Kapitel 2 Maschinenmodelle

In diesem Kapitel sollen verschiedene Modelle behandelt werden, wie ein System beschrieben und realisiert werden kann, das Algorithmen ausführt.

2.1. Interpretative Modelle

Hier sind im vorigen Kapitel schon zwei Modelle behandelt worden:

- die von-Neumann-Maschine
- die Wegener-Maschine.

Beide Modelle gehen davon aus, daß der Algorithmus beschrieben wird durch ein Maschinenprogramm im Speicher, das interpretiert wird.

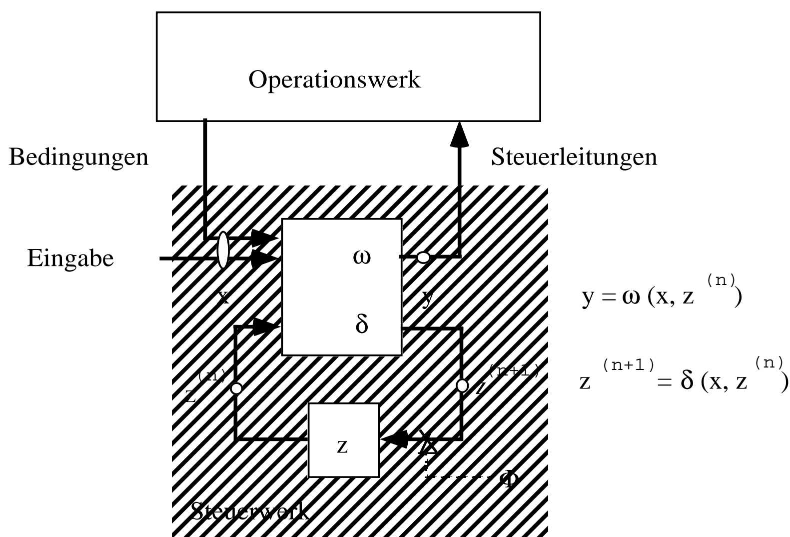
Dabei werden bei der Interpretation im Entschlüssler die Steuersignale für den Abwickler bzw. den datenmanipulierenden Teil erzeugt.

Das Programm ist dabei in viele Einzelschritte zerlegt worden.

2.2. Schaltwerksorientierte Modelle

2.2.1. Schaltwerk als Steuerwerk

Abwickler, Entschlüssler und Programmspeicher sind hier zusammengefaßt im Steuerwerk für den datenmanipulierenden Teil, das Operationswerk. Das Steuerwerk erzeugt die Steuersignale für das Operationswerk und verrechnet Bedingungen aus dem Operationswerk zu den Folgezuständen des Steuerwerks.



$y = \omega(x, z^{(n)})$

$z^{(n+1)} = \delta(x, z^{(n)})$

Dieses Bild modelliert das Steuerwerk in der allgemeinsten Form als Mealy-Automat, mit einem Zustand, codiert in dem Inhalt des Zustandsregisters z und allgemeinen Funktionen realisiert als Schaltnetz mit den Ausgängen y und $z^{(n+1)}$ für die Belegung der Steuerleitungen und den Folgezustand.

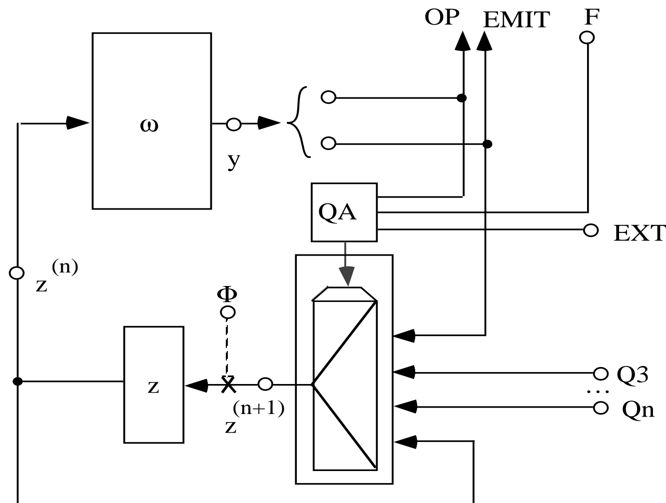
Das System wird vorangetrieben durch einen Takt Ø, der den Übergang von $z^{(n)} \rightarrow z^{(n+1)}$ bewirkt. Das Zustandsregister z übernimmt hier die Rolle des Befehlszählers IP und die Bedingungen die Rolle von F in der Wegener-Maschine.

Üblich sind drei Spezialisierungen des Mealy-Automaten.

- Speicherautomat $y = \omega(z^{(n)})$
  d. h. y hängt nur vom momentanen Zustand ab und kann als Speicher, adressiert durch den Zustand $z^{(n)}$, modelliert werden (Kontrollspeicher).
- der Ausgang y wird interpretiert als Steuerwort, das einerseits die Steuerleitungen aktiviert, andererseits aber auch die Erzeugung des Folgezustands beeinflußt: y besteht aus den Teilen Operationscode OP und EMIT: y .= OP : EMIT
  $\delta(x, z^{(n)}) = \delta_1(x, z^{(n)}, y)$;
- $z^{(n+1)}$ wird modelliert als Ausgang eines Multiplexers, der eine Funktion von $z^{(n)}$, externen Eingängen $Q_3, \dots Q_n$ und dem EMIT-Teil des Steuerworts ist.

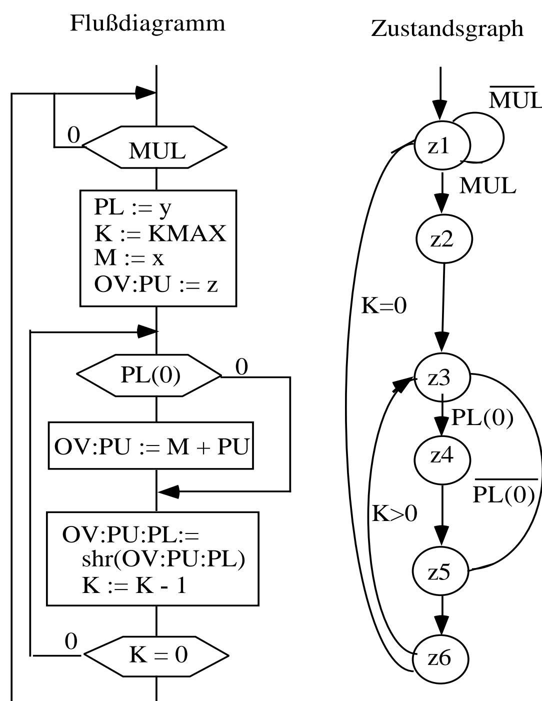
Der Multiplexer wird gesteuert durch eine Funktion QA , deren Eingänge der OP-Teil des Steuerworts, die Bedingungen F aus dem Operationswerk und externe Eingänge EXT sind.

Das erlaubt eine sehr enge Korrespondenz zwischen dem Modell einer Wegener-Maschine und diesem spezialisierten Schaltwerksmodell.



Beispiel: Ein Schaltwerk für den Multiplikationsalgorithmus

Der Algorithmus für die Multipliziermaschine aus Kapitel 1 soll in Form eines Schaltwerks realisiert werden. Eine mögliche Beschreibungsform für Schaltwerke sind Zustandsgraphen. Mit gegebenem Flußdiagramm des Algorithmus läßt sich ein Zustandsgraph unschwer ablesen. Man faßt in Flußdiagrammen alle Operationen, die sich parallel ausführen lassen, d. h. bei denen die Reihenfolge der Abarbeitung keinen Einfluß auf das Ergebnis hat, in jeweils einem Kästchen zusammen, das genau wie die Abfragen je einem Zustand des Schaltwerks zugeordnet wird.



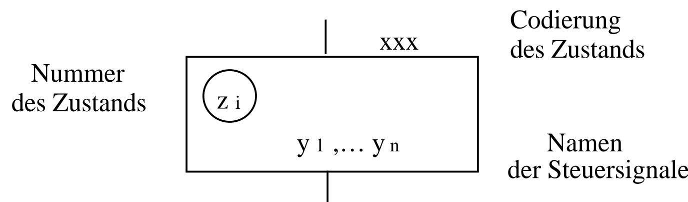
Für den Schaltwerksentwurf sind noch die Steuerleitungen zu kennzeichnen, die im jeweiligen Zustand aktiviert werden sollen.

2.2.2. ASM-Karten

Die Zusammenfassung von Zuständen und aktivierten Steuerleitungen leisten ASM-Karten (algorithmic state maschine).

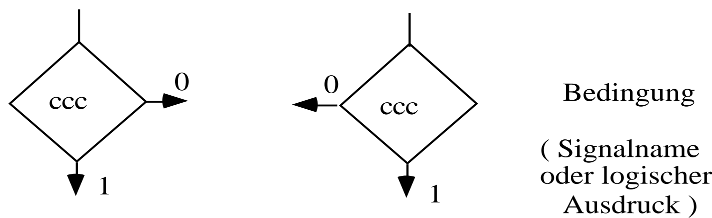
Sie enthalten drei verschiedene Komponenten

Aktionsboxen



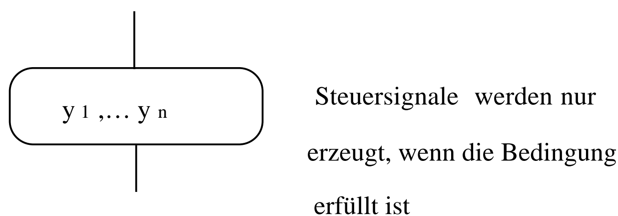
Im Zustand z_i, codiert als xxx, werden die Steuersignale $y_1, ..., y_n$ aktiviert.

Entscheidungsboxen



Im Zustand z_i wird eine Bedingung ccc abgefragt, die ein Signalname oder ein logischer Ausdruck sein kann, und entsprechend verzweigt.

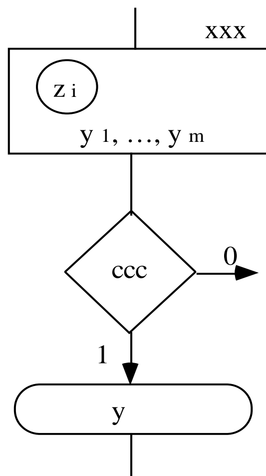
Bedingte Ausgangsboxen



Am Zustand z_i hängt eine Entscheidungsbox. Einer ihrer Ausgänge aktiviert zusätzlich Steuersignale

$y_{c1}, ..., y_{cm}$.

Damit lassen sich Mealy-Automaten beschreiben:



Im Zustand z_i mit der Codierung x, x, x werden die Steuersignale $y_1, ..., y_m$ erzeugt.

Wenn die Bedingung ccc erfüllt ist, wird zusätzlich das Steuersignal y erzeugt. Damit läßt sich jedes Schaltwerk beschreiben:

$$Y = (y_1, ..., y_m, y) = \omega\,(x, z^{(n)})$$

mit $X = (c, ...)$ und $z^{(n)} = (b_{p-1}, ..., b_0)_2$; $b_j \in \mathbb{B}$

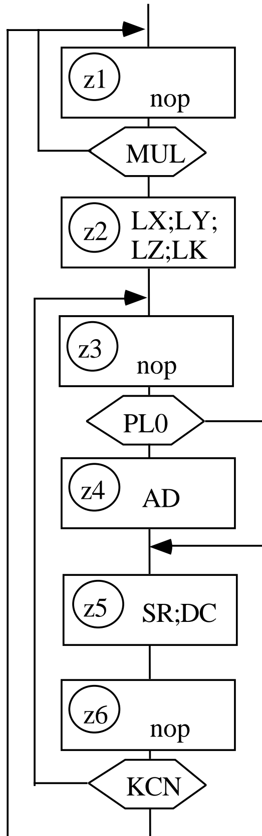
Man wird dafür Sorge tragen, daß die Bedingungen ccc, bei denen Steuersignale erzeugt werden, nicht direkt von den Signalen y im Opreationswerk abhängen, um asynchrone Rückkopplungen zu vermeiden.

