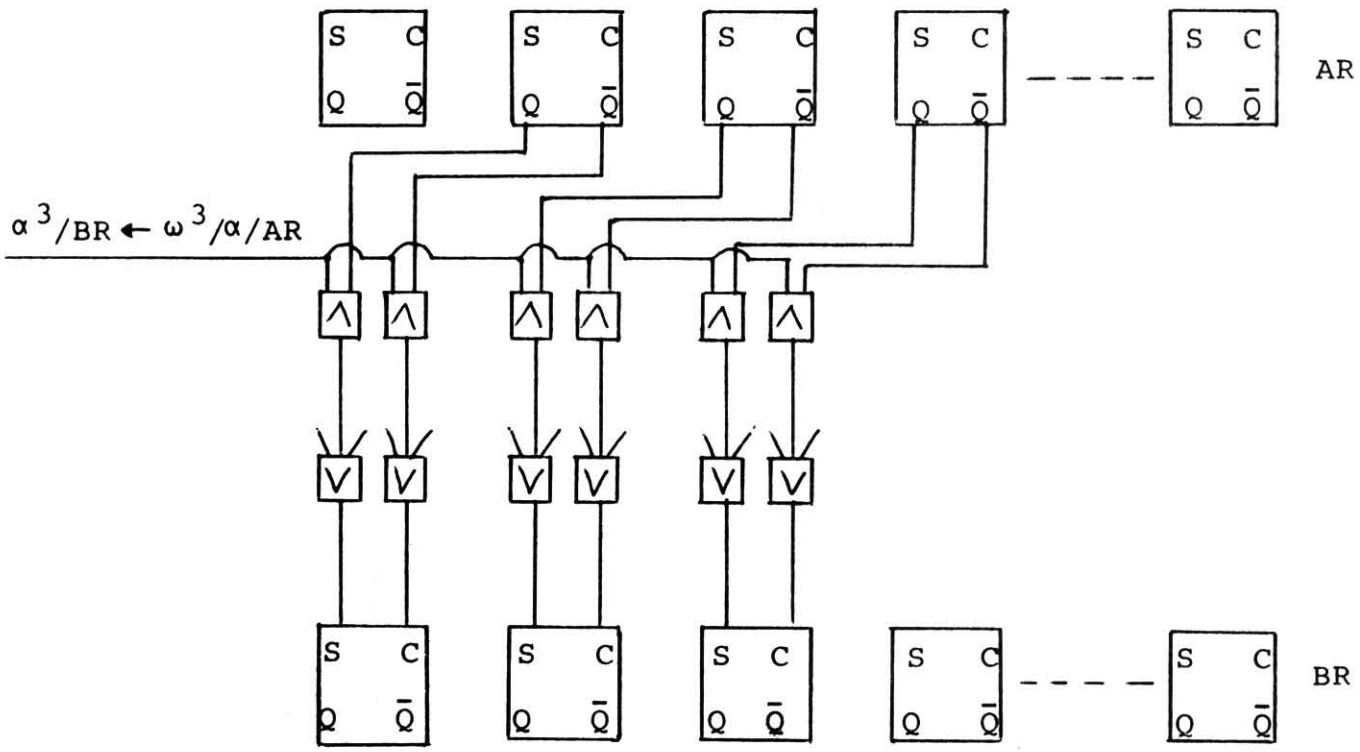


Bild 17

Ganz entsprechend können mit Hilfe des Compress-Operators beliebige Flipflops aus einem Register ausgewählt werden.

Außerdem besteht kein Unterschied zu einem Transfer, bei dem die rechte Seite die Catenation von Registerteilen und/oder Flipflops - auch verschiedener Register - ist. Bild 18 veranschaulicht die

Übersetzung des Statements

$$\omega^3/BR \leftarrow \omega^2/AR,\ CR_1$$

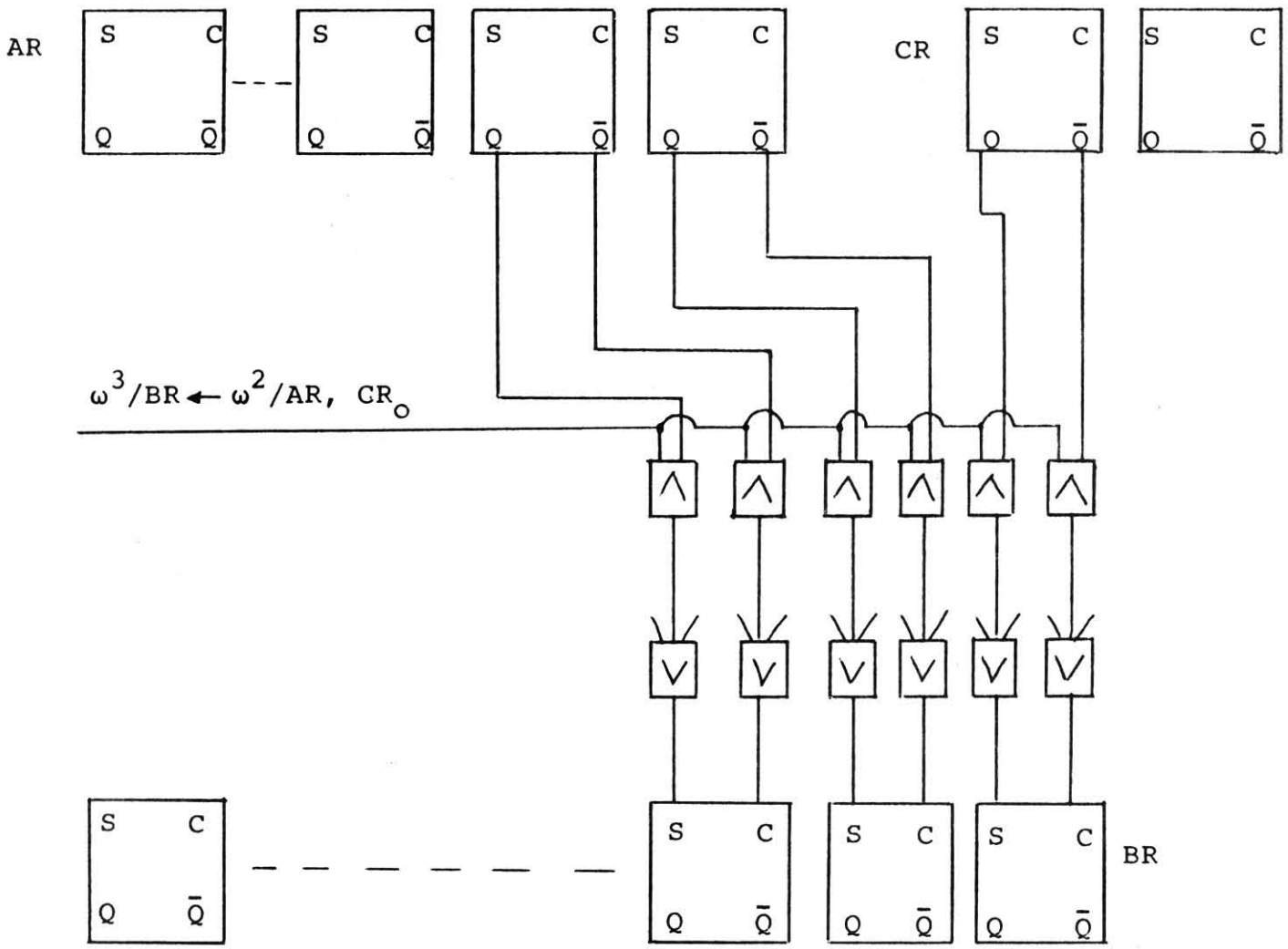


Bild 18

Weiterhin besteht kein Unterschied zu einem Transfer, bei dem die Quelle Konstanten sind. Dabei sind lediglich die Eingänge der AND-Gatter mit konstanten Spannungs-Levels statt mit den Ausgängen von Flipflops verbunden.

Des weiteren hat auch die Rotation eine analoge Hardware-Übersetzung. Der Unterschied liegt allein in der Leitungsverbindung zwischen

den Ausgängen des Quell-Registers und den Eingängen der AND-Gatter vor dem Zielregister; ansonsten bleibt die Hardware zu allen bisher behandelten Transfers gleich.

