INSTITUT FÜR INFORMATIK
UND PRAKTISCHE MATHEMATIK

Werkzeuge des MIMOLA-Hardware-Entwurfssystems

Klaus Kelle, Gerd Krüger, Peter Marwedel,
Lothar Nowak, Lutz Terasa, Franz Wosnitza

CHRISTIAN-ALBRECHTS-UNIVERSITÄT
KIEL

Werkzeuge des
MIMOLA-Hardware-Entwurfssystems

Klaus Kelle+, Gerd Krüger++, Peter Marwedel++,
Lothar Nowak++, Lutz Terasa++, Franz Wosnitza+

Bericht Nr. 8707
Juni 1987

Institut für Theoretische Elektrotechnik
der RWTH Aachen
Kopernikusstr. 16, 5100 Aachen

Institut für Informatik und Praktische
Mathematik der CAU Kiel
Olshausenstr. 40-60, 2300 Kiel

Abstract

This report describes the tools of a CAD system supporting the design of digital hardware. The CAD system is based upon the MIMOLA hardware description language. PASCAL-like algorithms (or "behaviour") as well as hardware structures (netlists) can be described in MIMOLA. Main emphasis is an the register transfer level of abstraction, but application levels and the gate level are also covered. The system currently contains six main tools, a common tool environment and some auxiliary programs. The six main tools perform data/control path synthesis, automatic generation of self-test programs, retargetable microcode generation, schematics generation, simulation and testability analysis.

Keywords

CAD for VLSI, hardware description languages, register transfer languages, hierarchical design, data path synthesis, self-test, automatic test generation, diagnostics, microcode generation, retargetability, schematics generation, register transfer simulation, functional testability, testability analysis,

Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie unter dem Förderkennzeichen NT 2816 A9 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Inhaltsverzeichnis

Wissenschaftlich-technische Ergebnisse

1 Hardware-Entwurf mit der Sprache MIMOLA

1.1 Einordnung

Durch die steigende Komplexität und die zunehmende Miniaturisierung sind der Entwurf, die Herstellung und der Test digitaler Systeme heute nur noch unter Einsatz von CAD-Systemen denkbar. Klassische CAD-Systeme unterstützen dabei vor allem die unteren Abstraktionsebenen wie die Gatter-Ebene, die Schalter-Ebene oder

Name der Ebene	Komponenten
algorithmische Ebene-n	Speicherzellen (Variable)
	Zuweisungen, Schleifen
algorithmische Ebene-1	dto.
RT-Strukturebene	ALUs, Busse, RAMs, Register
Logik-Ebene	Boolsche Ausdrücke
Gatter-Ebene	NAND- und NOR-Gatter
Schalter-Ebene	Schalter, Abschwächer,
	Verbindungen [Hay821]
symbolische Layoutebene	"Sticks"
Layout-Ebene	Rechtecke

Abb.1.1: Mögliche Abstraktionsebenen digitaler Systeme

In der Abbildung 1.1 wird zwischen n verschiedenen algorithmischen Ebenen unterschieden, um die übliche n-stufige Interpretation von Programmen wiederzugeben. Im Falle von Maschinenprogrammen, welche durch Mikroprogramme interpretiert werden', ist n gleich z. Stellen die Maschinenprogramme einen Interpreter einer höheren Programmiersprache (z.B. BASIC, PROLOG) dar, so ist n gleich 3. Für Maschinenprogramme, welche nicht durch Mikroprogramme interpretiert werden (z.B. RISCProgramme), ist n gleich 1.
Die unterschiedlichen Ebenen stellen stets nur **Interpretationsebenen** dar. **Compilierungen,** die Hochsprachen-Programme etwa auf RISC-Programme abbilden, sind durch die Abb. 1.1 nicht erfaßt. Beschreibungen auf der untersten algorithmischen Ebene werden auch Register-Transfer(RT-) Verhaltensbeschreibungen genannt.

Um der zunehmenden Komplexität Rechnung zu tragen, wurde mit dem MIMOLA-Hardware-Entwurfssystem MSS ein CAD-System entwickelt, welches schwerpunktmäßig die höheren Abstraktionsebenen unterstützt. Abgedeckt wird der Bereich von algorithmischen Beschreibungen bis zur RT-Strukturebene, von einzelnen Werkzeugen auch noch die Gatter-Ebene. Tab. 1.1 gibt eine Übersicht über die Werkzeuge des MSS:

- die Simulation von Hardwarestrukturen
- die automatische Erzeugung von Selbsttestprogrammen für speicherprogrammierbare Strukturen
- die Übersetzung von PASCAL-artigen Programmen in den Maschinencode (Zielmaschinen-unabhängiger Compiler)
- die Erzeugung von Hardwarestrukturen zu einem vorgegebenen Verhalten, beispielsweise zu einem vorgegebenen Befehlssatz (Synthese von RT-Strukturen)
- die Simulation von Verhaltensbeschreibungen
- die Umsetzung textueller Hardwarebeschreibungen in Blockdiagramme

Tab. 1.1 Werkzeuge im MSS

1.2 Die Sprache MIMOLA

1.2.1 Überblick

Die Sprache MIMOLA (machine independent microprogramming language) dient der Dokumentation des jeweils aktuellen Stadiums des Entwurfs. MSS und MIMOLA sind einander in hohem Maße angepaßt. Das bedeutet allerdings nicht, daß die Algorithmen des MSS eine Eingabe in MIMOLA voraussetzen. Vielmehr ließen sich in anderen Sprachen beschriebene Teilaspekte des Entwurfs in die intern benutzte Zwischensprache TREEMOLA übersetzen. Die folgende Darstellung der Sprache MIMOLA beschreibt also zugleich, welche Art von Systemmodellierung Voraussetzung für eine solche Übersetzung ist.

Anders als die meisten bekannten Sprachen unterscheidet MIMOLA grundsätzlich zwischen der Darstellung des **Verhaltens** eines Moduls und seiner **Struktur.** Aus der Verhaltensbeschreibung ist zu ersehen, zu welchen Ausgaben und zu welchen Änderungen an den internen Zuständen bestimmte Eingaben führen. Die Strukturbeschreibung spezifiziert, aus welchen einfacheren Moduln ein komplexes Modul zusammengesetzt ist. Struktur und Verhalten stellen zwei **alternative Sichten** eines Moduls dar. Mittels Synthese ist es möglich, aus einer Verhaltensbechreibung eine Strukturbeschreibung zu generieren. Verhalten und Struktur können beide simuliert werden. In der Simulation der Struktur muß dazu auf das Verhalten der lokalen Module zurückgegriffen werden. Die Korrektheit der Implementierung eines Verhaltens durch eine Struktur kann für Fallbeispiele durch Vergleich der Simulationsergebnisse oder allgemein durch Verifikation erfolgen.

Im Interesse einer hohen Akzeptanz der Sprache lehnt sich MIMOLA sowohl in der Verhaltensbeschreibung als auch in der Strukturbeschreibung soweit wie möglich an etablierte Sprachen an.

Dementsprechend steht für Verhaltensbeschreibungen (bis auf unten angeführte Ausnahmen) in MIMOLA eine Obermenge von PASCAL zur Verfügung. Dabei entspricht nicht nur die Syntax, sondern auch die Semantik derjenigen von PASCAL. So sind z.B. rekursive Prozeduraufrufe, dyna

mische Variable und Arrays mit beliebig vielen Dimensionen erlaubt. Zusätzlich zu PASCAL enthält MIMOLA Erweiterungen für systemnahes Programmieren.

Im Bereich der Strukturbeschreibung lehnt sich MIMOLA eng an ebenfalls auf PASCAL basierende Sprachen wie CAP [Ram80], KARL [Har84] oder. VHDL [VHDL] an. Strukturbeschreibungen bestehen aus einer Liste lokaler Moduln sowie einer Netzliste. Letztere beschreibt, wie die lokalen Module untereinander verbunden sind.

Struktur- und Verhaltensbeschreibungen können in MIMOLA baumartig geschachtelt werden.

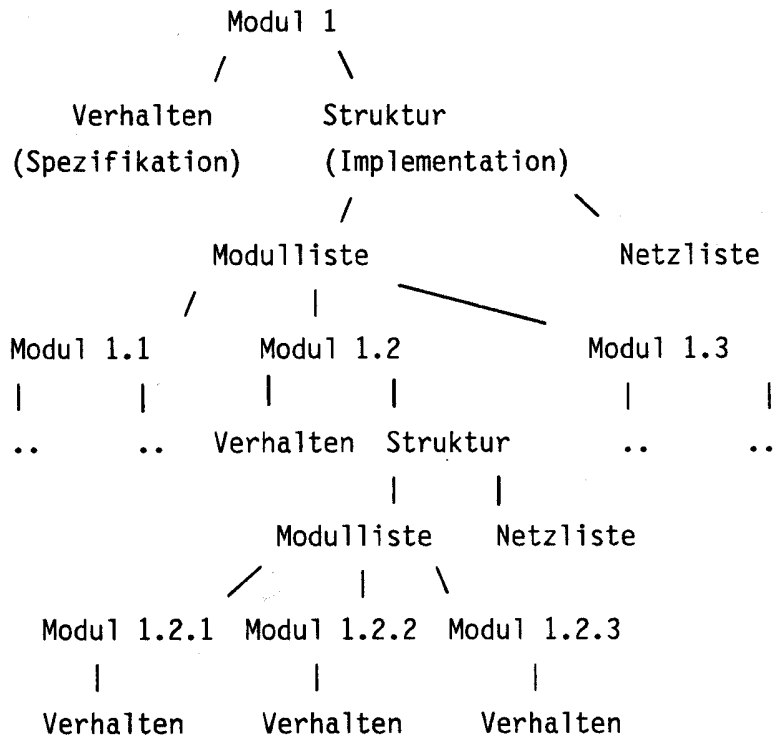


Abb. 1.2: Modul-Hierarchie

1.2.2 Verhaltensbeschreibung

Wie bereits erwähnt, erlaubt MIMOLA eine Beschreibung der Zusammenhänge zwischen den Eingaben, den Ausgaben und den internen Zuständen eines Moduls durch eine Obermenge von PASCAL. D.h., das Verhalten wird im Interesse der Akzeptanz der Sprache **imperativ** beschrieben. In M,IMOLA nicht enthalten sind lediglich die REAL-, SET- und Enumeration-Datentypen. Aufgrund der Hardwarenähe von MIMOLA sollen diese vom Benutzer selbst auf Bitstrings abgebildet werden. Als Erweiterung gegenüber PASCAL enthält MIMOLA u.a. folgende Fähigkeiten:

- Verarbeitung von Bitstrings
- wählbare Bindung von Variablen an Speicher
(z.B. statische (FORTRAN-) oder dynamische (PASCAL-) Bindung)
- Konvertierung zwischen allen Typen
- explizite Adreßrechnungen
- reichhaltiges Repertoire an Operatoren
- explizite Angabe von Nebenläufigkeiten und Parallelitäten

Beispiele (SR und SM seien vordefinierte Speicher)

```
TYPE word = (15:0);          (*16 Bit langer Bitstring
VAR pc : word AT SR [7];          (*Bindung von pc an SR [7]
VAR sp : word AT SR [6];          (*Bindung von sp an SR [6)
VAR a : word AT SM [sp+1];           (*dynamische Variable a
a := ref&(pc) + 2;              (*ref&: Adresse einer Variablen*)
a := pc "SHIFTLL" 5;            (*Schiebe links logisch  *)
PARBEGIN pc:=15; a:=7 PAREND    (*Parallelausführung     *)
```

Sofern ein Modul potentiell Teil eines anderen Moduls werden soll, ist es sinnvoll, die Verhaltensbeschreibung derartig einzuschränken, daß das exportierte Verhalten auf einer höheren Abstraktionsebene erkannt werden kann. Es zeigt sich, daß hierzu eine Modellierung analog zu abstrakten Datentypen [Lis77] geeignet ist.

```
MODULE B74S381(IN a,b:(15:0); FCT s:(2:0); OUT f : (15:0));
 BEHAVIOUR AtRTLevel IS
 BEGIN
 f<- CASE s OF
  0 . 0;
  1 . b - a;
  2 . a - b;
  3 . a + b;
  4 : a "XOR" b;
  5 : a "OR" b;
  6 : a "AND" b;
  7 : #FFFF            (* # = hexadezimal*)
 END
 END;
```

Wird das Modul B74S381 in einem komplexen System als Teil verwendet, so kann es diesen Operationen wie "+", "-" und "AND" zur Verfügung stellen. Im Unterschied zur Spezifikation exportierbarer Operationen in der Software muß in der Hardware auch ein Auswahlcode für die Operation spezifiziert werden. Dies geschieht in der syntaktischen Form eines CASE-Statements. Lediglich im Falle einer einzigen exportierbaren Operation darf der CASE-Rahmen entfallen.

Beispiel:

```
MODULE B74S00(IN a,b:(3:0); OUT f : (3:0));
 BEHAVIOUR AtGateLevel IS BEGIN f<- a "NAND" b;
 END;
```

Komplexe Ausdrücke und Sequenzen von Statements zur Spezifikation des exportierbaren Verhaltens sind nicht erlaubt.

Sofern die exportierten Operatoren nicht zur Liste der MIMOLA-Standardoperatoren gehören, kann deren Semantik zusätzlich prozedural beschrieben werden. Das Schlüsselwort **BEHAVIOUR** korrespondiert dann zum PASCALSchlüsselwort **PROGRAM.**

Beispiel:

```
MODULE Stack (INOUT data:(15:0); (*16 i/o lines*)
                  FCT ctl : (1:0) ; CLK clock : (0) );


   BEHAVIOUR AtRTLevel IS

    VAR Memory : ARRAY [0..1023] OF (15:0);
     Pointer: RANGE 0..1023 ;

    PROCEDURE push         (IN i : word);
     BEGIN
       PARBEGIN
        Memory[Pointer]:=i;
        Pointer := "INCR" Pointer;
       PAREND
     END;

    PROCEDURE pop          (OUT o : word);
     BEGIN
        PARBEGIN
        Pointer:= "DECR" Pointer; (*Zuweisung an Speicher*)
        o <- Memory[Pointer];            (*Zuweisung an Signal *)
       PAREND
     END;

    PROCEDURE read         (OUT o : word);
     BEGIN
      o <- Memory[Pointer]; END;

    PROCEDURE clear ; BEGIN
```

```
BEGIN_behaviour
AT clock DO
  CASE ctl OF 0 :
   push(data); 1 :
   pop (data); 2 :
   read(data); 3 :
   clear;
  END_case;
END_behaviour;
```

Eine solche Beschreibung kann auf zweierlei Arten genutzt werden: Der CASE-Teil ist global bekannt und dient in der Rechnersynthese und in der Codeerzeugung dazu, für push- und pop-Operationen geeignete Module und die benötigten Steuercodes zu erkennen. Die gesamte Verhaltensbeschreibung ist **lokal** bekannt und kann zur Synthese der lokalen Struktur des Moduls Stack genutzt werden. Bei diesem hierarchischen Vorgehen ist sowohl ein top-down-, als auch ein bottom-up-Design möglich, jenachdem, ob zunächst die Struktur des umgebenden Rechners oder zunächst die lokale Struktur des Stacks synthetisiert werden.

1.2.3 Strukturbeschreibung

Hardwarestrukturen werden in MIMOLA beschrieben als Menge von Moduln und deren Verbindungen untereinander.

Beispiel:

```
MODULE Stack (INOUT data:(15:0);
               FCT ctl : (1:0) ; CLK clock : (0) );
 STRUCTURE NotYetCompleted IS
  TYPE word = (15:0); addr = (
  9:0);
   PARTS     (*Beginn der Modulliste*)

    Memory : MODULE SRam «SIZE=1024»
    (INOUT d:word;                 (*Daten E/A       *)
    ADR a:addr;                    (*Adressen        *)
    FCT s:(0) ;                    (*Kontrolle       *)
               CLK   c:(0) );      (*Takt            *)
              BEHAVIOUR OfSRam IS
     BEGIN
     AT c DO
     CASE s OF
     0 : SRam[a]:=d;                    (*Schreiben*)
     1 : d <- SRam[a];                  (*Lesen         *)
     END;
     END;
    Pointer: MODULE Reg «SIZE=1»
                          (IN i : addr; OUT o: addr;
                           FCT        s:(0); CLK c:(0) );
```

```
            BEHAVIOUR OfReg IS
            BEGIN
            AT c DO
             CASE s OF
                 0 . Reg .= i;
                 1 : o <- Reg;
               END;
             END;

  CONNECTIONS         (*unvollständige Netzliste*)
 clock    -> Memory .c;
 clock    -> Pointer.c;
 Pointer                                          -> Memory .a;
 data     -> Memory .d;
END structure;
BEHAVIOUR

  PROCEDURE  push  ...
  PROCEDURE   pop   ...
  PROCEDURE  read  ...
  PROCEDURE clean ..

  BEGIN  behaviour  AT
   clock DO
     CASE ctl OF
      0  :  push(data);
      1  :  pop  (data);
      2  :  read(data);
      3 : clear;
     END-case;
 END behaviour;
```

Man beachte, daß bei den Blattmodulen Memory und Pointer lediglich noch das Verhalten angegeben wird.

1.2.4 Zusatzinformation

Verhalten und Struktur stellen zwei, u.a. von EDIF [EDIF] her bekannte Sichten ("views") eines Moduls dar. Spezifisch für die höheren Entwurfsebenen ist es, daß zu einem Modul weitere Informationen benötigt werden, welche sich weder diesen beiden noch der Layout-Sicht zuordnen lassen. Speziell erwähnt seien hier die Speicherreservierungen und die Programmtransformationen. Diese zusätzlichen Informationen stellen weit weniger als die klassichen Sichten ein umfassendes Bild eines Moduls dar. Im folgenden werden alle zu einem Modul zugehörigen Informationen (einschließlich der Verhaltens-, der Struktur- und der Layout-Sicht) als **Aspekte** eines Moduls bezeichnet.

Der Aspekt "Speicherreservierungen" zeichnet Speicherzellen zur Verwendung für bestimmte Zwecke aus. Für das MSS wichtig ist die Bekanntgabe der Zellen für den Programmzähler und den Befehlscode:

Beispiele:

```
LOCATIONS_For_Instructions SI [0..1023];
LOCATIONS_For_ProgramCounter PC ;
```

Der Aspekt "Programmtransformationen" ,dient der expliziten Darstellung semantischen Wissens.

Beispiel:

```
REPLACE          (*Ersetze                        *)
 &a.(15:0) < 0   (*Test auf < 0                   *)
WITH             (*durch                         *)
 &a.(15)         (*Test des Vorzeichens                        *)
END;
```

Diese Regeln drücken jeweils eine Implikationsrichtung aus. Für Aquivalenzen werden zwei Regeln benötigt.

1.3 Das MIMOLA-Entwurfssystem

Das MIMOLA-Hardware-Entwurfssystem enthält eine ganze Reihe von Werkzeugen zur Unterstützung des Entwurfs auf den höheren Ebenen. Genannt seien hier nur der Testprogramm-Generator, die RT-Synthese und der maschinenunabhängige Compiler. Alle Werkzeuge sind in einer gemeinsamen Umgebung eingebettet. Diese enthält einen System-Monitor, eine gemeinsame Datenbasis, einen Übersetzer von MIMOLA in die interne Zwischensprache TREEMOLA, sowie einen Übersetzer von TREEMOLA nach MIMOLA (vgl. Abb. 1.3).

Vor der eigentlichen Anwendung der Werkzeuge müssen ggf. die Struktursichten der Module expandiert werden [Mar87]. Diese Aufgabe wird ebenfalls von der Werkzeug-Umgebung übernommen. Auf diese Weise **ist ein hierarchischer EntwurfsprozeB möglich, obwohl die einzelnen Werkzeuge selbst keine Hierarchie kennen.** Aus Gründen der Portabilität wird die Datenbasis z. Zt. ähnlich wie im VENUS-System [Hör86] durch Files realisiert.

Die Werkzeuge werden im einzelnen in den folgenden Kapiteln vorgestellt.

Bis auf wenige Anweisungen zur Ablaufsteuerung im System-Monitor und die Darstellung von Blockdiagrammen ist das MSS vollständig in einer Untermenge von Standard-PASCAL implementiert. Dadurch ist das System hochgradig portabel. Infolgedessen konnten entsprechend den Wünschen der Anwender größere Teile des Systems u.a. auf folgenden Rechnern installiert werden: Siemens 7.760/BS2000, VAX 86.000/VMS, Eclipse/ MV10000, Apollo/Aegis, Sun/Unix. Die Installationszeiten für Erstinstallationen unter einem Betriebssystem liegen bei wenigen Tagen.

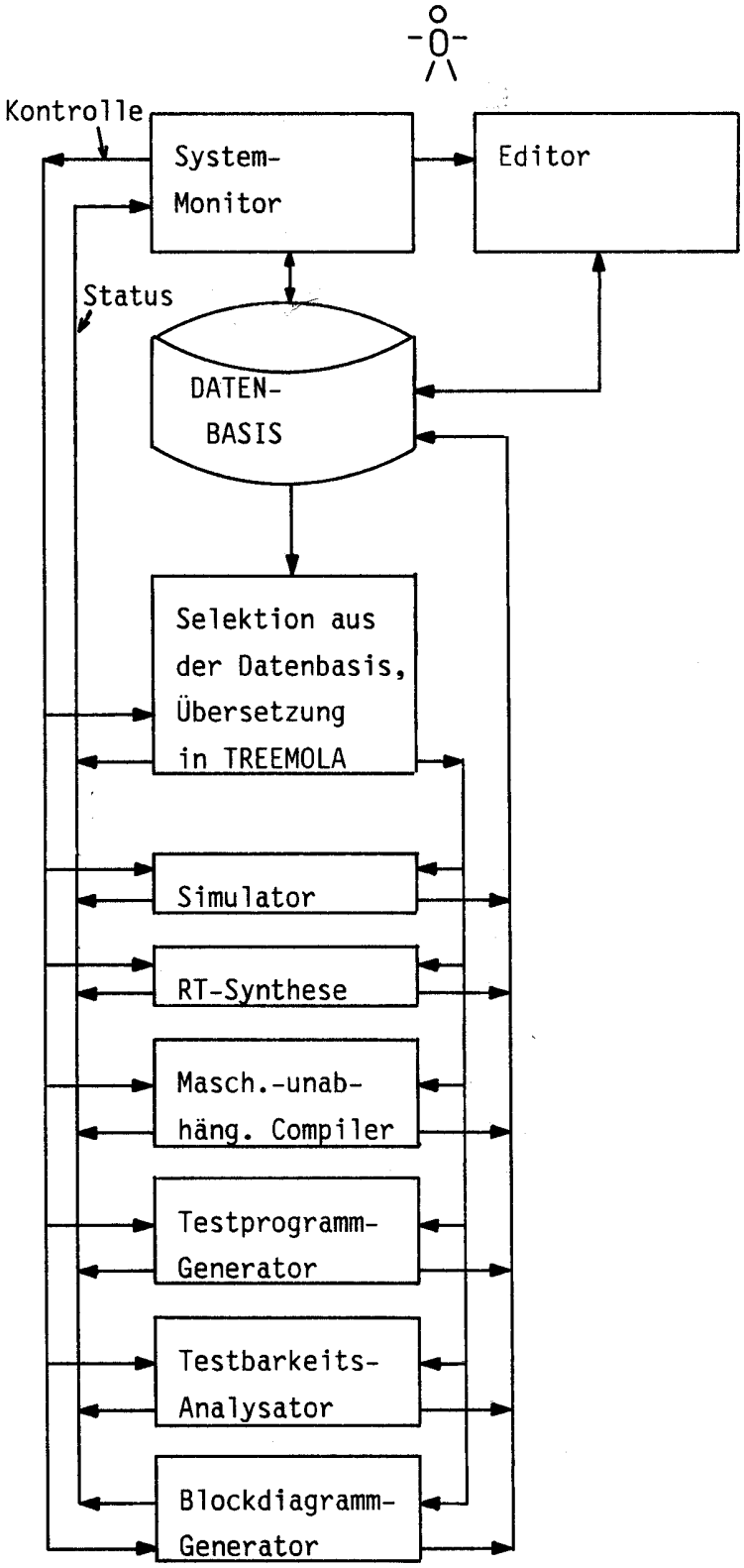


Abb. 1.3: Werkzeuge und Werkzeug-Umgebung im MSS

Jedes der Werkzeuge der Abb. 1.3 setzt die Kenntnis eines Teils der Aspekte des gegenwärtigen "current module under design" (CUD) voraus. Umgekehrt erzeugt jedes Werkzeug selbst solche Aspekte. Die entsprechenden Relationen zwischen Werkzeugen und Aspekten beschreibt Abb. 1.4: Aspekte sind jeweils in Rechtecken eingeschlossen, Werkzeuge sind einem der Balken zugeordnet. Optionale Eingaben eines Werkzeugs. sind durch 0 gekennzeichnet.

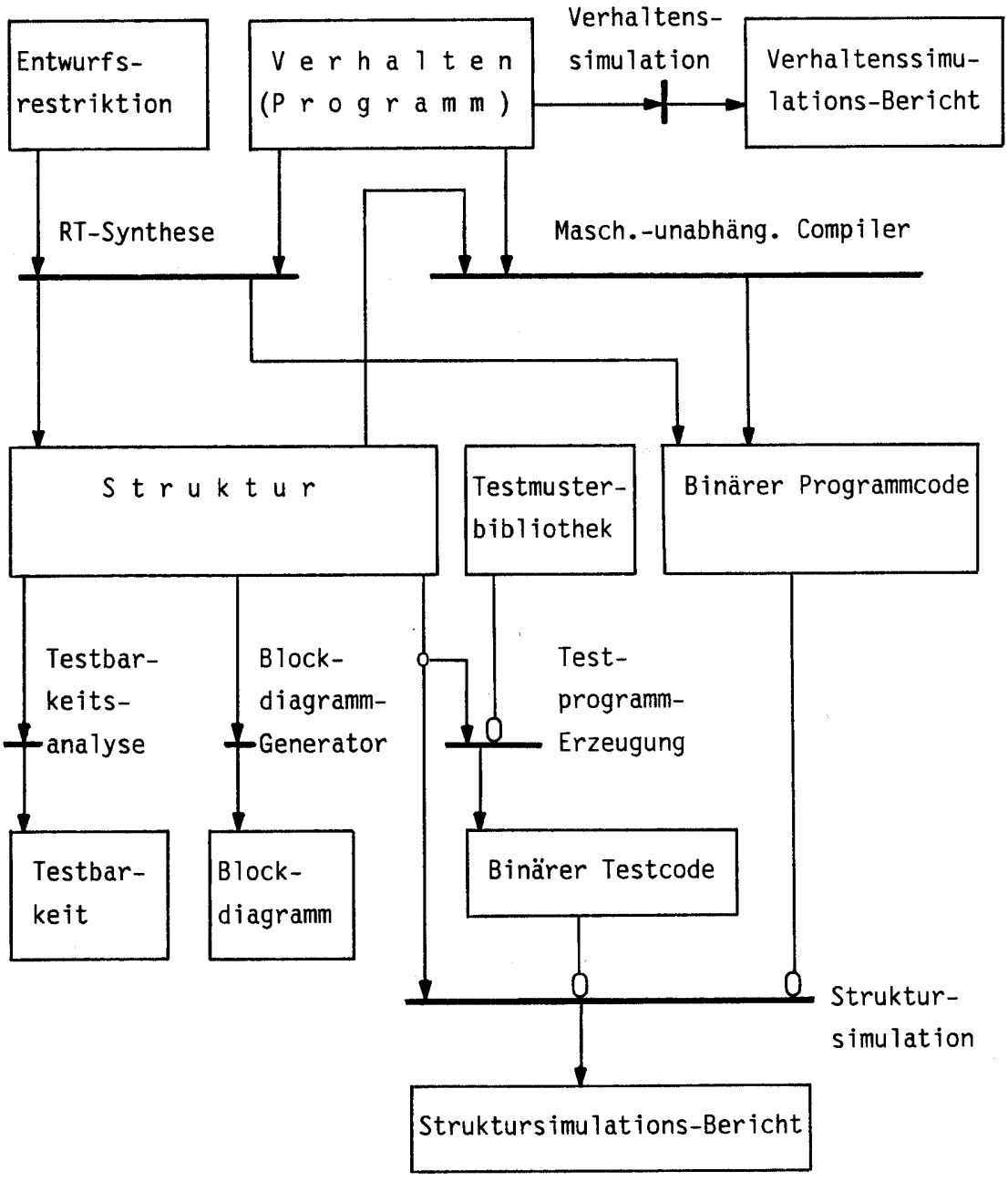


Abb. 1.4: Abhängigkeiten zwischen Werkzeugen und Aspekten

Jedem der hier aufgeführten Werkzeuge ist nachfolgend ein Kapitel gewidmet.