Die Umformung des Algorithmus "Multiplizieren" in eine ASM-Karte ist dann z.B. so realisiert:
In drei Zuständen werden Steuerleitungen aktiviert

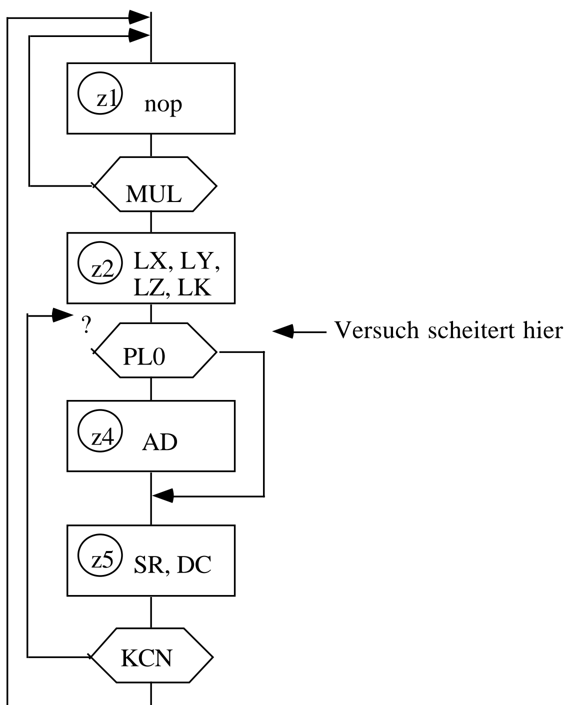
- in z_2 die Leitungen
  LX, LY, LZ und LK
  (Initialisierung)
- in z_4 die Leitung
  AD
  (Addition)
- in z_5 die Leitungen
  SR und DC
  (Schieben und Dekrementieren)

In den anderen Zuständen werden bedingte Sprünge realisiert:

- in z_1 die Abfrage auf das externe Signal MUL
- in z_3 die Abfrage auf das niederwertigste Bit von PL : PLO
- in z_6 die Abfrage auf den Überlauf des Zählers K : KCO

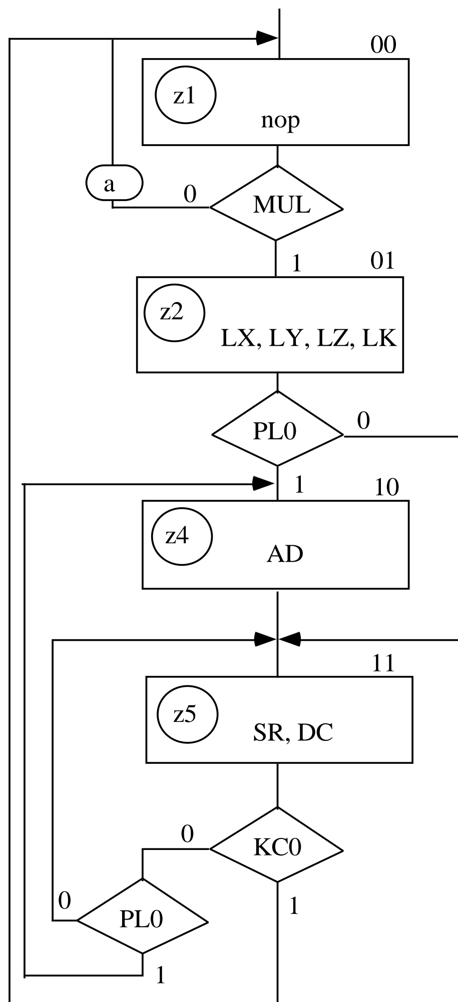


Es liegt nahe zu versuchen, die Zahl der Zustände zu reduzieren, indem man die Trennung zwischen datenverarbeitenden und Abfragezuständen auflöst, und die Abfragen an die darüberstehenden Zustände mit aktivierten Steuerleitungen bindet.



Die Zustandsreduktion ist aber nicht unproblematisch:
Da die Abfrage auf PLO hier an den Zustand z_2 gebunden ist, in dem die Initialisierung gemacht wird, kann der Rücksprung zu der Abfrage nicht separat erfolgen.

Erfolg bringt hier die Trennung der Abfrage auf PLO : einmal beim Eintritt in die Schleife aus dem Zustand z_2 und dann beim Verlassen von z_5 im Zweig mit KCO = 0 .



Ferner wird hier im Zustand z_1 bei der Abfrage auf MUL ein externes Signal a bei MUL = 0 erzeugt, womit das System einen Mealy-Automaten beschreibt.

Die Codierung der Zustände ist willkürlich.

Daraus leitet sich dann der klassische Schaltwerksentwurf

mit den Funktionen $\delta\,(x, z^{(n)})$ und $\omega\,(x, z^{(n)})$ ab:

sei $x = (KCO, PLO, MUL)$ und $z^{(n)} = (z_1{}^{(n)}, z_0{}^{(n)})$

und $y = (L^*, AD, SR, DC, a)$ dann ist die Funktionstafel

Zustand		Bedingung			Folgezustand		Steuerleitungen			
$z_1^{(n)}$	$z_0^{(n)}$	KCO	PLO	MUL	$z_1^{(n+1)}$	$z_0^{(n+1)}$	L*	AD	SR,DC	a
0	0	*	*	0	0	0	0	0	0	1
0	0	*	*	1	0	1	0	0	0	0
0	1	*	0	*	1	1	1	0	0	0
0	1	*	1	*	1	0	1	0	0	0
1	0	*	*	*	1	1	0	1	0	0
1	1	0	0	*	1	1	0	0	1	0
1	1	0	1	*	1	0	0	0	1	0
1	1	1	*	*	0	0	0	0	1	0

und daraus abgeleitet die gesuchten Funktionen

$\delta(x, z^{(n)})$

$$z_0^{(n+1)} = \overline{z_1^{(n)}}\,\overline{z_0^{(n)}}\,MUL + \overline{z_1^{(n)}}\,z_0^{(n)}\,PL0 + z_1^{(n)}\,\overline{z_0^{(n)}} + z_1^{(n)}\,z_0^{(n)}\,\overline{KC0}\,\overline{PL0}$$

$$z_1^{(n+1)} = \overline{z_1^{(n)}}\,z_0^{(n)} + z_1^{(n)}\,\overline{z_0^{(n)}} + z_1^{(n)}\,z_0^{(n)}\,KC0$$

$\omega(x, z^{(n)})$

$$L^* = LX = LY = LZ = LK = \overline{z_1^{(n)}}\,z_0^{(n)}$$

$$AD = z_1^{(n)}\,\overline{z_0^{(n)}}$$

$$SR = DC = z_1^{(n)}\,z_0^{(n)}$$

$$a = \overline{z_1^{(n)}}\,\overline{z_0^{(n)}}\,\overline{MUL}$$

Damit ist das Schaltwerk für die Ansteuerung des datenverarbeitenden Teils der Multipliziermaschine vollständig beschrieben.

2.3. Geschachtelte Maschinen

2.3.1. Maschinenhierarchien

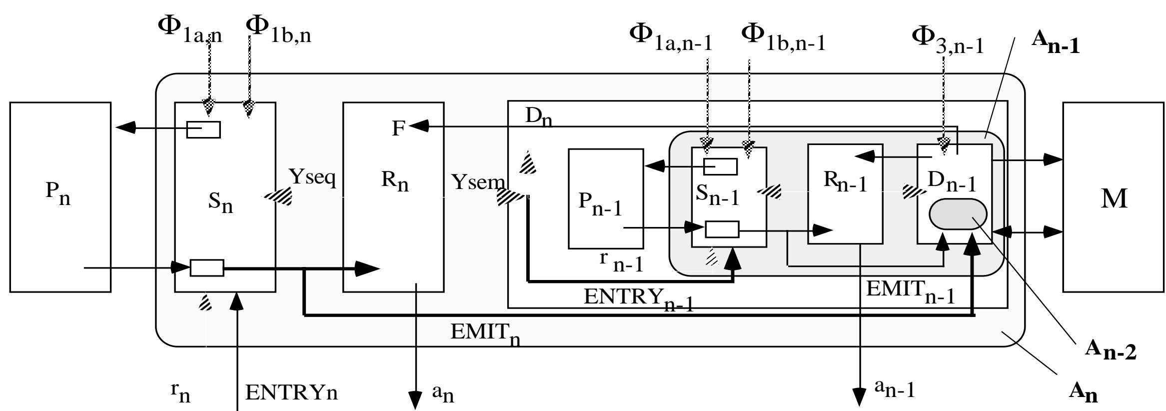
Ein Befehl wird in drei Phasen abgearbeitet:

φ_1 : Befehl holen: at $Ø_{1a}$ do IP := NEW
at $Ø_{1b}$ do IR := P(IP)

φ_2 : Befehl entschlüsseln: kein Takt nötig, wenn nur ein Schaltnetz da ist;

φ_3 : Befehl ausführen: ein Takt $Ø_3$ oder viele Takte; Übernahme der Ergebnisse in Register.

Wenn die Phase $Ø_3$ viele Takte braucht, um das Ergebnis des Befehls zu berechnen, dann wird der Datenmanipulator durch ein eigenes Schaltwerk realisiert oder durch **geschachtelte Maschinen** oder **Mehrebenenmaschinen**.



Sei A_n die äußere Maschine, A_{n-1}, A_{n-2}, ... innere Maschinen, die den Datenmanipulator und darin noch einmal den Datenmanipulator der inneren Maschine realisieren.

Man nennt A_{n-1} die **Mikromaschine** und A_{n-2} die **Nanomaschine**.

Im allgemeinen kommt man mit drei ineinandergeschachtelten Maschinen aus, häufig schon mit einer inneren Maschine A_{n-1}.

Wenn sie frei programmierbar ist, spricht man von einer **mikroprogrammierbaren** Maschine, sonst von einer **mikroprogrammierten** Maschine.

Das Programm in P_{n-1} nennt man das Mikroprogramm, ein Programm in P_{n-2} bezeichnet man als **Nanocode**.

Für den Programmierer ist von außen sichtbar die Maschine A_n. Sie kennt z. B. einen Befehl "sinus", der intern durch ein Mikroprogramm in P_{n-1} beschrieben ist und in der Mikromaschine mit vielen Taktschritten realisiert wird.

Der Vorteil mikroprogrammierter Maschinen ist der einfache Aufbau aus Schaltnetzen für

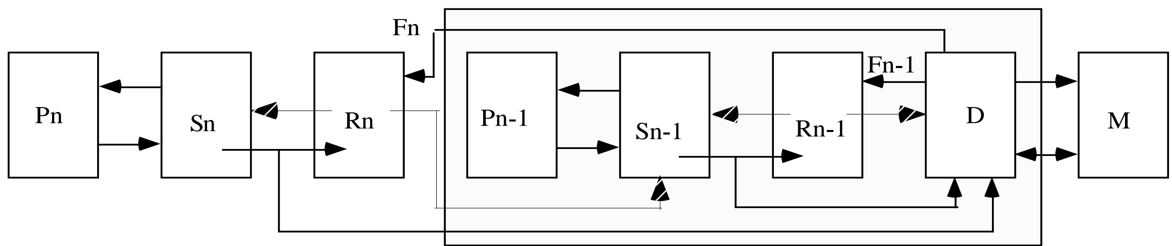
einige primitive Funktionen, die durch Mikroprogramme komplexe Funktionen realisieren. Erkauft allerdings mit längeren Befehlslaufzeiten. Daher geht der Trend hin zu komplexen Schaltnetzen, die erheblich schneller sind.

In der Abbildung wird ein Teil von Y_{sem} zur Ansteuerung des Datenmanipulators D_n verwendet, ein Teil als "Entry" in die Mikromaschine benutzt.

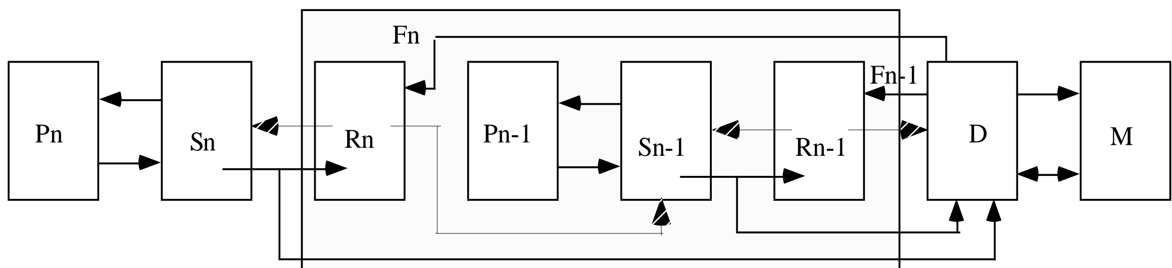
Man hat den typischen Fall eines komplexen Datenmanipulators vor sich, bei dem einige Befehle durch eine Mikromaschine realisiert sind.

Wird nun Y_{sem} vollständig als ENTRY in S_{n-1} benutzt, hat man eine Konfiguration dieser Art:

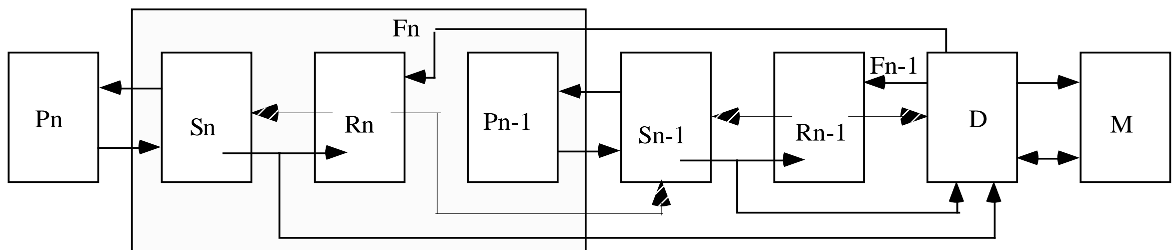
Mit einem Interpretationsrahmen über D_n hat man wie bisher einen komplexen Datenmanipulator.



Verschiebt man den Interpretationsrahmen über den Entschlüssler, kann man die gleiche Anordnung auch als **komplexer Entschlüssler** sehen.



Schiebt man den Interpretationsrahmen über S_n, dann kann man die Anordnung als



sprachinterpretierende Maschine, beschreiben. Man spricht dann auch von einem **language interpreting module** oder LIM:

Ein einfacher Befehl einer höheren Sprache, z.B. C, wird interpretiert, indem er in eine Vielzahl von Befehlen der niederen Sprache umgesetzt wird.