Bei dem Transfer

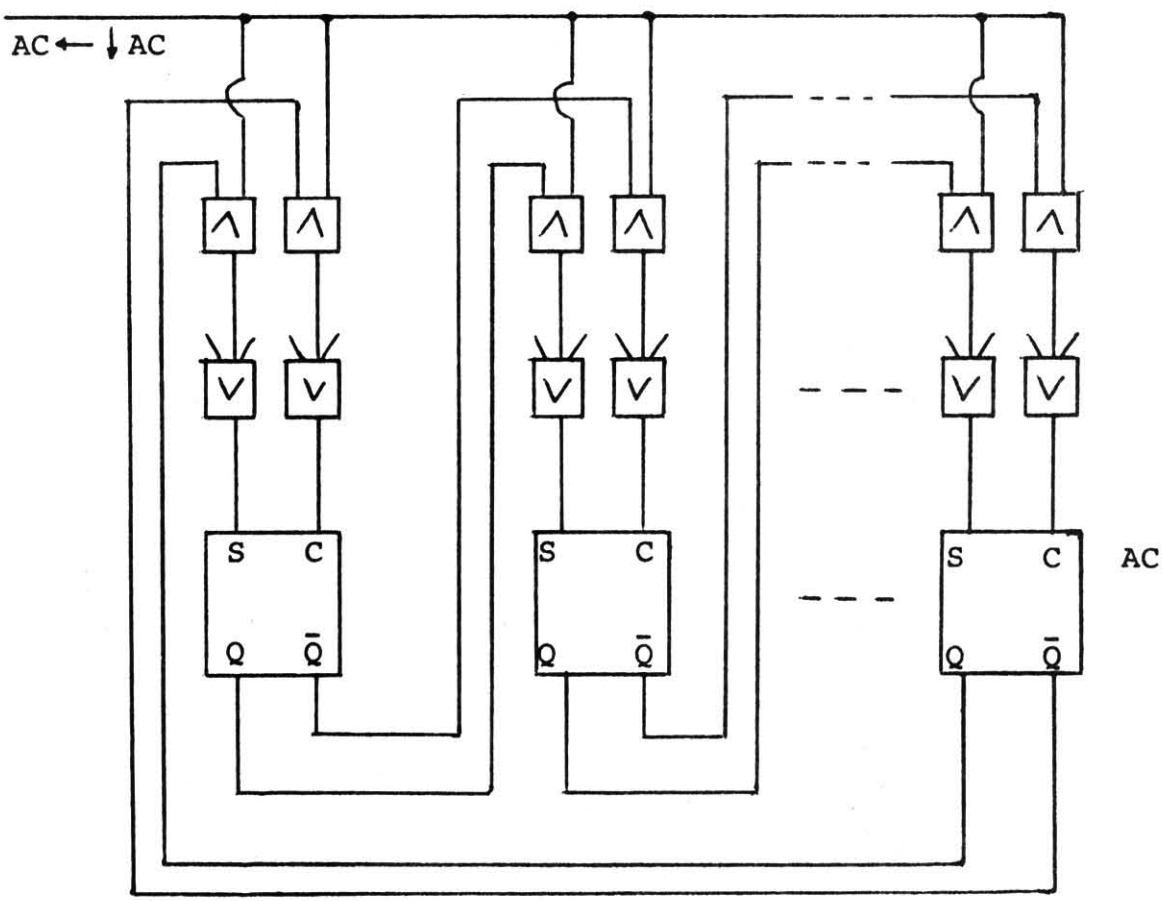
$$BR \leftarrow 2 \uparrow AR$$

z.B. liegt AR_2 auf BR_0 , AR_3 auf BR_1 usw.; auf dem vorletzten Flipflop von BR liegt AR_0 , auf dem letzten AR_1.

Oft wird die Rotation beim Rotieren eines Registers in sich selbst benutzt. Es soll als Beispiel das Register AC um eine Position nach rechts rotiert werden:

$$AC \leftarrow \downarrow AC$$

Da das Register AC gleichzeitig als Quelle und Transferziel dient, muß das AC-Register ein Master-Slave-Register darstellen; denn nur dann dürfen Ziel und Quelle identisch sein. Ist dies der Fall, so wird in die Hardware nach Bild 19 übersetzt.



Es soll nun ein weiterer AHPL-Operator behandelt werden: die Reduktion. Sie wird allgemein bei alternativen Transfers (siehe unten) und in Branch-Statements als Bedingung benutzt, doch ist natürlich auch ein Transfer in ein Flipflop möglich:

$$BR_1 \leftarrow \vee/AR$$

Bedeutung: die Q-Ausgänge der rechts vom Schrägstrich angegebenen Flipflops - in diesem Beispiel aller Flipflops von AR - werden als Eingänge des links vom Schrägstrich angegebenen Gattertyps genommen, in dem Beispiel also ein OR-Gatter. Dabei muß dieses Gatter so viele Eingänge haben, wie rechts vom Reduktionsstrich Flipflops angegeben sind. Aus Gründen der Assoziativität sind als Gatter nur AND und OR sinnvoll, doch kann der Hardware-Compiler eine zusätzliche Negation erkennen und entsprechend übersetzen. Der Ausdruck $\sim\vee/AR$ z.B. ergibt dann die Übersetzung nach Bild 20.

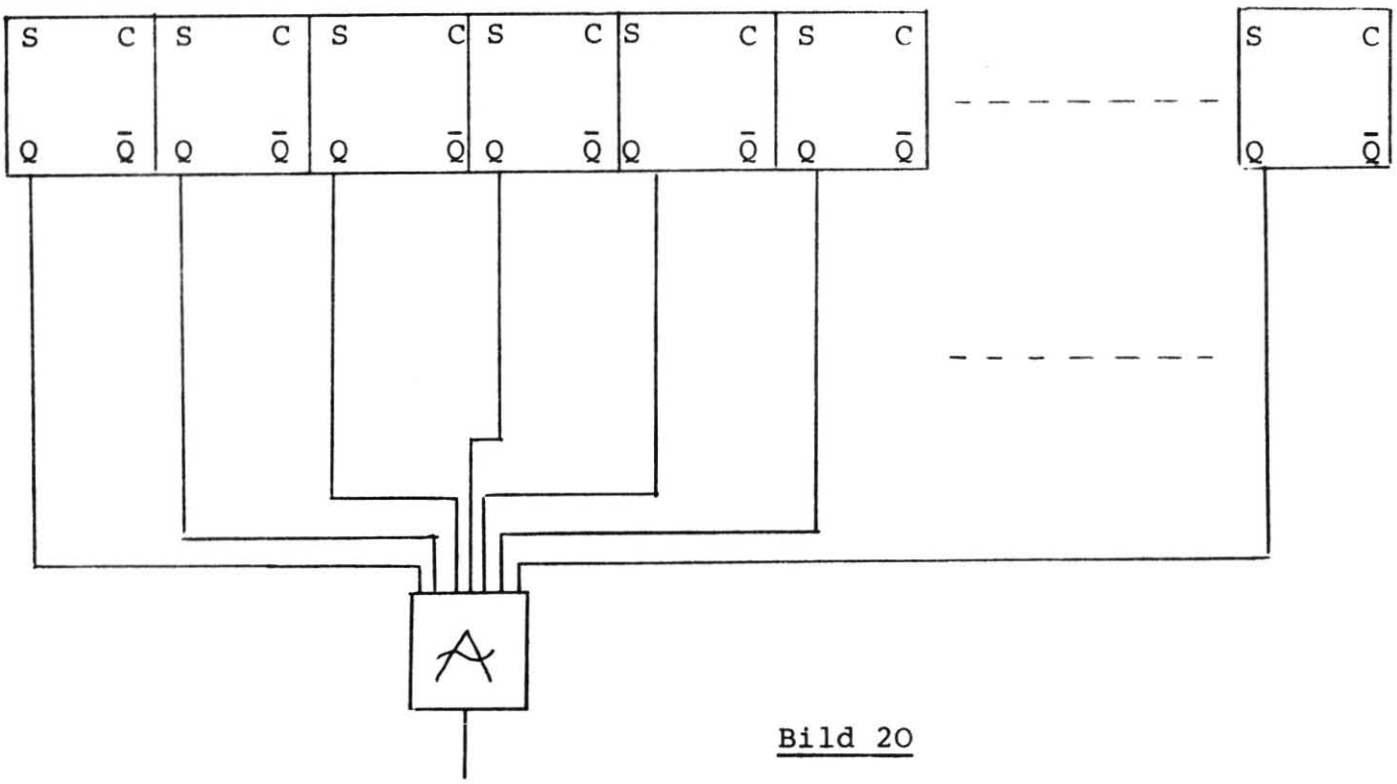


Bild 20

Ein Gatter mit entsprechend vielen Eingängen wird es möglicherweise nicht geben. Der Hardware-Compiler sollte das mit Hilfe einer Liste der verfügbaren Gatter feststellen und den Ausdruck entsprechend durch mehrere Gatter erzeugen.

Interessante Aspekte im Bezug auf den Hardware-Compiler bietet ein Alternative Transferstatement wie z.B. folgendes :

$$BR \leftarrow (AC \wedge F) \vee (AR \wedge \bar{F})$$

BR, AC und AR sollen dabei gleich lange Register sein und F ein Flipflop. Bild 21 stellt die Übersetzung dieser Transfers für ein einziges Bit dar.

Es fällt auf, daß dieser Transfer zu einem normalen Jam-Transfer einen kleinen Unterschied aufweist: zwar sind S- und C-Eingang des Ziel-Flipflops belegt, von den Quell-Flipflops aber werden unterdessen nur noch die positiven Ausgänge verwandt. Der Grund leuchtet ein: dadurch, daß der Transfer bedingt ist, benötigt man außer den beim normalen Jam-Transfer bereits notwendigen Gattern pro Bit zwei AND- und ein OR-Gatter; würde man auch die negativen Ausgänge transferieren, dann also das Doppelte. Die Verdoppelung verhindert man durch die Negation nach dem OR-Gatter.