1.4 Literatur

[EDIF] Electronic Design Interchange Format Steering Committee: EDIF Electronic Design Interchange Format, Version 1 0 0

[Har84] R. Hartenstein und K. Lemmert : KARL-III Language Reference Manual, Fachbereich Informatik der Universität Kaiserslautern, 1984

[Hay82] J.P. Hayes: A Unified Switching Theory with Applications to VLSI Design, Proceedings of the IEEE, Vol. 70, 10(1982), S. 1140-1151

[Hör86] E. Hörbst, M. Nett und H. Schwärtzel: VENUS : Entwurf von VLSISchaltungen, Springer, 1986

[Lis77] B. Liskov et al. : Abstraction Mechanisms in CLU, Communications of the ACM, Vol. 20, 8(1977), S. 564-576

[Mar87] P. Marwedel : On the use of hierarchies in the MIMOLA hardware design system, CompEuro Conf. an Computers and VLSI, Hamburg, 1987

[Ram80] F.-J. Rammig: CAP/DSDL : Preliminary Language Reference Manual, Forschungsbericht Nr. 129 der Abteilung Informatik der Universität Dortmund, 1980

[VHDL] CAD Language Systems Inc., UHDL Language Reference Manual, Draft Standard 1076/A, IEEE, Dec. 1986

2 Synthese von RT-Strukturen

Unter Synthese versteht man generell das Zusammenfügen von Teilen (Komponenten) zu einem "System" mit dem Ziel, ein vorgegebenes Verhalten zu erreichen. Automatische Syntheseverfahren besitzen dabei gegenüber einem manuellen Entwurf den Vorteil, daß das System mit großer Sicherheit auch tatsächlich das gewünschte Verhalten zeigt, der Entwurf also **korrekt** ist. Außerdem wird der Entwurfsprozeß durch eine automatische Synthese wesentlich beschleunigt. Vor allem aufgrund dieser Eigenschaften ist eine deutliche Steigerung des Interesses an Verfahren zur Synthese von RT-Strukturen festzustellen.

2.1 Spezifikation

2.1.1 Verhaltens-Spezifikation

Verhaltensspezifikationen digitaler Systeme können auf unterschiedlichen Abstraktionsebenen erfolgen. Eine übliche Form der Spezifikation besteht aus der Vorgabe eines zu implementierenden Befehlssatzes. Diese Form wird benutzt, wenn zu einer existierenden Rechnerfamilie, z.B. zur IBM/370-Familie, ein neuer, kompatibler Rechner entwickelt werden soll. In diesem Fall kann man die Semantik eines Befehlssatzes durch Angabe eines Interpreters spezifizieren. Derartige Interpreter sind als mikroprogrammierte Interpreter großer Mainframes oder auch aus dem Buch von Bell und Newell (siehe [Sie82]) bekannt. Der bei diesem Vorgehen resultierende Entwurf enthält (mindestens) zwei Interpretationsebenen, nämlich 1. die Interpretation durch den mikroprogammierten Interpreter und z. die Interpretation des Interpreters durch die Hardware.

In einer Reihe von Anwendungsfällen ist man bestrebt, den Effizienzverlust durch die zweistufige Interpretation zu vermeiden, indem man algorithmische Programme (ggf. nach einer Compilation) direkt durch die Hardware interpretiert. In diesem Fall wird das gewünschte Verhalten direkt durch Anwendungsprogramme repräsentiert. Eine derartige Vorgehensweise ist z.B. sinnvoll, wenn spezielle Bildverarbeitungsoder Simulationsrechner entworfen werden sollen.

Da ein Interpreter auch ein Programm darstellt, besteht die Verhaltensspezifikation in beiden Fällen aus einem ("algorithmischen" oder "imperativen") Programm. Beide Fälle können von einem Entwurfssystem behandelt werden, welches Programme als Verhaltensspezifikation erwartet.
Das MIMOLA-Synthesesystem [Mar85,Mar86] MSSH ist ein solches System. Es erwartet eine Programm-Beschreibung in MIMOLA. Aus Akzeptanzgründen enthält die heutige Version dieser Sprache bis auf wenige Ausnahmen

(z.B. REAL-Zahlen) PASCAL als Subset.

Beispiel für ein MIMOLA-Programm:

```
 PROGRAM rem;
CONST n=7;
LABEL       Lx;
TYPE                          integer = (15:0) ;    (*16 Bit langer
Bitstring*)
VAR         a,b               : integer;
 BEGIN
 a:=13; b:=5;
Lx: WHILE a > 0 DO
 BEGIN
  a:= a - b;
 END;
 a:=a + b;
```

2.1.2 Entwurfsrestriktionen

Es gibt nun eine Vielfalt von Rechnerstrukturen, welche das vorgegebene Verhalten zeigen können. Beispielsweise könnte man die Programme auf praktisch alle klassischen Rechner compilieren. Eine derartige Lösung wäre aber in der Regel zu langsam und es kommt darauf an, den Entwurfsraum auf die Menge der interessierenden Lösungen zu begrenzen. Diesem Zweck dient ein zweiter Teil der Spezifikation, nämlich die Entwurfsrestriktionen. Entwurfsrestriktionen werden im wesentlichen in

Form einer partiellen Strukturbeschreibung formuliert.
Minimal enthält diese Strukturbeschreibung eine Beschreibung des Programmzählers und der Datenspeicher, also z.B.:

```
PARTS
 PC: MODULE R15              (*Programmzähler *)
      (IN e : (15:0); OUT a : (15:0); CLK c:(0));
     CONBEGIN
      AT c DO R15:=e;
      a <- R15
     CONEND;
  SM: MODULE S4k              «SIZE=4096»          (*Hauptspeicher*)
       (INOUT ea:(15:0); ADR address:(15:0); FCT sel:(0);CLK c:(0));
       BEGIN
        CASE sel OF
          0 : AT c DO S4k[address]:=ea;
          1 : ea<-S4k[address];
        END
       END;


  SR: MODULE S8 «SIZE=8»     (*Registerspeicher*)
       PORT P1 (IN e:(15:0); ADR ad:(2:0); FCT s:(0); CLK c:(0));
       PORT P2,P3(OUT f:(15:0); ADR ad:(2:0); FCT s:(0)                  );
       CONBEGIN
        WITH P1 DO
         CASE s OF
         0 : AT c DO S8[ad]:=e;
         1.;
         END;
        WITH P2,P3 DO
         f<- CASE s OF
          0 : Wad];
          1 : TRISTATE;                          (*hochohmiger Ausgang*)
       CONEND;
```

Zusätzlich kann die Strukturbeschreibung einen Katalog verfügbarer arithmetisch/logischer Bausteine enthalten.

Beispiel:

```
TYPE word =
 (15:0);
MODULE 87483 «COST=3» (IN l,r : word; OUT f : word); BEGIN
   f <- 1 + r;
  END;
MODULE B74xy « COST=10»
     (IN a,b : word; FCT sel : (1:0);
  OUT f : FIELDS eq : (16); res : (15:0) END);
 CONBEGIN
   f.res<- CASE sel OF
            0 . a - b;
            1 : a + b;
            2 . a * b;
            3 : a "DIV" b;
           END;
   f.eq<- CASE sel OF
            0 . a <= b;
            1 . a < b;
            2 . a = b;
            3 . a <> b; END
```

So kann garantiert werden, daß der fertige Entwurf nur Exemplare vorentworfener Bausteine enthält. Offensichtlich ist dies im Fall der Realisierung eines Rechners aus diskreten ICs notwendig. Aber auch im Fall der Standardzellen-Technik existieren Kataloge vorentworfener Bausteine.

Als weitere Beschränkung kann eine maximale Breite des Steuerwortes vorgegeben werden. Alternativ kann dies durch die Beschreibung eines Befehlsspeichers oder in einem interaktiven Prozeß geschehen.

2.1.3 Vorschriften für die Abbildung auf die RT-Ebenen

Zusätzlich zu den oben erwähnten Teilen enthält die Spezifikation noch Information darüber, wie die Verhaltensspezifikation auf die unteren Abstraktionsebenen abzubilden ist.

Dazu gehören die "Locations"-Vereinbarungen, mit denen Speicherzellen für bestimmte Zwecke reserviert werden. Speicherzellen können reserviert werden für die Aufnahme

- von Benutzer-Variablen ("LOCATIONS For Variables"),
- des Programmzählers ("LOCATIONS For ProgramCounter"),
- von Hilfsvariablen ("LOCATIONS-For Temporaries"),
- von Befehlen ("LOCATIONS For Instructions").

Beispiel:

```
LOCATIONS For Variables       SM [0..4095];
LOCATIONS For ProgramCounter  PC ;
```

Zum anderen gehören dazu auch Vorschriften für die Ersetzung von höheren Sprachelementen, wie z.B. Unterprogramm-Aufrufen, durch Operationen auf der RT-Verhaltensebene. Dies geschieht mittels ProgrammtransformationsRegeln.

Beispiel:

```
MACRO TransformIntoRTBehaviour IS
 REPLACE
 &1: WHILE &expr DO &block
 WITH
  &1: IF &expr THEN BEGIN &block; &1 a: PC:=&a END
 END;
```

Mit &-Zeichen beginnende Identifikatoren stellen formale Parameter dieser Transformationsregel dar.

2.1.4 Funktion des Synthesesystems

Zusammengenommen ergeben die Beispiele dieses Abschnitts eine vollständige Eingabe für das MIMOLA-Synthessystem.
Aufgabe des MIMOLA-Synthesesystems ist es, die Beschreibung eine Rechners auf der RT-Strukturebene zu generieren, welcher das vorgesehene Verhalten zeigt, die Entwurfsrestriktionen einhält und ansonsten möglichst schnell ist. Aus Komplexitätsgründen ist es notwendig, da<. Entwurfsproblem in eine Reihe von Teilproblemen zu zerlegen, welche in folgenden vorgestellt werden.

2.2 Ersetzung von höheren Sprachelementen

Die Verhaltensspezifikation liegt in der Regel auf einem PASCAL-artigen Niveau vor. Dann enthält diese Spezifikation höhere Sprachelemente, deren Realisierung in Hardware nicht offensichtlich ist, wie z.B. FOR-Schleifen und Unterprogramm-Aufrufe. Um die Erzeugung von Hardware vorzubereiten, werden höhere Sprachelemente im ersten Syntheseschritt durch explizite Operationen auf Daten ersetzt. Dies geschieht durch Anwendung definierter Programmtransformations-Regeln. Mit dem oben angeführten Programm und der oben angeführten Regel ergibt sich daraus z.B.:

```
Lx:          IF a>0 THEN
BEGIN
 a .=a - b;
Lx a:            PC:=Lx;
END;
```

Variable im Sinne von PASCAL stellen ebenfalls höhere Sprachelemente dar. Solche Variable werden üblicherweise Hauptspeicherzellen zugeordnet. Die Adressierung dieser Zellen kann statisch sein (wie etwa in FORTRAN) oder dynamisch (wie etwa in PASCAL). Im letzteren Fall wird die Adresse gebildet als Summe einer veränderlichen Basisadresse und eines konstanten Offsets. Die Basisadresse wird dabei meist in einer Zelle des Registerspeichers SR gehalten. Unter dieser Annahme geht obiges Programm z.B. über in:

```
Lx: IF SM [ SR[0] + 5 ] > 0 THEN
        BEGIN
         SM [SR[O] +5 ] := SM [ SR[0] + 5 ] - SM [ SR[0] + 6 ]; Lx
a: PC:=Lx END;
```

Als letztes höheres Sprachelement enthält das Beispielprogramm noch bedingte Anweisungsblöcke. Diese werden im nächsten Transformationsschritt in eine Form überführt, die die Generation von Hardware weiter erleichtert.

2.3 Auflösung von bedingten Anweisungsblöcken

Bedingt auszuführende Anweisungsblöcke werden von klassischen Compilern in eine Sequenz aus bedingt auszuführenden Sprüngen und unbedingten Zuweisungen zerlegt.

Beispiel:

Der Block

```
Lx: IF SM [ SR[0] + 5 ] > 0 THEN
        BEGIN
         SM [SR[0] +5 ] := SM [ SR[0] + 5 ] - SM [ SR[0] + 6 ]; Lx a:
PC:=Lx END;
```

wird zerlegt in

```
Lx      : PC                :_ (IF SM[SR[0]+5] > 0 THEN Lx -t ELSE Ly);
Lx t:   SM [SR[0] +5 ]          := SM [ SR[0] + 5 ] - SM [ SR[0] + 6 ];
Lx a:   PC                                                    .= Lx;
Ly . ......
```

Diese Zerlegung bewirkt eine zwangsweise Sequentialisierung, welche eine mögliche Parallelverarbeitung behindert. Das MSS unterstützt daher auch die hardwaremäßige Realisierung von bedingten Ausdrücken und bedingten Zuweisungen, die beide eine Realisierung von bedingten Anweisungsblöcken ermöglichen, ohne eine zwangsweise Sequentialisierung zu erfordern.
Beispiel:

Die hardwaremäßige Realisierung des parallelen Blocks

```
Lx:       PARBEGIN
          PC                  := (IF SM[SR[0]+5] >0 THEN Lx ELSE Ly);
 /SM[SR[0]+5] > 0/ SM[SR[0]+5] := SM[SR[0]+5] - SM[SR[0]+6];
PAREND;
Ly: ...
```

in der /SM[SR[0]+5]>0/ dem aus CDL bekannten Label oder auch Dykstra's "guard" entspricht, ist in einem einzigen Befehlszyklus möglich. Durch eine Sprungoptimierung wurde noch die Anweisung "Lx a: PC:=Lx" entfernt und die gesamte Schleife kann in einem Befehl ausgeführt werden.

Es hängt von den jeweiligen Entwurfsrestriktionen ab, welche der Zerlegungsmöglichkeiten zu dem schnellsten Rechner führt. Leider kann aber in einer so frühen Phase des Entwurfsprozesses noch nicht mit vertretbarem Aufwand entschieden werden, welche Zerlegung dies ist. Andererseits ist eine solche Zerlegung Voraussetzung für die folgenden Entwurfsschritte.

Infolgedessen werden maximal 3 verschiedene Alternativen der Zerlegung bedingter Anweisungsblöcke erzeugt. Für jede der Alternativen werden die nachfolgenden Syntheseschritte getrennt durchlaufen. Erst wenn der jeweils benötigte Code bekannt ist, erfolgt eine Entscheidung zugunsten einer der Alternativen.

Mit diesem Ansatz ist es möglich, die Beschleunigung durch die Verwendung bedingter Ausdrücke und bedingter Zuweisungen genauer zu quantifizieren. Ein. Beispiel dafür zeigt Abbildung 2.1:

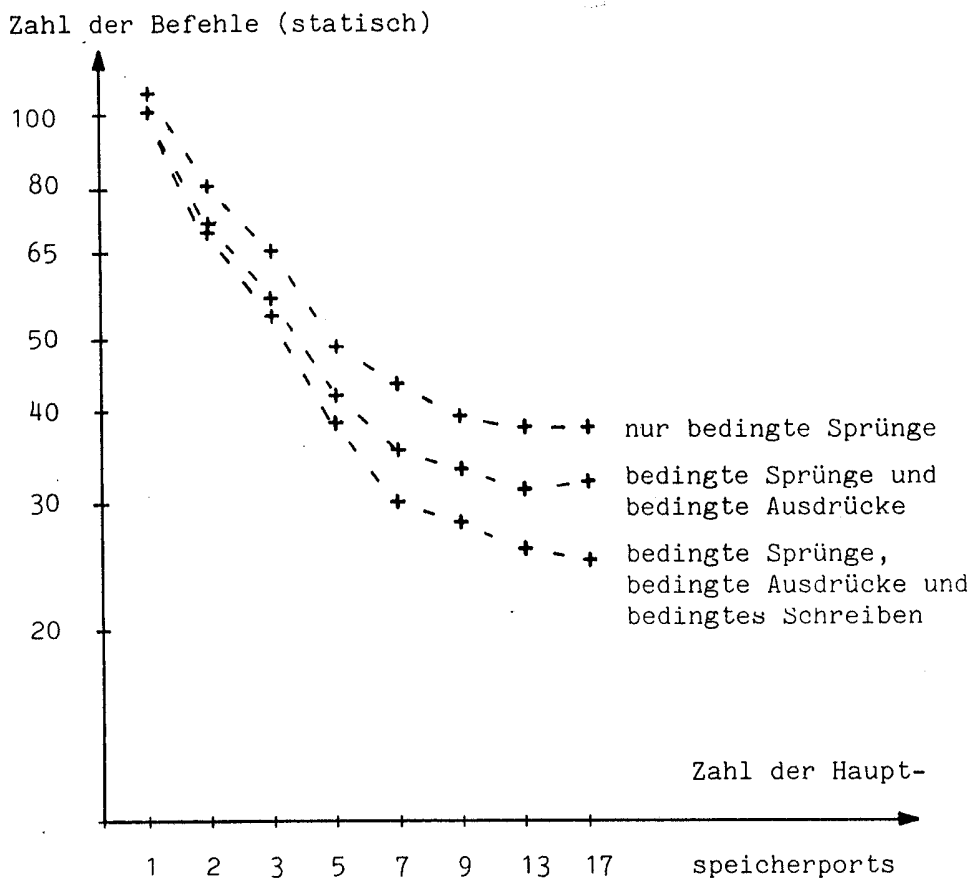


Abb. 2.1: Einfluß der Realisierung bedingter Blöcke

Erlaubt man zusätzlich zu bedingten Sprüngen die Benutzung bedingter Ausdrücke (mittlere Kurve), so ergibt sich gegenüber der ausschließlichen Benutzung bedingter Sprünge eine Reduktion sowohl der Zahl der dynamisch ausgeführten wie auch der Zahl der vom Compiler generierten Befehle. Dargestellt ist in der Abb. 2.1 nur die zweite Zahl. Wesentlich wird dieser Unterschied aber erst bei Rechnerstrukturen mit vielen Speicherports, also sehr parallelen Strukturen. Zur Erleichterung des Vergleichs ist die Zahl der Befehle in Abb.2.1 logarithmisch eingetragen.

Ebenfalls eine Reduktion der Zahl der Befehle ergibt sich, wenn zusätzlich bedingte Zuweisungen erlaubt werden (untere Kurve).

Diagramme dieser Art helfen bei der gegenseitigen Abstimmung von Prozessor- und Speicherstrukturen. künftiger, hochintegrierter Rechner.

2.4 Zerlegung von Ausdrücken

Im nächsten Syntheseschritt werden die parallelen Blöcke daraufhin analysiert, ob sie unter Einhaltung der Ressource - Restriktionen einschrittig ausführbar sind. Ist dies nicht der Fall, so werden die parallelen Blöcke in eine Sequenz einschrittig ausführbarer Blöcke zerlegt. Das Verfahren entspricht im Prinzip dem Aufbrechen komplexer Ausdrücke in eine Reihe von Maschinenbefehlen, wie es von üblichen Compilern vorgenommen wird. Das MIMOLA-System ist aber im Gegensatz zu üblichen Compilern nicht für eine spezielle Zielmaschine ausgelegt und erzeugt statt einzelner Maschinenbefehle Blöcke parallel ausführbarer Zuweisungen. Um für unterschiedliche Restriktionen generierte Rechner untereinander in bezug auf ihre Rechenleistung vergleichen zu können, muß das MSS für alle Restriktionen eine möglichst gute (am besten: die optimale) Zerlegung liefern. Es ist daher notwendig, gemeinsame Teilausdrücke bei der Zerlegung auszunutzen.

Eine besondere Bedeutung gemeinsamer Teilausdrücke im MSS ergibt sich auch dadurch, daß bei der Erzeugung bedingter Ausdrücke und bedingter Zuweisungen zusätzliche gemeinsame Teilausdrücke entstehen.

Während schnelle Algorithmen für eine optimale Zerlegung unter Vernachlässigung gleicher Teilausdrücke bekannt sind (vgl. [AhoJoh76, SetUll70]), liegen entsprechende Verfahren für den allgemeinen Fall nicht vor. Der Grund dafür liegt in der NP - Vollständigkeit des Problems. Lediglich für spezielle Fälle sind polynomielle Algorithmen bekannt. So betrachten Prabhala und Sethi [Pra80] Zerlegungen für eine kellerartige Verwaltung der Hilfszellen, spezielle Flußgraphen und einfache Maschinenbefehle.

Im MSS wird daher eine heuristische Methode zur Zerlegung der Ausdrücke benutzt. Sie basiert auf der Definition des Prädikats TreeTooBig und der Funktion MostComplex.

Sei t ein beliebiger, aufgrund der vorangegangenen Transformationen generierter Ausdruck oder eine Zuweisung. Dann sei TreeTooBig(t) definiert durch:

$$\text{TreeTooBig}(t) := \begin{cases} \textbf{true}, \text{ falls t aufgrund der Entwurfsrestriktionen} \\ \qquad\quad \text{nicht in einem Befehl ausführbar ist, und} \\ \textbf{false} \text{ sonst} \end{cases}$$

Die genaue Definition von TreeTooBig enthält als wichtige Kriterien die Zahl der Speicherreferenzen in t, die Zahl der verfügbaren Speicherports, die obere Grenze für die Befehlswortbreite und Abschätzungen der Kosten für die Implementierung arithmetisch/logischer Operationen. Beispielsweise ist TreeTooBig(t) = **true**, falls die Zahl der Referenzen auf einen Speicher die Zahl der Ports dieses Speichers übersteigt. Die Berücksichtigung einer beschränkten Zahl von Speicherzugriffen wird in vielen anderen Synthesesystemen (siehe z.B. [Gir84, Sou83, Tri87, Par86, Pau87]) umgangen, indem für jede Variable ein eigenes Register vorgesehen wird. Dadurch ergibt sich aber ein unbefriedigend hoher Hardwarebedarf.

Sei t wiederum ein Ausdruck oder eine Zuweisung. Dann liefere die Funktion MostComplex(t) den nach einem heuristischen Maß "größten" Unterausdruck von t, welcher einer Hilfsvariablen zugewiesen werden kann, ohne die Entwurfsrestriktionen zu verletzen. MostComplex wählt einen Unterausdruck gemäß der folgenden Prioritäten:

1. Zahl der Speicherreferenzen
2. Zahl der Referenzen auf Speicher, die nicht der Aufnahme von Hilfsvariablen dienen
3. Gleiche Teilausdrücke
4. Boolsche Ausdrücke
5. Von links nach rechts

Beispielsweise selektiert MostComplex unter Ausdrücken mit gleicher Zahl von Speicherreferenzen vorzugsweise solche, welche gleiche Teilausdrücke darstellen.

Das Verfahren zur Aufspaltung der Ausdrücke arbeitet nun wie folgt (die Prozedur Push hinterlegt .die erzeugten Zuweisungen zur Weiterverarbeitung in einem Stack):

```
PROCEDURE Chopping (s);
  BEGIN
  V t, mit t ist Unterausdruck von s, beginnend mit den Blättern von s
    DO BEGIN
    WHILE TreeTooBig(t) DO
    BEGIN
      e:=MostComplex(t);
      IF e=nil THEN Fehler('keine Lösung') ;
      Weise den durch e repräsentierten Ausdruck einer
       Hilfsvariablen zu; Sei e' der erzeugte Baum.
      Push(e');
      Ersetze e durch Lesen der Hilfsvariablen
   Ersetze alle Teilausdrücke, die gleich dem durch e
    repräsentierten sind, ebenfalls durch Lesen der
    Hilfsvariablen, sofern diese nicht bereits Teil eines Push
    übergebenen Ausdrucks waren END; END;
  Push(s)
 END;
```

Alg. 2.1 Zerlegung von Ausdrücken

Bei einem Hauptspeicherport würden für obiges Beispielprogramm die folgenden Zuweisungen auf dem Keller abgelegt:

```
SR [v]              := SM [ SR[0] + 5 ];
SR [v']             := SR[v] - SM [ SR[0] + 6 ];
SM [ SR[0] + 5 ]    := SR[v'];
```

v und v' stellen dabei (möglicherweise verschiedene) Adressen der Hilfsvariablen SR[v] und SR[v'] dar. Die Vergabe von Adressen wird zurückgestellt, um so zu einer geringeren Zahl benötigter Hilfszellen zu kommen.

2.5 Scheduling von Zuweisungen

Das Resultat des letzten Syntheseschrittes besteht aus einer Menge von Zuweisungen zu verschiedenen Speichern, wie etwa zum Hauptspeicher, zum Registerspeicher, zum Condition-Code-Register u.s.w. Es wäre jetzt eine Einschränkung der möglichen Parallelität, wenn pro generiertem Befehl jeweils nur zu einem Speicher eine Zuweisung ausgelöst werden könnte. Es sollen daher Befehle erzeugt werden, die Zuweisungen zu mehreren Speichern veranlassen können.
Damit entsteht das Problem, eine Zuordnung von Zuweisungen zu Befehlen zu finden, die die Programmausführungszeit minimiert und die Entwurfsrestriktionen einhält. Da dabei auch die Ausführungsreihenfolge der Zuweisungen festgelegt wird, wird dieser Syntheseschritt als "Scheduling von Zuweisungen" bezeichnet. Das Zuordungsproblem tritt in der gleichen Form auch in der Mikrocode-Kompaktierung auf.

Bei einer solchen Zuordnung ist darauf zu achten, daß Speicherzellen nicht überschrieben werden, bevor alle Zugriffe auf den alten Inhalt erfolgt sind und daß alle Zwischenergebnisse abgespeichert sind, bevor darauf zugegriffen wird. Hierzu ist eine sorgfältige Datenflußanalyse erforderlich. Klassische Definitionen [Pad80] und Verfahren sind dabei im MSS nicht direkt anwendbar, da diese keine parallelen Blöcke berücksichtigen.

Es wurde daher ein Definitionsgerüst aufgebaut, welches eine klare Beschreibung des implementierten Scheduling-Verfahrens erlaubt.

Seien s_i und s_j zwei Zuweisungen.

Def.:

1. s_i heißt **datenabhängig** von s_j (in Zeichen: s_i **da** s_j) g.d.w. s_j eine Speicherzelle mit einem Inhalt beschreibt, der von s_i benötigt wird.

Beispiel: Man betrachte die folgende Sequenz von Zuweisungen:

```
BEGIN
 s1: a:=b;
 s2: c:=2*a;
 s3: a:=a*c;
END;
```

Es gilt: s_2 **da** s_1, s_3 **da** s_1, s_3 **da** s_2

Def.:

2. s_i heißt **antidatenabhängig** von s_j (in Zeichen: s_i **ada** s_j) g.d.w. s_i eine Speicherzelle beschreibt, die von s_j gelesen wird, aber der von s_i erzeugte nicht der von s_j benötigte Inhalt ist.

Beispiel:

In der obigen Sequenz gilt: s_3 **ada** s_2, da s_3 eine Variable überschreibt, die von s_2 benötigt wird.

Offensichtlich können Antidatenabhängigkeiten höchstens dann entstehen, wenn Variablen mehrfach ein Wert zugewiesen wird, d.h. das sog. "single assignment"-Prinzip gerade nicht erfüllt ist. U.a. deshalb ist es sinnvoll, zwischen den Relationen **da** und **ada** zu unterscheiden. In der Literatur zur Mikroprogrammierung erfolgt diese Unterscheidung meist nicht (siehe z.B. [Mal78]).

Def.:

3. s_i heißt **potentiell datenabhängig** von s_j (in Zeichen: s_i **da*** s_j) g.d.w. zur Übersetzungszeit mit einem bestimmten Aufwand nicht ausgeschlossen werden kann, daß s_j eine Speicherzelle mit einem Inhalt beschreibt, der von s_i benötigt wird.