2.3.2. Taktung geschachtelter Maschinen

Betrachten wir zunächst die Takterzeugung für eine nicht geschachtelte Maschine, deren Entschlüssler ein Schaltnetz ist:

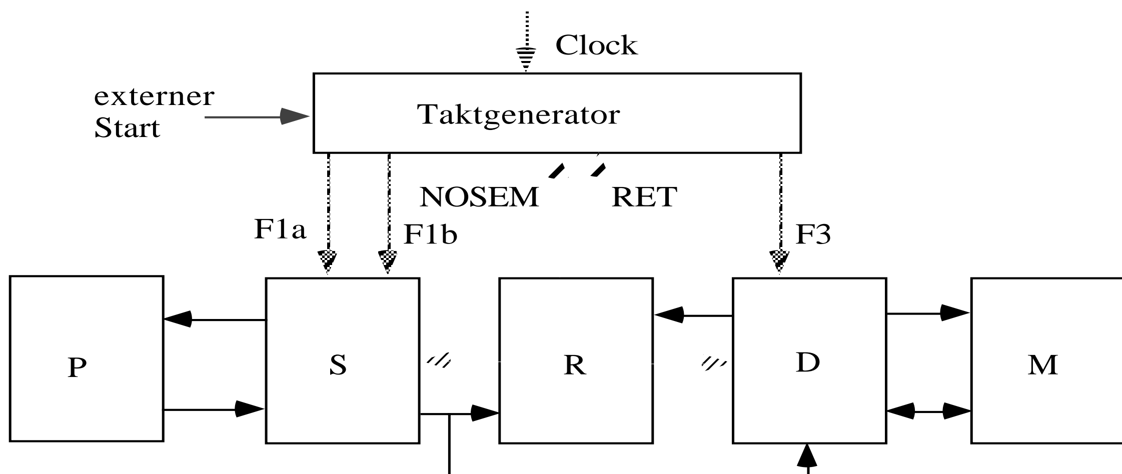
In Phase φ_1 müssen zwei Takte erzeugt werden, $Ø_{1a}$: IP := NEW und $Ø_{1b}$: IR := P(IP)

In Phase φ_2 kein Takt.

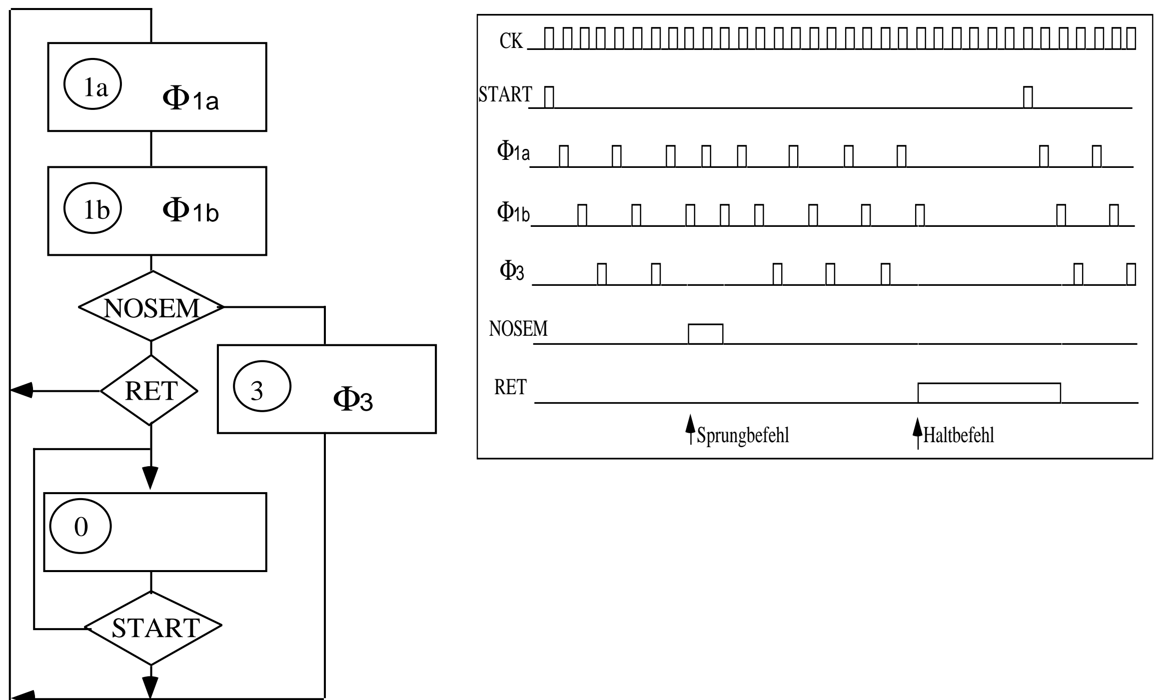
In Phase φ_3 ein Takt $Ø_3$ für die Übernahme der Ergebnisse oder viele Takte bei inneren Maschinen.

Die Takte werden in einem Taktgenerator erzeugt, der aus einem Clocksignal (Quarzgesteuerter Oszillator, 10 - 500 MHz) die Takte erzeugt und aus der Maschine zwei Signale erhält: NOSEM wenn ein Sprungbefehl entdeckt wird (ein Bit im Befehl), dann braucht $Ø_3$ nicht erzeugt zu werden, und RET wenn ein Haltbefehl entschlüsselt wird, womit der Taktgenerator gestoppt werden kann.

Er muß dann allerdings mit einem START-Signal wieder angeworfen werden.



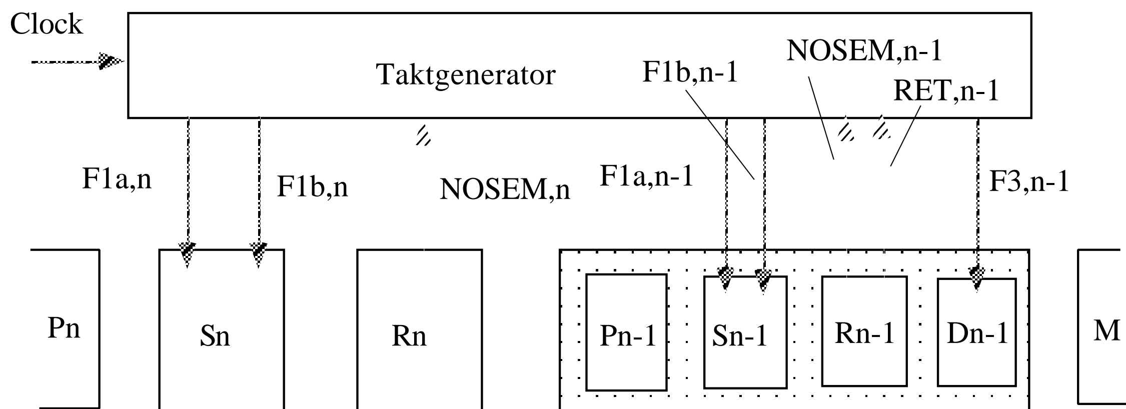
Die Funktion dieses Taktgenerators wird durch die folgende ASM-Karte und einen Impulplan beschrieben.



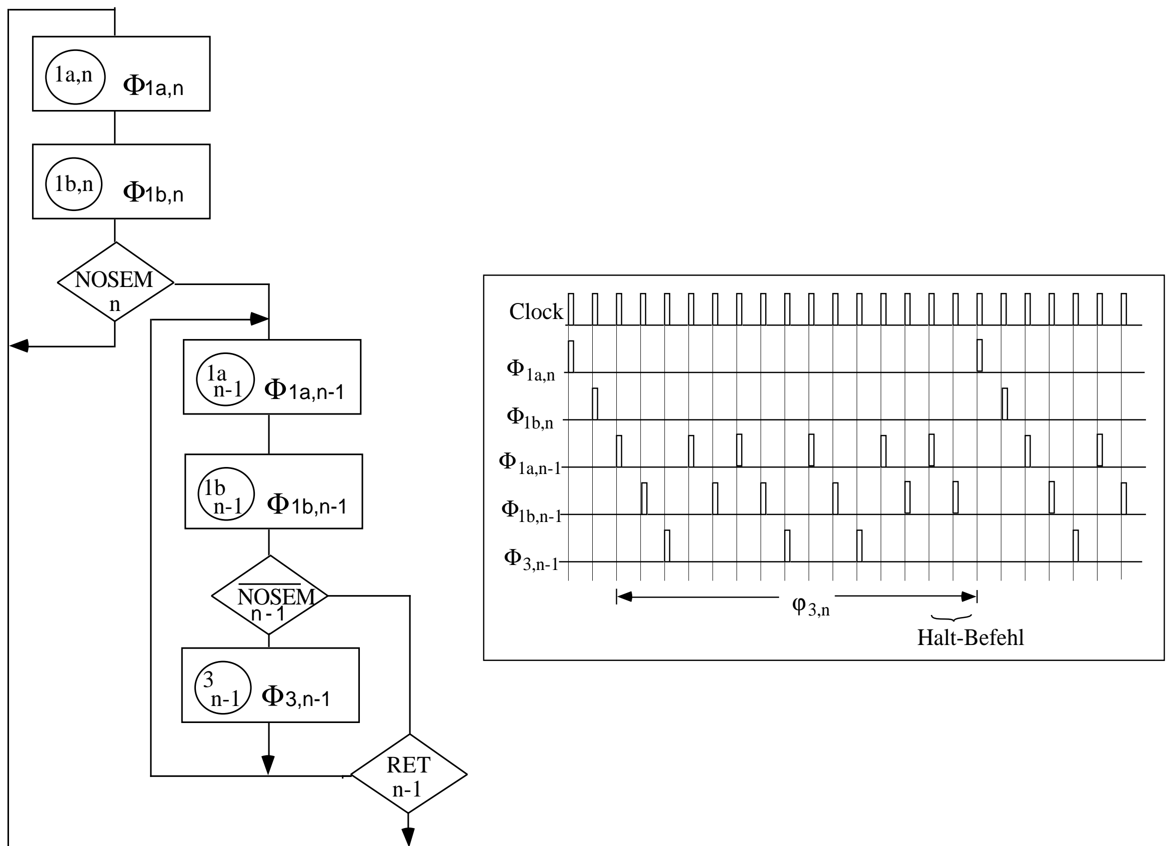
Man sieht, daß die Clock zwar weiterläuft; aber nach einem "Halt" werden bis zum nächsten START keine Takte mehr erzeugt.

Man spricht hier auch von **busy waiting**.

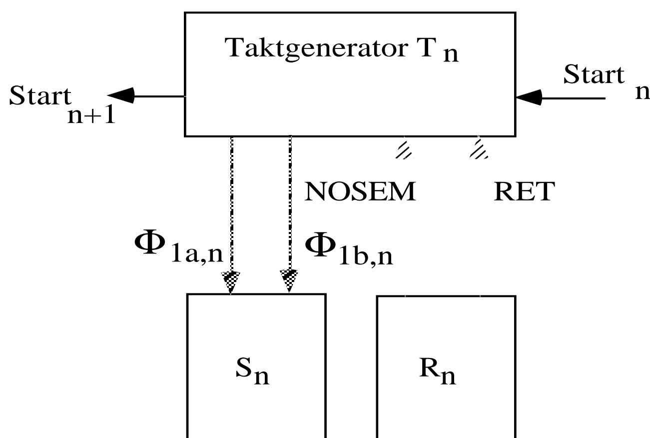
Betrachten wir nun die Taktung für zwei **ineinandergeschachtelte Maschinen**:



Auch hier läßt sich die Takterzeugung durch eine ASM-Karte beschreiben und der Zeitverlauf der Takterzeugung mit einem Impulsplan beispielhaft verdeutlichen.

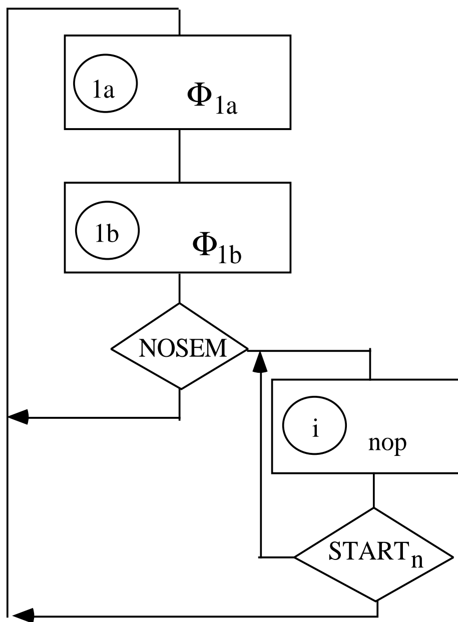


Der Taktgenerator läßt sich aus Teilen für die Maschine A_n zusammensetzen:

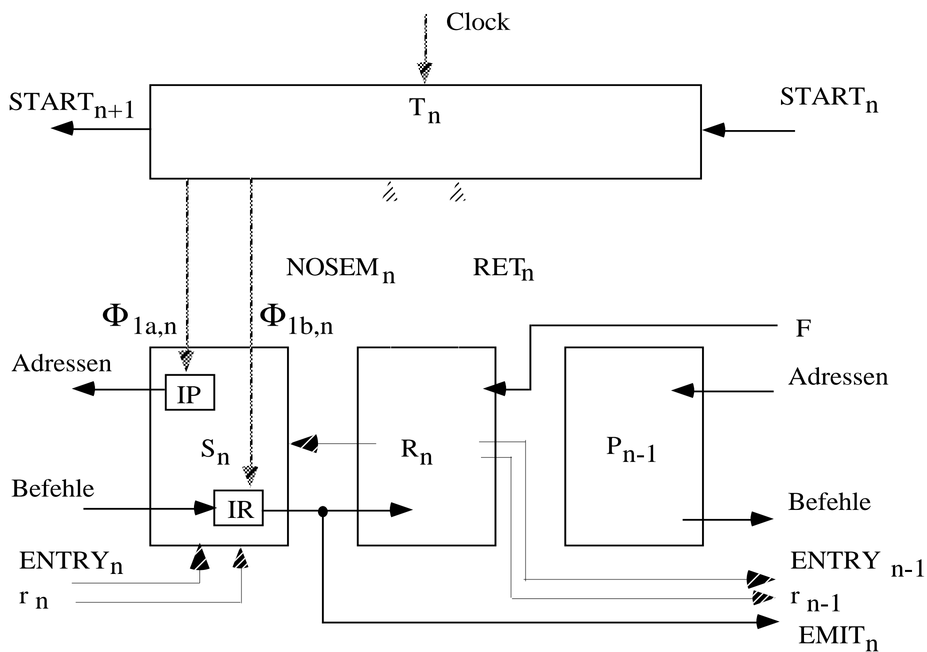


Das RET-Signal aus R_n ist das Startsignal für die nächsthöhere Ebene.