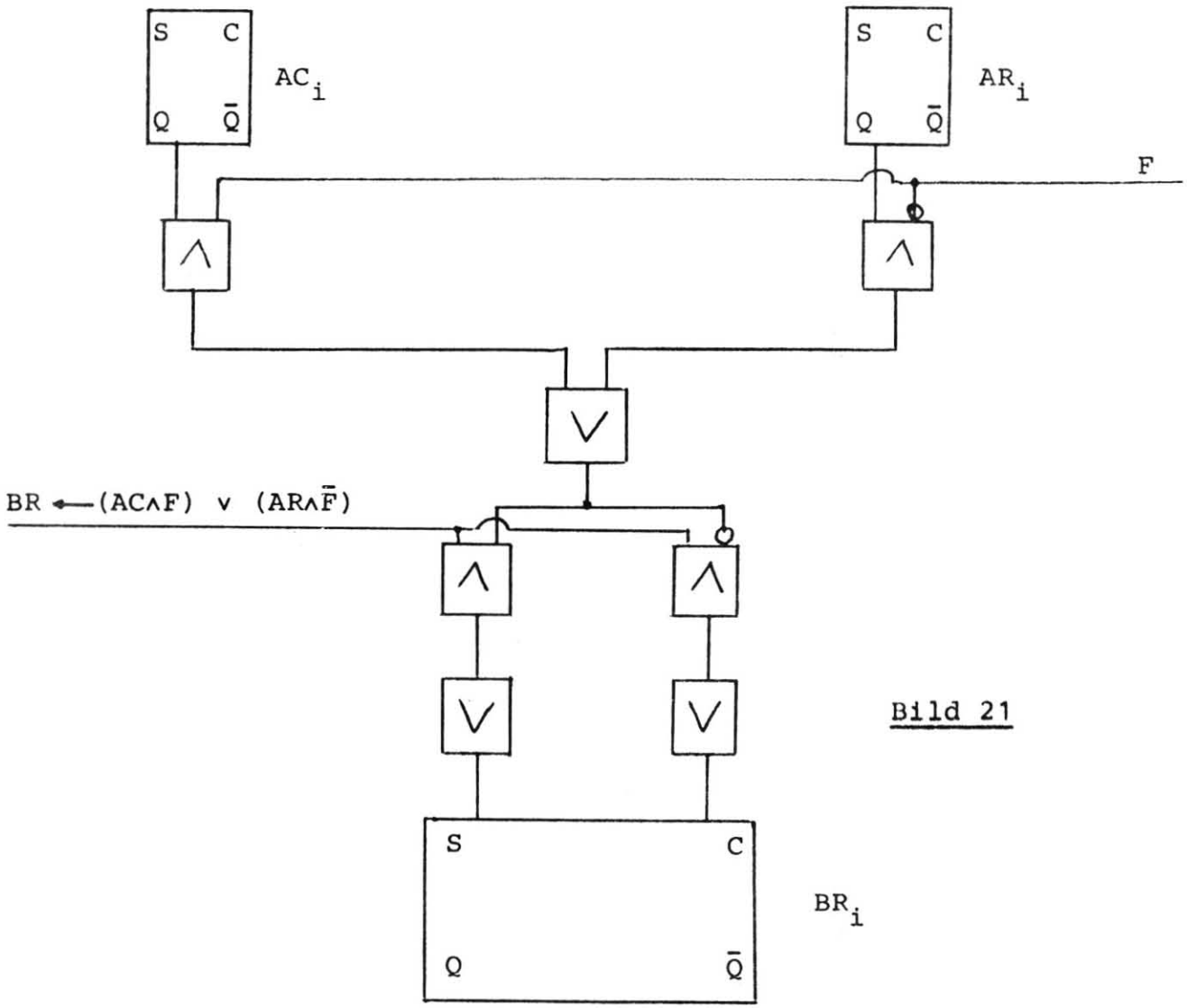


Bild 21

Ein solcher bedingter Transfer läßt sich bei entsprechender disjunkter Bedingung natürlich auch aus mehr als zwei Registern als Quelle durchführen. Für jedes weitere Register wird dann pro Bit ein 2-Input-AND-Gatter sowie ein weiterer Eingang beim OR-Gatter benötigt.

Es läßt sich leicht einsehen, daß das Alternative Transferstatement

$$BR \leftarrow (AC \wedge F) \vee (AR \wedge \bar{F})$$

aus obigem Beispiel durch folgende Sequenz ersetzbar ist:

```
[1]     → (2×F)+(4×F̄)
[2] BR ← AC
[3]     → (5)
[4] BR ← AR
[5] ........
```

Das Alternative Transfer wird also durch einen Branch Controler und zwei Simple Transfers ersetzt (Bild 22).

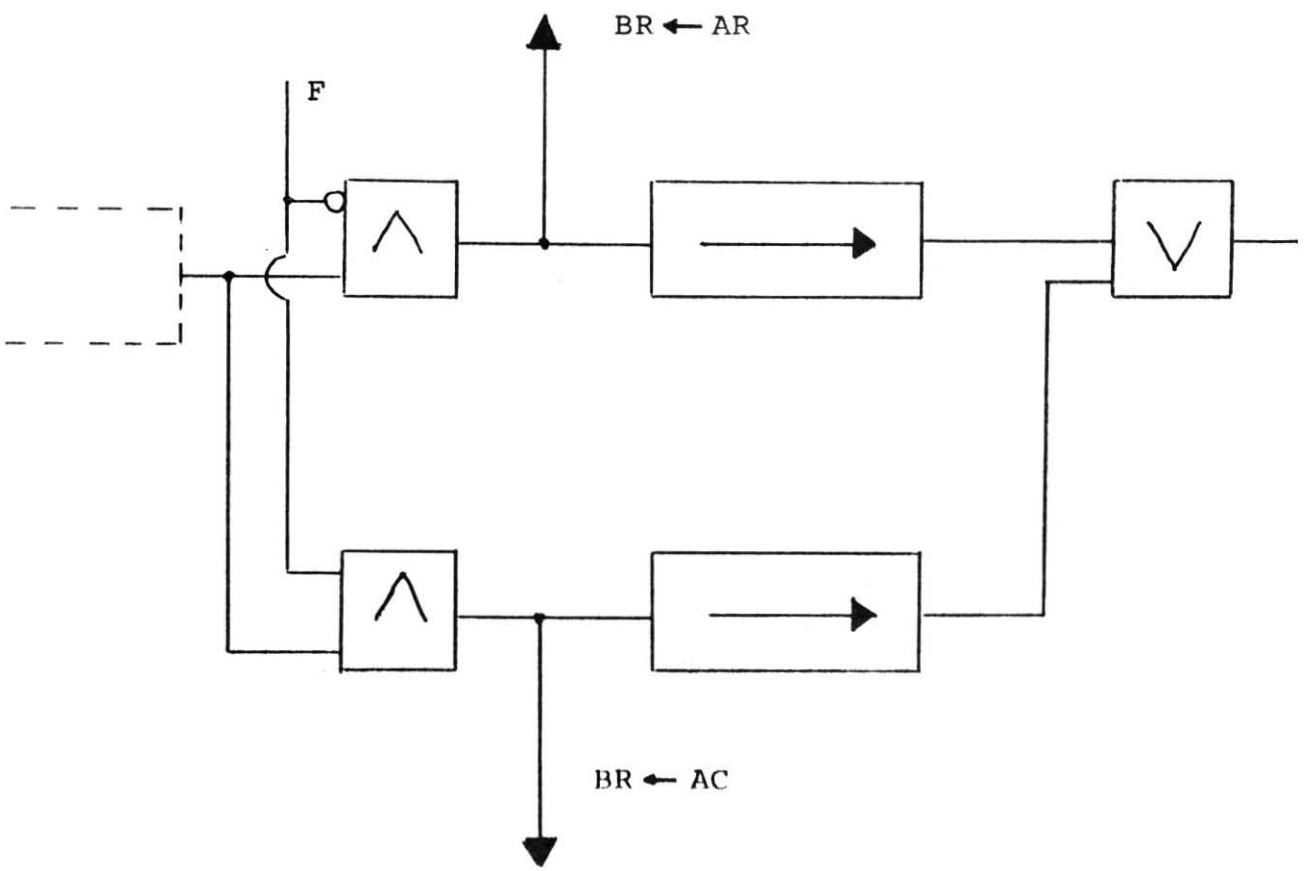


Bild 22

Beim Alternative Transfer benötigte man pro Bit vier AND-. ein OR-Gatter und eine Negation. Die Negation der Transfer-Bedingung sowie die OR-Gatter direkt vor den Flipflop-Eingängen sind dabei nicht berücksichtigt.

Bei der Beschreibung durch zwei Simple Transfers benötigt man hingegen pro Bit nur vier AND-Gatter, dafür aber einen Eingang mehr bei den OR-Gattern der Zielregister; außerdem für den gesamten Transfer im Controler zwei AND- und ein OR-Gatter sowie ein weiteres Delay-Element.

Dieses zusätzliche Delay-Element läßt sich allerdings durch eine weitere Modifizierung der Control-Sequenz noch einsparen :

```
[1]   → (2×F)+(4×F̄)
[2]   BR← AC, NØ DELAY
[3]   → (4)
[4]   BR← AR
```

Die Übersetzung zeigt Bild 23.

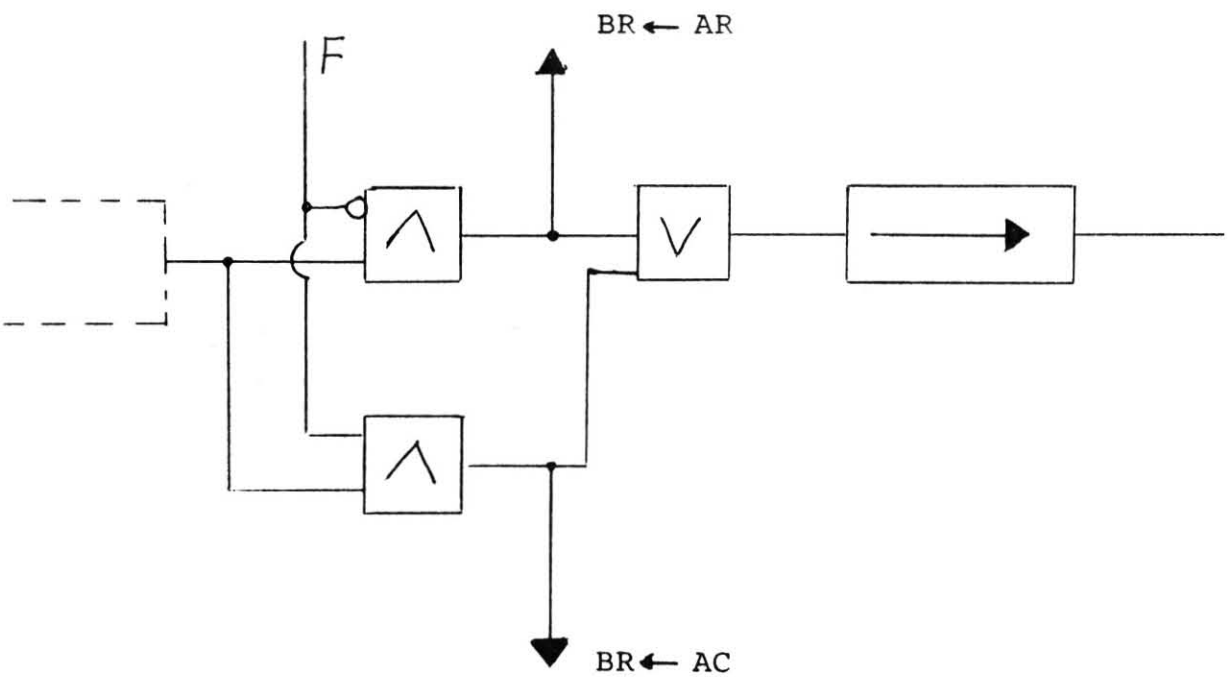


Bild 23

Das Ersetzen eines Alternative Transfers durch zwei Simple Transfers dürfe sich also bereits bei einer Registerlänge von drei oder vier Flipflops lohnen. Nicht berücksichtigt wurden dabei allerdings die Fan-Out-Bedingungen, die die Günstigkeit der einen oder anderen Transferart wesentlich beeinflussen können. Dieser Einfluß sollte vom Hardware-Compiler berücksichtigt werden und entsprechend zur Realisierung der günstigsten Möglichkeit führen.