4. Analog zu 3. wird die potentielle Antidatenabhängigkeit (in Zeichen: s_i **ada*** s_j) definiert.

Die genaue Form der Definitionen findet sich in [Mar85].

Auf dieser Grundlage läßt sich ein Scheduling-Algorithmus formulieren.

Seien $MI(s_i)$ bzw. $MI(s_j)$ die Befehle, denen s_i bzw. s_j zugeordnet werden. $MI(s_i) < MI(s_j)$ bedeute, daß $MI(s_i)$ vor $MI(s_j)$ ausgeführt wird. $MI(s_i) \leq MI(s_j)$ bedeute, daß $MI(s_i)$ vor $MI(s_j)$ ausgeführt wird oder daß $MI(s_i) = MI(s_j)$ ist. Dann sind nur solche Schedules zulässig, für die gilt:

1. $\forall\ s_i, s_j$ mit s_i **da*** s_j: $MI(s_j) < MI(s_i)$
2. $\forall\ s_i, s_j$ mit s_i **da*** s_j existiert kein k derart, daß s_k **ada*** s_i und $MI(s_j) \leq MI(s_k) < MI(s_i)$.

Gilt für zwei Zuweisungen s_i **ada*** s_j und s_j **ada*** s_i, so heißen s_i und s_j (direkt) **zyklisch voneinander abhängig.**

Die erste Implementierung dieses Syntheseschrittes basierte auf einem first-in first-out (FIFO-) Verfahren, welches zuerst von Dasgupta und Tartar angegeben wurde. Für s_i **da*** s_j wird bei diesem Verfahren stets s_j zuerst einem Befehl zugeordnet. In [Mar85] wird gezeigt, daß mit diesem Ansatz ein backtracking erforderlich werden kann, wenn zyklisch abhängige Zuweisungen existieren. Dieses backtracking wird vermieden, wenn man zu einem LIFO - Verfahren übergeht. In einem solchen Verfahren wird für s_i **da*** s_j stets s_i zuerst einem Befehl zugeordnet. Das LIFO-Verfahren führt außerdem in vielen Fällen zu einem reduzierten Bedarf an Hilfsspeicherzellen. Eine vollständige Beschreibung des Scheduling-Verfahrens findet sich in [Mar85].

2.6 Register-Allokation für Hilfsvariable

Die Vergabe von Adressen für Hilfsvariable soll so erfolgen, daß die Zahl der benötigten Hilfszellen minimal wird. Dieses Zuordnungsproblem wird meist mit Register - Allokation bezeichnet. Allgemein läßt sich das Problem auf das Färbungsproblem, oder -äquivalentauf das Cliquenproblem in Graphen zurückführen (vgl. z.B. [Ber85, MueDud0Ha84], siehe auch [Sch80]). Beschränkt man sich auf die Optimierung innerhalb verzweigungsfreier Codesequenzen, so entartet das

Färbungsproblem zum Färben von Intervallgraphen, was in linearer Zeit möglich ist. Die benötigte Zahl von Registern wird als "cutwidth of a linear arrangement" bezeichnet [Pfa86].

Die Lösung des Färbungsproblems läßt sich wie folgt formulieren: Sei:

V = { 1:vmax } die Menge der Indices der Hilfsvariablen,
Z = { 1:zmax } die Menge der Hilfszellen,
N = { 1:nmax } die Menge der Indices der MI.

Für k ∈ V seien die Funktionen Def(k) und Ref(k) definiert:

Def(k) : die Menge der Indices jener MI, welche Zuweisungen an die Hilfsvariable v_k enthalten,

Ref(k) : die Menge der Indices jener MI, welche lesende Referenzen auf die Hilfsvariable v_k enthalten.,

Für n ∈ N seien die Funktionen First(n) und Last(n) definiert:

First(n) : = { v | v ∈ V, Min(Def(k)) = n }

Last (n) : = { v | v ∈ V, Max(Ref(k)) = n }

Darin bedeuten Min() und Max() das kleinste bzw. das größte Element einer (nicht-leeren) Menge.

Gesucht ist eine Abbildung Location: V -> Z derart, daß eine minimale Zahl von Hilfszellen benutzt wird und es gilt:

Für alle j,k ∈ V, mit j ≠ k :
(location[j] ≠ location[k]) oder
Min(Def(j))) ≥ Max(Ref(k)) oder
Min(Def(k))) ≥ Max(Ref(j))

Das folgende Programm gibt das Verfahren zur Bestimmung der Abbildung Location wieder:

```
VAR
 Location : ARRAY[1 ..          vmax] OF 1                      zmax;
 (*Abb. V -> Z*)
 FreeList : SET OF 1 .                                          zmax;
 BEGIN
    FreeList:={1..zmax};
    FOR i:=1 to nmax DO         (*Schleife über MI                          *)
    BEGIN
    FreeList:=FreeList U Last(i);                          (*Zellen-Freigabe *)
    FOR ALL k e First(i) DO
    BEGIN
 z:=AnyElement(FreeList);                              (*Irgendein z s FreeList*)
 FreeList:=FreeList-(z);                                 (*Zellen-Reservierung *)
 Location[k]:=z;
 END;
END;
```

Alg. 2.2 Register - Allokation

In einer Schleife über die generierten Instruktionen (FOR i:=..) werden zunächst die Zellen jener Hilfsvariablen freigegeben,die in der Instruktion MI; zuletzt aelesen werden. Anschließend werden Zellen für die

Hilfsvariablen blockiert, die in Instruktion MI¡ definiert werden.

Die Komplexität des Verfahrens ist 0(nmax), d.h. proportional zur Zahl der generierten Instruktionen.
Das Verfahren liefert bei gegebener Liste von Mikroinstruktionen stets eine Lösung mit der minimalen Zahl benötigter Hilfszellen, da die Freigabe von Zellen stets vor der Blockierung erfolgt. Um insgesamt zu einer kleinen Zahl benötigter Hilfszellen zu kommen, werden durch das Scheduling schreibende Zugriffe auf Hilfsvariable möglichst dicht vor den lesenden plaziert.

2.7 Auswahl arithmetisch/logischer Bausteine

In diesem Syntheseschritt werden arithmetisch/logische Bausteine aus einer Bibliothek derart ausgewählt-, daß für alle generierten Befehle ausreichend viele Bausteine vorhanden sind. Dabei wird angenommen, daß jeder Operation eines Befehls ein Hardware - Baustein für die Dauer der Befehlsausführung fest zugeordnet werden kann (diese Annahme führt zu einer besonders einfachen Kontroll-Logik).

M : Die Menge arithmetisch/logischer Modultypen,
k_m : die (ggf. fiktiven) Kosten des Typs m, $m \in M$,
o_m : die Menge der vom Typ $m \in M$ bereitgestellten Operationen
$f_{i,j}$: die Häufigkeit des Vorkommens der Operation j in Befehl i (d.h. z.B. die Zahl der 16-Bit Additionen in diesem Befehl)

Zu bestimmen ist die Zahl der Exemplare x_m des Modultyps m derart, daß für alle Befehle ausreichend viele Bausteine zur Verfügung stehen und daß die Gesamtkosten $\sum_{m \in M} x_m * k_m$ minimal werden.

Sei F_i die Menge der in Instruktion i benutzten Operationen und F_i^* deren Potenzmenge:

$$F_i := \{ j \mid f_{i,j} > 0 \}$$
$$F_i^* := \mathcal{P}(F_i)$$

Ein offensichtlich hinreichendes Kriterium für eine ausreichende Zahl von Bausteinen ist das folgende:

$$\forall i, \forall g \in F_i^* : \sum_{\substack{m \in M \\ g \cap o_m \neq \emptyset}} x_m \geq \sum_{j \in g} f_{i,j} \qquad (1)$$

D.h. für alle Instruktionen i und alle Elemente g der Potenzmenge der in Befehl i benutzten Operationen ist die Summe der Exemplare jener Modultypen, die mindestens eine benötigte Operation ausführen können,

mindestens gleich der Gesamtzahl der ausgeführten Operationen.

Beispiel:

Gegeben sei der Befehl

SR[3] := SR[3] - SM[SR[0]+5]

Daraus resultieren folgende Ungleichungen:

Die Zahl der Bausteine, die "+" ausführen können, ist ≥ 1.
Die Zahl der Bausteine, die "-" ausführen können, ist > 1.

nie Zahl Klar Balisteine die „-„ oder „+" ausführen können, ist ≥ 2.

Mit $a_{g,m} := 0$, falls $(g \cap o_m = \emptyset)$
1 , falls $(g \cap o_m \neq \emptyset)$
folgt aus (1) :

$$\forall i, \forall g \in F_i^* : \sum_{m \in M} a_{g,m} * x_m \geq \sum_{j \in g} f_{i,j}$$

Für irgendein festes $g \in F^* = \bigcup_i F_i^*$ können diese
Ungleichungen nur erfüllt sein, wenn

$$\sum_{m \in M} a_{g,m} * x_m \geq \max_i \left(\sum_{j \in g} f_{i,j} \right)$$

Mit $b_g := \max_i \left(\sum_{j \in g} f_{i,j} \right)$ folgt :

$$\forall g \in F^* : \sum_{m \in M} a_{g,m} * x_m \geq b_g \qquad (2)$$

(2) zeigt, daß die Randbedingungen in Form von linearen Ungleichungen darstellbar sind. Damit ist die optimale Wahl der x_m auf eine ganzzahlige lineare Optimierung zurückgeführt. In der Implementierung wurde der Gomory-I Algorithmus benutzt.

Der Vorteil dieses Verfahrens liegt vor allem darin, daß eine globale Optimierung der arithmetisch / logischen Bausteine bei gleichzeitig geringer Komplexität möglich ist. Es stellte sich heraus, daß die Mächtigkeit der Menge F* in der Praxis kleiner ist, als zunächst erwartet wurde. Die folgende Tabelle zeigt die Mächtigkeit von F* (d.h. die benötigte Zahl von Ungleichungen) und die Laufzeiten, für den Gomory-I Algorithmus für drei praktische Anwendungsfälle:

Programm	Kern eines Parsers	Kern eines Experten-Systems	einfacher Logik-Simulator
| F* |	20	33	5
Zahl der Variablen	11	11	7
CPU-Zeit [ms] (1 Mips)	35	30	33

2.8 Zuordnung von Hardware-Ressourcen

Mit dem Abschluß des letzten Syntheseschrittes sind die wesentlichen Bausteine des "Rechenwerks" (engl. "data path") bekannt: Die Datenspeicher waren bereits Bestandteil der Spezifikation, arithmetisch/ logische Bausteine wurden durch die Synthese ausgewählt. Die Verhaltensbeschreibung wurde in Verhaltensbeschreibungen einzelner Befehle zerlegt. Jeder dieser Befehle enthält eine gewisse Menge von Operationen, wie z.B. Additionen oder Schreib-Operationen. Bislang wurde diesen Operationen aber noch keine Hardware-Ressource zugeordnet. Im allgemeinen ist die Zuordnung von Hardware-Ressourcen nicht eindeutig: Schreib- und Lese-Operationen können verschiedenen Ports eines Speichers zuordnet werden, arithmetisch/logische Operationen können alternativ von verschiedenen ALUS durchgeführt werden, und Konstanten können wahlweise von verschiedenen Befehlsfeldern oder auch von fest verdrahteten Konstanten generiert werden.

Betrachten wir als Beispiel den Befehl

```
SR[3] := SR[3] - SM[SR[O]+5]
```

Unter der Annahme, daß sowohl Baustein ALUO als auch ALU1 subtrahieren können, kann alternativ einer dieser Bausteine der Subtraktion zugeordnet werden. Die Zelle SR[3] kann entweder über Port P2 oder P3 gelesen werden. Entscheidet man sich für Port P2 und für ALUO, so macht dies eine physikalische Verbindung von P2 zu einem Eingang der ALUO notwendig.

Verallgemeinert gilt: Die Lösung des **Zuordnungsproblems impliziert die Verbindungsstruktur des Rechners.**
Die Zuordnung muß daher derart vorgenommen werden, daß die Kosten für Verbindungen möglichst gering werden. Da bislang die Kosten in Form der benötigten Chip-Fläche noch nicht vorhergesagt werden können, wird z. Zt. vom MSS die Zahl der **Verbindungen** möglichst gering gehalten.
Es zeigt sich, daß Spezialfälle des Zuordnungsproblems auf eine Variante des "Quadratischen Zuordnungsproblems", auf das sog. "tree-QAP" [Lop83] zurückgeführt werden können. Damit ist für eine optimale Lösung eine exponentielle Laufzeit zu erwarten. Im gegenwärtigen MSS wurde daher ein heuristisches Verfahren implementiert, welches auf einem Branchand-Bound Ansatz basiert. Dieses Verfahren ordnet nicht nur geeignete Hardware-Ressourcen zu, sondern entscheidet auch, welche Kontroll-Codes zu verwenden sind. Kann von einer ALU beispielsweise (abhängig vom Kontroll-Code) sowohl der rechte vom linken, als auch der linke vom rechten ALU-Eingang subtrahiert werden, so erfolgt die Auswahl des Codes derart, daß die Verbindungen minimiert werden. Im Rahmen dieser Optimierung wird auch die Kommutativität genutzt und es werden zueinander konverse Operationen wie "<" und ">" durcheinander ersetzt.

Als Ergebnis des Zuordnungsverfahrens kann es erforderlich werden, mehrere Ein- und Ausgänge miteinander zu verbinden. Das Synthesesystem fügt in diesem Fall die erforderlichen Multiplexer und TRISTATE-Treiber ein.

2.9 Optimierung der Befehlswortbreite

Durch eine geeignete Beschaltung der Steuereingänge aller in der RT-Struktur vorhandenen Module muß dafür gesorgt werden, daß die Module in einem bestimmten Befehl die gerade benötigte Operation ausführen. Das Synthesesystem benutzt zu diesem Zweck die "direct encoding" Methode [Agr76]: Jedem Steuereingang wird direkt ein Befehlsfeld zugeordnet. Da in einigen Befehlen die Beschaltung der Steuereingänge redundant ist, können sich die Befehlsfelder überlagern. Durch eine solche Überlagerung reduziert sich die Befehlswortbreite. Klassische Verfahren zur Wortbreitenreduktion (vgl. [Das79]) sind im Fall des MSS nicht anwendbar, da sie auf dem Codierungsmodell von Wilkes beruhen, in dem Multifunktionsbausteine nicht vorkommen. Es wurde daher ein eigenes Verfahren zur Überlagerung von Befehlsfeldern entwickelt und implementiert.

Sei l_i die Länge des Befehlsfeldes i in Bits.
Sei $c_{i,j}$ = **true**, falls die Felder i und j in mindestens einem Befehl nicht gleichzeitig redundant sind und =**false** sonst.
Gesucht ist eine Anordnung der Felder im Befehlsformat derart, daß insgesamt ein möglichst schmales Befehlswort resultiert.

Dieses Problem ist äquivalent zu einem Scheduling-Problem von Jobs der Ausführungszeit l_i mit durch $c_{i,j}$ beschriebenen Ressource - Konflikten. Dieses Problem ist NP - vollständig. Zur Lösung wird daher ein heuristisches Verfahren benutzt:

- Den jeweils längsten Feldern wird zuerst eine Position im Befehlsformat zugeordnet.
- Unter den gleich langen Feldern haben die Felder mit den meisten Konflikten Vorrang.
- Kann ein Feld aufgrund der $c_{i,j}$ mehreren Positionen innerhalb des Befehlsformates zugeordnet werden, so erfolgt die Auswahl nach der "Best fit" - Methode.

Dieses Verfahren stellt eine Erweiterung des sog. LPT- Scheduling dar. Die erzielten Reduktionen der Wortbreite schwanken von Anwendung zu Anwendung stark. Sie liegen üblicherweise zwischen 50% und 0%.

2.10 Binden der Programme an die Struktur

In diesem Syntheseschritt werden die zur Ausführung einer bestimmten Operation des Programms erforderlichen Hardware-Ressourcen explizit in das Programm einkopiert. Dadurch wird u.a. die Erzeugung von Benutzungsstatistiken vorbereitet.

2.11 Exemplarische Anwendung

Eine Anwendung des Synthesesystems soll im folgenden exemplarisch vorgestellt werden. Als Beispiel dient dazu das bei Wirth [Wir75] angegebene Programm **mergesort** zum Sortieren ganzer Zahlen (bei einem echten Entwurf würde man ein umfangreicheres Programm verwenden). Für dieses Beispiel soll die Programmlaufzeit und die Codemenge der mit dem MSS erzeugten Entwürfe mit einem konventionellen Rechner verglichen werden.

Die Ausführungszeit eines Befehls der mit dem MSS erzeugten Rechner sei 1 Mikrosekunde. Dieser Wert läßt sich technisch leicht erreichen. Mit den Ausführungshäufigkeiten für 20 zu sortierende Zahlen ergeben sich so die Zeiten der Abb. 2.2. Als Vergleich wird die Ausführungszeit des entsprechenden PASCAL-Programms auf einer Rechenanlage des Typs Siemens 7.760 benutzt. In beiden Fällen ist die Ein/Ausgabe nicht berücksichtigt.

Es könnte nun der Eindruck entstehen, daß dieses günstige Ergebnis durch die Konstruktion von Spezialrechnern für die jeweils eingegebenen Programme entsteht. Dies ist aber nicht der Fall, da voll programmierbare Rechner erzeugt werden und z.B. spezielle Konstanten (außer der Null) nicht hartverdrahtet

	MSS - Entwurf		SIEMENS 7.760
Hauptspeicherports	1	4	PASCAL V 3.1A10
Konstantenfelder	1	4	(keine
ALUS	1	4	Laufzeittests)
Laufzeit [Millisek.]	3,3	1,0	6,2
Programmcode [Bytes]	1143	755	936

Abb. 2.2: Vergleich von Laufzeiten und Codemengen

2.12 Zusammenfassung

Im Synthesesystem wird ein neues Verfahren zur Zerlegung des Problems in eine Reihe von Teilproblemen handhabbarer Komplexität angewandt. Das Verfahren ist vollständig implementiert und auf Rechnern verschiedener Hersteller verfügbar.

Um im Vergleich zu anderen Synthesesystemen Hardware einzusparen, werden Variable in adressierbaren Speichern und nicht (wie üblich) in eigenen Registern untergebracht. Wegen der beschränkten parallelen Zugriffsmöglichkeiten auf Speicher entsteht dadurch als erstes wesentliches Teilproblem das Problem der **Zerlegung komplexer Ausdrücke** in eine Menge einfacherer Ausdrücke. Im Synthesesystem wird ein neues Verfahren zur Zerlegung komplexer Ausdrücke angewandt, welches vorhandene gemeinsame Teilausdrücke zur Optimierung nutzt.

Soweit eine ausreichende Zahl von Hardware-Ressourcen gemäß der Entwurfsrestriktionen zulässig ist, generiert die Synthese Befehle, die eine **parallele Ausführung** mehrerer **Zuweisungen** bewirken. Dadurch soll die Ausführungsgeschwindigkeit im Vergleich zu üblichen sequentiellen Maschinen gesteigert werden. Die Suche nach geeigneten parallel ausführbaren Zuweisungen wird als **Scheduling** bzw. als **Kompaktierung** bezeichnet.

Es wird ein neues Schedulingverfahren vorgestellt, welches insbesondere die Erfordernisse paralleler Blöcke berücksichtigt. Um beim Scheduling möglichst viele Freiheiten zu haben, werden die Register zur Aufnahme von Zwischenergebnissen im Gegensatz zur gängigen Praxis nach statt vor dem Scheduling vergeben.
Für unterschiedliche Technologien liegen meist vorentworfene HardwareBausteine wie z.B. arithmetisch/logische Netzwerke vor. Solche Bausteine können pro Programmschritt jeweils eine Operation aus einer Liste von Operationen oder Funktionen ausführen. Sie heißen daher Multifunktionsbausteine. Die kostengünstige **Auswahl von Multifunktionsbausteinen** wird auf die lineare Programmierung zurückgeführt. Die Zahl der benötigten Variablen wird im Vergleich zu den Verfahren, die für einfache Bausteine existieren, wesentlich reduziert. Damit wird eine Lösung der linearen Programme in vertretbarer Zeit möglich.

Aufgrund des Flächenbedarfs von Leitungen in der VLSI - Technologie wichtig ist ferner die **Minimierung der Zahl benötigter Verbindungen** durch das MIMOLA-System. Eine solche Minimierung fehlt in vielen anderen Entwurfssystemen.

Die als Resultat der Synthese erhaltenen Rechnerstrukturen sind sowohl in bezug auf die Laufzeiten der Programme wie auch in bezug auf die Dichte des Codes zumindest nicht schlechter als konventionelle Rechner, häufig sogar schneller. Entgegen einer verbreiteten Auffassung ergibt sich sogar bei einer direkten Ansteuerung der Hardware-Bausteine durch Felder des aktuellen Befehls (sog. "direct encoding" [AgrRau76]) ein kompakter Programmcode.

Die Erfahrung zeigt, daß die automatisch erzeugten Strukturen trotz weitreichender Optimierungen von Benutzern gern modifiziert werden. MIMOLA bietet den Vorteil, daß diese Eingriffe durch Editieren der erzeugten MIMOLA-Beschreibungen zunächst einmal dokumentiert werden können. Eine Verifikation der Eingriffe kann mit dem Maschinenunabhängigen Compiler (vgl. Kap. 4) vorgenommen werden. Im Falle unzulässiger Eingriffe in die Struktur wird der Compiler nicht mehr in der Lage sein, das binäre Maschinenprogramm zu generieren und entsprechende Fehlermeldungen liefern.

2.13 Literatur

[Agr76] A.K. Agrawala und T.G. Rauscher: Foundations of Microprogramming,, Acadamic Press, New York, 1976

[AhoJoh76] A.V. Aho und S.C. Sethi : Optimal Code Generation for Expression Trees, Journal of the ACM, Vol. 23, 3(1976), S. 488-501

[Ber85] T. Berger : Ein Programm zur Speicheroptimierung im MIMOLA-Software-System (Diplomarbeit), Institut für Informatik und Prakt. Mathematik der Universität Kiel, 1985

[Das79] S. Dasgupta : The Organization of Microprogram Stores, Computing Surveys, Vol. 11, 1(1979), S. 39-65

[Gir84] E.F. Girczyc und J.P. Knight : An ADA to Standard Cell Hardware Compiler Based an Graph Grammars and Scheduling, Proc. ICCD, 1984, S. 726-731

[Lop83] E.B. Lopez : El problema de la asignacion cuadratica -Estudio del caso en que el grafo de flujos es un arbol, Cuadernos de Bioestadistica, Madrid, Vol. 1, 1983, S. 127-131

[Mal 78] P.W. Mallett: Methods of compacting microprograms, Dissertation, University of Southwestern Louisiana, Lafayette, 1978

[Mar85] Ein Software-System zur Synthese von Rechnerstrukturen und zur Erzeugung von Mikrocode, Habilitationsschrift, Universität Kiel, 1985

[Mar86] P. Marwedel : An Algorithm for the Synthesis of Processor Structures from Behavioural Specifications, Proc. EUROMICRO '86, S. 251-262

[MueDudOHa84] R.A. Mueller, M.R. Duda und S.M. 0'Haire
A Survey of Resource Allocation Methods in Optimizing Microcode Compilers, 17th Annual Microprogramming Workshop, 1984, S. 285-295

[Pad80] D.A. Padua, D.J. Kuck und D.H. Lawrie : High - Speed Multiprocessors and Compilation Techniques, IEEE Trans. Comp., Vol. C-29, 9(1980), S. 763-776

P. Pfahler, Universität-GH Paderborn, mündliche Mitteilung,1986

[Pfa86] B. Prabhala und R. Sethi : Efficient Computation of Expressions with Common Subexpressions, Journal of the ACM, Vol. 27, 1(1980), S. 146-163

[Pra80] F. Scholz : Registerzuteilung für Ausdrücke mit gemeinsamen Unterausdrücken auf dem Adressierungsniveau (Dissertation), Universität Karlsruhe, 1981

[Sch80] D.P. Siewiorek, C.G. Bell und A. Newell: Computer Structures: Principles and Examples, McGraw-Hill, 1982

R. Sethi und J.D. Ullman : The Generation of Optimal Code for Arithmetic Expressions, Journal of the ACM, Vol. 17, 4(1970), S. 715-728

[Sie82] J.R. Southard : MacPitts : An Approach to Silicon Compilation, Computer, Vol. 16, 12(1983), S. 74-82

H. Trickey : Flamel : A High-Level Hardware Compiler, IEEE Transactions an CAD, 1987, S. 259-269

[SetU1170] A.C. Parker et al. : MAHR : A Program for Datapath Synthesis, Proc. 23rd Design Automation Conf., 1986, S. 461-466

[Pau87] P.G. Paulin und J.P. Knight : Force-Directed Scheduling in Automatic-Data Path Synthesis, Proc. 24rd Design Automation Conf., 1987

[Wir75] N. Wirth : Algorithmen und Datenstrukturen, Teubner, Stuttgart, 1975

3 Automatische Erzeugung von Testprogrammen

3.1 Einleitung

Mit der wachsenden Integrationsdichte moderner digitaler Schaltungen bei fast gleichbleibender Zahl externer Anschlüsse wird es zunehmend schwieriger, Tests zu erstellen, die in zufriedenstellendem Ausmaß die Schaltung auf einwandfreie Funktion überprüfen. Die Testerzeugung für Entwurfsverifizierung und Fertigungskontrolle wird immer mehr zum Engpaß und entscheidenden Kostenfaktor bei der Entwicklung neuer Produkte. Für die Vereinfachung und Beschleunigung dieser Aufgabe werden dringend neue Werkzeuge gebraucht. Gleichzeitig muß bereits beim Schaltungsentwurf für ausreichende Testbarkeit gesorgt werden. Testerzeugung und Entwurf müssen in gegenseitiger Abhängigkeit einhergehen, um den Anforderungen an Entwurfszeit und Testqualität gerecht werden zu können.

Bisher übliche Verfahren zur Testerzeugung arbeiten auf der Gatterebene und konzentrieren sich auf kombinatorische Schaltungen. Sie scheitern bei sequentiell tiefen Schaltungen und werden bei komplexen kombinatorischen Netzen wegen des enormen Aufwandes an Arbeits- und Rechenzeit wirtschaftlich untragbar [Set85]. Eingebaute Testhilfen ("Scan Path" [Eic77]) und Selbsttest-Einrichtungen ("BILBO" [Köh79]) können Abhilfe schaffen. Eine bestimmte Schaltungstechnik und eine nicht unerhebliche Vergrößerung der Chipfläche müssen jedoch in Kauf genommen werden.

Speziell für Mikroprozessoren sind auch funktionale Testverfahren entwickelt worden. Sie versuchen anhand der Datenflußspezifikation des Benutzerhandbuchs algorithmisch eine Folge von Instruktionen herzuleiten, deren ordnungsgemäße Ausführung das einwandfreie Arbeiten des Prozessors gewährleisten soll [Sus83]. Ein üblicher Machinenbefehlssatz für sequentielle Datenverarbeitung wird vorausgesetzt. Die Implementierung einzelner Befehle und eine evtl. unterliegende Mikroprogrammebene bleiben unberücksichtigt.

Die Testerzeugung für digitale Systeme mit spezieller Struktur, insbesondere mikroprogrammierte und kundenspezifische Prozessoren, gilt nach wie vor als ein besonderes Problem.

3.2 Ein Werkzeug zur Testerstellung

Im Rahmen des hier beschriebenen Projekts ist ein Verfahren entwickelt worden, um für digitale Prozessorschältungen automatisch Selbsttestprogramme zu generieren [Krü86]. Das Verfahren ist in der Komponente MSST des MIMOLA-Entwurfssystems vollständig implementiert. Die wesentlichen Merkmale (für Testerzeugung und -Anwendung) sind:

- Das zu prüfende Prozessorsystem wird in einem speziellen Testlauf ("off line") mit einem automatisch erstellten Selbsttestprogramm getestet. Dieses Programm führt nacheinander "lokale" Testmuster an die Eingänge interner Register-Transfer-Module. Die resultierende ModulAusgabe wird (per Programm) im System mit dem vorausberechneten Sollwert verglichen. Bei Übereinstimmung setzt das Programm seinen normalen Ablauf fort, andernfalls wird zu einer vom Anwender zu definierenden Fehleradresse verzweigt.