Die ASM-Karte für die Taktsignalerzeugung zeigt das nächste Bild.

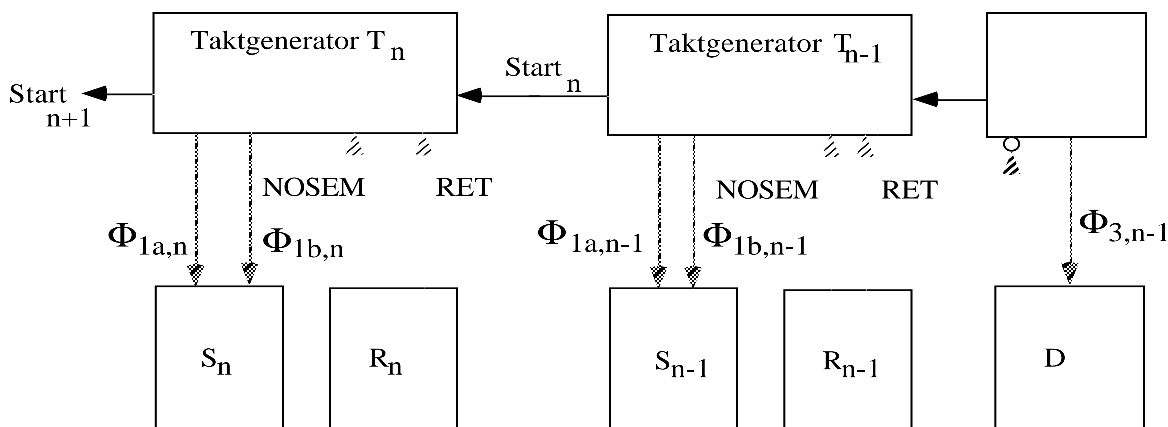


Die Anwendung dieser Generatormodule kann z. B. in einem sprachinterpretierenden Modul (LIM) erfolgen.



Schließlich braucht man dann noch ein Modul für die Erzeugung von $Ø_3$: mit $\neg$NOSEM wird der Generator angestoßen, erzeugt einen Takt und ein RET-Signal.

Die Takterzeugung einer geschachtelten Maschine geschieht dann mit drei Taktmodulen.



2.3.3. Beispiel: Interpretation eines Programms

Es soll das Programm der Primitiven Maschine aus Kapitel 1.2.

(3 4 * 5 6 * + Λ)

mit Hilfe der Simple Machine ausgeführt werden.

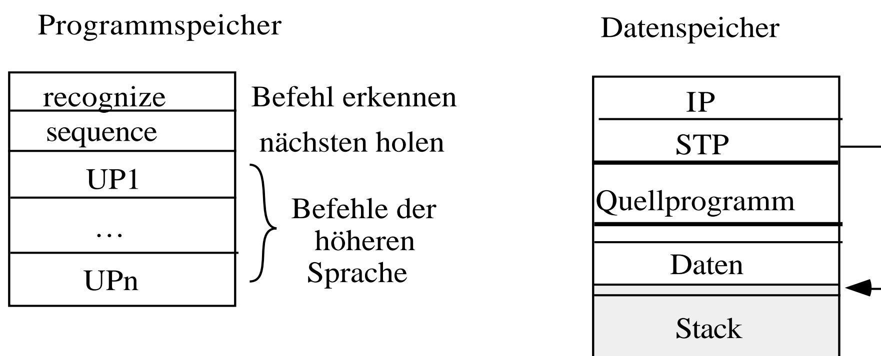
Das soll auf zwei Weisen geschehen:

a) durch Interpretation mit Software in einer 1-Ebenen-Maschine

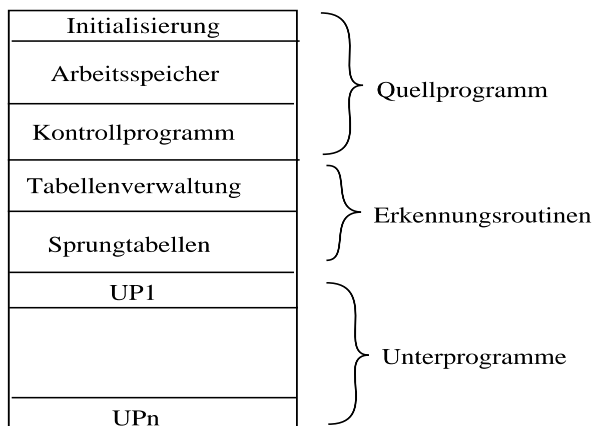
b) durch Anwendung des LIM-Schemas in einer 2-Ebenen-Architektur.

Damit soll die Beschleunigung bei der Befehlsabarbeitung sichtbar gemacht werden, die durch den Einsatz der zusätzlichen Hardware erreicht wird.

Die **Softwarelösung** hat einen Programm- und einen Datenspeicher mit folgender Organisation:



Insbesondere sind die Phasen φ_1 und φ_2 der Abarbeitung eines Befehls der Primitiven Maschine hier durch Programme "sequence" und "recognize" realisiert.
Werden beide Programme in einem Speicher realisiert sieht die Organisation z.B so aus:



Das zugehörige Programm ist

```
RECOG: (* Phase 2 : entschlüsseln* )
        DR  := ‘*’
        if (ACC ≠ DR)  then goto S1
        call MULT
        goto SEQ

S1:     DR  := ‘+’
        if (ACC ≠ DR)  then goto S2
        call ADD
        goto SEQ

S2:     DR  := ‘L’
        if (ACC ≠ DR)  then goto S3
        halt

S3:     call CONST

SEQ: (* Phase1 : Befehl holen )
        DP  := ‘IP’
        DR  := M ( DP ) (*hole IP*)
        DP  := DR
        DP  := DP + 1 (*modifiziere IP*)
        DR  := M ( DP )
        ACC  := DR (*lade P ( IP )*)
        DR  := DP
        DP  := ‘IP’(*speichere  IP zurück*)
        M ( DP )  := DR
        goto RECOG

CONST:
        DP  := ‘STP’ (*hole Stackpointer*)
        DR  := M ( DP )
        DP  := DR (*modifiziere STP*)
        DP  := DP + 1
        DR  := DP (*speichere STP zurück*)
        DP  := ‘STP’
        DR  := M ( DP )
        DP  := DR (*speichere Konstante*)
        DR  := ACC (*im ACC nach  M (STP)*)
        M ( DP )  := DR
        return
```

```
MULT:
        DP  := ‘STP’
        DR  := M ( DP ) (*hole Stackpointer*)
        DP  := DR
        DR  := M ( DP ) (*lade M ( STP )*)
        DP  := DP - 1 (*dekrementiere STP*)
        ACC  := DR
        DR  := M ( DP ) (*multipliziere*)
        ACC  := ACC * DR
        DR  := ACC
        M ( DP )  := DR (*speichere Ergebnis*)
        DR  := DP
        DP  := ‘STP’
        M ( DP )  := DR (*speichere STP zurück*)
        return

ADD:
        DP  := ‘STP’
        DR  := M ( DP ) (*hole Stackpointer*)
        DP  := DR
        DR  := M ( DP ) (*lade M ( STP )*)
        DP  := DP - 1 (*dekrementiere STP*)
        ACC  := DR
        DR  := M ( DP ) (*addiere*)
        ACC  := ACC + DR
        DR  := ACC
        M ( DP )  := DR (*speichere Ergebnis*)
        DR  := DP
        DP  := ‘STP’
        M ( DP )  := DR (*speichere STP zurück*)
        return
```

Es umfaßt 12 Befehle für Phase 2, 10 Befehle für Phase 1 und 11, 14 und 14 Befehle für die Operationen CONST, MUL und ADD: 61 Befehle insgesamt.

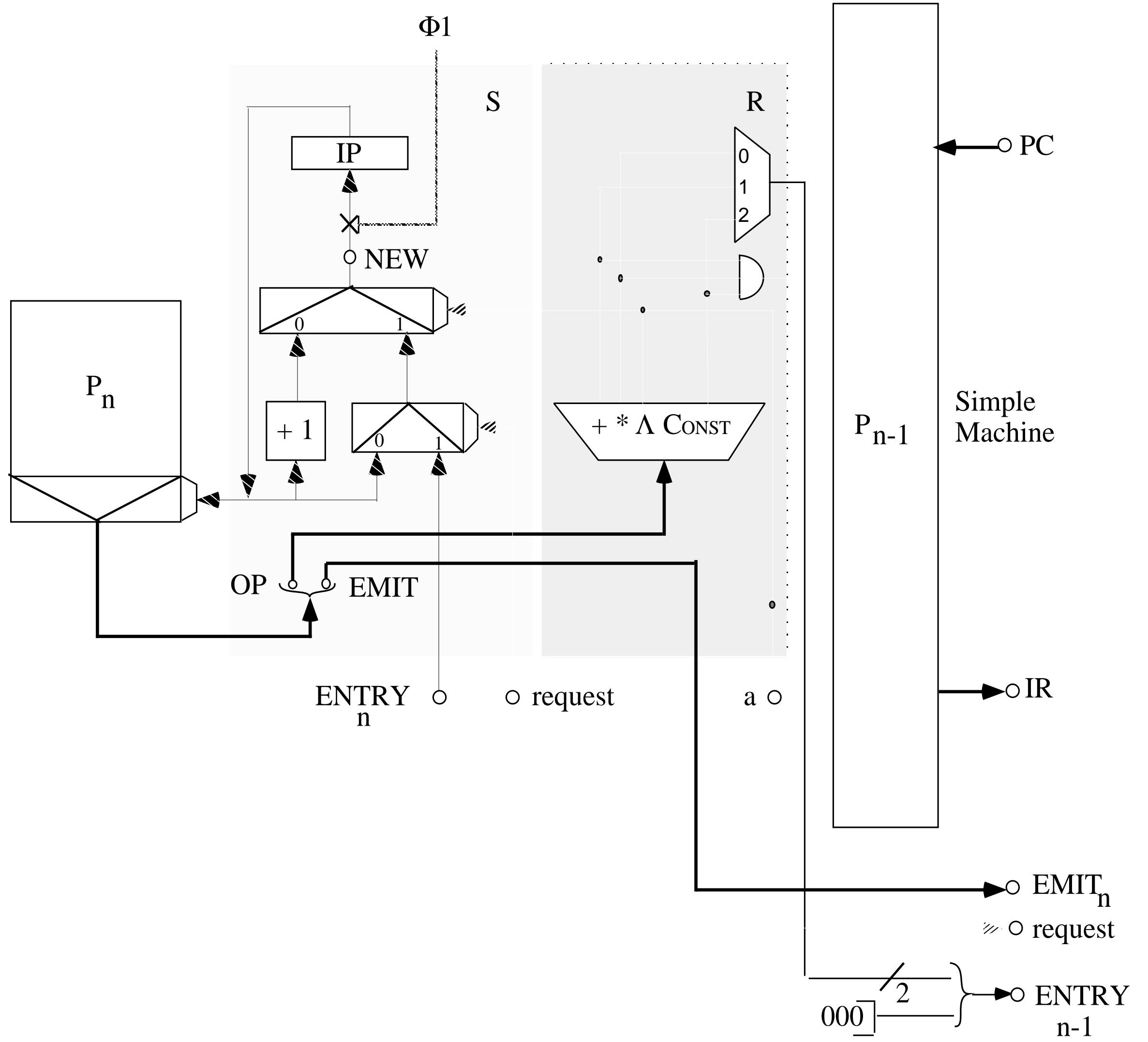
Die **Hardware-unterstützte Lösung** wendet das LIM-Schema an.

Das Programm zeigt dann die nächste Abbildung.