Alternative Transfers sind auch mit nicht vollständiger Bedingung erlaubt. Es sei zum Beispiel folgendes einfache Alternative Transferstatement mit den beiden Registern AC und MD und dem Flipflop F gegeben :

$$AC \leftarrow MD \wedge F$$

Bedeutung: Wenn die Bedingung F"1" ist, soll sich AC aus MD ergeben, im anderen Fall soll nichts geschehen.

Die Hardware-Übersetzung mit Controler zeigt Bild 24.

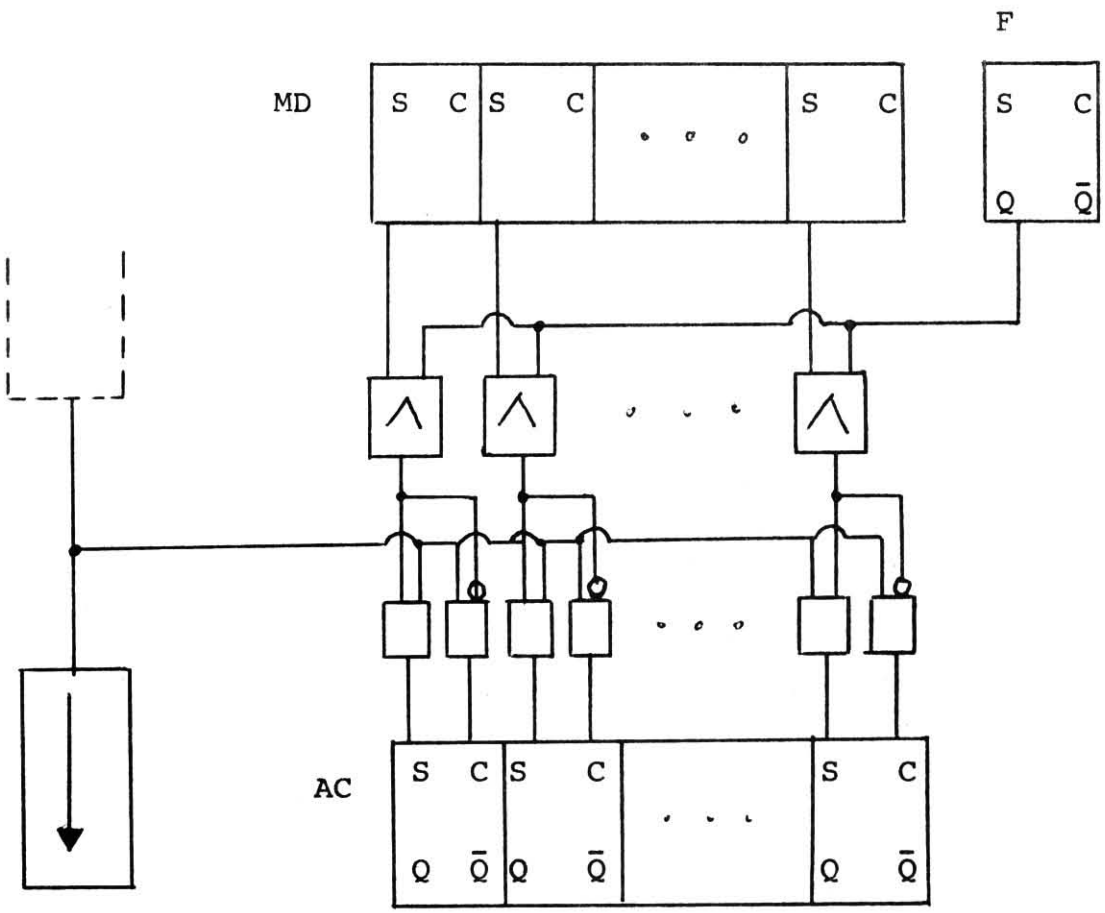


Bild 24

Die normalerweise vor den Eingängen des Zielregisters notwendigen OR-Gatter wurden der Übersicht halber weggelassen.

Dieser Transfer entspricht also genau einem Alternative Transfer mit mehreren Alternativen. Da nur eine "Alternative" vorhanden ist, werden natürlich die OR-Gatter eingespart.

Auch für diesen Transfer gibt es entsprechen dem vorangegangenen Beispiel eine Umschreibung mit einem Simple Simultanstatement :

[1] → (2×F)+(3×$\bar{F}$)
[2] AC← MD
[3] .....

Die Übersetzung ist Bild 25 zu entnehmen.

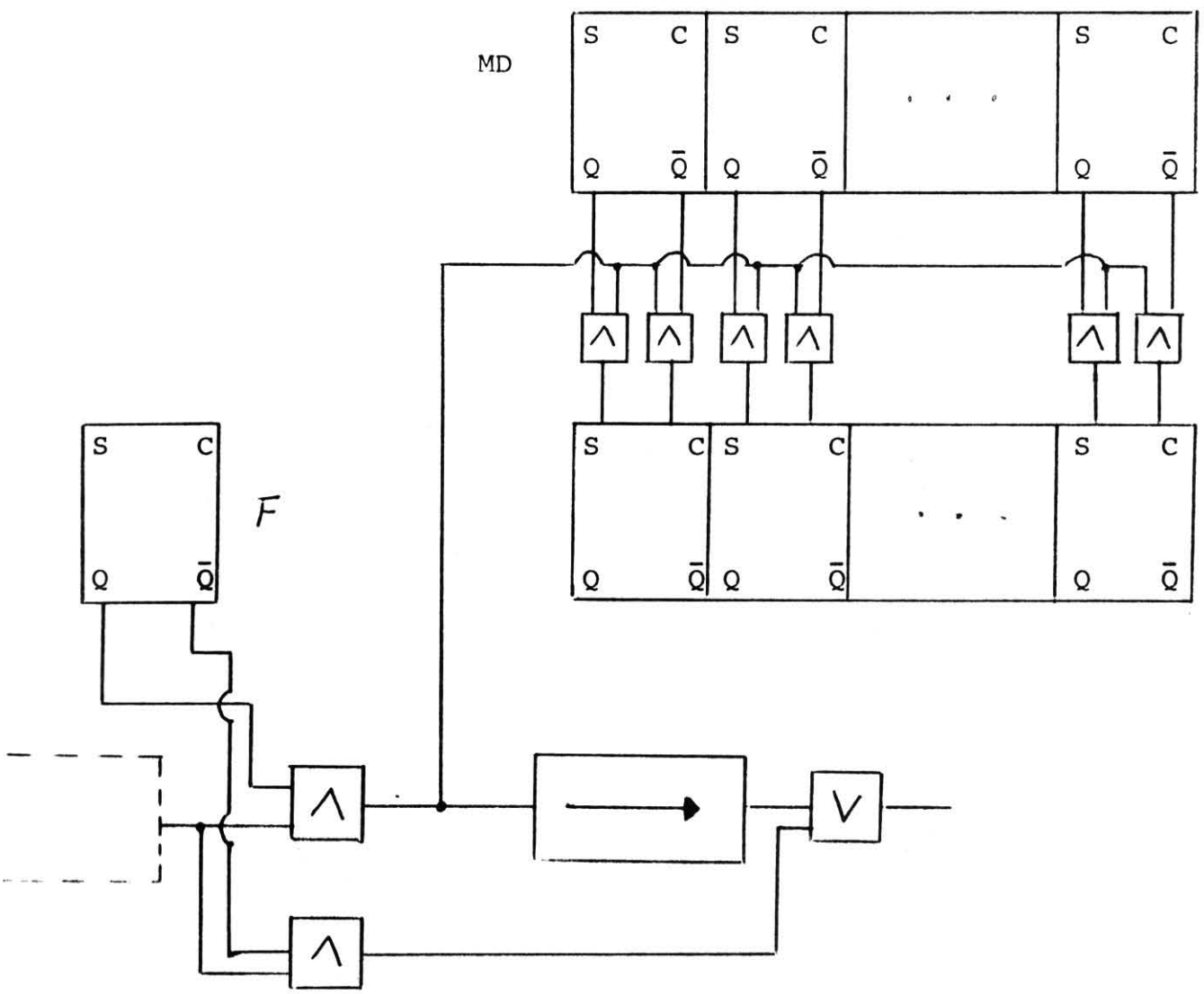


Bild 25

Wie zu sehen ist, benötigt man im Controler zwei AND- und ein OR-Gatter mit je zwei Eingängen. Dafür hat man n (n=Anzahl der Flipflops in MD) AND-Gatter und n Negationen eingespart. Außerdem wird das Delay umgangen, wenn die Bedingung nicht erfüllt ist; die Sequenz wird dadurch also schneller.

Außer einem direkten Transfer von Quellregister zu Zielregister gibt es die Möglichkeit eines Transfers über einen Bus. Auf die Vor- und Nachteile eines Bus-Transfers soll hier nicht weiter eingegangen werden, sondern es wird nur die Hardware-Übersetzung eines solchen erläutert.

Bild 26 zeigt als Beispiel den ABUS, der mit den beiden Registern AR und BR belegt werden kann und dessen Ausgang auf den zwei Registern CR und DR liegt. Es sind also folgende Transfers möglich:

ABUS ← AR	ABUS ← AR
CR ← ABUS	DR ← ABUS
ABUS ← AR	ABUS ← BR
DR ← ABUS	DR ← ABUS

Zur Demonstration wurden Register mit nur vier Flipflops gewählt (Bild 26).