- Die "lokalen" Modul-Testmuster werden in einer Bibliotheksdatei bereitgestellt. Sie sollen Fehler im Modulinneren und auf den Anschlußleitungen am Modulausgang erkennbar machen. Für jedes mögliche Fehlverhalten einzelner RT-Module können spezielle Muster in die Bibliothek eingetragen werden. Wenn keine vordefinierte Testmusterbibliothek zur Verfügung steht, so kann auf einfache Standardwerte zurückgegriffen werden.

- Ausgangsbasis für die Erzeugung der Selbsttestprogramme ist ein Register-Transfer-Modell des zu testenden Systems. Für die Erstellung eines solchen Modells genügt die zu jedem Prozessorsystem mitgelieferte Benutzerinformation (Befehlsformate, Operationscodes etc.). Zur Bildung eines realitätstreuen Systemmodells und in der Regel besseren Fehlermodellen [Mic82] sind nähere Kenntnisse über die tatsächliche Struktur des Systems von Vorteil. Details bis herunter zur Gatterebene können einbezogen werden.

  Neben den Datenpfaden werden auch Steuer- und Adreßlogik spezifiziert. Module in diesen Schaltungsteilen, die nicht über Datenpfade erreich

bar sind, können indirekt durch die Wahl spezieller Belegungen für die Steueroder Adreß- Eingänge beim Test der jeweils gesteuerten oder adressierten Module geprüft werden.
Eine Überwachung der Testprogramm-Laufzeit oder des Programmzählerstands genügt, um die von den lokalen Testmustern "aktivierten" Fehler (die zu einer Programmverzweigung führen) zu erkennen und grob in einer Menge von Modulen bzw. Moduloperationen zu lokalisieren. Aufwendige zusätzliche Testhardware ist dafür nicht erforderlich.
Der programmierte Selbsttest ist nicht abhängig von strengen Entwurfsregeln oder in die Hardware eingebauten Testhilfen (Scan Path etc.). Vorhandene Hardware-Unterstützung für das Testen wird, soweit im Modell des zu testenden Systems berücksichtigt, wie alle anderen Hardware-Teile vom Testprogramm in Anspruch genommen und auch selbst getestet.

Teilschaltungen von digitalen Prozessorsystemen (ICs, Platinen) können in ihrer vorgesehenen Systemumgebung geprüft werden. Für das separate Testen einzelner Teile kann jedoch auch eine einfache Testumgebung mit Prozessorstruktur konstruiert und zusammen mit der jeweiligen Teilschaltung als Systemmodell spezifiziert werden. Es werden dann Programme für dieses Testsystem generiert.

Die automatisch erstellten Selbsttestprogramme werden je nach Spezifikation der Prozessorsteuerung direkt im entsprechenden Mikro- oder Maschinencode ausgegeben und sind unmittelbar auf dem zu prüfenden Prozessorsystem ablauffähig. Zur Auswertung und Dokumentation stehen auch abstrahierte Ausgaben auf RT-Niveau in verschiedenen Detaillierungen zur Verfügung.

Durch die Modellierung des zu testenden Systems auf Register-TranferEbene verringert sich die Komplexität des Testerzeugungsprozesses gegenüber den herkömmlichen Methoden auf der Gatterebene ganz erheblich. Nur zur Erstellung der Modul-Testmusterbibliothek muß noch gelegentlich (im Rahmen einzelner RT-Module) auf diese Methoden zurückgegriffen werden.

Kürzere Test-Erzeugungszeiten erlauben einen vielseitigen Einsatz des Verfahrens. Außer zum Test des-fertigen Systems beim Hersteller und beim Anwender kann es bereits während des Entwurfs zur Testbarkeitsanalyse herangezogen werden. Die von einem implementationsunabhängigen RT-Modell abgeleiteten Selbsttestprogramme können ferner zur Simulation einer darunterliegenden Struktur verwandt werden und Somit zur Entwurfsvalidierung dienen.

3.2.1 Automatische Test-Erzeugung

Für einen automatischen Testgenerator erweist sich das MIMOLA-Rechnerentwurfssystem als günstiges Umfeld. Die Ebene der Hardware-Beschreibung (Register-Transfer-Struktur) ist diejenige, welche für die Testerzeugung bei hochkomplexen integrierten Schaltungen am besten geeignet erscheint. Auf dieser Ebene ist eine Partitionierung des Systems (in RT-Module) vorgegeben, die für den Test einerseits die Zahl der zu prüfenden Komponenten überschaubar macht (statt Zehntausende von Gattern kaum hundert RT-Module), andererseits aber die Komplexität der einzelnen Komponenten für eine herkömmliche Testerzeugung in erträglichen Grenzen hält.

In der MIMOLA-Strukturbeschreibung muß neben den Datenpfaden auch die Steuerlogik für die unterste Programmebene im Schaltungsmodell spezifiziert werden. Die erzeugten Selbsttestprogramme sind immer auf dieser, der modellierten Hardware am nähesten gelegenen Programmebene angesiedelt. Dies kann eine Mikro- oder Maschinenprogramm-Ebene sein. Es wird angenommen, daß die Instruktionen die Hardware direkt steuern und nicht durch ein weiteres Mikro- bzw. Nanoprogramm interpretiert werden.

Der Test eines komplexen Systems läßt sich nun zurückführen auf eine Reihe von Tests für die weit weniger komplexen RT-Module und ihre Verbindungen. Zusätzlich kann durch die Überwachung des Systemzustands (Belegung der Speicherzellen im System) geprüft werden, ob bei der Aktivierung einzelner Module Seiteneffekte auftreten, die zu einer fehlerhaften Veränderung des Systemzustands führen.

Es soll angenommen werden, daß die zum Testen der verschiedenen RT-Mo

dule benötigten "lokalen" Testmuster in einer Bibliothek zur Verfügung stehen (vgl. Abs. 3.4). Weiterhin gilt: Im allgemeinen ist es bei hochintegrierten Systemen nicht möglich, über externe Anschlüsse oder spezielle Testpunkte direkt auf die Ein- und Ausgänge der RT-Module zuzugreifen; d.h. es kann kein "in circuit"-Test für diese Module durchgeführt werden.
Es bleibt somit für MSST die Aufgabe zu lösen, einerseits die Testmuster von geeigneten Datenquellen (Speichern) innerhalb oder außerhalb des Systems bis zu den Eingängen des zu testenden Moduls fortzuschalten und andererseits die resultierende Modulausgabe einer internen oder externen Auswertung zuzuführen. Dabei müssen andere RT-Module kombinatorischer und sequentieller Art passiert werden. Es sind Folgen von Eingangsdaten und Steuersignalen (ein Testprogramm) zu generieren, derart daß im System die entsprechenden Pfade "sensibilisiert" und die gewünschten Operationen mit den Testmustern aktiviert werden.

3.2.2 Selbsttestprogramme

Wenn die Auswertung der Modultests im zu prüfenden System selbst vorgenommen wird, spricht man von Selbsttestprogrammen. Die Vorteile der Testauswertung mit externer Apparatur (Testautomaten) schrumpfen in dem Maße, in dem die Zugriffsmöglichkeiten auf schaltungsinterne Meß- und Einspeisungspunkte beschnitten werden. Bei einem vollintegrierten Prozessorsystem muß auch ein Testautomat sich darauf beschränken, den Ablauf eines Testprogrammes zu überwachen. Viele interne Signale sind von außen nur indirekt zu beobachten, teilweise müssen sie erst im System aufbereitet und über mehrere Stationen transferiert werden. Der Schritt zum Selbsttestprogramm ist dann kurz und angesichts des Kostenvorteils naheliegend.

Grundsätzlich ist heute unumstritten, daß mit sorgfältig erstellten Selbsttestprogrammen eine zufriedenstellende Fehlererkennungsrate erreicht werden kann [Bel83, Hun82, Bra84, Gär85]. Dies gilt insbesondere für ein Programm auf Mikrocode-Ebene, das im allgemeinen erweiterte Möglichkeiten zur Steuerung und Beobachtung einzelner RT-Module hat

[Rah72]. Was eine zufriedenstellende Fehlererkennungsrate ist, läßt sich nur empirisch im jeweiligen .Einzelfall ermitteln. Allgemein aussagekräftige Maße oberhalb von Gatterebene und "stuck-at"-Fehlermodell existieren nicht.
Die Lösung des über die Fehlererkennung hinausgehenden Problems., die Ursachen eines Fehlers in einzelnen Modulen zu lokalisieren, wird hier nur als sekundäres Ziel betrachtet. Der entscheidende Nachteil von Selbsttestprogrammen gegenüber anderen Testverfahren war bisher immer nur der erhöhte Aufwand für die besonders auf Mikrocode-Ebene mühsame Arbeit der manuellen Programmerstellung [Bel83].

Zwei Grundforderungen an das hier vorgestellte Verfahren können gemeinsam nur von Selbsttestprogrammen erfüllt werden. Um den Einsatzbereich des Testgenerators nicht einzuschränken, soll es grundsätzlich:

(a) keine Abhängigkeit von nicht im Systemmodell enthaltenen externen Testschaltungen oder evtl. einzusetzenden Testautomaten und

(b) keine Beschränkung auf spezielle, nach festem Schema entworfene oder hardwaremäßig mit Testhilfen (z.B. "scan path") versehene Systeme geben.

Für jedes Prozessorsystem, dessen RT-Struktur identisch oder nur funktional äquivalent modelliert werden kann, soll die Erzeugung von Selbsttestprogrammen im Prinzip möglich sein. Als minimale Ausstattung für die Realisierung eines programmierten Selbsttests muß das zu prüfende System lediglich aufweisen:

- Einen Vergleichsoperator, bzw. die Möglichkeit, einen Vergleich durch Test auf Null und vorangegangene Subtraktion (bzw. Addition des Komplements, "exklusiv Oder"-Operation oder Äquivalenzoperation mit dem negierten Vergleichswert) zu realisieren,

- die Möglichkeit, abhängig vom Vergleichsergebnis den Programmfluß zu ändern (bedingt zu springen) und so einen Fehler nach außen zu signalisieren.

Wohl jedes einsatzfähige Prozessorsystem erfüllt diese Bedingungen. Im übrigen entscheiden die Steuerbarkeit der Hardware und ihre Beobachtbarkeit, gesehen vom internen Vergleichsoperator, über die Fehlererkennungsrate des Selbsttestprogrammes. nie Fehlererkennungsrate ist im Zusammenhang zum gewählten Fehlermodell zu sehen und muß im Einzelfall durch Analyse des generierten Programmes oder durch eine Fehlersimulation ermittelt werden.

Entsprechend der MSS-Philosophie wird auch bei der Testerzeugung keine "vollautomatische" Lösung ohne Eingriffsmöglichkeit angestrebt. Der Mensch, hier der Testingenieur, muß in der Lage sein, seine durch keinen Automaten zu ersetzende Erfahrung in das Testprogramm mit einzubringen. Über die Erstellung des Systemmodells hinaus kann er hier durch die Angabe von Adreßwerten und Testmustern für die internen Module Fehlermodelle, Teststrategie und Testlänge selbst bestimmen und damit die Fehlerüberdeckung direkt beeinflussen.

Für den Einsatz beim Entwurf einer Rechnerstruktur soll das Testerzeugungssystem Hinweise geben auf schlecht testbare Schaltungsteile. Durch die Identifizierung von nicht direkt steuerbaren oder observierbaren Modulen und Verbindungswegen ist es dem Entwerfer möglich, durch entsprechende Maßnahmen (z.B. zusätzliche Leitungen) sofort Abhilfe zu schaffen.

3.3 Algorithmen für die Erzeugung von Selbsttestprogrammen

3.3.1 Symbolische Musterbefehle

Durch die Aufgabenstellung für das zu erzeugende Selbsttestprogramm ist dessen grobe Struktur vorgegeben. Die für die Realisierung benötigten Instruktionen lassen sich grob in zwei Klassen aufteilen:

- **Lade-Instruktionen** für den Test der speichernden Moduln und für die Bereitstellung von Eingangsdaten über sequentielle Pfade,

und für die durch das Vergleichsergebnis gesteuerte Verzweigung im Programmfluß.

Für beide Klassen können die in Abb. 3.1 gezeigten symbolischen Musterbefehle, bzw. Musterbefehls-Sequenzen, auf RT-Ebene definiert werden.

Lade-Instruktionen:

```
(I1)                              (I2)
PARBEGIN                          PARBEGIN
  Register    := Wert;              Speicher[Adresse] := Wert;
  Prog.zähler := Lnext;             Prog.zähler       := Lnext;
END;                              END;
Lnext: (* continue *)             Lnext: (* continue *)
```

Test-Instruktionen:

```
(I3)  Prog.zähler := (IF test.modul(Eingangsbelegung)
                         =  Sollwert
                      THEN Lnext
                      ELSE errorexit);
      Lnext:  (* continue *)
```

Variante für Sprung bei negierter Bedingung:

```
(I4)  Prog.zähler := (IF test.modul(Eingangsbelegung)
                         =  Sollwert
                      THEN Lnext2
                      ELSE Lnext);

      Lnext :  Prog.zähler := errorexit;
      Lnext2:  (* continue *) ...
```

Variante für den Test von Moduln im Programmzähler-Daten-Eingangspfad:

```
(I5)  Prog.zähler := test.modul(Eingangsbelegung);
                     (* Sprung nach Lnext2 *)

      Lnext :  Prog.zähler := errorexit;
      Lnext2:  (* continue *) ...
```

Abb. 3.1: Vollständiger Satz symbolischer Musterbefehle für die automatische Erzeugung von Selbsttestprogrammen

Zusammen mit zwei Varianten für Spezialfälle enthalten diese Muster alle geforderten Aktionen und sind somit ausreichend für die Implementierung des zu erzeugenden Selbsttestprogramms.
Die in Abb. 3.1 verwendeten Label Lnext und Lnext2 kennzeichnen die unmittelbar auf die aktuelle Instruktion folgenden Befehlsadressen.. Ein "PARBEGIN ... END"-Block zeigt an, daß die darin enthaltenen Anweisungen Bestandteil einer Instruktion sind und synchron ("parallel") zueinander ausgeführt werden (MIMOLA-Notation, [Jöh87]). "Prog.zähler" ist der (Mikro-) Programmzähler im System. Er adressiert den Befehlsspeicher. "test.modul" ist ein zu testendes Modul. "Eingangsbelegung" umfaßt Testmuster und Steuercode für eine Modul-Operation. "errorexit" steht für die Adresse, die im Fehlerfall angesprungen wird. Dies kann eine Konstante sein oder für den gegenwärtigen Programmzählerstand stehen. In der Variante (I5) muß die Eingangsbelegung so gewählt sein, daß sich der Ausgangssollwert "Lnext2" ergibt.

Die Aufgabe, Selbsttestprogramme im Mikro- oder Maschinencode eines zu testenden Prozessorsystems zu erzeugen, bedeutet nun, gemäß einer explizit (Testmusterbibliothek) oder implizit (Standardmuster) vorgegebenen Reihenfolge ein symbolisches Befehlsmuster auszuwählen, mit den konkreten Modulnamen und Eingangsbelegungen zu versehen und in den Steuercode des Zielsystems zu übersetzen.

Da die symbolischen Muster für alle Zielsysteme gleich bleiben, das Modell der Zielhardware und der zu erzeugende Steuercode sich mit jedem neuen zu testenden System jedoch ändern, wird ein spezieller "retargierbarer Übersetzer" ("retargetable compiler") für diese Aufgabe benötigt.

3.3.2 Retargierbare Code-Erzeugung

Ein retargierbarer Übersetzer ist ein für verschiedene Zielsysteme bzw. Zielcodes umstellbarer Codegenerator. Man kann bei existierenden retargierbaren Codegeneratoren im wesentlichen zwei verschiedene Ansätze unterscheiden [Gan82]:

- **Interpretative Code-Erzeugung** erstellt zunächst (durch Interpretation des Quellcodes) einen Zwischencode für eine virtuelle Maschine und benötigt dann für jedes Zielsystem eine spezielle Komponente zur Übersetzung in den Zielcode. In dieser Komponente ist neben dem CodeErzeugungsalgorithmus implizit auch eine Beschreibung des Zielsystems enthalten.
- **Code-Erzeugung durch Mustererkennung** trennt Hardware-Beschreibung und Code-Erzeugungsalgorithmus. Quellanweisungen und Zielinstruktionen werden in eine gleichartige, baum- oder listenförmige Struktur gebracht. Mustererkennung wird eingesetzt, um für die zu übersetzenden Anweisungen passende Instruktionen im Zielcode zu finden.

Für die Erzeugung der Selbsttestprogramme scheiden interpretative Ansätze aus. Die Erstellung eines Zwischencodes bietet Vorteile, wenn die Übersetzung aus verschiedenen Quellsprachen unterstützt werden soll. Die symbolischen Muster-Instruktionen für das Selbsttestprogramm können jedoch in jedem gewünschten Format bereits vorgegeben werden. Die eigentliche Code-Erzeugung würde beim interpretativen Ansatz zielsystemabhängig bleiben. Der für die Testerzeugung vorgesehene Einsatzbereich im Rahmen des MIMOLA-Rechnerentwurfssystems MSS2 schließt derartige Lösungen aus. Häufige Entwurfsiterationen und der Einsatz zur Testbarkeitsanalyse verlangen die völlige Unabhängigkeit von Verfahrensablauf und Zielsystem.

Das im MSS2 vorhandene System zur retargierbaren Mikrocode-Erzeugung arbeitet auf der Basis der Mustererkennung. Im Gegensatz zu anderen in der Literatur bekannten Verfahren [Veg82, Mul83], die auch für die Erzeugung von Mikrocode geeignet sind, kommt es ganz ohne handerstellte und zielsystemabhängige Komponenten aus [Mar85]. Dennoch ist weder dieses noch die anderen bekannten Verfahren zur Übersetzung der Selbsttestprogramm-Musterbefehle verwendbar. Die symbolischen Musterbefehle enthalten anders als übliche Quellprogramme auf RT-Ebene mit den zu testenden Moduln und deren Eingangsbelegungen bereits vollständig an die Hardware gebundene Elemente, die in die zu erzeugenden Codesequenzen einzubauen sind.

Für die Erstellung des Selbsttestprogramm-Codes bestehen weitere spezielle Anforderungen, die von den bestehenden Code-Erzeugungssystemen im Allgemeinen nicht berücksichtigt werden:

- Statt immer möglichst kurzer (d.h. laufzeitoptimierter) Codesequenzen mit Instruktionen aus einer meist nur kleinen Untermenge aller möglichen Befehle, ist es für Testzwecke erforderlich, möglichst alle Bereiche der Hardware anzusprechen und alle Steuercodes zu verwenden. Die Programmlaufzeit spielt eine dagegen untergeordnete Rolle.
- Über die bei (Mikrocode-) Compilern üblichen Transformationen und Optimierungen hinaus sollen z.B. Konstanten (Testmuster, Sollwerte) auch über verschachtelte und wertverändernde Pfade (Module ohne direkte Durchschaltefunktion) herbeigeführt werden können.

- Für den Test auf Nebeneffekte und zur Beschleunigung der Code-Erzeugung sollen alle in den Speicherzellen des Zielsystems entstehenden Werte protokolliert werden und in den nachfolgenden Instruktionen für die Konstantenerzeugung zur Verfügung stehen. Beim Test auf Nebeneffekte als Abschluß des Testprogrammes sollen die Werte noch einmal ausgelesen und überprüft werden.

Da neben den bestehenden retargierbaren Codegeneratoren auch keine Testerzeugungsverfahren auf RT-Ebene (oder anderen Abstraktionsebenen) bekannt sind, die Selbsttest-Programmcode für Prozessorsysteme liefern können, wurde für die vorstehenden Anforderungen das im folgenden beschriebene spezielle Verfahren zur Code-Erzeugung für Selbsttestprogramme entwickelt. Es verbindet die bei Codeund Testerzeugung üblichen Methoden und führt sowohl eine Mustererkennung für die symbolischen Befehlsmuster als auch eine Pfadsensibilisierung für bestimmte Signale (Testmuster, Testantworten, Sollwerte) durch.

3.3.3 Mustererkennung und Pfadsensibilisierung

Das zu testende System wird dargestellt als ein gerichteter Graph (Verbindungsgraph). Dieser Graph läßt sich direkt ableiten aus dem mit

MIMOLA beschriebenen Systemmodell. Die Knoten des Graphen sind die spezifizierten Moduln, Festkonstanten, Busse und Instruktionsfelder (Ausgangs-Unterbereiche des Befehlsspeichers bzw. -Registers). Die Knoten enthalten neben dem Modulnamen auch die Information über die zur Verfügung stehenden Operationen und die dafür benötigten Steuercodes. Die Kanten repräsentieren die Verbindungen zwischen den Modulen. Ihre Richtung folgt dem für die jeweilige Verbindung vorgesehenen Datenfluß.

Das zu erzeugende Selbsttestprogramm ist eine Folge von Steuerbefehlen. Jeder Steuerbefehl (wie auch die symbolischen Muster in Abb. 3.1) wird repräsentiert durch eine Liste von RT-Anweisungen (Statements), die innerhalb eines Befehlszyklus synchron auszuführen sind. Die RT-Anweisungen werden (zwecks Mustervergleich) wie folgt modelliert:

Eine Register-Transfer-Anweisung (RT-Statement) ist ein gerichteter Baum (im folgenden auch **Datenflußbaum** oder kurz **RT-Baum** genannt). Die Wurzel des Baumes ist die Datensenke, d.h. ein zu ladendes speicherndes Modul. Die Blätter des Baumes sind die Datenquellen (Instruktionsfelder, Festkonstanten, speichernde Moduln ohne Adreß- oder Steuereingang). Für jede in der Anweisung spezifizierte Operation existiert ein innerer Knoten. Die Kanten geben die Richtung des Datenflusses an.

Der in Abb. 3.2 als Beispiel gezeigte Datenflußbaum für die symbolische Testinstruktion (I3) enthält in seinen Blättern symbolische Werte, die im Einzelfall durch die konkreten Konstanten ersetzt werden. Der Knoten "Eingangsbelegung" wird dann aufgeteilt in je einen Knoten für jeden Subport des zu prüfenden Moduls ("test.modul"). Für die bedingte Zuweisung mittels "IF ...THEN...ELSE"-Anweisung steht im Baum ein Knoten mit der entsprechenden MIMOLA-Standardfunktion "SELECT2". Sie wird realisiert durch einen Multiplexer, der vom Vergleicher-Knoten ("=") gesteuert wird.

Für die Code-Erzeugung durch Mustererkennung werden nun i.a. Instruktionen des Zielsystems in der gleichen Darstellungsform zum Vergleich angeboten [Gan82]. Für diese Instruktionen ist der Steuercode bereits bekannt und nur die Direktdatenund Adreßfelder müssen noch mit den aktu

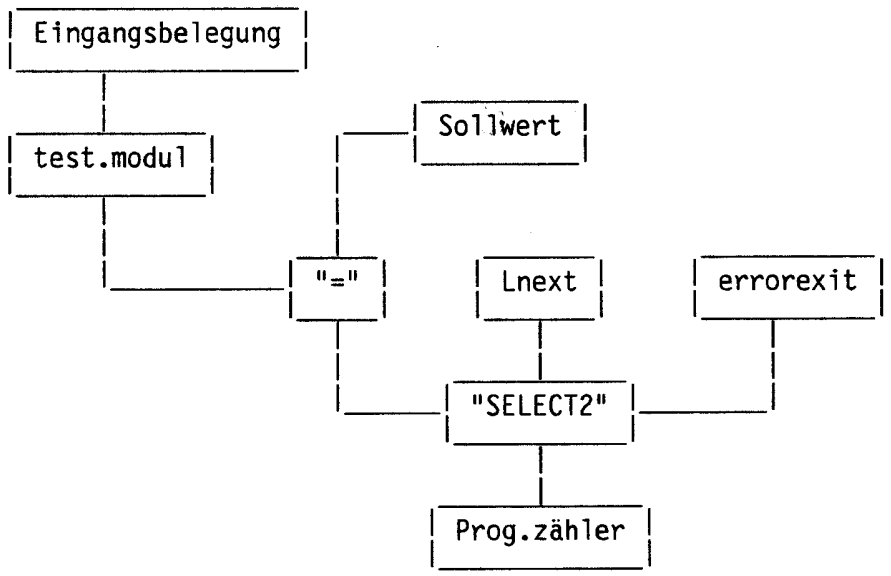


Abb. 3.2: Die RT-Anweisung der symbolischen Test-Instruktion als Baum

ellen Werten gefüllt werden. Insbesondere bei Systemen mit großer Steuerwortbreite (horizontale Mikrocodierung) führt dieser Ansatz jedoch wegen der großen Zahl verschiedener Instruktionen zu Problemen. Eine explizite Ansteuerung bestimmter Moduln für Testzwecke (die u.U. eine sonst unsinnige Instruktion ergibt) wäre kaum möglich.

Code-Erzeugung durch Mustererkennung bedeutet hier, für alle RT-Bäume einer zu übersetzenden Instruktion Teilbäume des Hardware-Verbindungsgraphen mit gleicher Struktur und gleichen Operations-Knoten (Modul-Knoten mit passender Operation) zu finden. Insbesondere müssen auch die explizit angesprochenen Moduln (Programmzähler, getestetes Modul) darin enthalten sein. Die Teilbäume für verschiedene Anweisungen dürfen sich nur in identischen Zweigen (gemeinsame Teilausdrücke der Statements) überlappen.

Die Generierung der Steuercodes und der Konstanten für einen im Verbindungsgraphen gefundenen Datenflußbaum entspricht der von der Testerzeugung auf Gatterebene bekannten **Pfadsensibilisierung.** Steuercodes und Konstanten in einer Instruktion sind jedoch i.a. nicht voneinander unab

hängig. Instruktionsfelder für Daten und Steuercode können überlappen. Zudem ist es vom Wert der Konstanten und dem aktuellen Inhalt der speichernden Moduln abhängig, welcher Pfad (bzw. Teilbaum) im Verbindungsgraphen am besten auf das zu übersetzende Muster paßt. Mustervergleich und Pfadsensibilisierung müssen daher simultan und, um alle Moduloperationen eines Systems testen zu können, für jede Belegung der Konstanten erneut ablaufen.
Für die Suche im Verbindungsgraphen werden die jeweils benötigten Steuercodes als zusätzliche Konstanten an die Modulknoten des Datenflußbaums gehängt. Die Belegungen aller Instruktionsfelder mit Steuercode, Adressen oder Daten ergeben zusammen das aktuelle Instruktions-Codewort.
Einmalige Erzeugung von Code für symbolische Instruktionen und dessen mehrfache Verwendung durch Einsetzen der Konstanten könnte zur Vereinfachung des Verfahrens ohne Einschränkung der Anwendbarkeit nur für die Lade-Instruktionen eingesetzt werden. Mit den für MSST verwendeten Algorithmen sind diese einfachen Instruktionen ohne Pfadverzweigung jedoch nicht mehr wesentlich für die Gesamtlaufzeit.

3.3.4 Transformationsregeln für Datenflußbäume

Im allgemeinen ist es nicht möglich, die Datenflußbäume einer zu übersetzenden Test- oder Lade-Instruktion vollständig und unverändert im Verbindungsgraphen des Zielsystems wiederzufinden. Es müssen daher Transformationen vorgenommen werden. Für die symbolischen Musterbefehle gelten die folgenden Regeln, die auch mehrfach und rekursiv angewandt werden dürfen:

(T1) Zwischen allen Knoten der Datenflußbäume können Knoten eingefügt werden, die eine identische Abbildung der Eingangsdaten auf den Ausgang vornehmen (z.B. Multiplexer, Busse, Addition der Konstante Null, etc.).