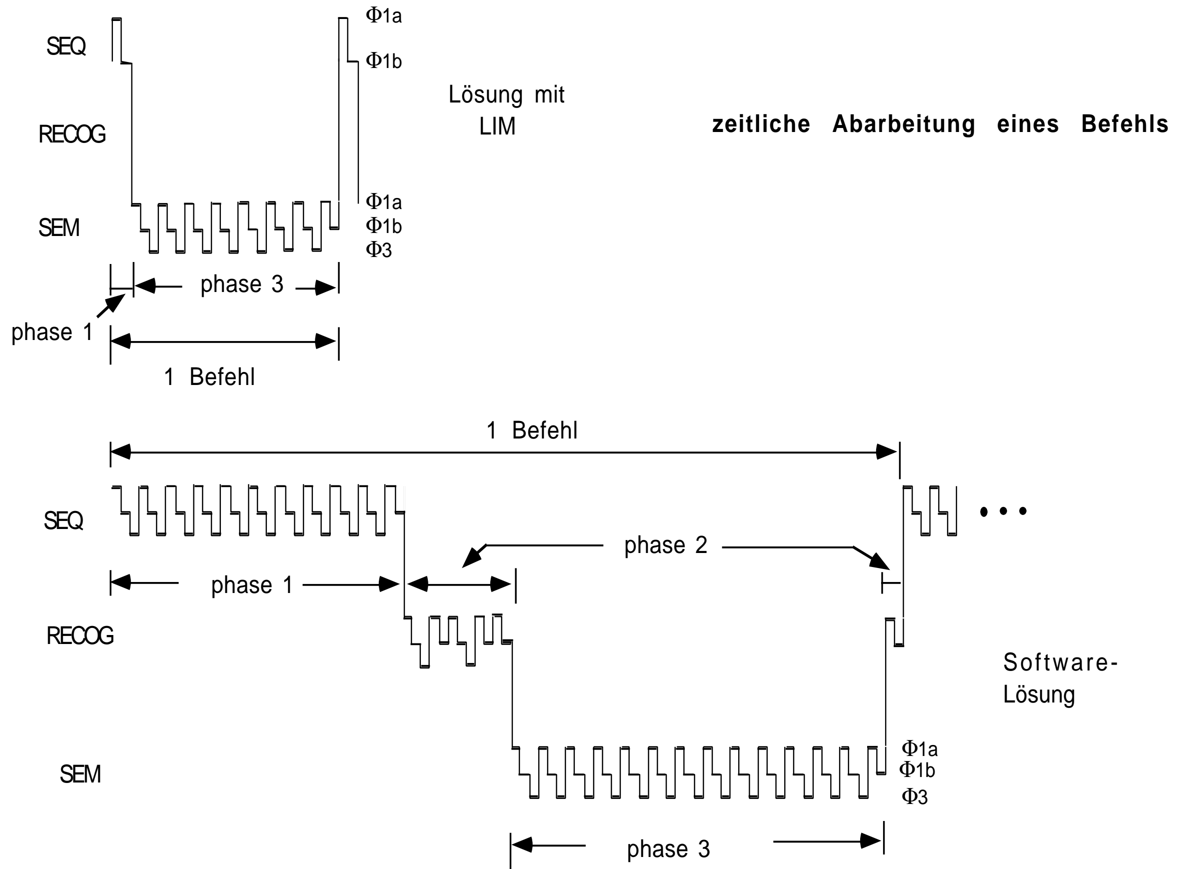
Programmspeicher der äußeren (PRIMITIVE) Maschine	Programmspeicher der inneren (SIMPLE) Maschine
	M (DP - 1) := M (DP) * M (DP - 1)
000 CONS 3	00 000 MUL DR := M (DP)
001 CONS 4	001 ACC := DR
010 * -	010 DP := DP - 1
011 CONS 5	011 DR := M (DP)
100 CONS 6	100 ACC := ACC * DR
101 * -	101 DR := ACC
110 + -	110 M (DP) := DR
111 L -	111 halt
	M (DP - 1) := M (DP) + M (DP - 1)
	01 000 ADD DR := M (DP)
	001 ACC := DR
	010 DP := DP - 1
	011 DR := M (DP)
	100 ACC := ACC + DR
	101 DR := ACC
	110 M (DP) := DR
	111 halt
	10 000 CONST DP := DP - 1
	001 DR := EMIT
	010 M (DP):= DR
	011 halt

In P_n stehen die Befehle des Quellcodes, in P_{n-1} die Unterprogramme, die den Befehlen, hier den Stackbefehlen der Primitiven Maschine, zugeordnet sind: in P_n 8 Befehle und in P_{n-1} 20 Befehle in der jeweiligen Maschinensprache der äußeren bzw. inneren Maschine.

Die zusätzliche Hardware ist das Register IP und das Schaltnetz des LIM-Moduls, das die Befehlsfortschaltung und -entschlüsselung übernimmt.


Sprachinterpretierender Modul
Φ1
S
R
IP
NEW
0
1
2
+ 1
+ * Λ CONST
P
n
OP
EMIT
ENTRY
n
request
a
PC
P
n-1
Simple
Machine
IR
EMIT
n
request
2
000
ENTRY
n-1

Den zeitlichen Gewinn zeigt ein Impulsdiagramm für beide Lösungen.



Bei der LIM-Lösung werden φ_1 und φ_2 durch wenige Takte realisiert, während die Softwarelösung jeweils ein Programm ablaufen läßt.

2.4. DLX-Maschine

2.4.1. Befehlsstruktur

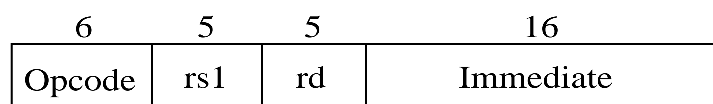
Es ist dies eine prototypische Maschine, eingeführt von Hennessy und Patterson als eine Mixtur aus einer Reihe von Maschinen Anfang der 80er Jahre:
AMD 29000, MIPS R 2000, Motorola 88000, SPARC-1, HP-850, SUN-4/260, DEC-3100, Intel i-860.

Eigenschaften:

- 32-Bit Adressbreite, d. h. 4 G Byte Adressraum
- 32-Bit Speicherzugriffsbreite
- 32 Universalregister (general purpose register, GPR) von Wortbreite; davon R1 - R31 normale Register, R0 ≡ 0
- 32 Gleitkommaregister für einfach-genaue GK-Zahlen. Sie können auch als 16 doppelt genaue Register F0, F2, ..., F14 angesprochen werden.
- Speicher byteadressiert mit ausgerichteten Adressen (d. h. Adressfortschaltung springt jeweils um 4 Byte).
  Die Befehle sind 32 Bit lang, ausgerichtet an den Wortgrenzen. Die Adressen 4n, 4n+1, 4n+2, 4n+3 bilden ein Wort.
- typische Befehle von RISC-Architekturen (reduced instruction set computer, RISC).

Die Befehle haben alle eine einheitliche Länge von hier 32 Bit. Es gibt drei Befehlsformate mit jeweils 6 Bit Opcode:

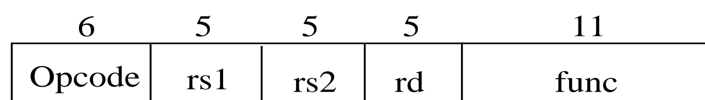
i-Typ Befehle



Quelle Senke

in denen ein Quellregister rs1, eine Senke rd und ein Immediatefeld definiert sind.

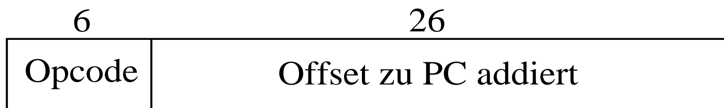
R-Typ-Befehle



Quelle1 Quelle2 Senke

in denen 2 Quellregister rs1 und rs2 und eine Senke rd definiert sind, mit denen die typischen 3-Adressbefehle realisiert werden: der neue Inhalt von Register rd ergibt sich als Inhalt von Register rs1 verknüpft mit dem Inhalt von Registre rs2: <rd> := <rs1> op <rs2>.

I-Typ-Befehle



die Sprünge beschreiben.

Die Befehle gliedern sich auf in

Datentransportbefehle

zum Datentransport zwischen Registern und Speicher (Load-Store-Architektur) oder Festkomma- und Gleitkommaregistern oder Spezialregistern.

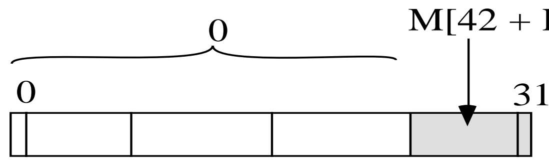
Die Speicheradressierungsart ist nur die Adressangabe als Inhalt eines GPR und 16-Bit Displacement.

LB, LBU, SB	: Bytebefehle - laden, vorzeichenlos, speichern
LH, LHU, LS	: Halbwortbefehle - laden, vorzeichenlos, speichern
LW, SW	: Ganzwortbefehle - laden, speichern
LF, LD, SF, SD	: GK - laden / speichern, einfach, doppelt genau
MOVI2S, MOVI2I	: Transport vom/zu GPR zu/vom Spezialregister
MOVF, MOVD	: GK-Transport einfach/doppelt genau
MOVFP2I, MOVI2FP	: Transport vom/zu GK-Register zu/von FK-Register

Beispiele:

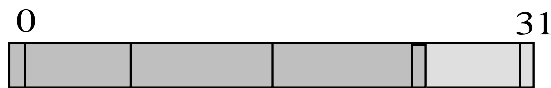
Bei der Beschreibung der Wirkung der Befehle kennzeichnet ## die Konkatenation von Bits, x^n die Anzahl Bits vom Typ x, $_m z$ die Länge des Bitstrings z und p_k das k-te Bit des Bytes p.

LBU R2, 42 (R3) lade Byte vorzeichenlos



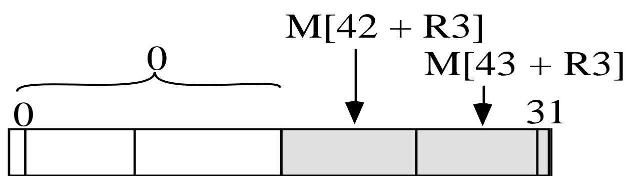
$$R2 <--_{32} 0^{24} \#\# M[42 + R3]$$

LB R2, 42 (R3) lade Byte mit Vorzeichen



$$R2 <--_{32} (M[42 + R3]_0)^{24} \#\# M[42 + R3]$$

LHU R2, 42 (R3) lade Halbwort vorzeichenlos



$$R2 <--_{32} 0^{16} \#\# M[42 + R3] \#\# M[43 + R3$$

SD 40 (R3), F0 speichere doppelt genauen GK-Wert aus GK-Registern F0, F1 in M (40 + R3) und M (44 + R3).

Arithmetisch-logische Befehle

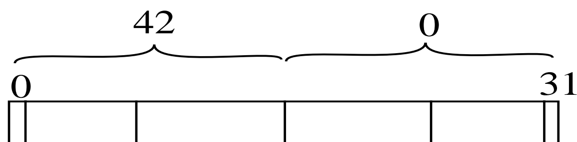
Operationen auf Festkommazahlen oder logischen Werten in GPR's.

Bei Überlauf wird ein vorzeichenbehafteter Arithmetik-Trap erzeugt.

ADD, ADDI, ADDU, ADDUI	addiere mit/ohne Immediate, mit/ ohne Vorzeichen
SUB, SUBI, SUBU, SUBUI	subtrahiere mit/ ohne Immediate, mit/ohne Vorzeichen
MULT,MULTU, DIV, DIVU	Multiplikation und Division, mit/ohne Vorzeichen Operanden sind GK-Register, 32 Bit
AND,ANDI	logisches UND mit/ohne Immediate
OR, ORI, XOR, XORI	logisches ODER und XOR mit/ohne Immediate
LHI	Laden der oberen Hälfte des Registers mit Immediate
SLL, SRL, SRA, SLLI, SRLI, SRAI	Schieben - um Immediate/ Wert in Register links logisch, rechts logisch, rechts arithmetisch
S___, S___I	Bedingung in Register laden Bedingung kann sein LT, GT, LE, GE, EQ, NE

Beispiele:

ADD R1, R2, R3	addiere	$R1 \leftarrow R2 + R3$
LHI R2, # 42	lade Konstante 42_{10} (Immediate) in die obere Hälfte von R2	$R2 \leftarrow_{32} 42 \# \# 0^{16}$



$R2 \leftarrow_{32} 42 \#\# 0^{16}$

$42_{10} = 00000000\ 00101010_2$

SLLI R1, R2, # 5	schiebe logisch links R2 um 5 Stellen	R1 ← R2 << 5
SLT R1, R2, R3	setze R1 nach Vergleich von R2 und R3	R1 ← (if (R2 < R3) then 1 else 0)

Steuerungsbefehle

Bedingte Verzweigungen und Sprünge; PC-relativ oder mit Register.

BEQZ, BNEZ	: Verzweigen bei GPR gleich/ ungleich Null; 16 Bit Offset von PC + 4
BFPT, BFPF	: Testen des Vergleichsbit im GK-Statusregister und Vezweigen 16-Bit Offset von PC + 4
J, JR	: unbedingte Sprünge, 26 Bit Offset von PC (relativer Sprung) oder PC <-- R (absoluter Sprung)
JAL, JALR	: UP-Sprung; Retten von PC + 4 in R31 (!) Ziel: PC + 26-Bit Offset oder PC <-- R
TRAP	: Übergang zum Betriebssystem bei vektorisierter Adresse
RFE	: Rückkehr zum Benutzercode nach einem trap der Benutzermodus wird rückgespeichert

Beispiele:

BEQZ R4, name	branch not equal zero	PC ← (if (R4 != 0) then name else PC+4)
JR R3	jump register unbedingter Sprung	PC ← R3
JAL name	jump and link register	R31 ← PC+4 PC ← name

dabei muß für Sprungziel "name" eine Adresse stehen mit

$((PC+4) - 2^{25}) \leq name < ((PC+4) + 2^{25})$

Gleitkommabefehle

Es werden Gleitkommabefehle mit einfach oder doppelt genauen GK-Zahlen spezifiziert.