Wie Bild 26 zeigt, besteht der Bus pro Bit nur aus einem OR-Gatter und den notwendigen Leitungen zu und von diesem Gatter sowie einem AND-Gatter pro Bit und Register, mit dem er beschaltet werden kann. Das Takten eines Bus-"Inhaltes" (von einem Inhalt kann man eigentlich nicht sprechen, da der Bus kein Speicherelement darstellt) in ein Zielregister geschieht wie beim Jam-Transfer von Register zu Register, mit der Ausnahme, daß der Bus natürlich nicht die negativen Registerinhalte mitführt und daher Negationsglieder benutzt werden müssen.

Das Beschalten eines Buses andererseits kann aus technischen Gründen nicht durch Pulse geschehen.

Levels sind in einem puls-orientieren System aber nur als Ausgänge von Registern und Flipflops vorhanden. Man muß daher zunächst in einem Takt ein Master-Slave-Flipflop setzen, dessen Ausgang an den entsprechenden AND-Gattern am Eingang des Buses liegt. Damit

führt der Bus den Registerinhalt. Im nächsten Takt wird dieser Inhalt in das Zielregister übernommen und danach kann das Bus-Eingangs-Flipflop wieder auf Null gesetzt werden.

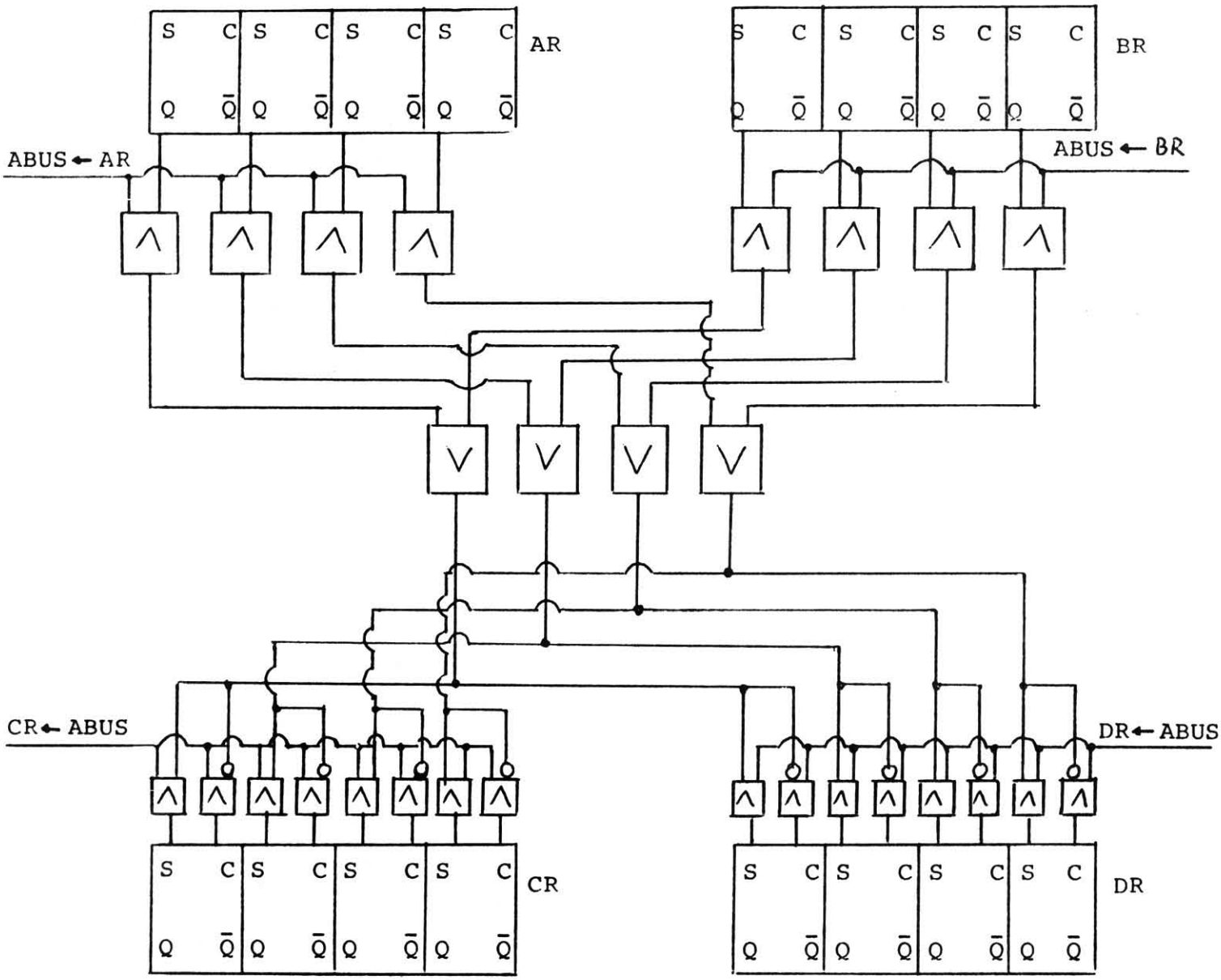


Bild 26

Ein Bus-Transfer dauert bei einem puls-orientierten System also grundsätzlich zwei Takte. Die Notation

ABUS ← AR
DR ← ABUS

wird vom Hardware-Compiler also übersetzt in :

ARABUSFF ← 1
DR ← ABUS ; ARABUSFF ← 0

ARABUSFF ist dabei das Master-Slave-Flipflop, dessen Ausgang auf den AND-Gattern von AR zum ABUS liegt.

Die Übersetzung zeigt Bild 27.

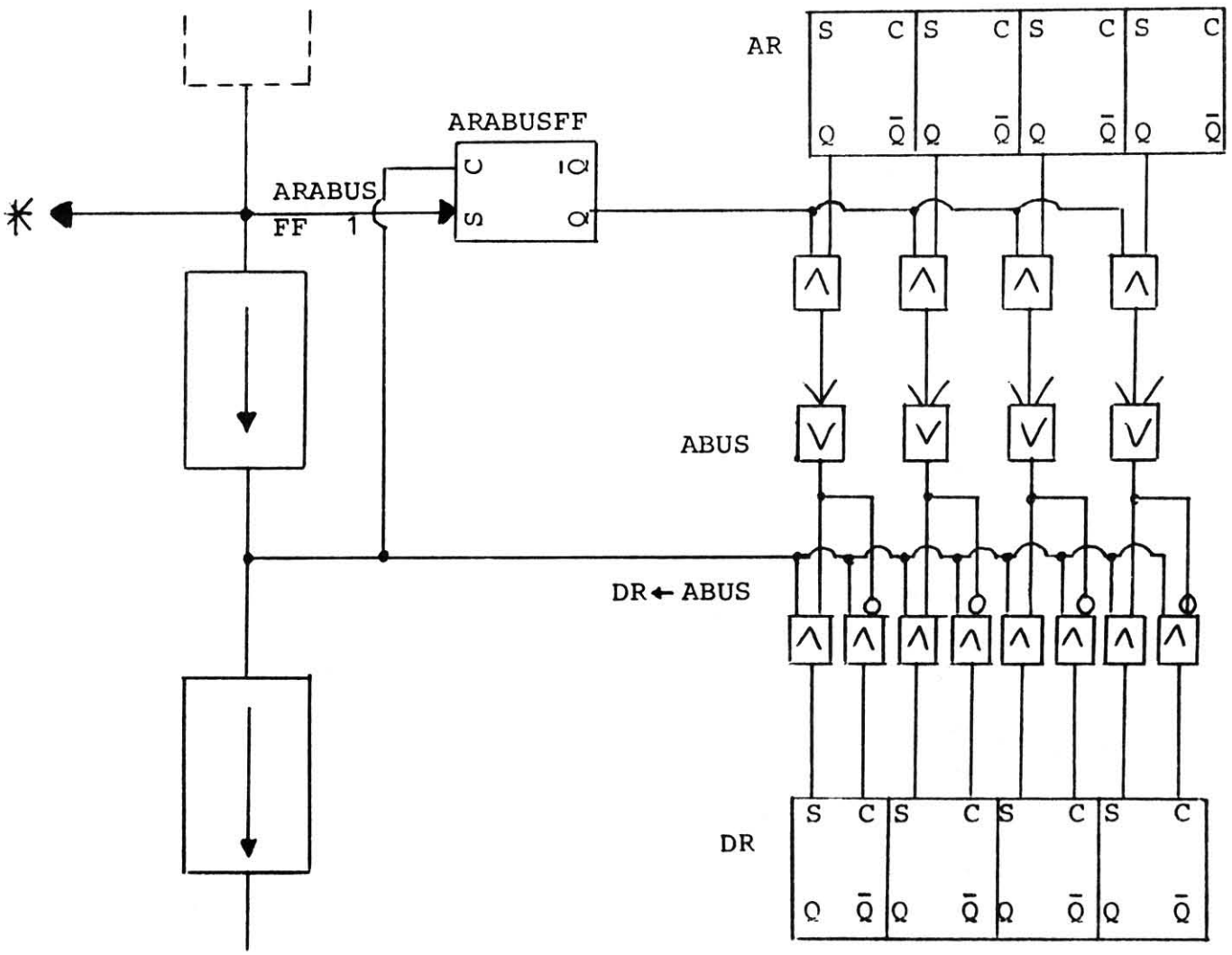


Bild 27

Sofern im vorangegangenen Takt dieser Bus nicht benutzt wurde, kann das Setzen des Flipflops natürlich simultan mit einem anderen Transfer ausgeführt werden, was in Bild 27 die abgehende, mit * bezeichnete Leitung andeutet. Ein Transfer über einen Bus würde dann nur noch einen Takt benötigen. Eine solche Möglichkeit sollte vom Hardware-Compiler erkannt und ausgenutzt werden.