(T2) Knoten mit invertierbaren Operationen (Ausgabefunktionen) dürfen eingefügt werden:

- Zwischen dem Knoten "test.modul" und dem Vergleicherknoten (Abb. 3.2) in den Instruktionen (I3) und (14) aus Abb. 3.1 . Der Sollwert am Vergleicher ist entsprechend zu ändern. Die Testaussage bleibt unberührt
- Zwischen Vergleicherknoten und "SELECT2" in (I3) und (I4). Falls die eingefügten Operationen eine Negation des Vergleichsergebnisses verursachen, sind "THEN"- und "ELSE"-Zweig am "SELECT2"-Knoten zu vertauschen.
- Zwischen den Knoten "Prog.zähler" und "test.modul" in der Variante (I5). Die "Eingangsbelegung" in den Blattknoten ist derart zu wählen, daß der Sprung nach Lnext2 erhalten bleibt.

(T3) Alle Konstanten-Knoten (Blätter) können aufgespalten werden in Knoten für Unterbereiche des von ihnen repräsentierten Bitbereichs. Die Werte der neuen Konstanten-Knoten müssen konkateniert den Wert der alten Konstanten ergeben.

(T4) Alle Konstanten-Knoten (Blätter) können durch Operationsknoten ersetzt werden. Es sind alle Operationen zulässig, für die die aktuelle Konstante im Wertebereich der Ausgabefunktion liegt. An die Operanden-Eingänge müssen neue Konstanten gelegt werden, derart daß die ursprüngliche Konstante am Ausgang als Operationsergebnis erscheint.

(T5) Ein Vergleicherknoten ("=") kann durch eine Prüfung auf Ungleichheit ("<>")ersetzt werden . "THEN"- und "ELSE"-Zweig am "SELECT2"-Knoten sind dann zu vertauschen.

(T6) Der Vergleicherknoten ("=") kann ersetzt werden durch einen Test auf Null ("=0", bzw. "<>0"). Ein Knoten für die Subtraktion des Sollwertes vom Ausgangswert des geprüften Moduls ist einzufügen. An Stelle der Subtraktion sind als Varianten der Regel (T6) auch folgende MIMOLA-Standardoperationen möglich: Addition des Zweierkomplements vom Sollwert, exklusiv Oder-Operation mit dem Sollwert, Equivalenz-Operation mit dem negierten Sollwert. Bei Test

mit "<>0" sind "THEN"- und "ELSE"-Zweig am "SELECT2"-Knoten zu vertauschen.

(T7) Zwischen allen Knoten der Datenflußbäume können speichernde ModulKnoten eingefügt werden. Der Baum wird dadurch aufgetrennt und der Speicher-Knoten wird Wurzel des abgetrennten Teilbaumes. Dieser wird in eine vor der aktuellen Instruktion zu plazierende spezielle Lade-Instruktion übertragen. An seine Stelle tritt eine "LeseOperation" für das eingefügte speichernde Modul. Es entsteht ein **sequentieller Pfad.** Die Adresse der aktuellen Instruktion im Befehlsspeicher erhöht sich um Eins. Die Konstanten im Eingangsbereich des Programmzählers müssen entsprechend geändert werden.

Der vorstehende Satz von Transformationsregeln ist erweiterbar. Er ist vollständig für alle Systeme, deren Verbindungsgraph durch Anwendung der Regeln auf die RT-Bäume der symbolischen Musterbefehle völlig überdeckt werden kann. Vollständigkeit bedeutet: Wenn das Zielsystem in der Lage ist, einen Musterbefehl auszuführen, dann sind die Regeln hinreichend, um die mit den aktuellen Konstanten belegten RT-Bäume an den Verbindungsgraphen des Zielsystems anzupassen und auf diese Weise den Steuercode zu generieren.

Es wäre daher bereits am Verbindungsgraphen erkennbar, ob und wie die gegebenen Regeln für ein bestimmtes System erweitert werden müssen. Für keins der bisher modellierten Systeme war eine Erweiterung notwendig. Ein Anspruch auf Vollständigkeit für alle jetzt und in Zukunft existierenden Systeme ist aber nicht möglich und wäre auch nicht beweisbar.

3.4 Testmusterbibliotheken

Neben der Hardware-Spezifikation benötigt die in MSST implementierte automatische Testerzeugung für jede spezifizierte Operation der im Systemmodell enthaltenen Module spezielle Bitmuster als Eingangsbelegungen für die systeminternen Moduleingänge. Diese "lokalen" Testmuster sollen Fehler im Modulinneren und auf den Anschlüssen am Modulausgang erkennbar machen, d.h. zu einer vom Sollwert

Das hier beschriebene Testverfahren auf RT-Ebene ist nicht beschränkt auf die üblichen restriktiven Fehlermodelle (z.B. Haftfehler). Für die Modellierung von Fehlern im Inneren der RT-Module steht dem Anwender ein allgemeiner Rahmen (Abweichungen vom spezifizierten Verhalten) zur Verfügung, den er je nach Schaltungstyp und Erfahrung mit bisher beobachteten Fehlern einschränken und so ein angemessenes Fehlermodell selbst bestimmen kann. Er trägt damit jedoch auch einen Großteil der Verantwortung für einen erfolgreichen, d.h. die tatsächlich auftretenden Fehler erkennenden Test.
Die für die Testerzeugung bereitzustellenden Eingangsbelegungen entsprechen dem "primitive cube of a logic fault" beim D-Algorithmus [Rot66]. Genau wie dort auf einer tieferen Ebene für alle im Fehlermodell enthaltenen Abweichungen vom Sollverhalten ein "fehlererkennendes" Bitmuster spezifiziert wird, so kann dies auch hier für die RT-Module geschehen. Der Unterschied besteht in der möglicherweise viel komplexeren Modulfunktion und darin, daß die Eingangswerte nun nicht mehr nur aus Einzelbits, sondern aus unterschiedlich langen Bitvektoren bestehen.

Der Versuch, die lokalen Testmuster für modulinterne Fehler bei unbekanntem Modulaufbau nur anhand der ausgeführten Operation automatisch zu generieren, läuft immer auf das Implizieren einer bestimmten Struktur hinaus und könnte zwangsläufig nur bei einer eingeschränkten Menge von Schaltungen zu guten Ergebnissen führen [Mar84]. Ein erfahrener Testingenieur ist flexibler und wird in der Regel bessere Lösungen finden. MSST bietet ein Werkzeug zur Automatisierung der bei komplexen Schaltungen nicht mehr überschaubaren Arbeiten für die Erstellung von Testprogrammen: Signalfortschaltung, Teststeuerung und Testcodeerzeugung auf Mikroprogrammebene.

Die für den Test erforderlichen Eingangsbelegungen werden dem Testerzeugungssystem in einer "Testmusterbibliothek", getrennt nach Modultyp und gegebenenfalls Speichereingangsport, Inkarnation ("part") und Moduloperation, übergeben (Abb. 3.3). Auch die Reihenfolge der einzelnen Modultests kann hier bestimmt werden. Bei Speichern umfaßt die Eingangsbelegung neben den Dateneingängen auch die Adreßeingänge. Es lassen sich alle bekannten funktionalen Speichertestprozeduren spezifizieren, die

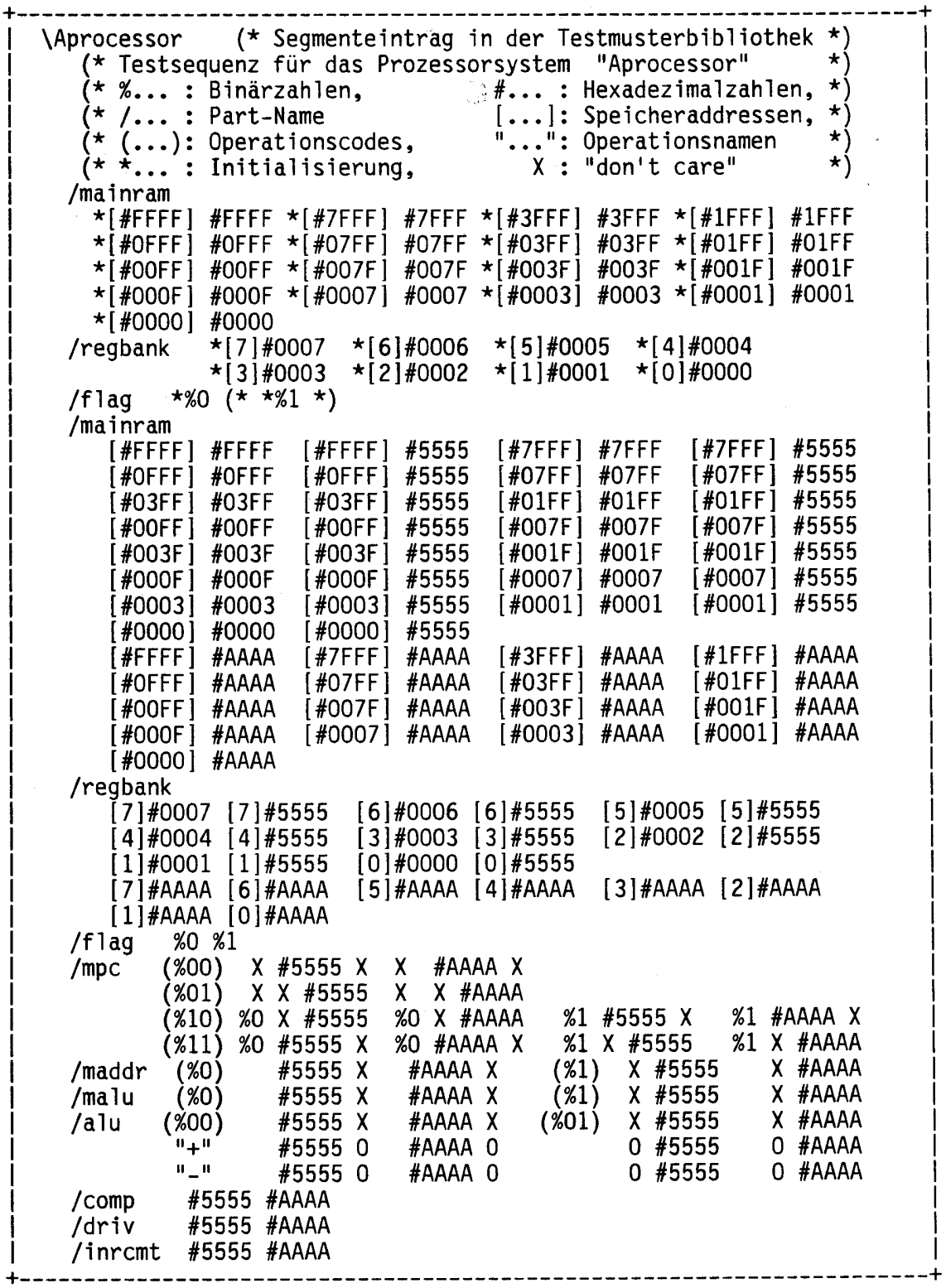
dann in ein Selbsttestprogramm umgesetzt werden. Für kombinatorische RT-Module können in der Bibliothek entweder alle möglichen Eingabewerte (bei geringer Eingangswortbreite) oder zufällige oder deterministische Testmuster angegeben werden. Deterministische Testmuster können für RT-Module mit bekannter Gatterstruktur mit den bewährten algorithmischen Methoden hergeleitet werden. Die Komplexität eines einzelnen Moduls ist mit diesen Verfahren im allgemeinen noch gut handhabbar.
Enthält die Bibliothek für eine Moduloperation keinen Eintrag, so kommen einfache Standard-Testmuster (alternierende Folgen "01...01" bzw. "10.. 10") zum Einsatz. Der Test wird derart durchgeführt, daß jedes für die Operation relevante Eingangsbit des Moduls einmal auf "O" und einmal auf "1" gelegt wird. Dadurch ist sichergestellt, daß auch ohne Testmusterbibliothek eine minimale Funktionsprüfung und ein Test auf Haftfehler ("stuck-at's") und Kurzschlüsse zwischen logisch und physisch benachbarten Bitleitungen erfolgen kann. Für fast alle üblichen arithmetischen und logischen Funktionen erfüllen die Standardmuster die Bedingung zum Erkennen dieser Fehler. Die in Abb. 3.3 gezeigte Belegung der Testmusterbibliothek entspricht in Reihenfolge und Musterauswahl dem Standardtest für einen einfachen Beispielprozessor [Krü87].

Selbsttestprogramme können für alle diejenigen Operationen generiert werden, für die sich das Sollresultat ermitteln läßt, d.h. eine der folgenden Bedingungen muß gelten:

(a) die zu testende Operation ist im Satz der MSS-Standard-Operationen enthalten,

(b) neben den Testmustern ist auch das Sollresultat in der Testmusterbibliothek spezifiziert,

(c) die zu testende Operation ist in der Hardware-Spezifikation prozedural durch MSS-Standardoperationen beschrieben (MSS2 Vers. 4.0).



```
\Aprocessor    (* Segmenteintrag in der Testmusterbibliothek *)
   (* Testsequenz für das Prozessorsystem  "Aprocessor"      *)
   (* %... : Binärzahlen,          #... : Hexadezimalzahlen, *)
   (* /... : Part-Name             [...]: Speicheraddressen, *)
   (* (...): Operationscodes,      "...": Operationsnamen    *)
   (* *... : Initialisierung,        X : "don't care"        *)
  /mainram
    *[#FFFF] #FFFF *[#7FFF] #7FFF *[#3FFF] #3FFF *[#1FFF] #1FFF
    *[#0FFF] #0FFF *[#07FF] #07FF *[#03FF] #03FF *[#01FF] #01FF
    *[#00FF] #00FF *[#007F] #007F *[#003F] #003F *[#001F] #001F
    *[#000F] #000F *[#0007] #0007 *[#0003] #0003 *[#0001] #0001
    *[#0000] #0000
  /regbank   *[7]#0007  *[6]#0006  *[5]#0005  *[4]#0004
             *[3]#0003  *[2]#0002  *[1]#0001  *[0]#0000
  /flag   *%0 (* *%1 *)
  /mainram
     [#FFFF] #FFFF  [#FFFF] #5555  [#7FFF] #7FFF  [#7FFF] #5555
     [#0FFF] #0FFF  [#0FFF] #5555  [#07FF] #07FF  [#07FF] #5555
     [#03FF] #03FF  [#03FF] #5555  [#01FF] #01FF  [#01FF] #5555
     [#00FF] #00FF  [#00FF] #5555  [#007F] #007F  [#007F] #5555
     [#003F] #003F  [#003F] #5555  [#001F] #001F  [#001F] #5555
     [#000F] #000F  [#000F] #5555  [#0007] #0007  [#0007] #5555
     [#0003] #0003  [#0003] #5555  [#0001] #0001  [#0001] #5555
     [#0000] #0000  [#0000] #5555
     [#FFFF] #AAAA  [#7FFF] #AAAA  [#3FFF] #AAAA  [#1FFF] #AAAA
     [#0FFF] #AAAA  [#07FF] #AAAA  [#03FF] #AAAA  [#01FF] #AAAA
     [#00FF] #AAAA  [#007F] #AAAA  [#003F] #AAAA  [#001F] #AAAA
     [#000F] #AAAA  [#0007] #AAAA  [#0003] #AAAA  [#0001] #AAAA
     [#0000] #AAAA
  /regbank
     [7]#0007 [7]#5555  [6]#0006 [6]#5555  [5]#0005 [5]#5555
     [4]#0004 [4]#5555  [3]#0003 [3]#5555  [2]#0002 [2]#5555
     [1]#0001 [1]#5555  [0]#0000 [0]#5555
     [7]#AAAA [6]#AAAA  [5]#AAAA [4]#AAAA  [3]#AAAA [2]#AAAA
     [1]#AAAA [0]#AAAA
  /flag   %0 %1
  /mpc   (%00)  X #5555 X  X  #AAAA X
         (%01)  X X #5555  X  X #AAAA
         (%10) %0 X #5555  %0 X #AAAA   %1 #5555 X   %1 #AAAA X
         (%11) %0 #5555 X  %0 #AAAA X   %1 X #5555   %1 X #AAAA
  /maddr  (%0)    #5555 X   #AAAA X    (%1)  X #5555    X #AAAA
  /malu   (%0)    #5555 X   #AAAA X    (%1)  X #5555    X #AAAA
  /alu   (%00)    #5555 X   #AAAA X   (%01)  X #5555    X #AAAA
          "+"     #5555 0   #AAAA 0          0 #5555    0 #AAAA
          "-"     #5555 0   #AAAA 0          0 #5555    0 #AAAA
  /comp    #5555 #AAAA
  /driv    #5555 #AAAA
  /inrcmt  #5555 #AAAA
```

Abb. 3.3: Beispiel einer MSS2-Testmusterbibliothek

3.5 Fehlererkennung

Das Selbsttestprogramm findet Fehler durch Vergleich der zu prüfenden Signale mit vorausberechneten Sollwerten. Bei Erkennung des ersten Fehlers ist das primäre Ziel des hier verfolgten Tests erreicht, es besteht Gewißheit, daß das System nicht korrekt arbeitet, und das Programm. kann abgebrochen werden ("go-no-go"-Test). Durch den Abbruch des normalen Programmflußes wird auch das Auftreten des Fehlers nach außen signalisiert. Eine minimale Umgebung für das Testprogramm ist dazu vom Anwender zu schaffen:

- Die automatisch generierte symbolische "STOP"-Instruktion am Ende des Programmes ist durch eine Befehlsfolge zu ersetzen, die den fehlerfreien Testablauf extern sichtbar macht.

- Zur Verzweigung im Fehlerfall ist ein Fehlerlabel ("errorexit") zu definieren. An der entsprechenden Stelle im Programmspeicher ist eine Befehlsfolge abzulegen, die eine Meldung nach außen bewirkt. Statt einem festen Fehlerlabel kann auch der jeweils aktuelle Programmzählerstand als "errorexit" gewählt werden, was im Fehlerfall zu einer Endlosschleife direkt in der fehlererkennenden Befehlsfolge führt.

Die Verzweigungslogik (modelliert als bedingungsgesteuerter Multiplexer vor dem Programmzählereingang) wird in allen ihren Operationen durch die Einstellung von konstanten Bedingungen "TRUE" oder "FALSE" geprüft.

Mit einem Selbsttestprogramm können vor allem Datenpfade (Signalpfade mit Vergleichsmöglichkeit) direkt geprüft werden. Für die Module auf diesen Pfaden ergibt sich folgende Aussage:

> Jeder "nichtredundante" Fehler, der eine Veränderung einer Ausgabeoder Zustandsübergangsfunktion eines Moduls bewirkt, kann erkannt werden.

Ein Fehler wird in diesem Zusammenhang als "nichtredundant" bezeichnet, wenn ein den Fehler aktivierendes lokales Testmuster (das zu einer Ab

weichung vom Ausgangs-Sollwert führt) an den Eingängen des betroffenen Moduls per Programm einstellbar ist. Wäre dies nicht der Fall, so könnte der Fehler auch im Normalbetrieb niemals auftreten.
Module, deren Ausgangssignale nicht direkt oder über sequentielle Pfade einem Vergleich zugeführt werden können, sind nicht unmittelbar testbar und werden als Problemstellen vom System (MSST) erkannt und gemeldet. Für Bausteine im Adreßpfad eines Speichers ist über einen vollständigen Test der Zellenauswahl Abhilfe möglich. Module in reinen Steuerzweigen, etwa Dekoder oder auch Instruktionsregister sind nur über die von ihnen gesteuerten Operationen indirekt zu prüfen. Es empfiehlt sich daher, über die Testmusterbibliothek beim Test der so gesteuerten Module Bitmuster zu verwenden, welche die für das Modul spezifizierten Operationen eindeutig voneinander unterscheiden. Bei einwandfreiem Verhalten aller Operationen erübrigt sich ein weitergehender Test der Steuerpfade.

Eine Sonderbehandlung ist möglich für Module, die nicht per Vergleich geprüft werden können, aber im Daten-Eingangspfad des (Mikro-) Programmzählers liegen. Anstelle der spezifizierten Testmuster werden für diese Module Eingangsbelegungen verwendet, die zusammen mit den automatisch generierten Steuersignalen einen unbedingten Programmsprung zum übernächsten Befehl bewirken. Als nächster (zu überspringender) Befehl wird ein unbedingter Sprung zum Fehlerlabel generiert.

Wenn die indirekten Testmöglichkeiten, z.B. wegen ungenügender Beobachtbarkeit der gesteuerten oder adressierten Module, nach Analyse der generierten Testprogramme als nicht ausreichend erkannt werden, so ist bei einem Schaltungsentwurf das Einbringen von zusätzlichen Verbindungen zu erwägen, um einen direkten Vergleichstest zu ermöglichen.

Für die indirekten Prüfungen läßt sich keine allgemeingültige Qualitätsaussage machen. Die Verantwortung für indirekte Tests liegt beim Anwender des Testprogrammgenerators MSST. Die indirekt zu prüfenden Module werden dem Anwender gemeldet. Eine automatische Erzeugung indirekter Tests findet nur für Module im Eingangsbereich des Programmzählers statt. Im Einzelfall kann eine Fehlersimulation mit dem generierten Programm für modellierte "stuck-at"- Fehler auf der Gatterebene statt fin-

den, wenn das Gattermodell und ein entsprechender Fehlersimulator zur Verfügung stehen.
Das generierte Selbsttestprogramm seht in verschiedenen Formaten zur Verfügung. Als Dokumentation wird ein Programm-Listing in einer "high level" Prefix-MIMOLA Notation ausgegeben. Bei jeder Instruktion .steht als Kommentar auch der Mikro- bzw. Maschinencode in hexadezimaler und (bei Wortbreite <= 32) auch in binärer Form. Warnungen zeigen an, für welche Module oder Operationen keine Testsequenz erzeugt werden konnte und was der Grund dafür war. Die erzeugten Programme liegen außerdem in interner Zwischensprache vor und können in dieser Form vom Simulator im MSS als Stimuli verarbeitet werden.

3.6 Fehlerlokalisierung

Die genaue Bestimmung des Fehlerortes und damit der Fehlerursache ist von Bedeutung, wenn eine Reparatur vorgenommen werden soll. Bei integrierten Schaltungen ist dies jedoch nur möglich, sofern sich auf dem Chip redundante Schaltungsteile befinden, die im Bedarfsfall hinzugeschaltet werden können (üblich bei großen Speicherbausteinen). Für den Hersteller ist es wichtig, häufig auftretende Fehler zu identifizieren, um einen Hinweis auf kritische Stellen im Entwurf oder im Herstellungsprozess zu erhalten.

Das hier beschriebene Testverfahren zielt in erster Linie auf einen schnellen "go-no-go"-Test, der beim Anwender wie beim Hersteller von Prozessorsystemen gleichermaßen einfach ohne Zusatzhardware einsetzbar ist. Ferner sollen die beim Testprogramm-Erzeugungsvorgang gewonnenen Erkenntnisse über die Testbarkeit einzelner Schaltungsteile zur Verbesserung beim Entwurf dienen. Die Lokalisierung von Fehlern ist ein sekundäres Ziel von MSST. Die Erarbeitung einer effizienten Diagnosestrategie für automatisch erstellte Selbsttestprogramme wäre eine über den Rahmen des Projekts hinausgehende Erweiterung.

Die Tests für die einzelnen Moduloperationen werden nacheinander über kurze und direkte Pfade, möglichst ohne sonst unbeteiligte Module zu

passieren, durchgeführt. Bei Modulen, die dennoch passiert werden müssen, kommen, wenn die Codierung es zuläßt, immer dieselben Durchschaltefunktionen zum Einsatz. Dadurch wird erreicht, daß ein Großteil der Moduloperationen zum erstenmal durch"die dafür vorgesehene Prüfsequenz angesprochen werden. Beim Auftreten eines Fehlers ist die erkennende Instruktionsfolge durch Programmzählerstands- oder Laufzeitüberwachung zu ermitteln. Fehler sind dann insbesondere in der oder den erstmalig angesprochenen Moduloperationen zu suchen.

Eine von MSST ausgegebene Tabelle aller Moduloperationen zeigt die Adressen der Programmschritte, in denen die Operationen erstmals, bzw. zum letzten Mal auftreten und wie oft sie insgesamt aktiviert werden. Optional erscheint im ausgegebenen Programm zu jeder Instruktion eine Liste aller aktivierten Moduloperationen. Die erstmalig angesprochenen sind darin gesondert markiert.

3.7 Anwendungen

Eine komplexe Aufgabenstellung, wie die hier beschriebene automatische Erzeugung von Selbsttestprogrammen ist nicht von vornherein in allen Einzelheiten überschaubar. Die Entwicklung der Algorithmen und die Erstellung von MSST gingen deshalb einher mit unzähligen Probeläufen für die verschiedensten Beispielschaltungen. Fehler und Unzulänglichkeiten wurden erkannt und beseitigt. Neben einfachen Phantasie-Prozessoren gibt es jedoch auch Anwendungen für real existierende und zum Teil auch kommerziell eingesetzte Systeme. Die wichtigsten davon sollen kurz vorgestellt werden.

- Ein Telefonprozessor

Als erstes reales Anwendungsbeispiel diente ein Telefonprozessor, projektiert von der Firma Siemens (München) als Ein-Chip CMOS Baustein. Der Aufbau ähnelt dem herkömmlicher einfacher Mikroprozessoren. Die unterste Programmebene ist das

Telefonprozessor (Standardmuster für alle Moduln) besteht aus ca. 1200 Maschinenbefehlen und wird von MSST in etwa 800 CPU-Sekunden auf einem Siemens 7.760 Rechner erzeugt.

- Ein Universalrechner mit innovativem Architekturkonzept

Ein größeres Anwendungsbeispiel ist ein nach den Prinzipien des MIMOLAHardware-Entwurfs in Kiel entwickelter und aufgebauter Rechner, genannt SAMP, für "selftimed, asynchroneous, microprogrammed, parallel" [Now86].
Mit MSST wurden Testprogramme für verschiedene Ausbaustufen des SAMP generiert. Zum ersten Mal wurden die Programme auch real eingesetzt und auf der Hardware des SAMP zum Laufen gebracht. Mit Standard-Selbsttestprogrammen (ohne externe Testmustereingabe) konnten eine Reihe von Verdrahtungsfehlern und einige IC-Ausfälle erkannt und auf die entsprechende RT-Komponente zurückgeführt werden. Andere Fehler, die für die Programme nicht erkennbar wären, sind bisher nicht aufgetreten. Die Erstellung von Testprogrammen in "Handarbeit" wäre für den SAMP nahezu unmöglich gewesen. Bei einem 142 Bit breiten Mikroprogrammwort hätten sich höchstwahrscheinlich weit mehr Fehler in den Programmen als in der Hardware gefunden. MSST konnte ca. 3000 fehlerfreie Mikroinstruktionen in weniger als 20 Minuten CPU-Zeit generieren.

- Ein System für kaufmännische Anwendungen

Der bisher umfangreichste Einsatz für komplexe und kommerziell genutzte Systeme wird in Kooperation mit der Firma Nixdorf (Paderborn) vorgenommen. Unter anderem wurde ein RT-Modell für die CPU des Bankenrechners NR22 erstellt. Dieser Rechner wird durch fest in PROMs codierte vertikale Mikroinstruktionen gesteuert. Steuerwerk und Operationseinheit arbeiten in einer Pipeline zeitlich versetzt. Unter ausschließlicher Verwendung von Standardmustern entstanden verschiedene Selbsttestprogramme mit 2000 bis 3000 Mikroinstruktionen. Die Laufzeiten für MSST lagen dabei zwischen 40 und 60 Minuten CPU-Zeit (Siemens 7.760). Die erzeugten Codesequenzen haben sich inzwischen auf der Original-Hardware als ab

lauffähige Testprogramme erwiesen. Ein Einsatz im Prüffeld wird vorbereitet.
- Ein Disk-Controller

Für einen mikroprogrammierten Disk-Controller mit einem speziellen Sequencer in Gate-Array Realisierung wurde ein MIMOLA-Modell innerhalb von zwei Tagen erstellt. Am dritten Tag standen ablauffähige Selbsttestprogramme zur Verfügung. Diese Arbeit wurde von Mitarbeitern der Firma Nixdorf nach viertägiger Einführung in das MIMOLA-System (insbesondere MSST) geleistet. Die Programme enthielten ca. 900 Mikroinstruktionen und wurden auf einer VAX unter VMS in rund 100 CPU-Sekunden generiert.