ADDD, ADDF	: Addition von DP- oder SP-Zahlen
SUBD, SUBF	: Subtraktion von DP- oder SP-Zahlen
MULTD, MULTF	: Multiplikation von DP- oder SP-Zahlen
DIVD, DIVF	: Division von DP- oder SP-Zahlen
CVTF2D, CVTF2I CVTD2F, CVTD2I CVTI2F, CVTI2D	: Konvertierungsbefehle; : CVTx2y konvertiert von Typ x zum Typ y I <-> FK, D <-> DP, F <-> SP (integer, double, floating)
____D, ____F	: DP- und SP-Vergleich; "____" ist LT, GT, GE, LE, EQ, NE setzt Vergleichsbit im GK-Statusregister

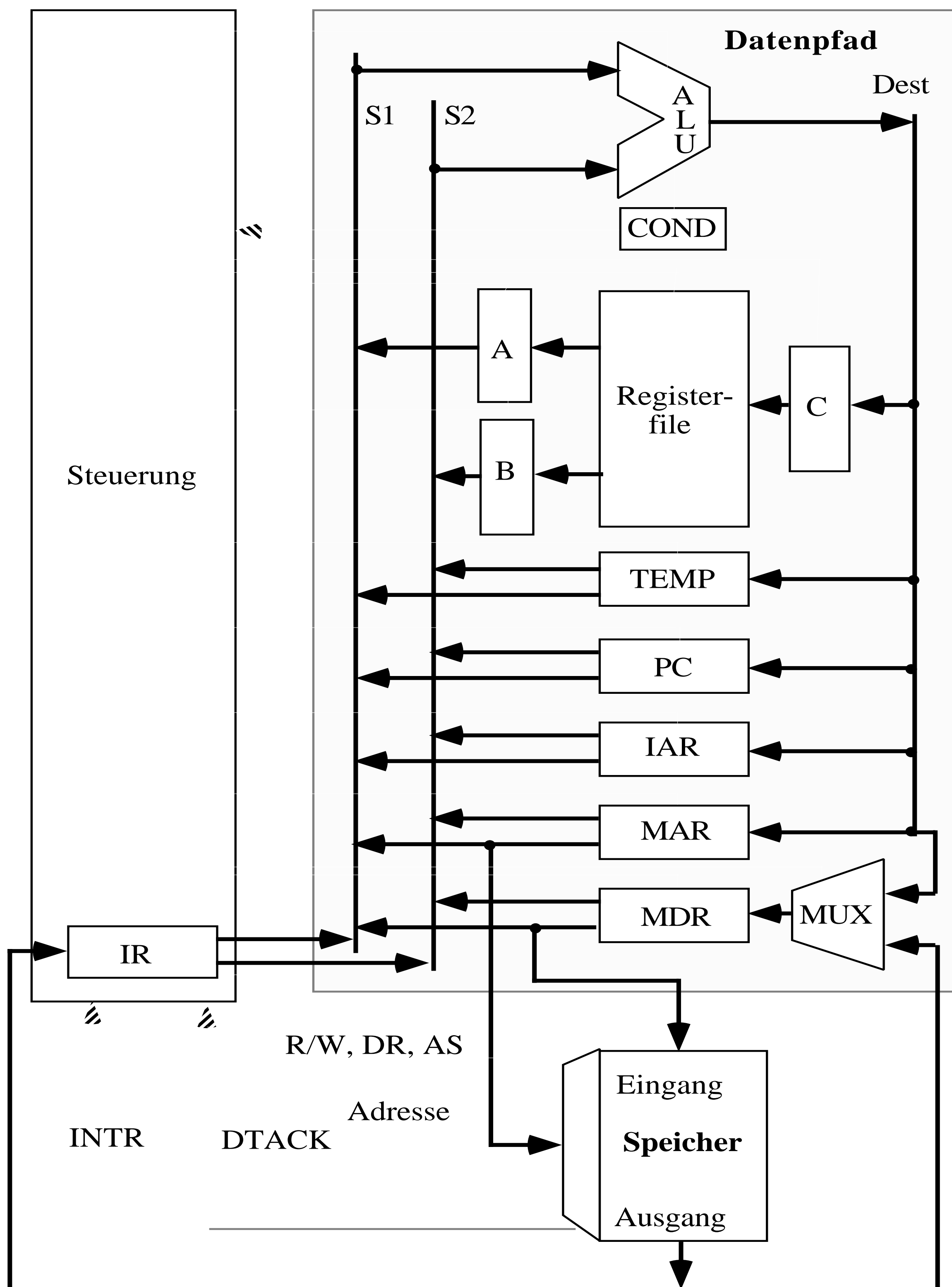
Beispiele:

ADDD F0, F4, F6	doppelt genaue Addition	F0 # # F1 ← F4 # # F5 + F6 # # F7
DIVF F0, F4, F6	einfach genaue Division	F0 ← F4 : F6
CVT/2F R7, F0	konvertiere Integerzahl in R7 in einfach genaue GK-Zahl in F0	
EQF F1, F2	prüfe auf Gleichheit von F1 und F2	GK-Statusregisterbit EQ ← (if (F1 = F2) then 1 else 0)

2.4.2. Datenpfade und innerer Aufbau der DLX

Die Organisation des datenverarbeitenden Teils der DLX zeigt das folgende Bild.
Es ist eine 3-Adress-Maschine, kenntlich an den 3 internen Datenbussen S1, S2 und Dest.

Quellen von S1 sind PC, das Interrupt-Adressregister IAR, das Speicheradressregister MAR, das Speicherdatenregister MDR und die temporären Register TEMP und A, und das Instruktionsregister IR. Senke für S1 ist die arithmetisch-logische Einheit ALU.
Quellen von S2 sind ebenfalls PC, IAR, MAR, MDR, TEMP, IR und das Register B, Senke ist die ALU.



A und B sind temporäre Register, in die der Inhalt der im Befehl angesprochenen Register rs1 und rs2 zwischengespeichert wird.

Der Bus Dest hat als Quelle den Ausgang der ALU. Senken sind die Register TEMP, PC, IAR, MAR, MDR und ein Latch C am Eingang des Registerfiles, in das Ergebnisse übernommen werden, bevor sie in das im Befehl spezifizierte Register rd übernommen werden.

Die ALU kennt folgende Operationen:

Dest	.=		
Dest	.=	S1 + S2	(2-Komplementdarstellung der Zahlen)
		S1 - S2	
		S1	
		S2	
		S1 << S2	(logisch links schieben)
		$S1 <<_a S2$	(arithmetisch links schieben)
		S1 >> S2	(logisch rechts schieben)
		S1 & S2	(logisches AND)
		S1 v S2	(logisches ODER)
		S1 ∣ S2	(logisches XOR)
		0/1	(Vergleich von A auf S1 und TEMP auf S2 auf $=, \neq, <, \leq, >, \geq$)

Vier Steuerleitungen reichen aus, um die ALU anzusteuern.

Zum Ansprechen der Register werden im Befehl 15 Bit benötigt:
rs1: 5 Bit, rs2: 5 Bit, rd: 5 Bit.

Die Durchschaltung auf die temporären Register benötigt 3 Steuerleitungen:
A := R (rs1); B := R (rs2); R (rd) := C .

Die Sammelleitung S1 hat 8 Quellen:

A, PC, TEMP, MAR, MDR, IAR, 0^{16} ## $IR_{16..31}$ (immediate-Wert);

0^{28} ## 100 (Konstante 4)

=> 3 Steuerleitungen werden gebraucht.

Das gleiche gilt für die Sammelleitung S2, auch sie hat 8 Quellen:

B, PC, TEMP, MAR, MDR, IAR, $(IR_{16})^{16}$ ## $IR_{16..31}$ (16-Bit-Ganzzahl);

$(IR_6)^6$ ## $IR_{6..31}$ (25-Bit Sprungadresse)

=> 3 Steuerleitungen zur Auswahl der Quelle.

Schließlich die Zuweisung der Sammelleitung Dest an TEMP, PC, C, MAR, IAR, MDR oder IR und die Zuweisung von M (MAR) an IR oder MDR:

Auch dafür reichen 3 Bit, da die Operationen sich gegenseitig ausschließen.

Schließlich muß festgelegt werden, woher MDR geladen wird: vom Speicher oder von Dest, d. h. ob gelesen oder geschrieben wird (R/W).

Der Datenpfad der DLX-Maschine braucht also mindestens

	15	(für das Registerfile)
+	4	(für die ALU)
+	3	(Auswahl für S1)
+	3	(Auswahl für S2)
+	3	(Auswahl für Dest und M(MAR))
+	1	(Lesen / Schreiben)
+	3	(für Übernahme in A oder B oder C)
	32	Steuerleitungen

Dazu kommen noch zwei Synchronisationsleitungen für den Speicher:

DR signalisiert das Bereitstehen von Daten von der CPU,

AS (address strobe) signalisiert die Gültigkeit einer Adresse im MAR

und vom Speicher her eine Quittungsleitung DTACK (data transfer acknowledge).

Nach außen hin gibt der Datenpfad ein Signal COND, das signalisiert, ob der Datenpfad A eine Null führt oder nicht und von außen in die Maschine kommt ein Interruptsignal INTR hinein.

2.4.3. Befehlsausführung der DLX-Maschine

Die Befehlsausführung geschieht in fünf Schritten:

1. Befehl holen (instruction fetch)
   Dauer: mindestens 2 Takte
   MAR <- PC; IR <- M (MAR)

   Dabei wird hier davon ausgegangen, daß der Speicherzugriff ohne Wartezyklen abläuft und die in MAR spezifizierte Adresse im Hauptspeicher liegt.

2. Befehlsentschlüsselung (instruction decode)
   Dauer: 1 Takt

   Dabei werden R-Typ- und i-Typ-Befehle unterschieden:
   R-Typ-Befehle: A <- RS1; B <- RS2; PC <- PC + 4
   i-Typ-Befehle: A <- RS ; B <- Rd ; PC <- PC + 4 .

3. Befehlsausführung
   Dauer: 1 Takt

   Die Aktionen richten sich nach der Art des Befehls:

   - Speicherzugriff (memory reference ; i-Typ)
     MAR <- A + $(IR_{16})^{16}$ ## $IR_{16..31}$; MDR <- B
     (Vorbereitung des Speicherzugriffs)
   - ALU-Befehle (Zwischenspeichern des 2. Operanden in TEMP)
     R-Typ-Befehl: TEMP <- B
     i-Typ-Befehl: TEMP <- $(IR_{16})^{16}$ ## $IR_{16..31}$
   - unbedingte Sprünge (branch / jump ; i-Typ-Befehl)
     TEMP <- PC + $(IR_6)^6$ ## $IR_{6..31}$; COND <- true

     Das relative Sprungziel aus dem Befehl (24 Bit) wird zum Inhalt von PC addiert und zwischengespeichert.
   - bedingte Sprünge (i-Typ-Befehl)
     TEMP <- PC + $(IR_{16})^{16}$ ## $IR_{16..31}$; COND <- (A = 0)

4. Befehlskomplettierung
   Dauer: mindestens 1 Takt

- ALU-Befehle: C <- A <u>op</u> TEMP
- Speicherzugriff: lesen: MDR <- M (MAR)
  schreiben: M (MAR) <- MDR
- bedingte Sprünge:
  if (COND) then PC <- TEMP

5. Rückschreibeschritt
   Dauer: 1 Takt

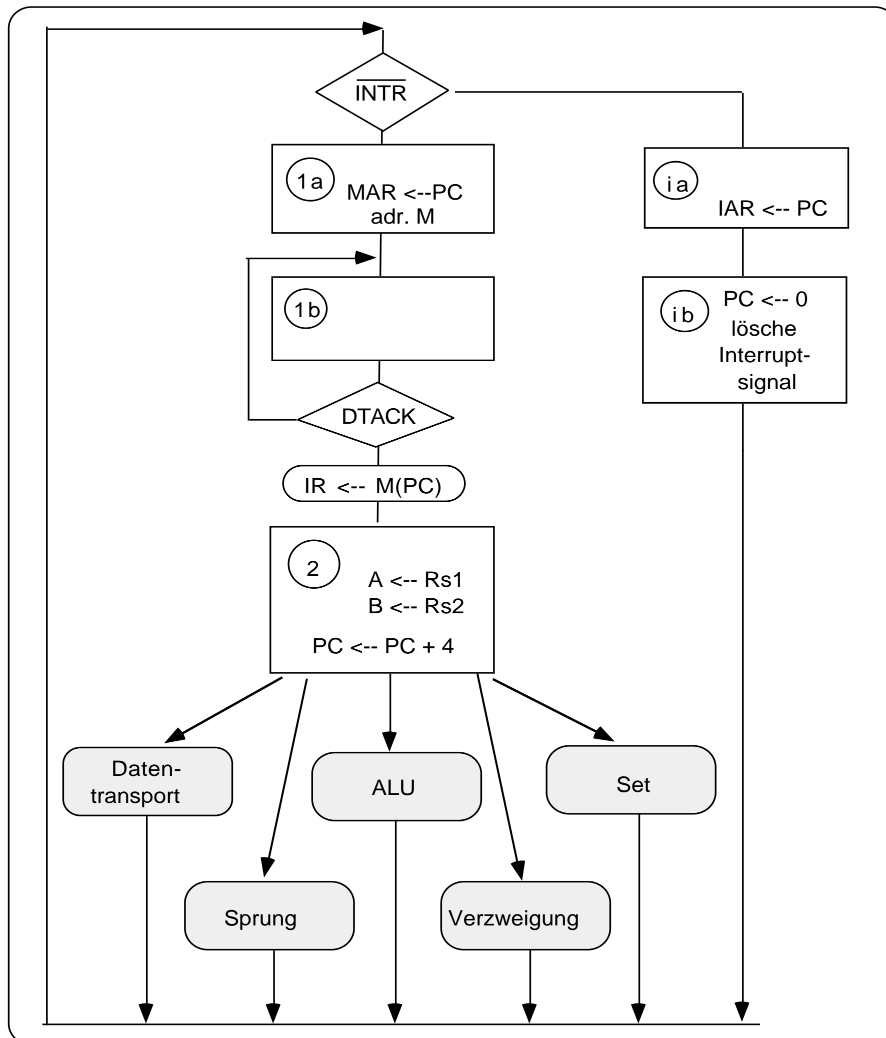
   ALU-Befehl: Rd <- C
   lesen: Rd <- MDR

2.4.4. Zustandsdiagramme der Befehlsabarbeitung

Die genauere Abarbeitung der Befehle soll nun an Hand von ASM-Karten gezeigt werden.