Level Controler

Die vorangegangenen Betrachtungen in diesem Kapitel wurden mehr der Vollständigkeit halber und aus didaktischen Erwägungen angestellt als aus praktischen. In der Praxis nämlich bringt der puls-orientierte Controler viele Probleme mit sich - z.B. beim Busing, aber auch Noice- und Laufzeitschwierigkeiten -. Diese entfallen beim level-orientieren Controler ohne Mehraufwand, so daß man heute beinahe ausschließlich Level-Controler verwendet.

Der Unterschied zwischen level- und puls-orientiertem Controler besteht darin, daß bei ersterem keine Delay-Elemente, wie sie oben vorgestellt wurden, benutzt werden, sondern Master-Slave-D-Flipflops Darin liegt, jedenfalls soweit es den Controler betrifft, der einzige Unterschied.

Der logische Unterschied ist folgender: beim puls-orientierten Controler wird ein Puls dazu benutzt, um einen Transfer auszuführen. Dieser Puls wird dann während der Pause der Master-Clock in einem Delay gespeichert. Sind z.B. wie beim Busing Levels erforderlich, so wird ein spezielles Flipflop durch einen Puls gesetzt und im nächsten Takt der Ausgangslevel dieses Flipflops benutzt.

Beim level-orientierten Controler dienen die Ausgangslevel der D-Flipflops dazu, einen Transfer vorzubereiten; ausgeführt wird der Transfer durch den nächsten Takt. Nach diesem Takt liegt der Level am Ausgang des nächsten Delay-Elementes.

Bild 28 stellt den Simple Transfer

$$A \leftarrow B$$

dar, wobei A und B Flipflops sind, und zwar bei einem level-orientierten Controler. Das Master-Slave-D-Flipflop im Controler wird dabei schematisch durch ein Rechteck mit einem D dargestellt. Eine

Taktleitung zu diesem Element wird angenommen.

Wie das Bild zeigt, hat sich außer dem Delay-Element gegenüber dem puls-orientierten Controler nur eines geändert: die Flipflops erhalten eine Taktleitung; es werden hier also getaktete S-C-Flipflops benutzt.

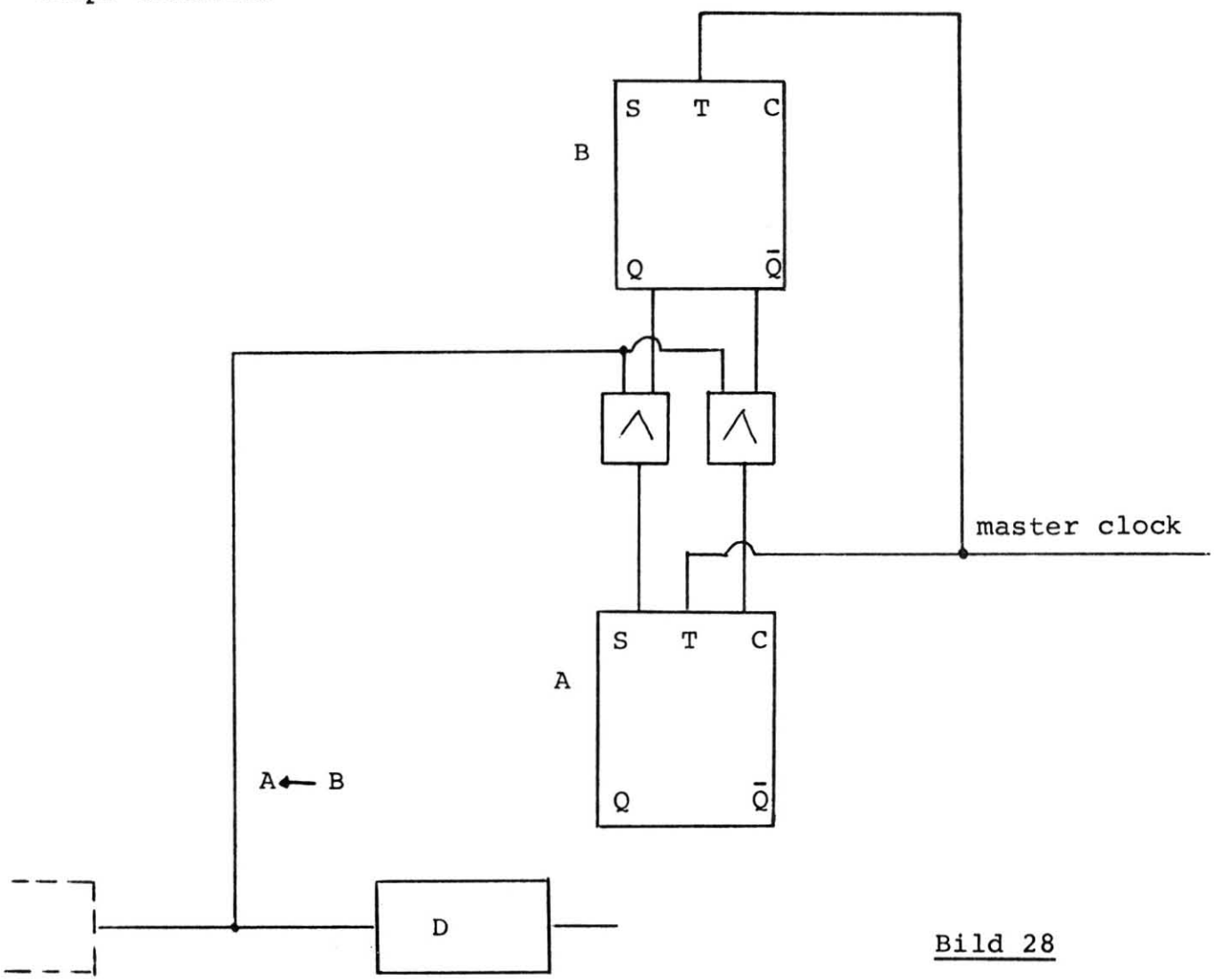


Bild 28

Hier der Ablauf des Transfers: wenn auf der Leitung vor dem D-Flipflop ein Level erscheint, liegen nach entsprechender Laufzeitverzögerung an den Ausgängen der beiden AND-Gatter und damit an den S- und C-Eingängen des Ziel-Flipflops die Werte von Q und $\bar{Q}$ des Quellflipflops. A reagiert jedoch darauf erst, wenn an seinem Takteingang der Takt erscheint.

Analog zu diesem Simple Transfer laufen alle Transfers ab, die beim puls-orientierten Controler vorgestellt wurden. Benutzt werden dabei S-C-Flipflops und Master-Slave-Flipflops, sowie flangengetriggerte Flipflops, die in ihrem Verhalten diesen entsprechen.

Bisweilen verwendet man auch D-Flipflops; allerdings muß dann der Master clock über ein Gatter geführt werden, was wieder zur Berücksichtigung von Fan-out-Bedingungen führt. Im Hardware-Compiler sollte jedoch auch diese Möglichkeit eingeschlossen sein.

Bild 29 zeigt die Realisierung eines Transfers mit D-Flipflops.

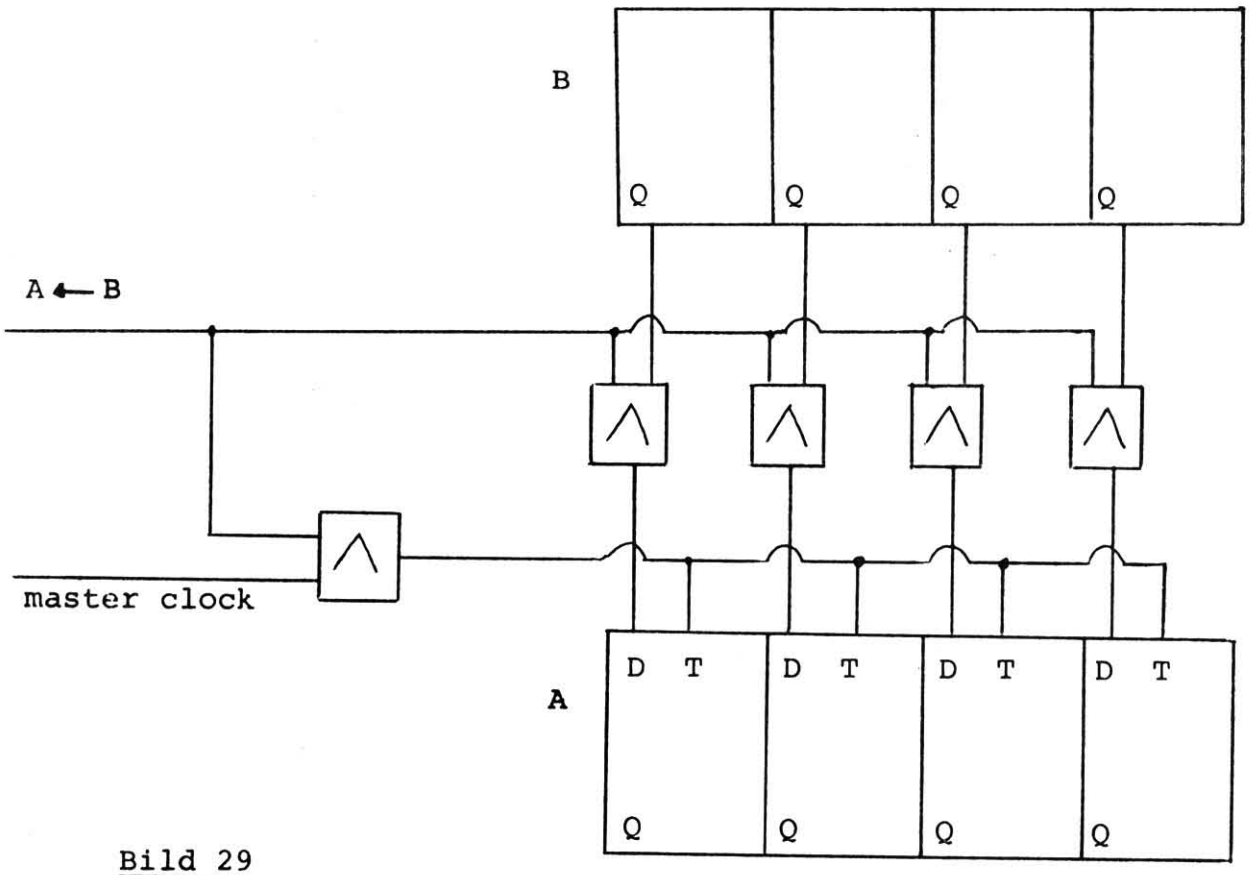


Bild 29

Recht einfach realisieren läßt sich das Busing bei einem Level-Controler, wie in folgendem Beispiel veranschaulicht:

Der Transfer vom Register MD in AC soll über den Bus ABUS erfolgen. Dazu wird durch den im Controler vorhandenen Level das MD-Register mit ABUS verbunden und gleichzeitig die Verbindung von ABUS über die AND-Gatter zu den Eingängen von AC hergestellt. Mit dem darauffolgenden Takt übernimmt das Register AC den Inhalt von MD. Dieser Bus-Transfer läuft also in einem Takt ab.