3.8 Zusammenfassung

Es sind ein Konzept und das dazugehörige Werkzeug entwickelt worden, um auf der Register-Transfer-Ebene in kurzer Zeit Hardware-Tests zu generieren, die ohne aufwendige externe Testautomaten auskommen und auch nicht auf eingebaute Testhilfen angewiesen sind. Die für einzelne Komponenten zur Verfügung stehenden Testmuster können in einer Bibliothek abgelegt werden. Für verschiedene Systeme, in denen diese Komponenten auftreten, wird dann automatisch der Steuercode für ein Selbsttestprogramm generiert, das den lokalen Einzeltest ausführt. Umfangreiche Anwendungen haben den Wert des Verfahrens in der Praxis bestätigt.

3.9 Literatur

[Be1831 C.Bellon et al.: "Automatic Generation of Microprocessor Test Programs", Test Technology Newsletter Vol.4, No.3, Jan.1983, S. 3-9

[Bra84] D.Brahme, J.A.Abraham: "Functional Testing of Microprocessors", IEEE Transactions an Computers, Vol.C-33, No.6, Jun. 1984, S. 475-485

[Eic77] E.B.Eichelberger, T.W.Williams: "A Logic Design Structure for LSI Testability", 14th Design Automation Conference, Jun.1977, S. 462-468

[Gan82] M.Ganapathi, C.N.Fischer, J.L.Hennessy: "Retargetable Compiler Code Generation", ACM Computing Surveys, Vol. 14, No. 4, Dec. 1982, S. 573-592

[Gär85] A.Gärtner: "Optimierte Selbsttestprogramme für Mikroprozessoren", Verlag TÜV Rheinland, Köln 1985

[Hun82] A.Hunger: "Neues Verfahren zum Selbsttest von Mikroprozessoren", Verlag TÜV Rheinland, Köln 1982

[Jöh87] R.Jöhnk, P.Marwedel: "MIMOLA Language Reference Manual, Revision 2", Institutsbericht (in Vorbereitung), Institut für Informatik und Praktische Mathematik, Universität Kiel 1987

[Köh79] B.Könemann, J.Mucha, G.Zwiehoff: "Built - In Logic Block Observation Techniques", IEEE 1979 International Test Conference, S. 37-41

[Krü86] G.Krüger: "Automatic Generation of Self-Test Programs - A New Feature of the MIMOLA Design System", 23rd Design Automation Conference, 1986

[Krü87] G.Krüger: "Ein Verfahren zur automatischen Erzeugung von Selbsttestprogrammen für digitale Prozessorsysteme", Dissertation, Universität Kiel 1987

M.Marhöfer: "Modellierung digitaler Schaltungen für Testanwendungen", Interner Bericht Nr. 6, Juni 1984, Institut für Informatik IV, Univ. Karlsruhe

P.Marwedel: "Ein Software-System zur Synthese von Rechnerstrukturen und Erzeugung von Mikrocode", Habilitationsschrift, Institut für Informatik und Praktische Mathematik, Universität Kiel 1985

[Mic82] A.Miczo: "Fault Modelling for Functional Primitives", IEEE 1982 International Test Conference Proceedings, S. 43-49

[Mul83] R.A.Mueller, J.Varghese: '"Flow Graph Machine Models in Microcode Synthesis", length International Microprogramming Workshop 16), 1983, S. 159-167

[Now86] L.Nowak: "SAMP : Entwurf und Realisierung eines neuartigen Rechnerkonzepts", Diss., Institut für Informatik und Prakt. Math., Universität Kiel, 1986

[Rah721 C.V.Ramamoorthy, L.C.Chang: "System Modeling and Testing Procedures For Microdiagnostics", IEEE Transactions an Computers Vol. C-21, No. 11, Nov. 1972, S. 1169-1183

[Rot661 J.P.Roth: "Diagnosis of Automata Failures: A Calculus and a Method". IBM Journal of Research and Development, Vol1.19, July 1966, S. 278-291

[Set851 S.C.Seth, V.D.Agrawal: "Cutting Chip-Testing Costs", IEEE Spectrum Vol.22, No.4, April 1985, S. 38-45

[Sus831 A.K.Susskind: "Overview of Microprocessor Testing", IEEE International Conference an Computer Design (ICCD) 1983, S. 45-48

[Veg82] S.R.Vegdal: "Phase Coupling and Constant Generation in an Optimizing Microcode-Compiler", 15th Annual Microprogramming Workshop (MICRO-15), 1982, S. 125- 133

4 Retargierbarer Codegenerator

Zu den wichtigsten Werkzeugen des MIMOLA Design Systems gehört der retargierbare Codegenerator. Dieser Codegenerator ist in der Lage eine gegebene algorithmische Beschreibung in den binären Maschinencode einer in MIMOLA spezifizierten Rechner-Hardware zu übersetzen. Als Anwendungsbereich dieses Codegenerators wurde bislang vor allem die Codeerzeugung für Benutzerprogramme gesehen.
Im Zusammenhang mit dem Testprogramm-Generator ergab sich aber ein weiterer Anwendungsbereich: Spezielle Testprogramme möchte man häufig nicht - wie bei MSST nötig - durch Testmuster spezifizieren, sondern in algorithmischer Form. Die maschinenunabhängige Codeerzeugung im MIMOLA Design System kann genutzt werden, um ein algorithmisch in MIMOLA beschriebenes Testprogramm in den binären Maschinencode zu übersetzen. Einen großen Vorteil bietet dabei die "Retargierbarkeit": Werden im Entwicklungsprozeß Hardware-Änderungen erforderlich, so kann nach einer entsprechenden Änderung der Hardware-Beschreibung unmittelbar wieder binärer Maschinencode erzeugt werden. Ein neuer Compiler ist nicht erforderlich.

Der Einsatz des bisherigen Codegenerators MSSV/MSSC [Mar84,Mar85] wurde jedoch durch relativ lange Übersetzungszeiten behindert. Aufbauend auf neuen Ideen zur Realisierung eines solchen Compilers wurde im Frühjahr 1987 mit seiner Implementierung begonnen. Im Bereich der retargierbaren Codeerzeugung wurden teilweise völlig neue Prinzipien verwandt.

4.1 Allgemeine Prinzipien

Kern des Übersetzungs-Verfahrens ist der Connection-Operation-Graph **(CO-Graph).** Er repräsentiert einerseits die Verbindungsstruktur der Hardware. Andererseits enthält er auch die zur Ausführung einzelner Operationen nötige Belegung des Instruktionswortes (den Maschinencode). Das Verfahren basiert im wesentlichen auf einem Pattern-Matching Algorithmus. Dieser versucht die durch das Programm gegebenen Zuweisungen (dargestellt als Baum) auf den durch die Hardware-Beschreibung gegebenen

CO-Graphen abzubilden. Ist eine solche Teilstruktur gefunden, wird damit der gewünschte Datenfluß innerhalb der Hardware festgelegt. Die Auswahl der Flußkanten bestimmt die auszuwählenden Multiplexer-Eingänge und damit ihren Kontroll-Code. Die Auswahl der Operatoren bestimmt die Kontroll-Codes der Funktions-Einheiten.
Dieses Verfahren unterscheidet sich im wesentlichen von dem im MSSV verwandten durch die Abbildung auf den CO-Graphen sowie die konsequente Rückführung aller Ressource-Konflikte auf Instruktions-Konflikte. Beiden Verfahren gemeinsam ist die Verwendung der reinen Hardware-Struktur-Beschreibung als Grundlage der Codeerzeugung. Andere in der Literatur angegebene Verfahren verlangen darüberhinaus Ziel-maschinenabhängige Hilfen des Benutzers. Der Aufwand zu ihrer Erstellung ist beträchtlich.
So verlangt das Codegenerations-System von Mueller [MuV83,MVA84] die manuelle Vorgabe von Flußgraphen für einzelne Teiloperationen. Das System von Vegdahl [Veg82,Veg82a,Veg83] verlangt die Angabe von Routinen, die die Möglichkeiten zur Konstanten-Erzeugung innerhalb der Hardware beschreiben. Ein wesentlicher Unterschied wird auch in der Betrachtung alternativer Lösungen (Versionen) gesehen. Obwohl schon Mallett [Mal78] in einer grundlegenden Arbeit darauf eingeht, wird die Existenz mehrerer Lösungsmöglichkeiten in allen Systemen ignoriert. Auch die bei Bode [Bod84] exemplarisch angegebenen Kompaktierungsverfahren [YST74,RaT74, DaT76,Fis79,Fis81] sind nicht für mehrere Versionen einer Teiloperation konzipiert. Gerade aber die Vielfalt der Lösungsmöglichkeiten ist nötig, um für parallele Rechnerstrukturen kompakten Code zu erzeugen. So zeigten verschiedene Anwendungen des Codegenerations-Systems (z.B.: [Kle86,Now86]), daß in realistischen Fällen bis zu 100 alternative Lösungsmöglichkeiten für einzelne Teiloperationen bestehen können. Aufgrund dieser Ergebnisse wird in dem beschriebenen Codegenerations-System besonderes Gewicht auf die Behandlung von Versionen gelegt.

Das hier vorgestellte Verfahren kann in drei Phasen gegliedert werden: In der ersten Phase, der Preallocation-Phase, wird aus der MIMOLAHardware-Beschreibung der CO-Graph aufgebaut. In der zweiten und dritten Phase erfolgt die Übersetzung der einzelnen Zuweisungen (Allocation) sowie ihre Zuordnung zu den Instruktionen (Scheduling).

4.2 Preallocation-Phase

Das Ziel dieser Phase ist der Aufbau des Connection-Operation-Graphen **(CO-Graph),** der die nachfolgende Allocation-Phase erleichtert. Er repräsentiert vollständig die vorgegebene Hardware. Sein Aufbau erfolgt in mehreren Schritten:

4.2.1 Aufbau eines Struktur-äquivalenten Graphen

Die MIMOLA-Hardware-Beschreibung besteht im wesentlichen aus einer Parts-Declaration und einer Netzliste. Die Parts-Declaration enthält die Beschreibungen der verwandten RT-Module. Diese werden durch ihre Schnittstelle (Eingänge, Ausgang, Bitbreiten), die ausgeführten Operationen sowie die dazugehörigen Steuercodes beschrieben. Jedem dieser RT-Module wird ein Knoten des aufzubauenden Graphen zugeordnet. Diese Knoten sind dann entsprechend der Netzliste durch gerichtete Kanten miteinander zu verbinden. Die Richtung der

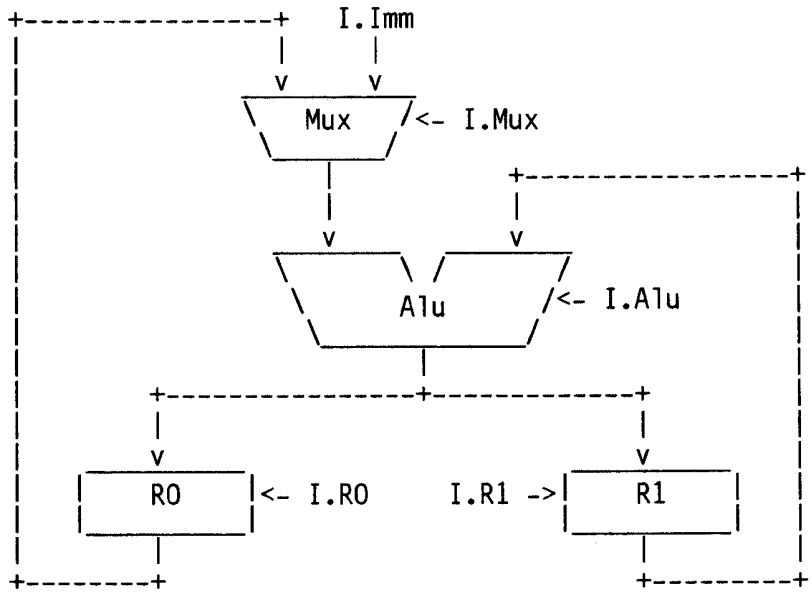


Abb. 4.1: Beispiel-Hardware

Speichern und Registern werden jeweils zwei Knoten zugeordnet: ein Knoten als Datenquelle für alle Leseoperationen, ein zweiter als Datensenke für die Schreiboperationen. Die Blätter dieses Graphen werden von Datenquellen (Speicher, -Register, (globale) Eingangs-Pins, Instruktionsfelder und festverdrahtete Konstanten) gebildet. Die Datensenken (Speicher, Register und (globale) Ausgangs-Pins) werden durch einen gemeinsamen, ausgezeichneten Knoten zusammengefaßt. Bis auf die Aufspaltung der Speicher und Register in zwei Knoten ist die Struktur dieses Graphen äquivalent zu der der Hardware.

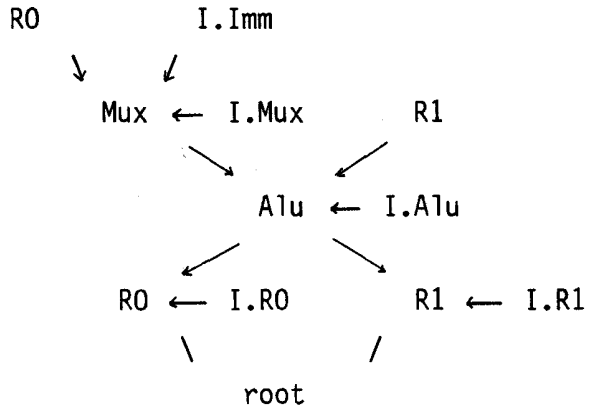


Abb. 4.2: Struktur-äquivalenter Graph

4.2.2 Auffächern der Modul-Knoten

Sei **M** ein ein RT-Modul repräsentierender Knoten, **IN(M)** die Menge der hineinführenden Kanten (die Eingänge), **OUT(M)** die Menge der herausführenden Kanten (die Ausgangsverzweigungen) und **FCT(M)** die Menge der von M ausgeführten Operationen. Dann kann M ersetzt werden durch eine Menge von Knoten **Fi**, mit i ϵ FCT(M), IN(Fi)<=IN(M) und OUT(Fi)=OUT(M). IN(Fi) umfaßt jetzt lediglich noch die von der Operation i benötigten Eingänge des Moduls M.

Da die einzelnen Operationen eines Moduls (pro Ausgang) nur alternativ genutzt werden können, stellen die Knoten Fi die möglichen Alternativen **("Versionen")** dar. Sie werden durch einen Versions-Knoten V zusammengefaßt. Insbesondere können alle Otlfi(Fi) durch OUT(V) ersetzt werden:

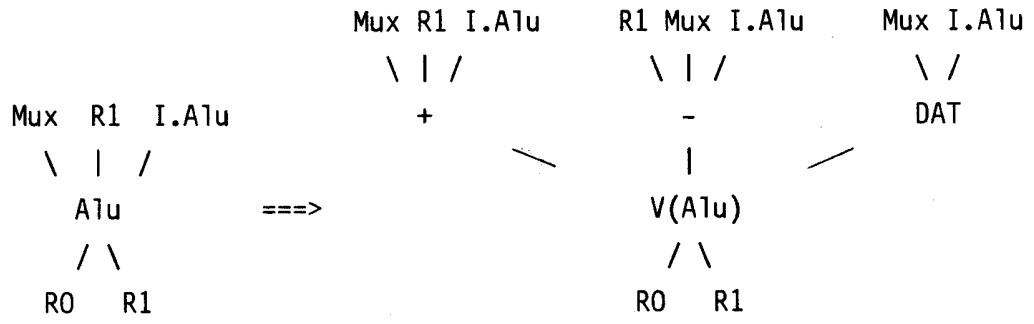


Abb. 4.3: Auffächern für Operationen
(DAT: Durchschalten des linken Eingangs)

4.2.3 Einführen von VIAs

Als **VIAs** (Umwege) werden Operationen mit einer speziellen Eingangsbelegung bezeichnet, die unter dieser Belegung die Daten eines anderen Eingangs unverändert weitergeben. Als Beispiel sei die Addition einer '0' oder die AND-Operation mit einem Bitstring von '1' genannt. Solche Operationen können in Verbindung mit ihrem Nullelement zum Durchschalten von Daten verwandt werden. Sie werden als zusätzliche Versionen in den CO-Graphen aufgenommen. Die Forderung nach einer speziellen EingangsBelegung sei im folgenden als **Assertion** bezeichnet und im Graphen durch ein '!' gekennzeichnet.

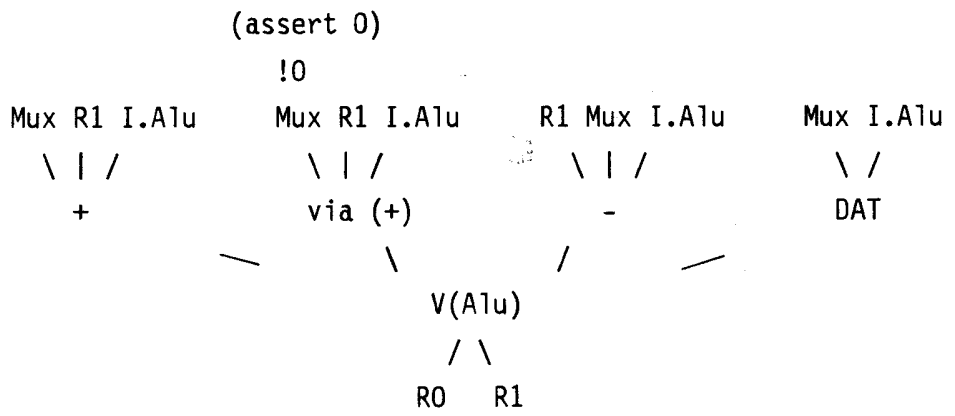


Abb. 4.4: Eintragen der VIA-Operationen

Ebenso können die benötigten Operations-Auswahl-Codes als Assertions an den Kontroll-Eingängen eingetragen werden (%... bezeichne eine Binärzahl):

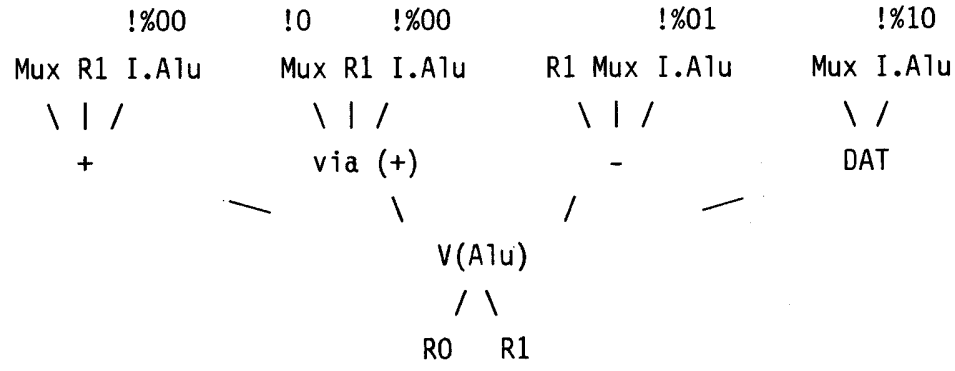


Abb. 4.5: Ausschnitt des CO-Graphen mit Assertions

4.2.4 **Allokation der Konstanten**

Die Assertions stellen Konstanten dar, die an dem betreffenden Punkt von der Hardware zur Verfügung zu stellen sind. Dies kann gewöhnlich durch eine geeignete Programmierung erreicht werden. Dadurch können jeder

Assertion eine oder auch mehrere Belegungen des Instruktionswortes mit ’0’ oder ’1’ zugeordnet werden, die die betreffende Konstante erzeugen. Die Belegung sei minimal in dem Sinne, daß alle zur Konstanten-Erzeugung irrelevanten Bits des Instruktionswortes mit Dont-Cares ('X') belegt sind. Alle möglichen Alternativen ("Versionen") zur Konstanten-Erzeugung werden im sogenannten Instruction-Tree **(I-Tree)** zusammengefaßt. Dieser I-Tree ist eine kompakte Darstellung alternativer Belegungen des Instruktionswortes.

Vor einer Definition dieses I-Trees seien zunächst noch zwei andere Begriff definiert:

Zwei Bitstrings A=(an,...,a0) und B=(bn,...,b0) sind **01X-kompatibel,** gdw. $\bigcup$ i e {0,...,n}

ai = '0' => bi = '0' v bi = 'X', ai = '1' => bi = '1' v bi = 'X'.

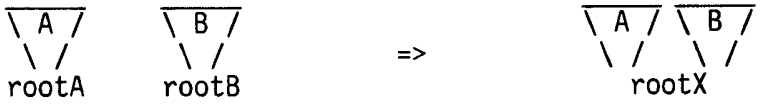
Die Menge **Fields** sei die Menge der möglichen Unterbereiche (Felder) des Instruktionswortes.

Der **I-Tree** ist eine baumartige Struktur mit folgenden Eigenschaften:

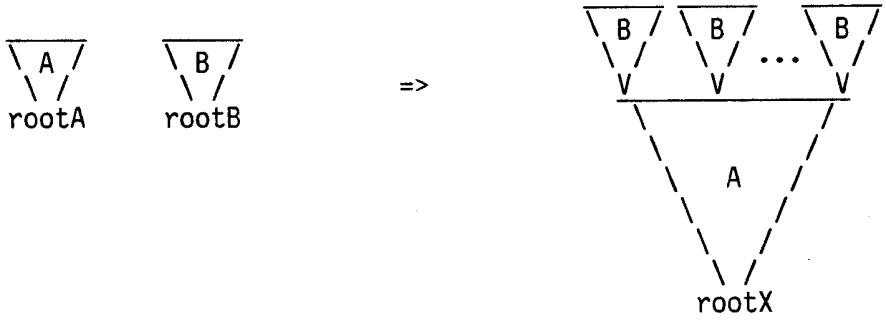
- Jeder Knoten ist einem Feld des Instruktionswortes zugeordnet. Er enthält genau eine mögliche Belegung dieses Feldes mit einem Bitstring aus '0','1' oder 'X'.
- Alle Söhne eines Knotens sowie deren Teilbäume sind kompatibel zum Vater. D.h.: Sie sind entweder verschiedenen (disjunkten) Feldern zugeordnet oder ihre Belegungen selbst sind 01X-kompatibel.
- Benachbarte Teilbäume stellen alternative Belegungen des Instruktionswortes dar.
- Jeder Pfad durch den I-Tree entspricht genau einer möglichen Belegung des Instruktionswortes. Felder die in einem solchen Pfad nicht auftreten, sind per Definition mit Dont-Cares belegt.
- ROOT(I-Tree) ist ein ausgezeichneter Knoten (die Wurzel). Sein Inhalt ist jedoch ohne Bedeutung.

SET ist eine Abbildung von Fields x Bitstring -> I-Tree, die ein Feld F des Instruktionswortes mit dem gewünschten Bitstring W belegt. Diese Belegung wird durch den I-Tree X dargestellt:

MERGE ist eine Abbildung von I-Tree x I-Tree -> I-Tree, die zwei I-Trees A,B überführt in den I-Tree X (die Alternativen A und B):



CUT ist eine Abbildung von I-Tree x I-Tree -> {I-Tree,{0}), die zwei I-Trees A,B überführt in den I-Tree X (den Schnitt von A und B). Dieser Schnitt kann auch leer sein.



Anmerkung:

Die Funktion CUT verläuft in zwei Phasen: Zunächst wird der I-Tree B an alle Blätter des I-Trees A kopiert. In der zweiten Phase werden alle inkompatiblen Pfade des neu entstandenen I-Trees X gelöscht.

Für die geschachtelte Funktion CUT(Il,CUT(I2,CUT(I3,CUT(..,In)..))) sei im folgenden die abkürzende Schreibweise CUT(I1,I2,I3,..,In) erlaubt.

Damit sind die Grundlagen geschaffen, die es gestatten verschiedene Probleme der Code-Erzeugung auf die Behandlung von I-Trees zurückzuführen:

Im vorangegangenen Abschnitt wurde gezeigt, daß jede Assertion durch einen I-Tree dargestellt werden kann. Die Ausführung einer Operation verlangt nun die Erfüllung all ihrer Assertions. Insbesondere ist ihr Kontroll-Code zur Verfügung zu stellen.
Sei I der I-Tree der einzigen Assertion einer Operation F, dann gewährleistet I die Ausführung dieser Operation F.
Seien Il und 12 die I-Trees zweier Assertions einer Operation F, dann kann dieser Operation der I-Tree I12=CUT(I1,I2) zugeordnet werden. 112 ist notwendige Bedingung für die Ausführung von F. Waren I1 und 12 die einzigen Assertions von F, ist 112 auch hinreichende Bedingung. Analog kann bei mehreren Assertions verfahren werden.
Dadurch ist die Möglichkeit geschaffen jeder Operation F des CO-Graphen einen I-Tree zuzuordnen, der ihre Ausführung gewährleistet. Im folgenden soll dargestellt werden, wie diese I-Trees gefunden werden.

Zunächst sind die einzelnen Assertions zu erfüllen: Im günstigsten Fall ist der zur Assertion gehörige Leitungspunkt selbst mit einer passenden hartverdrahteten Konstanten verbunden. In diesem Fall steht der gewünschte Wert immer zur Verfügung. Der zugehörige I-Tree besteht lediglich aus der Wurzel (alles Dont-Care).

Ist direkt über der Assertion ein Feld des Instruktionswortes zu finden, so ist dieses mit dem gewünschten Wert zu belegen. Damit bestimmt die Assertion eine Teilbelegung des Instruktionswortes. Die übrigen Felder werden durch sie nicht berührt und entsprechen damit Dont-Cares. Der zugehörige I-Tree ist durch I=SET(Feld,Wert) gegeben.

Ist direkt über der Assertion keine hartverdrahtete Konstante und kein Instruktionsfeld zu finden, so muß versucht werden, den benötigten Wert über Umwege durch andere Module zur Verfügung zu stellen. D.h.: es wird ein Pfad durch den CO-Graphen gesucht, der lediglich über DAT- bzw. VIAOperationen zu einer passenden Konstanten oder zu einem Instruktionsfeld führt.

Sei I die bisher notwendige Bedingung für die Erfüllung der Assertion und liege eine DAT-/VIA-Operation; repräsentiert durch den I-Tree D, auf dem Pfad, so muß I verschärft werden zu I=CUT(I,D).

Bestehen jedoch mehrere Möglichkeiten (beispielweise I1 und 12) den gewünschten Wert zur Verfügung zu stellen, so ist I zu erweitern zu I=MERGE(I1,I2).

Kann eine Assertion nicht erfüllt werden (I={0}), so kann der gesamte dazugehörige Operations-Baum entfernt werden. Sind jedoch alle Assertions einer Operation erfüllt, so sind die entsprechenden Eingänge für die folgenden Schritte ohne Bedeutung und werden entfernt. Der zugehörige I-Tree (hier dargestellt durch '[...]') wird dem OperationsKnoten zugeordnet:

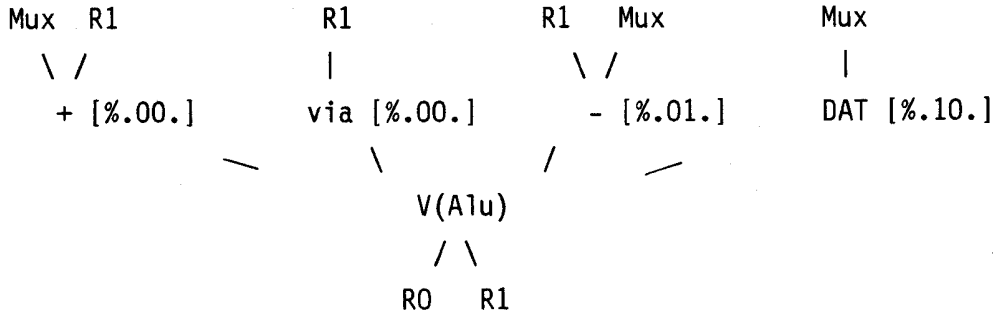


Abb. 4.6: Allokation der Konstanten

4.3 Allocation-Phase

Das Ziel dieser Phase ist die Zuordnung der Operatoren einer Zuweisung zu den vorhandenen RT-Modulen sowie die Erzeugung des dazugehörigen Codes. Da hier jedoch nur einzelne Zuweisungen (und keine Instruktionen) betrachtet werden, werden in dieser Phase wiederum nur Teilbelegungen des Instruktionswortes erzeugt, gegebenenfalls jedoch mehrere Versionen. Ihre Darstellung erfolgt

Die Zuordnung geschieht durch einen Algorithmus, der zu einem vorgegebenen Baum (die Zuweisung des Programms) einen ähnlichen Teilbaum innerhalb des CO-Graphen sucht. Ähnlich bedeutet in diesem Fall, daß neben den vorgegebenen Operatoren-lediglich DAT-/VIA-Operationen im Teilbaum auftreten dürfen. Zusätzlich gilt die Forderung, daß alle in diesem Teilbaum gefundenen Instruktionswort-Belegungen miteinander.kompatibel sein müssen. Gibt es mehrere solcher Teilbäume, werden dadurch mehrere Versionen erzeugt.
Ähnlich der Konstanten-Allokation werden gleichzeitig benötigte Operationen durch die CUT-Funktion miteinander verknüpft, Alternativen durch die MERGE-Funktion. Das Ergebnis (der zugehörige I-Tree) wird der Wurzel der gesuchten Zuweisung zugeordnet. Damit sind alle Versionen dieser Zuweisung allein durch den I-Tree repräsentiert.