2.4.4.1. Abarbeitung auf der obersten Ebene

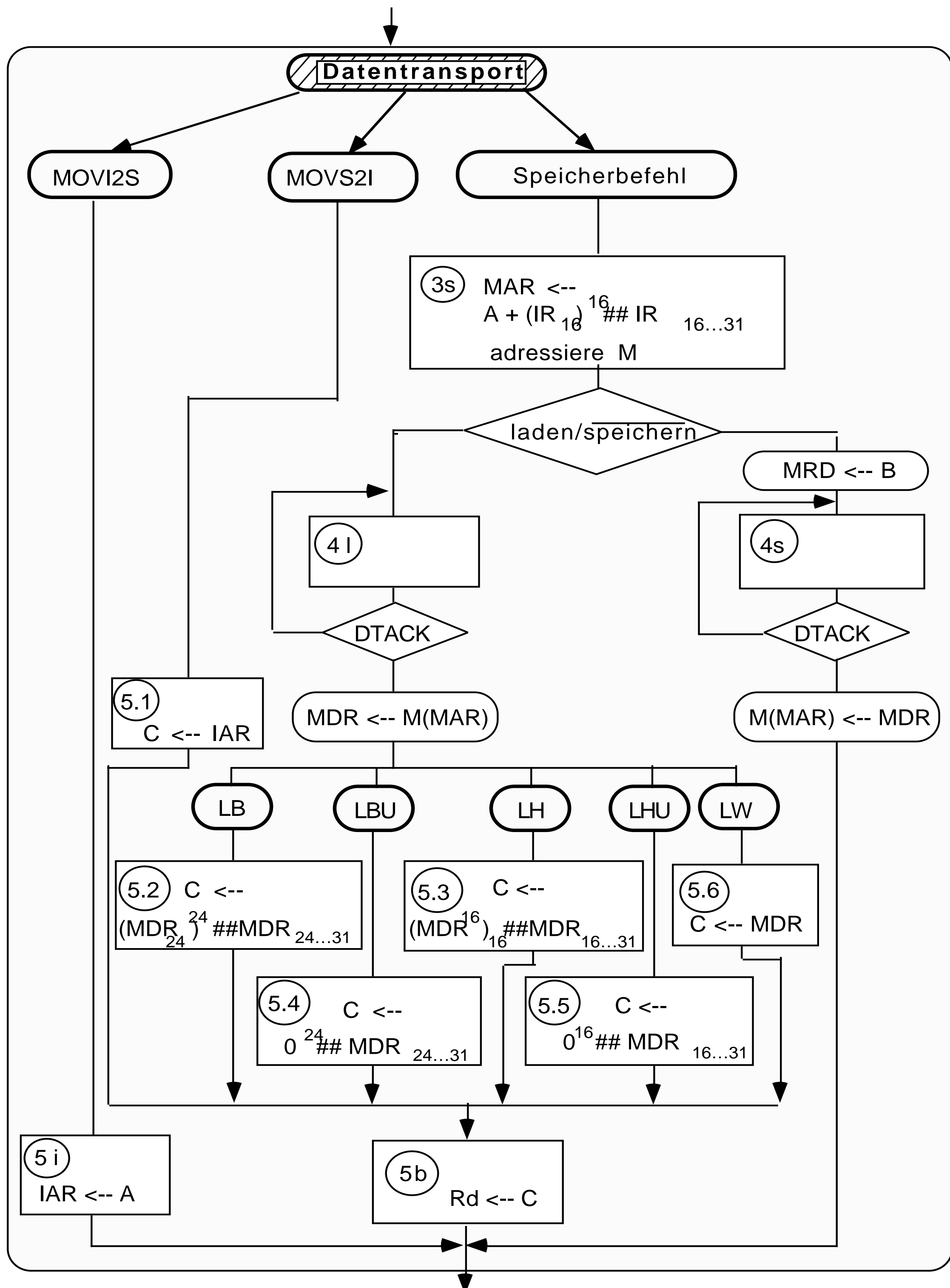
Befehl holen und entschlüsseln, 5 Zustände und 5 Untergruppen:



Die GK-Operationen sind dabei nicht gezeigt. Sie erfordern eine eigene Behandlung, da sie mehrere Schritte brauchen oder ggf. eigene Hardware.

Je nach Ergebnis der Entschlüsselung zweigt sich die Behandlung der Befehle auf in

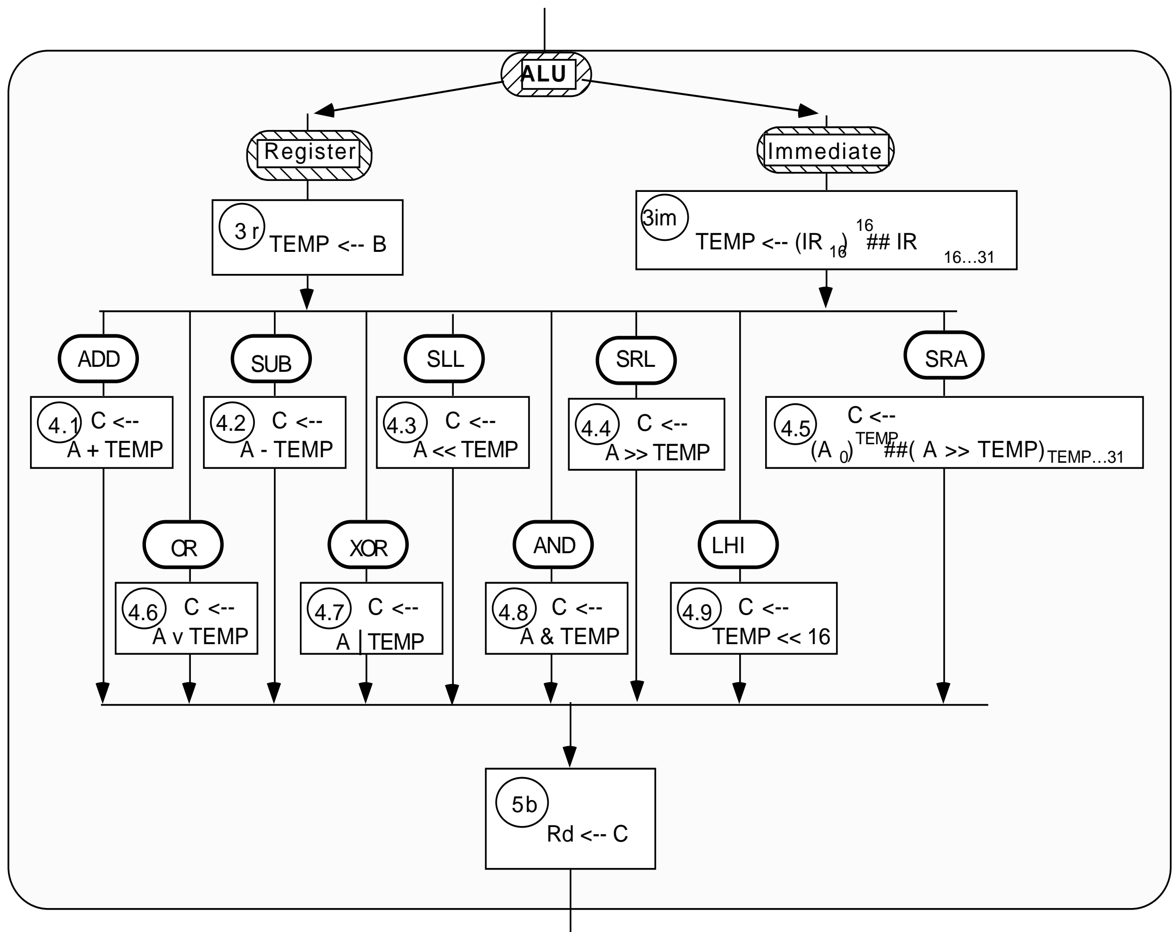
2.4.4.2. Datentransportbefehle



Die dick umrandeten ovalen Kästen bezeichnen die einzelnen Daten transportierenden Befehle; insgesamt hat das Diagramm 11 Zustände.

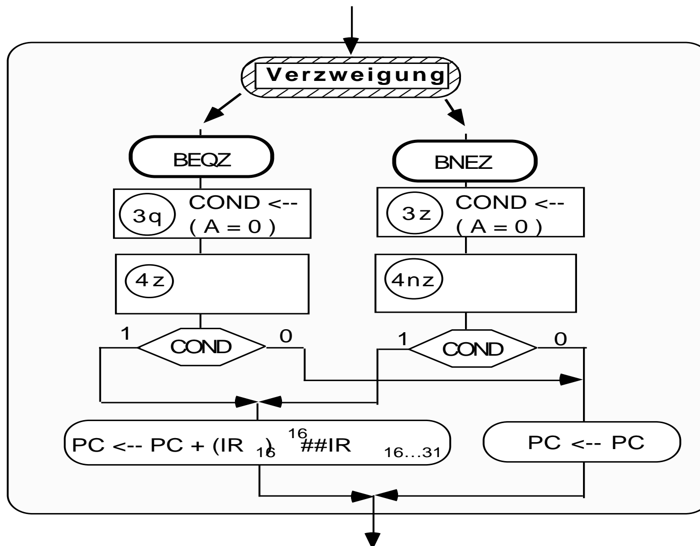
2.4.4.3. Datenbearbeitende Befehle

Insgesamt läßt sich das Diagramm mit 12 Zuständen darstellen



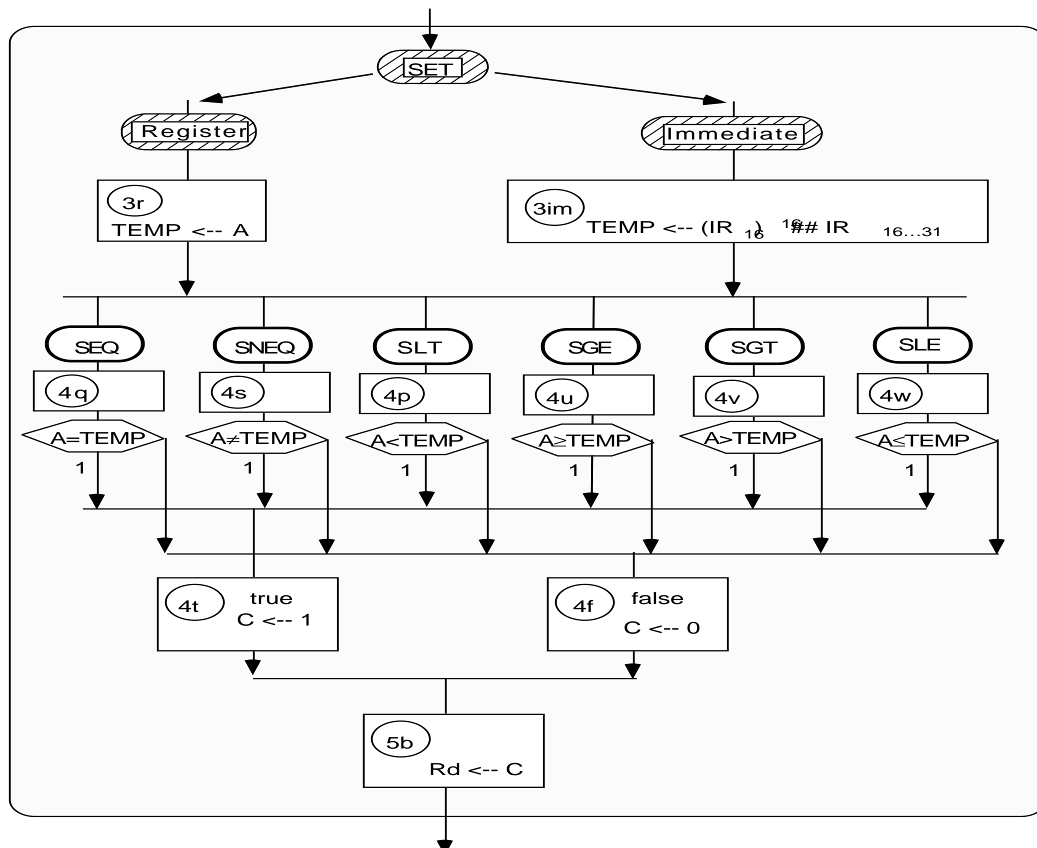
2.4.4.4. Verzweigungsbefehle

bedingte Sprünge, gesteuert durch COND, in dem der Vergleich festgehalten wird
4 Zustände.