Bild 30 zeigt die Übersetzung :

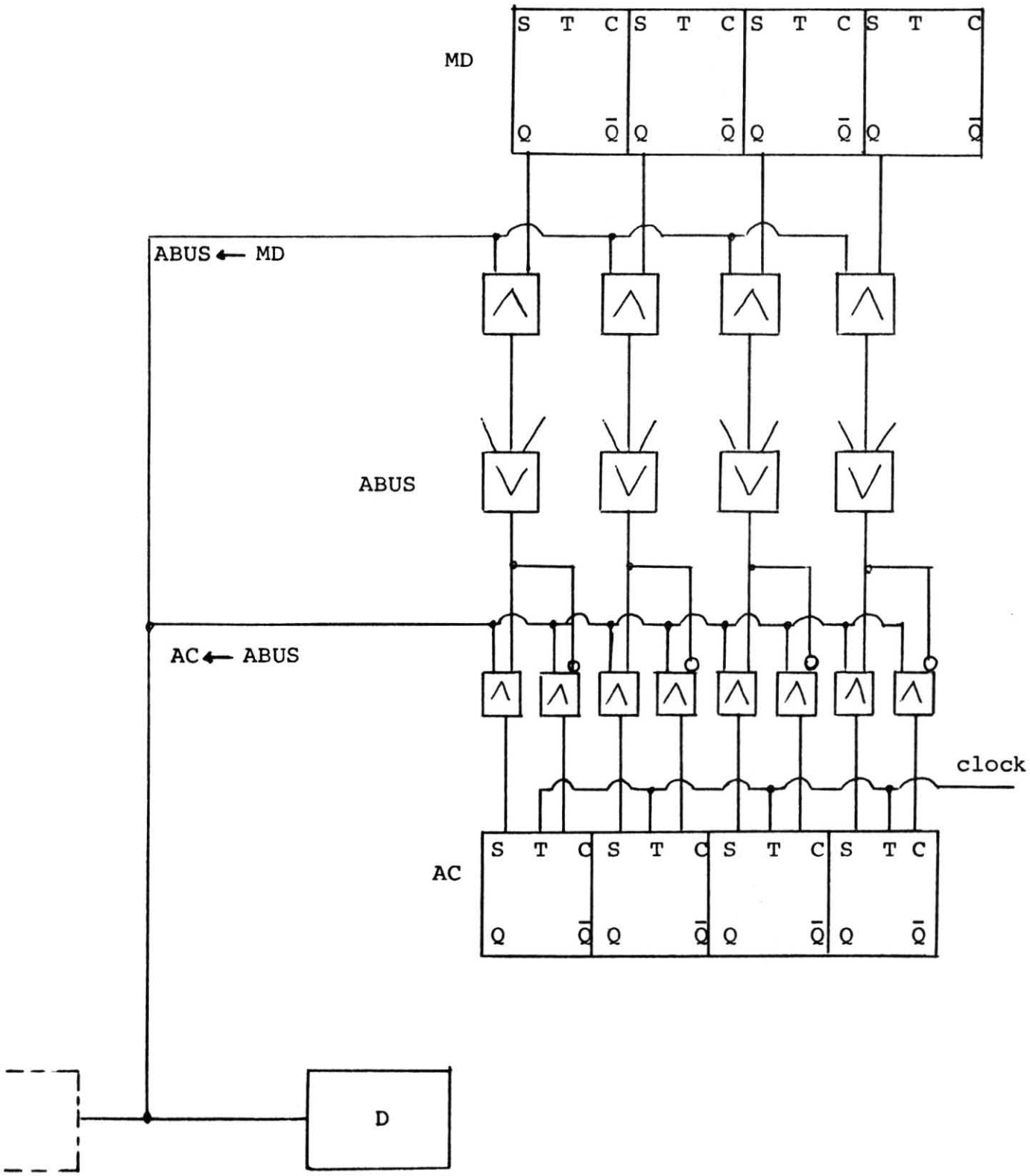


Bild 30

4.3 Unterprogrammtechnik

Der Übersetzungsprozess eines AHPL-Kontrollprogrammes ist ähnlich dem der Übersetzung eines Programmes, das in einer problem-orientierten Programmier-Sprache geschrieben wurde.

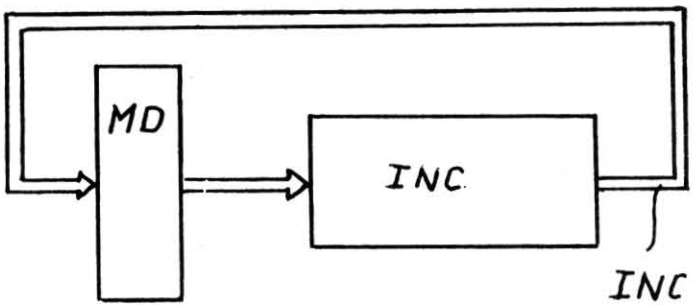
Im ersteren Falle wird eine 'hardwired control unit' erzeugt, im letzteren ein Programm, das auf einem Rechner ablaufen kann.

Eine 'hardwired control unit' stellt praktisch einen Hardware-Interpreter dar, der es ermöglicht, ein erzeugtes Maschinenprogramm zur Objektzeit ablaufen zu lassen. Ausgehend von einer vorgegebenen Anzahl von Grundelementen (Register, Kontrollflipflops und einem oder mehreren Speichern) und einem bestimmten Maschinenbefehlsvorrat kann ein AHPL-Kontrollprogramm dazu benutzt werden, um einen Rechner zu simulieren oder mit Hilfe eines Hardwarecompilers die Zentraleinheit eines Rechners zu entwerfen. Eine weitere Möglichkeit besteht darin, AHPL als eine Microassemblersprache aufzufassen. Ein AHPL-Programm wird dann in ein Microprogramm übersetzt, das Statement für Statement nach einem bestimmten Microbefehlsformat in einem RØM (Read Only Memory) gespeichert wird und mit Hilfe eines Microleitwerkes zur Objektzeit interpretiert werden kann.
Abweichend von der Unterprogrammtechnik problemorientierter Programmiersprachen wird in AHPL diese Technik nur dazu benutzt, um ein Programm übersichtlich zu gestalten. Durch Definition einer Funktion oder einer Subroutine entwirft der Designer neue Bauelemente eines Rechners, deren Funktion durch die entsprechende Statementfolge beschrieben wird. Das so entstehende Blockelement kann dann durch Angabe des Namens in einem Kontrollprogramm verwendet werden. Dabei ist folgendes zu beachten: die Eingangsparameter im Funktions- oder Subroutineaufruf stellen ganz bestimmte Register- oder Flipflop-Anschlüsse dar; ändern sich diese Parameter in einem anderen Aufruf, so muß das durch das Unterprogramm definierte Bauelement vom Hardware-Compiler neu generiert werden, denn ein fest verdrahteter Anschluß kann keinen 'Dummy' darstellen. (Ausnahme: Busstruktur)

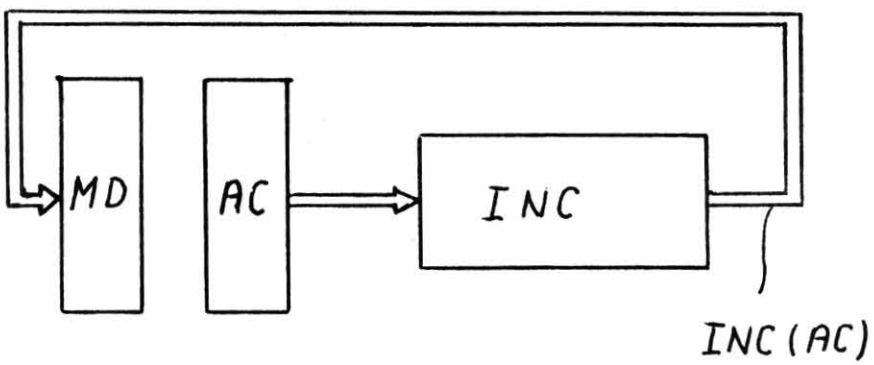
Beispiel: In einem Kontrollprogramm treten folgende Statements auf :

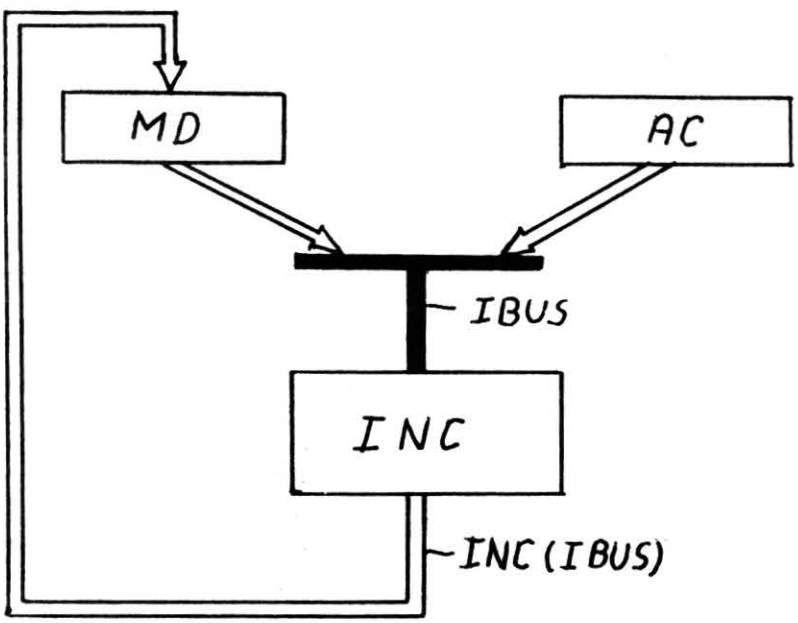
MD ← INC(MD)

MD ← IND(AC)



a) Duplizierung der INC-Subroutine





b) keine Duplizierung notwendig durch Bus mit Zweierstruktur

Zweierstruktur bedeutet, daß der Transfer in zwei Schritten durchgeführt wird, d.h.