Dieses Verfahren kann angewandt werden unter der Annahme, daß alle Ressource-Konflikte zu entsprechenden Instruktionsfeld-Konflikten führen. Die Annahme gilt, falls keine Seiteneffekte auftreten. Störende Seiteneffekte sind solche, die zu ungewollten Zustandsänderungen führen (Lade-Operationen). Auch Konflikte auf ungenutzten Bussen können zumindest zu einer ungewollten Leistungsaufnahme führen. In beiden Fällen ist eine Sonderbehandlung in der nachfolgenden Scheduling-Phase nötig. Alle übrigen Seiteneffekte (z.B.: Operations-Auswahl einer unbeteiligten Funktions-Einheit) können ignoriert werden.

Kann eine Zuweisung nicht allokiert werden, liefert der Allokations-Algorithmus die mögliche Fehler-Ursache und den möglichen Fehler-Ort. Als Ursachen kommen in Betracht:

- Ein gewünschter Operator ist nicht vorhanden.
- Alle Operatoren sind vorhanden, jedoch in ungeeigneter Weise (für die betreffende Zuweisung) miteinander verbunden.
- Ein ähnlicher Teilbaum wurde gefunden, jedoch sind die Instruktionswort-Belegungen inkompatibel.

Während im ersten Fall die Hardware unzureichend für die vorgegebene Zuweisung ist, könnte in den beiden anderen Fällen die Zuweisung lediglich zu komplex sein. In diesem Fall wird die Aufspaltung der Zuweisung durch Hilfszellen versucht: Als Hilfszellen stehen dem System vom Benutzer auszuzeichnende Register oder Speicherbereiche zur Verfügung.
Anhand des Fehler-Ortes und Informationen aus dem CO-Graphen kann nun eine geeignete Hilfszelle ausgewählt werden. Die Zuweisung wird versuchsweise in zwei Teile zerlegt: die Zuweisung eines Teilausdrucks an die Hilfszelle sowie dem restlichen Teil der Zuweisung mit der einkopierten Hilfszellen-Anwendung. Beide Teile werden nun erneut dem Allokations-Algorithmus angeboten. Dieses Verfahren wird rekursiv angewandt. Die Wahl der zu verwendenden Abbruchkriterien ist von entscheidender Bedeutung für die Ausführungszeit der Allocation-Phase.

4.4 Scheduling-Phase

In dieser Phase liegen dem System eine Menge von allokierten Zuweisungen vor, die in geeigneter Weise einzelnen Instruktionen zuzuordnen sind. Die Zuweisungen selbst stehen durch das Programm in zwei Relationen zueinander: Sie können datenabhängig oder antidatenabhängig voneinander sein. Beide Relationen bilden jeweils eine Halbordnung.

Eine Zuweisungen S2 ist **datenabhängig** S1, gdw. S1 eine Zelle beschreibt, deren Inhalt in S2 gelesen werden soll.

Beispiel: SEQBEGIN R1 := ... ; ... := R1 SEQEND

Eine Zuweisung S2 ist **antidatenabhängig** S1, gdw. S2 eine Zelle überschreibt, deren alter Inhalt in S1 noch gelesen werden soll.

Beispiel: PARBEGIN ... := R1 ; R1 := ... PAREND

Anmerkung: Eine formale Definition der Begriffe datenabhängig und anti-datenabhängig befindet sich bei [Mar85].

4.4.1 Aufstellen der Halbordnungen

Seien S1,S2 Zuweisungen und S2 antidatenabhängig S1, dann darf S2 nicht vor S1 ausgeführt werden (wohl aber parallel zu S1).

Wir schreiben: S1 <= S2

Sei nun GE(S1) die Menge aller Zuweisungen, die antidatenabhängig S1 sind, dann ist damit S2 E GE(S1).

Seien S1,S2 Zuweisungen und S2 datenabhängig Sl, dann muß S2 nach S1 ausgeführt werden.

Wir schreiben: S1 < S2

Sei nun GT(S1) die Menge aller Zuweisungen, die datenabhängig S1 sind, dann ist damit S2 s GT(S1).

Besteht weder eine Antidaten- noch eine Datenabhängigkeit, so sind die Zuweisungen voneinander unabhängig.

Die einzelnen Zuweisungen werden auf ihre Daten- bzw. Antidatenabhängigkeit untersucht und die betreffenden Halbordnungen aufgebaut.

4.4.2 Packen der Zyklen

Aufgrund kreuzweiser Antidatenabhängigkeiten können Zuweisungen zyklisch voneinander abhängig sein. D.h.: innerhalb der GE-Relation existiert ein Pfad mit: S1 <= S2 <= ... Sn <= S1. Diese Relation kann nur erfüllt werden durch: S1 = S2 = ... Sn , also durch die parallele Ausführung aller Zuweisungen dieses Zykels.

Daher werden in diesem Schritt alle Zykel bestimmt und die I-Trees Il,...,In ihrer Mitglieder auf Kompatibilität überprüft. Da alle Mitglieder eines Zykels gemeinsam in eine Instruktion gepackt werden müssen, werden sie im folgenden gemeinsam durch den I-Tree Icyc mit Icyc = CUT(Il,...,In) repräsentiert.

Anmerkung: Zykel innerhalb der GT-Relation können nicht auftreten und brauchen daher nicht betrachtet zu werden.

4.4.3 Packen der NOOPs

Als **NOOPs** (no operation) werden hier diejenigen Operationen bezeichnet, die das Laden eines Speichers oder Registers verhindern (NOLOAD-Operation). Zusätzlich sei das Inkrementieren des Programmzählers in die Liste der NOOPs aufgenommen. Aufgrund der GT-Relationen ist die Menge der Zuweisungen, die möglicherweise gemeinsam in einer Instruktion ausgeführt werden können, beschränkt. Alle Speicher und Register, die in dieser Menge nicht als Datensenken auftreten, müssen daher mit der NOLOAD-Operation belegt werden. Ebenso muß der Programmzähler inkrementiert werden, falls sich nicht ein expliziter Sprung unter der betrachteten Menge von Zuweisungen befindet.

Seien N1,...,Nn die I-Trees der ausgewählten NOOPs, dann ist Inoop mit Inoop = CUT(N1,...,Nn) eine notwendige Bedingung für die im folgenden zu packende Instruktion.

4.4.4 Packen der Zuweisungen

Zu dieser bisher gefundenen Teilbelegung Inoop des Instruktionswortes sind nun möglichst viele Zuweisungen zu packen. Die Reihenfolge ist durch die GE-Relation teilweise vorgegeben. Wir unterscheiden zwei verschiedene Zustände in denen sich eine Zuweisung S befinden kann: - S ist **aktiv:** GT(S)={0} und GE(S)={O}

- S ist **inaktiv:** sonst

Daher gilt: Eine aktive Zuweisung hat keinen Nachfolger und kann damit potentiell in die letzte Instruktion gepackt werden.

Der Algorithmus für das Packen von Zuweisungen geht rückwärts vor, d.h. die (zeitlich) letzte Instruktion wird zuerst gepackt. Zunächst ist die aktuelle Instruktion mit Inoop vorzubesetzen: Iinst := Inoop. Dann wird aus der Menge der aktiven Zuweisungen eine ausgewählt (z.B.: x) und es wird versucht diese zu Iinst zu packen. Das Ergebnis ist der Schnitt Iinst = CUT(Iinst,Ix). War der Versuch erfolgreich, so wird x aus allen

Mengen GE der übrigen Zuweisungen entfernt. Damit gehen weitere Zuweisungen in den aktiven Zustand über. War der Versuch nicht erfolgreich, wird eine andere aktive Zuweisung ausgewählt. Dies wird solange fortgesetzt bis keine aktive Zuweisung mehr existiert oder alle getestet wurden. Die verbliebenen Zuweisungen sind zur augenblicklichen Instruktion nicht kompatibel und müssen daher in eine andere (vorhergehende) Instruktion gepackt werden.

Da nicht alle der potentiell möglichen Zuweisungen gepackt wurden, gibt es u.U. Speicher oder Register, die weder als Datensenke noch als NOOP-Version in der aktuellen Instruktion auftreten. Ihre NOOPs sind der Instruktion noch hinzuzufügen. Multiport-Speicher erfordern an dieser Stelle eine Sonderbehandlung.

Im folgenden wird die nächste (vorhergehende) Instruktion eröffnet. Daher können alle bisher gepackten Zuweisungen aus den GT-Mengen der übrigen entfernt werden. Danach erfolgt das weitere Packen wie oben beschrieben.

Dem folgenden Beispiel (Abb.4.7) liegt eine reale Hardware zugrunde (AMD 29203). Aufgrund ihrer 2-Adreß-Befehle verlangt sie die Verwendung gemeinsamer Schreib-/Lese-Adressen bei den Hilfszellen-Ersetzungen. Die angegebene Scheduling-Reihenfolge ist nur eine der möglichen. Verständnishalber wurde hier eine Heuristik gewählt, die unter mehreren aktiven Zuweisungen jeweils die erste auswählt (RQ: Register, SR: Speicher).

4.4.5 Packen der TRISTATE-Versionen

Um Seiteneffekte zu vermeiden, die unbenutzte Busse beschreiben, sind diese mit der TRISTATE-Operation zu belegen. Die entsprechenden I-Trees sind mit den Instruktionen zu schneiden. Sie wurden bereits während der Preallocation-Phase als Schnitt aller am Bus beteiligten TRISTATEOperationen erzeugt.

Zuweisungen auf Register-Transfer-Level:

```
PARBEGIN
 SR[2]:=SR[1]+SR[3];
 SR[1]:=(((RQ+SR[2])+(SR[3]+SR[2])) + RQ) + ((SR[3]+SR[4])+SR[2]);
PAREND;
```

Hilfszellen-Ersetzung (SR[11..15]):

```
PARBEGIN
 BEGIN
  { 1} SR[11]:=SR[1];              { 2} SR[11]:=SR[11]+SR[3];
  { 3} SR[2]:=SR[11];
 END;
 BEGIN
  { 4} SR[12]:=SR[3];              { 5} SR[12]:=SR[12]+SR[2];
  { 6} SR[13]:=RQ+SR[2];           { 7} SR[13]:=SR[13]+SR[12];
  { 8} SR[14]:=SR[3];              { 9} SR[14]:=SR[14]+SR[4];
  {10} SR[14]:=SR[14]+SR[2];       {11} SR[15]+SR[13]+RQ;
  {12} SR[15]:=SR[15]+SR[14];      {13} SR[1]:=SR[15];
 END;
PAREND;
```

Liste der Zuweisungen mit möglicher Scheduling-Reihenfolge:

Stmt#	GE	GT	Reihenfolge	
1	13	2	4	
2	-	3	2	
3	-	-	1	<- letzte Instr.
4	-	5	12	
5	3	7	11	
6	3	7	13	<- erste Instr.
7	-	11	10	
8	-	9	8	
9	-	10	7	
10	3	12	6	
11	-	12	9	
12	-	13	5	
13	-	-	3	

Abb. 4.7: Beispiel zum Scheduling

4.4.6 Auswahl der schnellsten Version

Die Instruktionswort-Belegungen der betrachteten Instruktionen sind bekannt. Sie liegen als I-Tree vor. Es gibt jedoch u.U. mehrere mögliche Versionen. Damit besteht die Möglichkeit, die jeweils schnellste dieser Versionen auszuwählen.

Da die I-Trees jedoch über keine derartige Information verfügen, ist die Ressource-Auswahl anhand des CO-Graphen für jede einzelne Version zu rekonstruieren. Dies kann durch Kompatibilitäts-Checks zwischen den I-Trees der Operationen des CO-Graphen und der betrachteten Instruktions-Version geschehen. Ist die Ressource-Auswahl bestimmt, kann anhand von Timing-Angaben die Laufzeit dieser Version bestimmt werden.

4.5 Ergebnisse

Das beschriebene Verfahren wurde implementiert. Gegenüber dem bereits existierenden Codegenerator MSSV/MSSC hat es entscheidende Vorteile. So bestätigte sich in den untersuchten Beispielen, daß die Allokation von Zuweisungen um einen Faktor 50 schneller ist als in MSSV. Die Hilfszellen-Ersetzung ist wesentlich verbessert worden, so daß auch sehr komplexe Ausdrücke in akzeptabler Zeit zu übersetzen sind. Die verwandte Scheduling Strategie ist ebenfalls entscheidend gegenüber der von MSSC verbessert worden. Insbesondere werden auch in komplexen Fällen Lösungen gefunden, die mit der in MSSC verwandten Strategie nicht zu finden waren. Die folgenden Übersetzungszeiten wurden an einer SIEMENS 7760 gemessen. Die Ziel-Hardware ist ein Spezial-Prozessor, aufgebaut aus Bitslice-Bausteinen der AMD2900-Serie. Er hat eine MikroprogrammBreite von 74 Bit. Das übersetzte Programm benötigt 121 Instruktionen.

5.2 Ansatz zum Entwurf eines Blockdiagramm - Generators

5.2.1 Problembeschreibung und Diskussion

Die Entwicklung eines Blockdiagramms aus einer Schaltungsbeschreibung läßt sich grob in folgende Punkte gliedern:

- Extraktion der relevanten Daten aus der Beschreibung, wie etwa Funktionseinheiten und Verbindungen.

- Angabe der Auflösungsstufe der hierarchischen Darstellung.

- Bestimmung der Anordnung der FE's in der Ebene.

- Festlegung des Verlaufs der Verbindungen zwischen den FE's.

- Abbildung der so angeordneten Objekte auf rechtwinklige Polygonzüge.

- Ergänzung des entstandenen Diagramms um Hinweise, z.B. die Beschriftung der FE's und der Verbindungen (s. Abb. 5.1).

Die Problembereiche 'Anordnung der FE's' und 'Verlegen der Verbindungen' sind den entsprechenden Problemen aus der VLSI-Technologie verwandt. Sie sind dort unter den Begriffen 'Placement' bzw. 'Routing' bekannt. Placement- und Routing-Algorithmen sind in der Regel sehr komplex, da sie versuchen, die bei der jeweiligen Anordnung auftretende Kosten zu minimieren. Die Kriterien, die Kosten für die Erstellung eines Layout liefern, sind z.B. die aus einer Anordnung der FE's resultierenden Verbindungslängen, die Gesamtfläche der Anordnung, die Zahl der Überschneidungen von Verbindungen, die Verbindungsdicke, die Größe der FE's, usw. Diese Randbedingungen können jedoch bei der Blockdiagrammerzeugung vernachlässigt werden. Allerdings ist es sinnvoll, den Lösungsraum der Probleme 'Placement' und 'Routing' durch die Einführung speziell auf die Blockdiagrammerzeugung abzielender Randbedingungen einzuschränken.

Ein Hauptziel dieser Arbeit war, mit dem gesamten Verfahren (im folgenden 'Blockdiagrammgenerator' genannt) einen schnellen Überblick über eine Schaltung zu ermöglichen. Dieses ist insbesondere im iterativen Entwurfsprozeß wichtig, den das MSS gestattet. Die verwendeten Algorithmen sollten daher niedrige Komplexität besitzen (es wurden in der Implementierung Algorithmen der Komplexität $O(n^2)$ verwendet, wobei n die Anzahl der darzustellenden FE's ist). Die auftretenden NP-vollständigen Pobleme mußten aus diesem Grunde mit Heuristiken annähernd gut gelöst werden.

Das zentrale Problem während der Erstellung eines Layouts ist das automatische 'Placement'. Die dazu bekannten Verfahren stellen entweder eine Anordnung der FE's durch sukzessives 'Anhäufen' ('Clustering') von FE's zusammen, wobei iterative oder konstruktive Verfahren Verwendung finden, oder versuchen, eine global optimale Lösung der Anordnung zu erreichen. Insbesondere das letztere Problem entspricht im wesentlichen dem 'Quadratic Assignment Problem', dessen Lösungsverfahren schon bei kleinen Anzahlen von FE's hohe Laufzeiten besitzen. (Eine Übersicht über die bekannten Verfahren findet sich in [Pre86].) Aufgrund des oben genannten Zieles sollten in dieser Arbeit laufzeitintensive Verfahren keine Verwendung finden. Ebenso erschien es sinnvoll, im 'Placement' die Struktur einer gegebenen Schaltung eingehen zu lassen, um Übersichtlichkeit des Diagrammes und Verifizierbarkeit der Schaltung (z.B. nach einem Syntheseschritt) zu ermöglichen. Eine nähere Darstellung des verwendeten 'Placement'-Verfahrens befindet sich im Abschnitt 5.3.

Für die im 'Routing' auftretenden Probleme, wie die Minimierung von Überschneidungen und Längen von Verbindungen sind zahlreiche Lösungsansätze bekannt [Ake72]. Das in dieser Arbeit verwendete Verfahren stützt sich im wesentlichen auf die in [Ake72] vorgestellte Methode, zu einer FE ein Verbindungsnetz zu generieren, das die FE mit allen Nachfolgern verbindet. Das Netz soll dabei einem 'Steiner-Baum' entsprechen. Aus Geschwindigkeitsgründen wurde jedoch auf die Verwendung des 'Lee-Algorithmus' (wie in [Ake72] angegeben) verzichtet und ein schnelleres Verfahren entwickelt, das u.U. nur suboptimale Lösungen liefert. Auf die in der VLSI- und Platinenverdrahtung auftretenden Probleme mit mehreren Verdrahtungsebenen brauchte im 'Routing' nicht eingegangen zu werden.

5.2.2 Allgemeiner Lösungsansatz

Folgende Randbedingungen sollen bei der Erstellung eines Blockdiagrammes eingehalten werden:

- Ausrichtung des Diagrammes in Signal- (Daten-)flußrichtung
- Vermeiden von Verbindungskreuzungen, wenn möglich
- Vermeiden von Verbindungsecken, wenn möglich
- keine Überlappung von FE's und Verbindungen
- Darstellung von Bussen vorzugsweise ohne Ecken.

In verschiedenen Arbeiten [Kum86,Ven86,Aou86] sind Wege zur Erzeugung von Blockdiagrammen aufgezeigt worden. Insbesondere in [Kum86] wird versucht, das Problem der Blockdiagrammerzeugung stark in Teilprobleme zu zerlegen und diese getrennt voneinander zu lösen. Da die dort vorgestellte Aufteilung sinnvoll erschien, wurde sie im großen und ganzen in die vorliegende Arbeit übernommen. Allerdings mußten aufgrund von Unzulänglichkeiten in den dort verwendeten Verfahren die Algorithmen neu formuliert oder neue Algortihmen eingeführt werden.

In den Arbeiten [Ven86] und [Aou86] werden grundsätzlich ähnliche Ansätze wie in [Kum86] verfolgt, sie unterscheiden sich jedoch in der Wahl der Algorithmen. Unterschiede treten insbesondere beim automatischen 'Placement' (dem zentralen Problem) auf; es erschien vorteilhaft, die verschiedenen Ansätze durch Abstraktion der Schaltung auf einen Graphen zu verallgemeinern.

5.2.3 Spezieller Lösungsansatz

Um das 'Placement' und das 'Routing' graphentheoretisch behandeln zu können, wird zunächst die darzustellende Schaltung auf einen gerichteten Graphen ('Schaltungsgraph') abgebildet. Dabei werden die

- FE's als Knoten und die
- Verbindungen als Kanten aufgefaßt.

Die Kanten sind entsprechend dem Datenfluß gerichtet. Bidirektionale Verbindungen werden gemäß MIMOLA-Konvention als eine einfache Kante definiert. Mikroprogrammwort-Verbindungen werden im Graphen nicht berücksichtigt, da kein Knoten für das Mikroprogrammwort definiert wird.

Dieser so erhaltene Graph besitzt folgenden Eigenschaften, die sich direkt aus der Struktur der Schaltung ergeben:

- Er besitzt Schlingen, wenn es FE's mit Rückkopplungen gibt.
- Er besitzt Zyklen, wenn die Schaltung funktionelle Schleifen enthält.
- Er besitzt Quellen und Senken, wenn es FE's gibt, die nur Ausgänge oder nur Eingänge haben.
- Er besteht aus mehreren (schwachen) Zusammenhangskomponenten, wenn die Schaltung entsprechend viele, funktionell unabhängige Teile enthält.
- Er besitzt Mehrfachkanten, wenn es FE's gibt, die an andere FE's über mehrere Kanten derselben Richtung verbunden sind.

Um Beziehungen zwischen FE's herausstellen zu können, wird eine Gewichtungsfunktion auf den Verbindungen erklärt. Diese Funktion läßt sich auf eine Kantenbewertung des Schaltungsgraphen abbilden. Durch die Zuordnung von natürlichen Zahlen zu den Verbindungen (Kanten) läßt sich

die 'Stärke' einer Verbindung zwischen zwei FE's messen. Diese Stärke kann etwa bestimmen, wie dicht- zusammen zwei FE's in der Anordnung stehen, wodurch eine Steuerung des 'Placements' möglich wird. Die Verbindungsgewichtung kann entweder'-mit voreingestellten, Erfahrungsoder benutzerdefinierten Werten durchgeführt werden.

Während des 'Placements' brauchen zwei Verbindungstypen nicht berücksichtigt zu werden: die Schlingen und die Mehrfachkanten. Die Schlingen werden in den Schaltungsgraphen nicht aufgenommen und die Mehrfachkanten durch eine einzige Kante derselben Richtung ersetzt. Die Ersetzung berücksichtigt dabei die Gewichte der einzelnen Kanten.

Der Schaltungsgraph kann jetzt formal definiert werden:

Def. 5.1:

Der **Schaltungsgraph** einer Schaltung S ist ein gerichteter Graph $G = (V_S, E_S, f_S)$ mit:

i) V_S ist die Menge der FE's,

ii) $E_S < V_S \times V_S$ ist die Menge der Verbindungen mit:
$(x,y) \in E_S$:<=>
es existiert eine Verbindung von FE x nach FE y,

iii) $f_S : E_S \dashrightarrow N_0$ eine natürliche Bewertungsfunktion auf den Kanten.

Anmerkung: Die Relation "<" soll die Teilmengenbeziehung darstellen.

5.3 Placement

5.3.1 **Problemstellung**

Unter dem **Placement** versteht man die Anordnung der FE's in der Ebene. Dazu wird zweckmäßigerweise ein Rechteckraster zugrundegelegt. Da gemäß den Vorgaben aus Abschnitt 5.2.2 es zunächst nicht auf die Größe der FE's bzw. exakten Längen der Verbindungen ankommt, wird für die Anordnung in der logischen Phase eine spezielle graphentheoretische Struktur defininiert: das natürliche Gitter.

Def. 5.2:

Das **natürliche Gitter** ist ein ungerichteter Graph G = (V,E) mit:

i) $V := \mathbf{N}^2$ ist die Menge der **Gitterpunkte,**

ii) ein Knoten $k \in V$ besitzt die **Koordinate** $(x_k,y_k) \in \mathbf{N}^2$

iii) $E \subset V \times V$ mit folgender Eigenschaft:
$(v,w) \in E :\Leftrightarrow (x_v = x_w)$ und $(|y_v - y_w| = 1)$ oder
$(y_v = y_w)$ und $(|x_v - x_w| = 1)$.
E heißt die Menge der **Gitterkanten.**

Abbildung 5.2 zeigt einen endlichen Ausschnitt aus einer Darstellung des natürlichen Gitters.

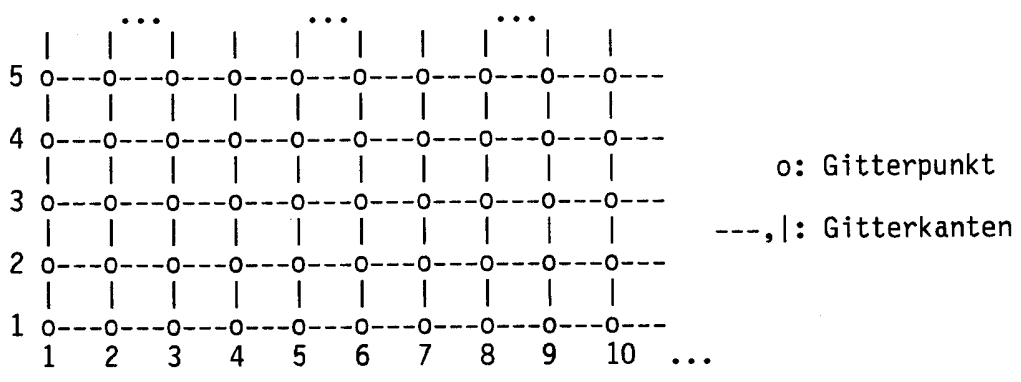


Abb. 5.2: Natürliches Gitter

	MSSV/MSSC	neuer Comp.
Allocation	1475 s	28 s
Scheduling	252 s	69 s
# Befehle	132	121

Abb. 4.8: Laufzeit-Vergleich

Das Codegenerations-System benutzt in den Bereichen Allokation und Scheduling neuartige Verfahren. So wurde besonderes Gewicht auf die Behandlung alternativer Lösungen (Versionen) gelegt. Sie werden in kompakter Form durch die Instruktionswort-Belegungen repräsentiert und als I-Trees dargestellt. Spezielle Operationen, die auf diesen I-Trees definiert sind, erlauben eine schnelle Überprüfung von RessourceKonflikten. Die Repräsentation aller Versionen durch einen I-Tree eröffnet in der Scheduling-Phase die Möglichkeit, die Auswahl einer bestimmten Version so spät wie möglich zu treffen ("delayed binding" [FLS82]). Neuartig ist die Betrachtung von NOOP-Versionen (s.o.) in der Kompaktierung. Sie stellen Seiteneffekte dar, die in maschinenunabhängigen Codegeneratoren zu berücksichtigen sind. Alle klassischen Kompaktierungsverfahren nehmen auf sie keine Rücksicht.