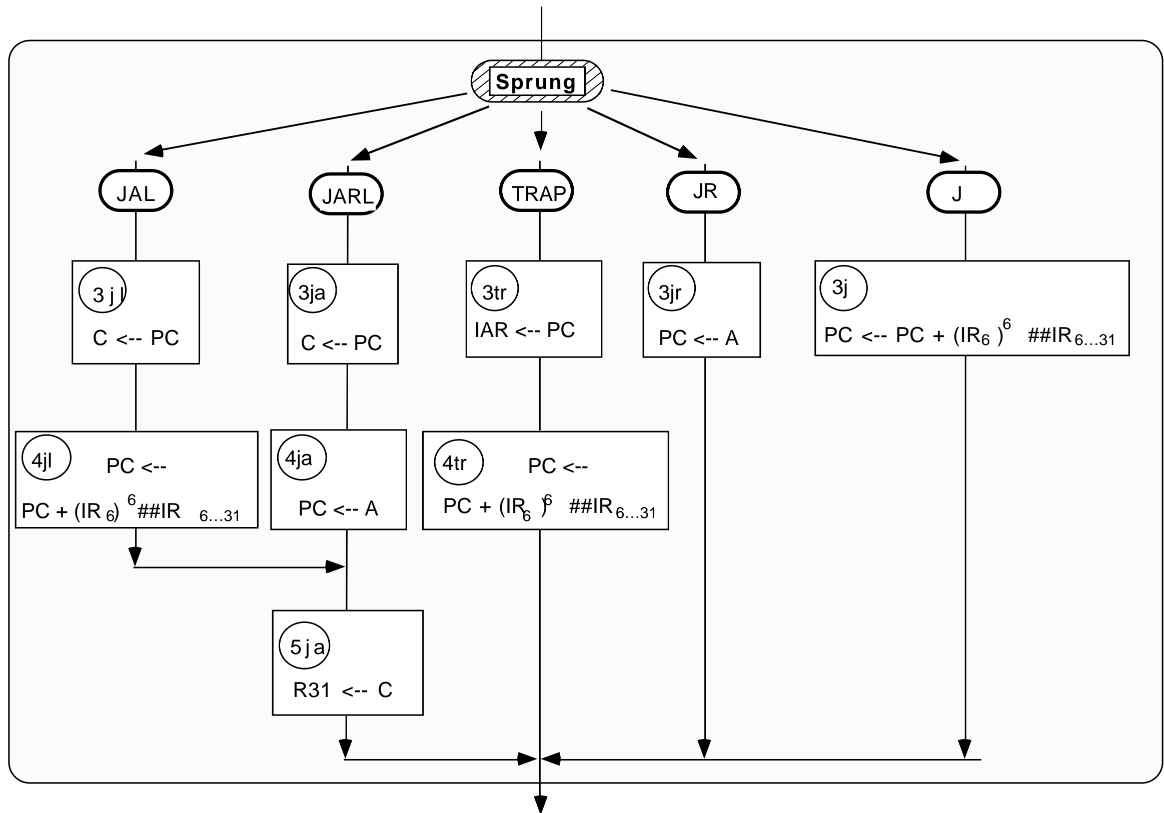
2.4.4.5. Vergleichsbefehle

bei denen am Ende der logische Wert des Vergleichs zwischen dem Wert in A und dem Register TEMP in das im Befehl spezifizierte Zielregister geschrieben wird, beschrieben in 11 Zuständen.



2.4.4.6. <u>Sprungbefehle</u>

unbedingte Sprünge, 9 Zustände



Zählt man die Anzahl der verschiedenen Zustände durch, dann findet man

Anfang	(1a, 1b, ic, ib, 2)	:	5
ALU	(3r, 3im, ADD, SUB, SLL, SRL, SRA, OR, AND, XOR, LHI, 5b)	:	12
Verzweigung	(3q,3z, 4z,4nz)	:	4
SET	(4q, 4s, 4p, 4u, 4v, 4w) (3r, 3im und 5b sind mit ALU gemeinsam)	:	6
Sprung	(3je, 3ja, 3tr, 3jr, 3j, 4jl, 4jA, 4tr, 5ja)	:	9
Datentransport	(3s, 4e, 4s, 5.1, 5.2, 5.3, 5.4, 5.5, 5i) (5b ist mit ALU gemeinsam)	:	9
	Summe der verschiedenen Zustände		45

Man kann die Zustände auf 5 begrenzen, dann sind die in den bisherigen Zuständen auszuführenden Aktionen abhängig vom jeweiligen Befehl; die ASM-Karten sind dann umzuschreiben als Beschreibungen eines Mealy-Automaten.

2.4.5. Codierung des OP-Code-Teils des Befehls

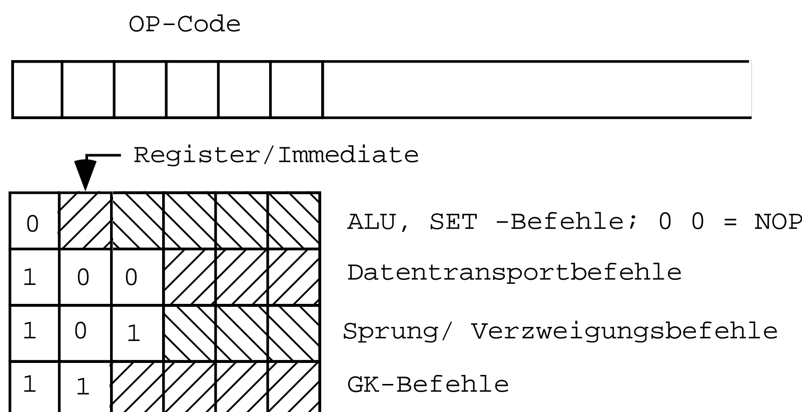
Mit Kenntnis der einzelnen Zustände bei der Befehlsausführung kann man nun versuchen, die für den OP-Code im Befehl vorgesehenen 6 Bits den Befehlen zuzuordnen.

Die wesentliche Unterscheidung zwischen den Befehlen auf Grund des OP-Codes muß gemacht werden beim Übergang von Zustand 2 -> 3 .

Die Aufteilung in Befehlsgruppen legt die entsprechende Aufteilung der Bits des OP-Code nahe:

Datentransport:	MOVI2S, MOVS2I, Speichern LB, LBU, LH, LHU, LW	:	8 Befehle
	Diese Gruppe braucht 3 Bit.		
ALU-Befehle:	Reg./Imm. -> ADD, SUB, SLL, SRL, SRA, OR, AND, XOR, LHI	:	9 Befehle
SET:	Reg./Imm. -> SEQ, SNEQ, SLT, SGE, SGT, SLE	:	6 Befehle
	Die Gruppen ALU und SET lassen sich zusammen mit 5 Bit codieren: 1 Bit für die Unterscheidung Register/Immediate-Befehl und 4 Bit für die 15 Befehle.		
Sprung:	JAL, JALR, J, JR, TRAP	:	5 Befehle
Verzweigung:	BEQZ, BNEZ	:	2 Befehle
	Diese beiden Gruppen brauchen zusammen 3 Bit.		

Damit könnte die Zuordnung so aussehen:



In der Gruppe ALU/SET sollte das erste Bit Null sein, dann ist 000 000 der Befehl NOP (tue nichts).

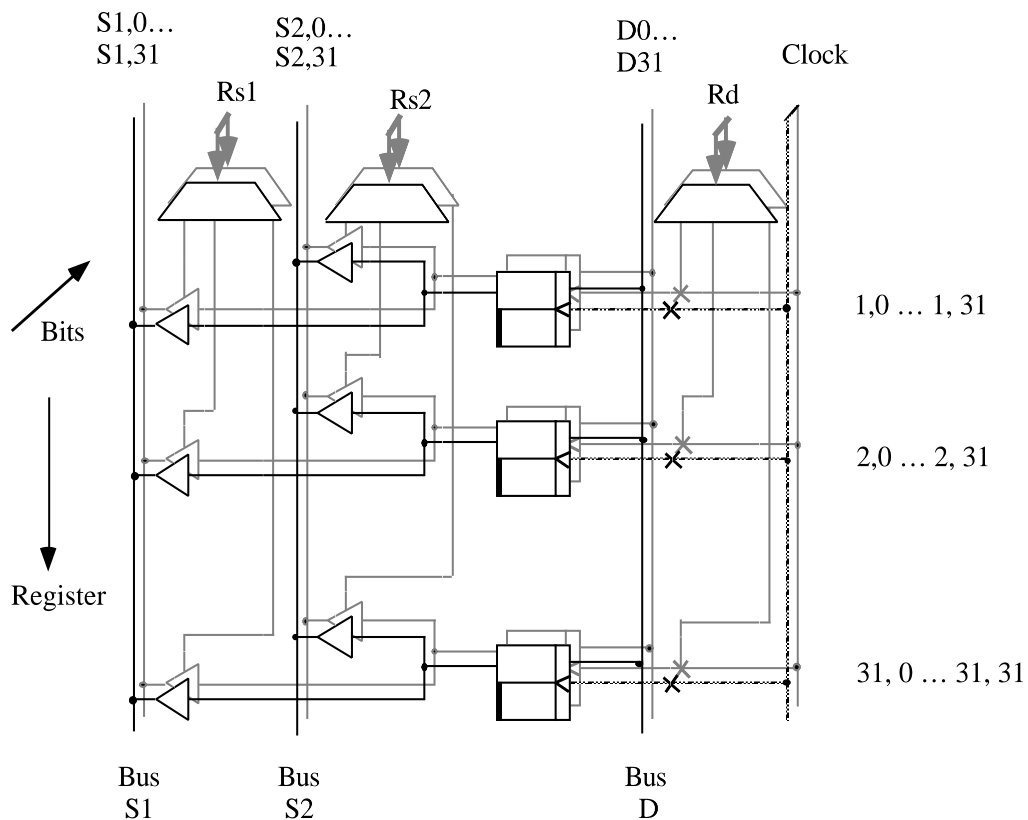
Die Codierung für die anderen Gruppen ist mehr oder weniger willkürlich.

Die 16 GK-Befehle erhalten eine eigene Gruppe.

2.4.6. Beschleunigung der DLX-Maschine

Die Maschine kann beschleunigt werden, indem das Registerfile nicht als dual-port-memory ausgebildet wird sondern als ein Registersatz, dessen Inhalt auf zwei Busse geschaltet werden kann: auf S1 und S2, und der von einem dritten Bus Dest her geladen werden kann.

Dann entfallen die Latches A, B und C und damit jeweils ein Takt in der Abarbeitung.



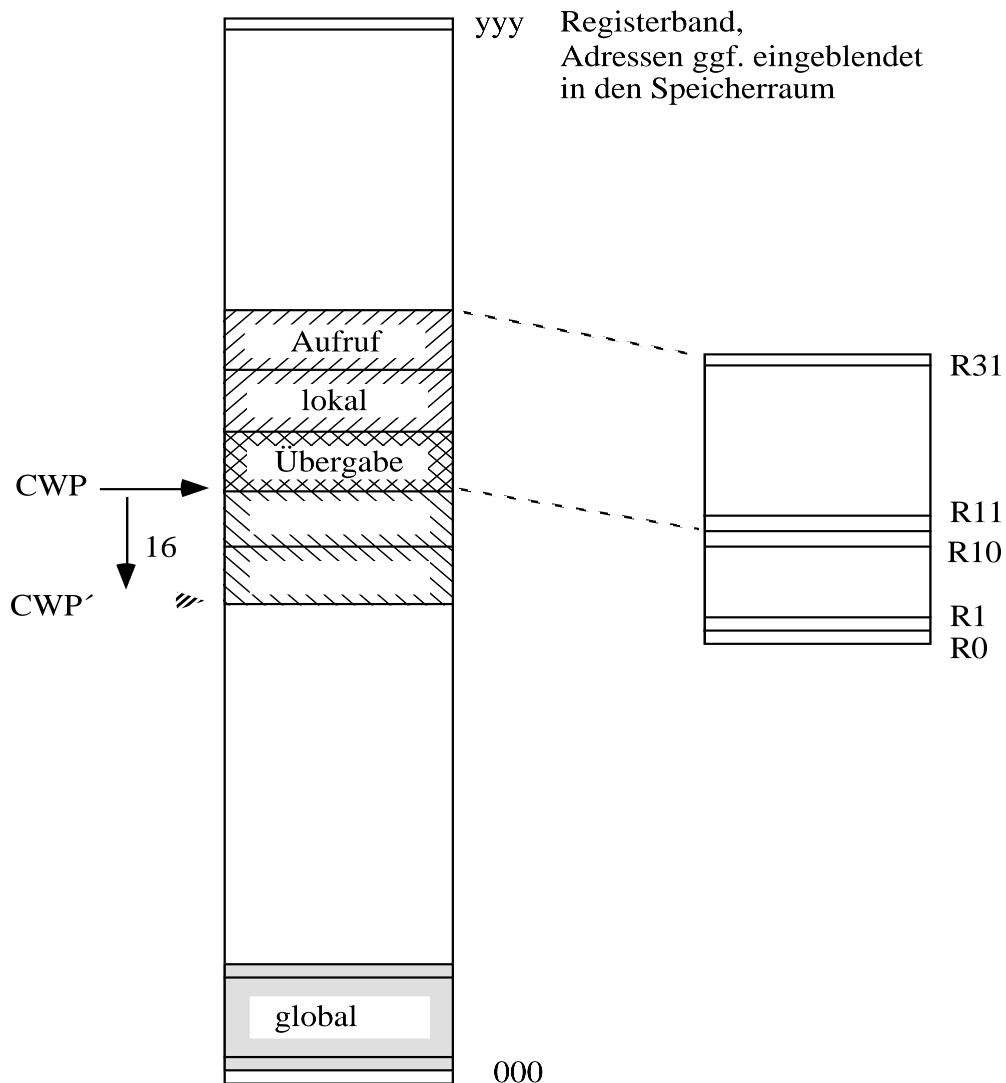
Ferner kann der PC mit einem Addierwerk versehen werden (praktisch als ladbarer Zähler ausgebildet).

Das Registerfile ist ein schneller Halbleiterspeicher von $(32 + 32) * 4 = 256$ Byte mit 15 Adresseingängen, 62 Datenausgängen und 32 Dateneingängen und zwei Synchronisationsleitungen für die Ausgänge und einer für den Eingang.

Der schnelle Speicher kann größer ausgelegt sein und dann kann, unsichtbar für den Benutzer, ein Sprung über die Rückkehradresse R31 ein Umschalten eines Registerfensters bedeuten:

Das Register ist als Band angeordnet mit z. B. an fester Stelle R_1 ... R_{10} für globale Daten und R_{11} - R_{30} durch einen Pointer (CWP - current window pointer) als momentan gültiger Registersatz gekennzeichnet.

Schaltet man beim Unterprogrammsprung den CWP z. B. um 15 weiter, so erhält man einen Überlappungsbereich, in dem Register der aufgerufenen Prozedur Werte der aufrufenden enthalten. Damit ist die Parameterübergabe sehr einfach.



Bei Sprachen vom Typ "Pascal" bestehen Schwierigkeiten, da hier Variable der aufrufenden Prozedur in der aufgerufenen immer noch sichtbar sind. Eine Möglichkeit, diese Schwierigkeit zu umgehen, ist die Einblendung des Registerfiles in den Adressraum des Rechners.