Aus MD ← INC(MD) wird:

IBUS ← MD

MD ← INC(MD

Aus MD ← INC(AC) wird:

IBUS ← AC

MD ← INC(IBUS)

Realisierung der INC-Subroutine

Das MD-Register habe 18 Komponenten. Der Aufruf INC(MD) soll einen Vektor darstellen, dessen Komponenten das duale Abbild des incrementierten Registerinhaltes darstellen soll.

Die Implementierung der IND-Subroutine beruht darauf, daß sich ein Bit in einem Zähler ändert, wenn alle Bits zu seiner rechten den Wert '1' haben.

```
    [1]  SUBRØUTINE INC(MD)
B   [2]  MDNEW← 18 ⍴ 0
B   [3]  i← 17
    [4]  MDNEW[i] ← MD[i] ⊕ (∧/ω[17-i](18)/MD)
B   [5]  i← i-1
B   [6]  i : 0, (≥,<) → (4,7)
    [7]  INC(MD) ← MDNEW
    [8]  RETURN
```

In dieser Subroutine sind eine Reihe neuer Statementtypen enthalten.

Zeile 1 stellt einen Subroutineaufruf dar, dessen Syntax früher schon behandelt wurde.

Zeile 4
Zeile 7 stellen Connectionstatements dar.

Zeile 2
Zeile 3
Zeile 5
Zeile 6 stellen Bookkeeping-statements dar.

In Zeile 2 wird mit Hilfe des APL-Reshape-Operators ein Vektor MDNEW definiert, dessen 18 Komponenten zunächst zu 0 gesetzt werden. MDNEW ist eine Zwischenvariable, deren Komponenten iterativ in Schritt 4 belegt werden. In Statement 7 wird der komplett belegte Vektor MDNEW dem Funktionswert INC(MD) zugeordnet. Statement 8 stellt analog zum FØRTRAN RETURN-statement den Rücksprung ins rufende Programm dar. (Unnötige Sprachvermischung!)

Besonderheiten eines Connectionstatements

1) Linke und rechte Seite werden durch einen Pfeil nach links getrennt. Dieser Pfeil stellt keinen Transferpfeil dar.

2) Die linke Seite eines Connectionstatements stellt einen Vektor oder Skalar dar und repräsentiert die Ausgangsleitungen eines Teilschaltnetzes, das auf der rechten Seite dieses Statements beschrieben wird.

3) Die rechte Seite eines Connectionstatements ist eine skalare oder vektorielle bool'sche Funktion, die direkt als logisches Teilschaltnetz realisierbar ist.

4) Argumente obiger Bool'scher Funktion können sein:
   - Konstanten (Vektoren von Oen und 1en)
   - Flipflops
   - Register oder Busse
   - Variablen, die auf der linken Seite anderer Connectionstatements auftreten
   - Ausgänge anderer Unterprogramme

5) Variable können mit variablen Indizes versehen werden, die in bookkeeping-statements berechnet werden.

Besonderheiten von Bookkeepingstatements

1) Sie erlauben eine Schleifenorganisation.

2) Alle AHPL-Branch-Statements können als Bookkeeping-Statements verwendet werden.

3) In einem Bookkeepingstatement sind alle arithmetischen Operatoren erlaubt (im Gegensatz zu Transferstatements).

4) Bookkeeping-Statements dienen nur zur Berechnung von Indizes, die in Connectionstatements gebraucht werden.

5) Bookkeeping-Statements werden durch das Zeichen B vor der Statementnummer gekennzeichnet.

Allgemeiner Aufbau einer Subroutine

Subroutine-Aufruf

Dimensionierung von Zwischenvariablen

Belegung der Zwischenvariablen

Zuordnung Zwischenvariable Subroutinename

Returnstatement

Der allgemeine Aufbau einer Funktion ist analog zu der einer Subroutine. Es entfällt hier lediglich die Dimensionierung der Zwischenvariablen (sie sind alle vom Typ Skalar); aus gleichem Grund können Bookkeeping-statements entfallen.

Die beinahe 1:1 Übersetzung der oben beschriebenen IN (MD)-Subroutine ist in Bild 31 dargestellt. Die Belegung der Zwischenvariablen MDNEW erfolgt im Statement Nr. 4.

$$MDNEW_i \leftarrow MD_i \oplus (\wedge/\omega^{17-i}(18)/MD)$$

Für i = 17 ergibt sich :

$$MDNEW_{17} \leftarrow MD_{17} \oplus (\wedge/\omega^{0}(18)/MD) = MD_{17} \oplus 1 = \overline{MD_{17}}$$

Für i = 16 ergibt sich :

$$MDNEW_{16} \leftarrow MD_{16} \oplus (\wedge/\omega^{1}(18)/MD = MD_{16} \oplus MD_{17}$$

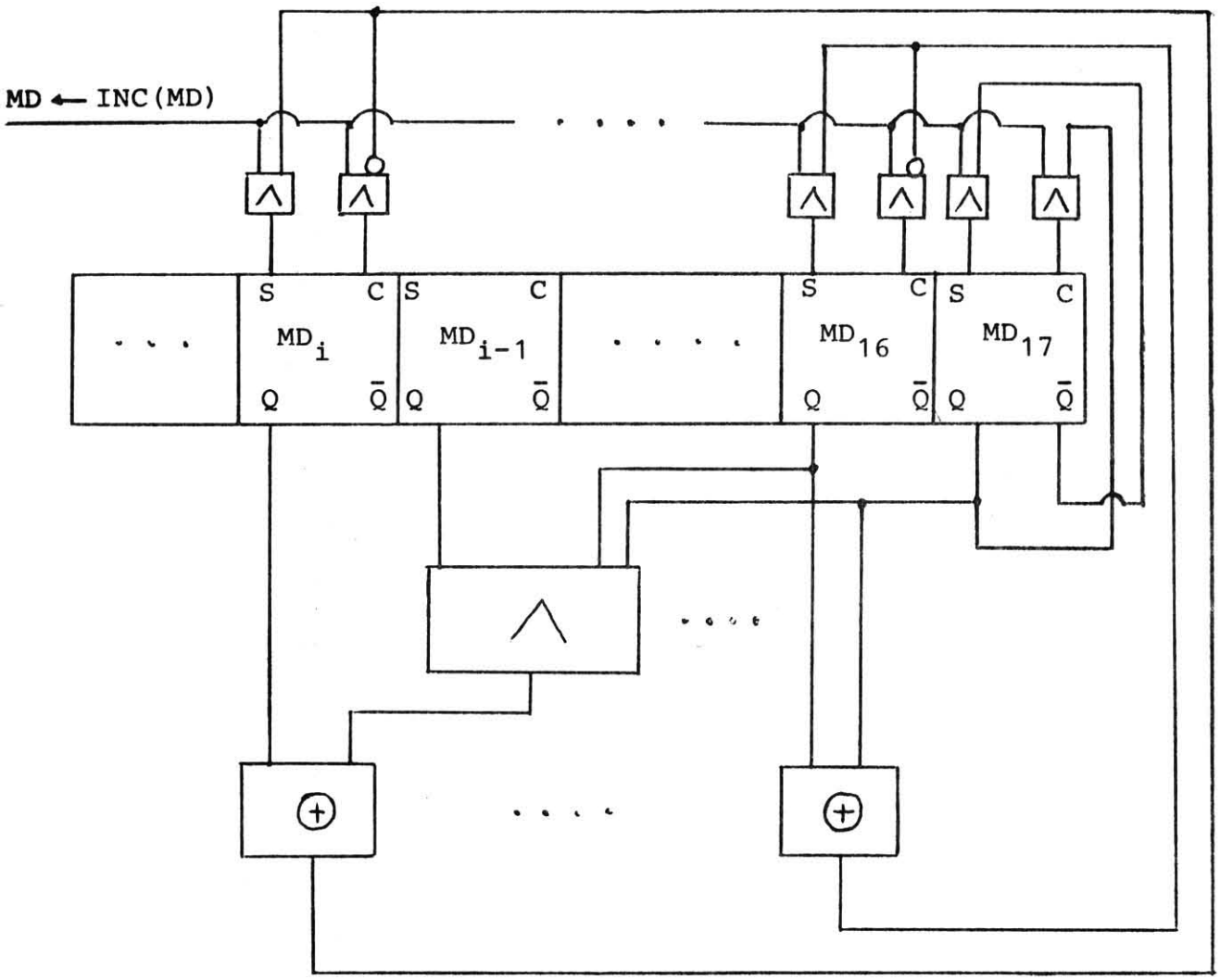


Bild 31

Literaturverzeichnis

K. Iverson	A Programming Language, New York, John Wiley & Sons, Inc., Jan. 1962
F.J. Hill und G.H. Peterson	Digital Systems: Hardware Organization and Design, New York, John Wiley & Sons, Inc. 1973
A.D. Falkoff und K.E. Iverson	"APL 360": User's Manual, International Business Maschines Corporation, 1968
I. Kupka und N. Wilsing,	1973 "Syntax und Semantik des Dialogsprachenkonzepts HDL", Bericht des Instituts, Informatik der Universität Hamburg