4.6 Literatur

[Bod84] A. Bode : Mikroarchitekturen und Mikroprogrammierung: For male Beschreibung und Optimierung, Informatik-Fachberichte, Bd. 82, Springer, Berlin, 1984

[DaT76] S. Dasgupta, J. Tartar : The Identification of Maximal Pa rallelism in Straight-Line Microprorams, IEEE Transactions an Computers, Vol. C-25, 10 (1976), S.986-992

[Fis79] J.A. Fisher : The Optimization of Horizontal Microcode within and beyond Basic Blocks: an Application of Processor Scheduling with Resources, (Dissertation), New York Univ., C00-3077-161, 1979

[Fis81] J.A. Fisher : Trace Scheduling: a Technique for Global Microcode Compaction, IEEE Transaction an Computers, Vol. C-30, 7 (1981), S. 478-490

[FLS82] J.A. Fisher, D. Landskov, B.D Shriver : Microcode Compacti on: the State of the Art, Technical Report, TR 82-3-3, Uni versity of Southwestern Louisiana, Lafayette, 1982

[Kle86] M. Klein : Entwurf eines mikroprogrammierbaren Coprozessors für die effiziente Abwicklung von Algorithmen des logischen Entwurfs, (Diplomarbeit), Fachbereich Elektrotechnik, Uni versität Kaiserslautern, 1986

[Mal78] P.W. Mallett : Methods of Compacting Microprograms (Disser tation), University of Southwestern Louisiana, Lafayette, 1978

[Mar84] P. Marwedel : A Retargetable Compiler For A High - Level Microprogramming Language, ACM SIGMICRO Newsletter, Vol. 15, Nr.4, 1984, S.267-274

[Mar85] P. Marwedel : Ein Software-System zur Synthese von Rechnerstrukturen und zur Erzeugung von Mikrocode, Habilitationsschrift, Institut für Informatik und Praktische Mathematik, Universität Kiel, 197 Seiten, 1985

[MuV83] R.A. Mueller, J. Varghese : Flow Graph Machine Models in Microcode Synthesis, 16th Annual Microprogramming Workshop (MICRO-16), 1983, S. 159-167

[MVA84] R.A. Mueller, J. Varghese, V.H. Allan : Global Methods in the Flow Graph Approach to Retargetable Microcode Generation, 17th Annual Microprogramming Workshop (MICRO-17),1984 S. 275-284

[Now86] L. Nowak : SAMP: Entwurf und Realisierung eines neuartigen Rechnerkonzeptes, (Dissertation), Institut für Informatik und Praktische Mathematik, Universität Kiel, 198 Seiten, 1986

[RaT74] C.V. Ramamoorthy, M. Tsuchiya : A High Level Language for Horizontal Microprogramming, IEEE Transactions an Computers Vol. C-23, 8 (1974), S. 791-801

[Veg82] S.R. Vegdahl : Local Code Generation and Compaction in Optimizing Microcode Compilers (Dissertation), Bericht CMU CS-82-153, Carnegie-Mellon Universität, Pittsburgh, 1982

[Veg82a] S.R. Vegdahl : Phase Coupling and Constant Generation in an Optimizing Microcode Compiler, 15th Annual Microprogramming Workshop (MICRO-15), 1982, S. 125-133

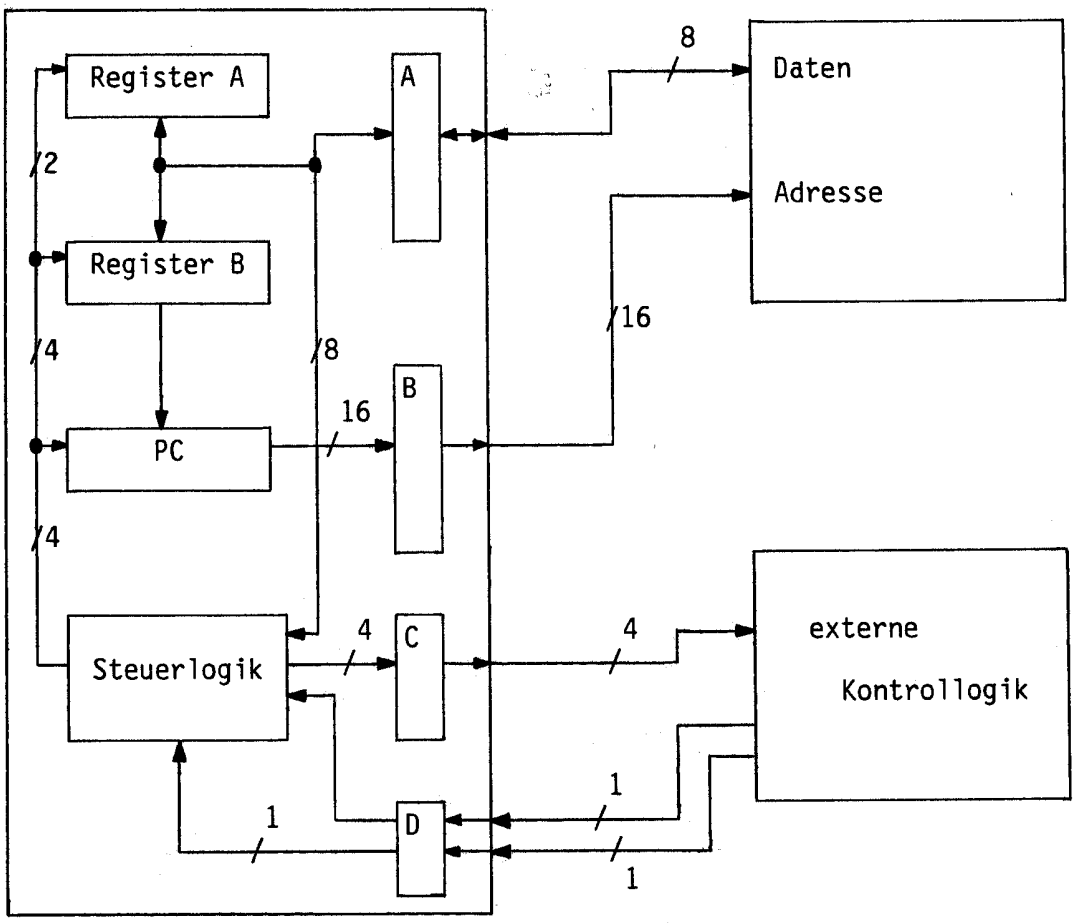
[Veg83] S.R. Vegdahl : A New Perspective an the Classical Microcode Compaction Problem, SIGMICRO Newsletter, Vol. 14, 1 (1983), S. 11-14

[YST74] S.S Yau, A.C. Schowe, M. Tsuchiya : On Storage Optimization for Horizontal Microprograms, Proc. 7th Annual Workshop an Microprogramming, 1974, S. 98-106

5 Erzeugung von Blockdiagrammen

5.1 Einleitung

Mit dem neu eingeführten Systemteil MSSG im MIMOLA - System wurde die Möglichkeit geschaffen, vom Benutzer des Systems entworfene oder durch die Synthese erzeugte Hardware - Strukturen graphisch darzustellen. Die Forderung nach einer solchen Darstellungsweise erwuchs insbesondere aus der Tatsache, das aus einer textuellen Hardware - Beschreibung ab einer gewissen Komplexität die Struktur der beschriebenen Schaltung nicht oder nur unvollständig ersichtlich ist. Dieses Problem ist bei vonHand-Entwürfen nicht unbedingt relevant, da sich i.a. der Entwickler schon vor der Anwendung des MIMOLA - Systems über die Struktur im klaren ist. Andererseits ist die aus einer Verhaltensbeschreibung synthetisierte Schaltung schwer oder gar überhaupt nicht zu verstehen, wenn sie in Textform vorliegt; eine geschickte Namensvergabe für die Funktionseinheiten könnte das Verstehen erleichtern, was jedoch durch die Automatisierung der Namensvergabe naturgemäß nicht erreicht werden kann. Aus diesem Grunde sollte die Umsetzung einer Hardware - Beschreibung in die Form von sogenannten 'Blockdiagrammen' durchgeführt werden. In einem Blockdiagramm werden die Funktionseinheiten und deren Verbindungen untereinander graphisch dargestellt, die Funktionseinheiten (FE) als Rechtecke, die Verbindungen als rechtwinklige Linienzüge, die zwischen den FE's verlaufen. Die FE's werden als 'Black Boxes' dargestellt, d.h. ihre (möglicherweise) existierende innere Struktur ist nicht nach außen sichtbar. Gleichwohl wird dadurch eine Anpassung der Blockdiagramm - Generation an eine Hierarchie von FE's erleichtert (Abb. 5.1), indem die 'Auflösungsstufe' der Darstellung entsprechend der Expansionsstufe der Schaltung (zum Thema 'Expansion' s. Abschnitt 1) gewählt wird.



Prozessor

Abb. 5.1: Beispiel einer Modulhierarchie
(A,B,C,D: Treiber der Prozessor-Schnittstelle, Hierarchie:
Prozessor -> Register,Steuerlogik -> Gatter -> ...)

Das Placement entspricht somit einer Abbildung, die jedem Knoten des Schaltungsgraphen eindeutig einen Knoten des natürlichen Gitters zuordnet. Eine solche Abbildung soll durch ein Verfahren erreicht werden, das sich auf eine Kostenfunktion stützt. Dabei sollen einerseits funktionelle Gesichtspunkte der darzustellenden Schaltung, etwa die Zusammengehörigkeit von Modulen (Multiplexer --> Speicher, Register --> Inkrementer) sowie ästhetische Forderungen (kurze Verbindungen, keine Umwege in Verbindungen) beachtet werden. Alle Forderungen können in der Regel nicht gleichzeitig optimal erfüllt werden, so daß Prioritäten gesetzt werden müssen. Dieses geschieht durch geeignete Wahl von Verbindungsgewichten (und damit der Kantenwerte im Schaltungsgraphen). Eine Kostenfunktion bewertet nun die Lage zweier Knoten zueinander, die durch eine Kante verbunden sind, nachdem für beide je ein Gitterpunkt gewählt wurde. Zusätzlich zu dem Kantenwert wird dabei die resultierende Länge der Kante und die Anzahl der Ecken, die im Kantenverlauf auftreten, bewertet. Für eine vollständige Anordnung aller Knoten sind noch die Kosten zu berechnen, die durch Überschneidungen von Kantenverläufen entstehen. Eine global optimale Anordnung aller Knoten des Schaltungsgraphen ist eine, für die die Gesamtkosten minimal sind. Das Auffinden einer solchen Anordnung ist dem 'Quadratic Assignment Problem' äquivalent [Pre86] und damit NP-vollständig. Um eine hohe Geschwindigkeit des Verfahrens zu gewährleisten, ist es notwendig, das Problem zu reduzieren.

Zunächst ist es nicht erforderlich, alle möglichen Anordnungen zu prüfen, da gewisse Randbedingungen (s. Abschnitt 5.2.2) eingehalten werden sollen. Weiterhin legt eine graphentheoretische Analyse des Schaltungsgraphen nahe, das 'Placement' in zwei Teilprobleme zu zerlegen, **die Spaltenanordnung** und die **Zeilenanordnung.** In beiden Problemen tritt die Kantenbewertung wiederum als Kostenfaktor auf.

5.3.2 Spaltenanordnung

Wie in Abschnitt 5.2.2 angegeben, sollen die Knoten des Schaltungsgraphen in Datenflußrichtung angeordnet werden. Die Art des Graphen (er ist gerichtet) legt folgenden (informell gegebenen) Anordnungsalgorithmus nahe:

1. Setze Column := 1.
Setze Nodes_To_Take := Vs und Nodes_Taken := {}.

z. Alle Knoten aus Nodes_To_Take, die keinen Vorgänger haben, erhalten Spalte Column.
Füge diese Knoten zu Nodes_Taken hinzu und nimm sie aus Nodes_To_Take heraus.

3. Ist Nodes_To_Take leer, so STOP, sonst gehe nach 4.

4. Alle Knoten aus Modes Taken werden aus den Vorgängermengen der Knoten in **Nodes_To_Take** gestrichen.

5. Setze Column := Column + 1 und gehe nach z.

Offensichtlich lassen sich mit diesem Algorithmus nur azyklische Graphen anordnen. Da ein Schaltungsgraph nicht notwendigerweise azyklisch sein muß (und es in der Regel auch nicht ist), wird dieser zunächst auf seinen reduzierten Graphen abgebildet [Wag70), welcher azyklisch ist. Die Knoten im reduzierten Graphen heißen Zusammenhangskomponenten und entsprechen den Teilen der Schaltung, die nur aus funktionellen Schleifen bestehen. Nach der Anordnung des reduzierten Graphen muß noch jeweils innerhalb jeder Zusammenhangskomponente eine Spaltenanordnung der Knoten getroffen werden. Diese Anordnung ist aufgrund der existierenden Zyklen nicht eindeutig zu treffen und geschieht durch die Betrachtung der Kantenbewertungen und der Lage der einzelnen Zusammenhangskomponenten zueinander.

Da jede Anordnung innerhalb einer Zusammenhangskomponente bedingt, daß Rückkopplungen auftreten, soll eine Anordnung gefunden werden, in der die Wertesummen der Rückkopplungskanten minimal ist. Dieses soll durch ein Verfahren erreicht werden, das speziell für das vorliegende Problem entwickelt wurde. Dieses Verfahren sucht innerhalb eines zyklischen Graphen einen azyklischen Teilgraphen mit derselben Knotenmenge und einer Teilmenge T der Kanten, so daß die Summe der Werte der Kanten aus T maximal ist.

5.3.3 Zeilenanordnung

Jedem Knoten des Schaltungsgraphen soll nun eine Zeile zugeordnet werden. Da diese Zuordnung die Spaltenanordnung nicht mehr beeinflussen soll, bedeutet das, daß die FE's in den Spalten permutiert werden müssen. Die Permutationen sollen derart geschehen, daß die eingangs erwähnten Forderungen für die Übersichtlichkeit der Kantenverläufe möglichst erfüllt werden. Dazu werden zunächst, einem Vorschlag in [Ven86] folgend, Hintereinanderreihungen ('Strings') von FE's durch ihre direkten Nachfolger gebildet. Diese Strings entsprechen gerichteten Wegen durch den Schaltungsgraphen und lassen sich als 'Datenpfade' durch die Schaltung interpretieren. Auf einem solchen Weg hat jeder Knoten, der nicht Anfangs- oder Endknoten ist, genau einen direkten Vorgänger und genau einen direkten Nachfolger. Da durch die Spaltenanordnung erreicht werden sollte, daß die Vorgänger einer FE möglichst links und die Nachfolger möglichst rechts von der FE zu liegen kommen, erstrecken sich die Wege über die Spalten hinweg. Haben FE's mehrere Vorgänger bzw. Nachfoger, so treten bei der Wahl der Wege Mehrdeutigkeiten auf. Diese könne durch Betrachten der Verbindungsgewichte aufgelöst werden.

Die Wahl der Wege wird nun folgendermaßen vorgenommen:

a) Jeder Knoten in Spalte 1 begründet einen Weg.

b1) Existiert zu einem Knoten kein direkter Nachfolger in der nächsthöheren Spalte, so beendet dieser Knoten den Weg, auf dem er liegt.

b2) Existiert genau ein direkter Nachfolger zu einem Knoten k, so setzt er den Weg fort, auf dem k liegt.

b3) Existieren mehrere direkte Nachfolger zu einem Knoten k, so ist aus den Kantenbewertungen der 'beste' direkte Nachfolger für k zu bestimmen, der den Weg von k fortsetzt.

Haben mehrere Knoten denselben direkten Nachfolger N erhalten, so setzt der Knoten N den Weg von nur genau einem Knoten fort. Das bedeutet wiederum, daß die restlichen Knoten 'ihre' Wege beenden. Um die

Zahl der Wege aus Übersichtlichkeitsgründen möglichst klein zu halten, ist es sinnvoll, die Wege der restlichen Knoten über evtl. weitere ('nächstbeste') direkte Nachfolger, fortzusetzen. Aus diesem Grunde wird die Wahl der 'besten' Nachfolger nicht je einzelnem Knoten, sondern für je zwei aufeinanderfolgende Spalten durchgeführt. Dabei sollen die Knoten der höheren Spalten denen der niedrigeren derart zugeordnet werden, daß die Summe der Werte der gewählten Kanten maximal wird. Dieses Problem ist ein lineares Zuordnungsproblem und wird in der vorliegenden Arbeit mit dem Verfahren 'Ungarische Methode' [Tin76,Hei77] gelöst.

Durch die Einschränkung, bei der Wahl der besten Nachfolger nur solche in der nächsthöheren Spalte zu betrachten, kann es vorkommen, daß zwei Wege über mehrere Spalten hinweg durch eine oder mehrere Kanten verbunden sind. Durch Zusammenfassung solcher Wege zu einem einzigen kann die Wegezahl weiter verringert werden. Die auch hier wiederum evtl. auftretenden Mehrdeutigkeiten können durch lineare Zuordung gelöst werden. Nach Definition der Wege muß noch festgelegt werden, welcher Weg welcher Zeile zugeordnet wird. Dazu wird zunächst zwischen je zwei Wegen eine **Affinität** erklärt, die sich aus der Anzahl und der Werte aller Kanten ergibt, die Knoten der beiden Wege verbinden. Danach wird versucht, die Wege in eine lineare Ordnung zu bringen, wobei je zwei Wege innerhalb der linearen Ordnung desto dichter beieinander liegen sollen, je höher ihre Affinität ist. Die Berechnung dieser linearen Ordnung geschieht mit einem speziell entworfenen Algorithmus, der das Problem mit einer Heuristik löst, da die Gesamtzahl der linearen Ordnungen bei n Wegen n! beträgt.

Nach der linearen Anordnung der Wege wird noch eine zeilenweise Kompaktierung durchgeführt. Zwei nebeneinander liegende Wege kommen in derselben Zeile zu liegen, wenn die Spaltenbereiche, die sie überdecken, sich nicht überlappen. Es hat sich jedoch als günstig erwiesen, diese Kompaktierung vom System her nur optional anzubieten, da die durch die bisherigen Verfahrensschritte entstehenden Anordnungen die Tendenz haben, in der Spaltenausdehnung erheblich größer zu werden als in der Zeilenausdehnung. Das kann jedoch die Übersichtlichkeit negativ beeinflussen.

5.4 Routing

5.4.1 Problemstellung

Mit dem Verfahrensteil 'Routing' werden alle Verbindungen, die im Blockdiagramm gezeigt werden sollen, zwischen den FE's verlegt. Wie in Abschnitt 5.2 angedeutet, wird dabei jeder FE ein 'Verbindungsnetz' zugeordnet, das die FE mit allen ihren Nachfolgern verbindet. Durch die Definition des natürlichen Gitters ist es möglich, das Verbindungsnetz graphentheoretisch zu beschreiben. Dem Verbindungsnetz entspricht somit eine zusammenhängende Menge von Kanten des natürlichen Gitters, d.h. einem zusammenhängenden Teilgraphen. Dieser Teilgraph ('Kantennetz') soll nun möglichst wenige Kanten und möglichst wenige Ecken besitzen. Ein Teilgraph mit minimaler Kantenzahl entspricht somit einem Steiner-Baum (im entsprechenden Sinne wie in der euklidischen Geometrie definiert). Da als Voraussetzung keine Verbindungen FE's überlappen dürfen, dürfen in keinem Kantennetz Knoten enthalten sein, die Koordinaten von FE's entsprechen. Aus diesem Grunde wird zunächst die gesamte FE-Anordnung im natürlichen Gitter affin abgebildet, indem die Koordinaten der FE's verdoppelt werden. Dadurch entsteht ein zusammenhängender Teilgraph K im natürlichen Gitter, der keine Knoten von FE's enthält. Werden die Kantennetze so definiert, daß sie Teilgraphen von K sind, so ist obige Forderung erfüllt. Abbildung 5.3 verdeutlicht diesen Zusammenhang.

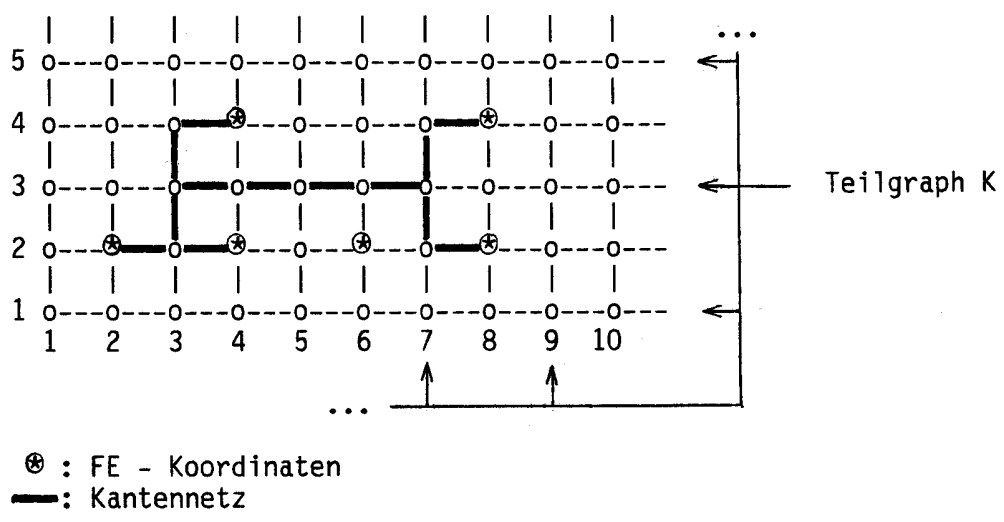


Abb. 5.3: Kantennetz

Das Auffinden eines Steiner-Baumes ist wiederum ein NP-vollständiges Problem. Mit einer Heuristik wird deshalb versucht, für jede FE ein Kantennetz mit möglichst wenig Kanten und möglichst wenig Ecken zu definieren. Eine Ecke ist im Sinne "des natürlichen Gitters ein Tripel (k1,k2,k3) von Knoten aus V, so daß gilt:

$$(x_{k1}=x_{k2} \text{ und } |y_{k1}-y_{k2}|=1 \text{ und } |x_{k2}-x_{k3}| = 1 \text{ und } y_{k2}=y_{k3}).$$

5.4.2 Das Routing - Verfahren

Das Kantennetz zu einer FE wird aus senkrechten und waagerechten Ketten, den **Segmenten,** zusammengesetzt. Die Segmente werden dabei wie folgt definiert:

Ein Segment ist eine (ungerichtete) Kette innerhalb des natürlichen Gitters, wobei alle Knoten dieser Kette entweder gleiche x- oder gleiche y-Koordinaten besitzen.

Ein Segment heiße senkrecht, wenn die x-Komponenten und heiße **waagerecht,** wenn die y-Komponenten aller seiner Knoten gleich (und ungerade) sind.

Die Bestimmung der Segmente wird folgendermaßen vorgenommen:

0. Sei eine FE M mit einer Menge von Nachfolgern DM gegeben.

1. Teile DM in die Menge der Nachfolger mit einer höheren Spalte als M (DMG) und die mit einer kleineren oder gleichen Spalte (DMK) als M auf.

2. Bestimnme in beiden Teilmengen DMG und DMK jeweils die FE mit der höchsten (= s_{DMG}) bzw. niedrigsten (= s_{DMK}) Spalte.

3a. Für die Teilmenge DMK definiere ein waagerechtes Segment W, das Knoten k1...km, m = x_M - s_{DMK} + 2, mit folgenden Koordianten enthält:
$x_{k1} = s_{DMK} - 1$, $x_{k2} = s_{DMK} + 1$, ... , $x_{kn} = x_M + 1$.

3b. Für die Teilmenge DMG definiere ein waagerechtes Segment W, das Knoten k1...kn, $n = s_{DMG} - x_M - 2$, mit folgenden Koordinaten enthält:
$x_{k1} = x_M + 1$, $x_{k2} = x_M + 3$, ..., $x_{kn} = s_{DMG} - 1$.
Besteht das waagerechte Segment für die Menge DMG aus nur einem Knoten, so wird es nicht weiter betrachtet.

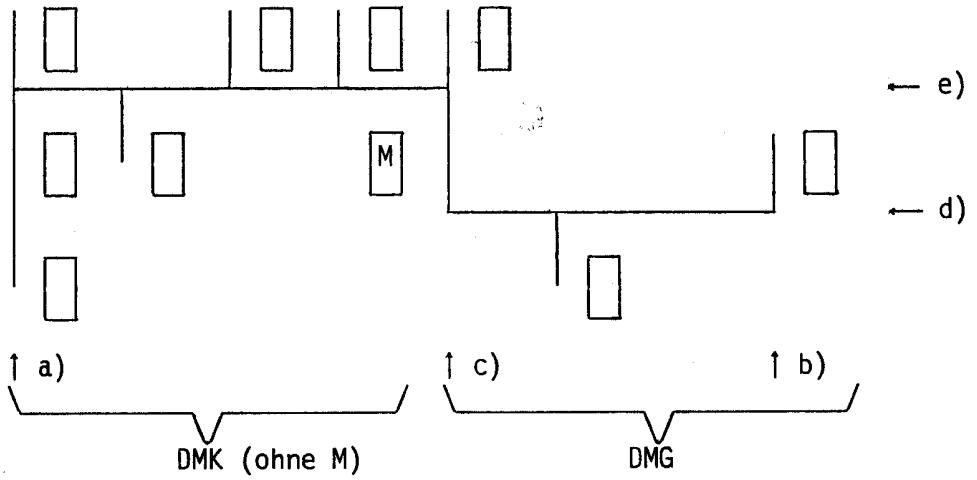
4. Die y-Komponente der Knoten in einem waagerechten Segment wird folgendermaßen ermittelt: In jeder Menge DMG und DMK bestimme jeweils die Knoten mit der höchsten und niedrigsten Zeile. Suche den Mittelwert zwischen diesen Extrema. Ist dieser Wert ungerade, so nimm ihn als y-Komponente. Ist er gerade, so erniedrige ihn um 1 und nimm diesen Wert als y-Komponente.

5. Für jede Spalte c, in der sich Elemente aus DM (= Nachfolger von M) befinden, bestimme die niedrigste (= rcmin) und die höchste (= rcmax) Zeile, die von diesen Elementen belegt werden. Definiere ein senkrechtes Segment S, das Knoten k1...kc mit folgenden Koordinaten enthält:
$x_{k1} = x_{k2} = \ldots = x_{kc} = c - 1$,
$y_{k1} = rcmin$, $y_{k2} = rcmin + 1$, ... , $y_{kc} = rcmax$.

6. Ist nach den Schritten 4 und 5 noch kein senkrechtes Segment in der Spalte $x_M + 1$ enthalten, so definiere dort ein senkrechtes Segment S.

7. Die y-Komponenten der Knoten des Segments S werden wie folgt ermittelt: Bestimme das Minimum und Maximum der Zeilen, die durch Elemente aus DM in Spalte Mx + 2 und den waagerechten Segmenten aus Punkt 3 belegt werden. Definiere S als Kette von Knoten zwischen diesen Extrema mit jeweils gleicher x-Komponente $x_M + 1$.

Abbildung 5.4 zeigt das Ergebnis dieser Schritte im Zusammenhang.



a) = Spalte s_{DMK}, b) = Spalte s_{DMG}, c) = Spalte $x_M + 1$
d) = waagerechtes Segment zu DMG, e) = waagerechtes Segment zu DMK

Abb. 5.4: Routing

5.5 Rasterung

5.5.1 Abbildung auf das kartesische Koordinatensystem

Bis zu diesem Verfahrenszeitpunkt sind die FE's und die Verbindungen nur logisch angeordnet worden, d.h. über die Größe der FE's und die exakten Längen der Verbindungen ist nichts bekannt. Auch fehlen noch die Anschlüsse der Verbindungen an die FE's. Es gilt jetzt, jede FE und jede Verbindung in das kartesische Koordiantensystem derart abzubilden, daß die Forderung nach rechtwinkligen Linienzügen erfüllt wird. Weiterhin ist es zweckmäßig, die einzelnen Linien an natürlichen Koordinaten beginnen und enden zu lassen. Aus diesem Grunde sei folgende Definition getroffen:

Def. 5.3:

Seien $x,y,z \in N$ und $x <= y$,

a) Eine **natürliche waagerechte Strecke** ist die Menge
$W_{xy}^{z} := \{ (w,z) \mid w \in R, x <= w <= y, z = const \}$.

b) Analog ist eine **natürliche senkrechte Strecke** die Menge
$SS_{xy}^{z} := \{ (z,w) \mid w \in R, x <= w <= y, z = const \}$.

c) Die natürlichen Zahlen x und y in obigen Definitionen heißen **Anfangs-** und **Endpunkte** der jeweiligen Strecken. Die Zahl z heißt **Spur** der jeweiligen Strecke.

Die bisher erzeugte logische Anordnung soll auf eine Menge von natürlichen Strecken abgebildet werden. Bevor das eigentliche Verfahren, das dieses leistet, vorgestellt wird, sind noch einige grundsätzliche Vorüberlegungen nötig.

A) Da der Datenfluß innerhalb des Diagramms von links nach rechts, d.h. im Sinne eines Koordinatensystems: in X-Richtung nach steigenden Koordinaten, verlaufen soll, sei festgelegt, daß die Eingänge einer FE an der linken Seite, die Ausgänge an der rechten Seite zu liegen kommen.

B) Die Größe der FE's ist zunächst völlig unbestimmt. Um eine Höhe und eine Breite festlegen zu können, ist es sinnvoll, die Zahl der Anschlüsse an eine FE als Größenkriterium zu verwenden. In MIMOLA wird pro FE nur ein Ausgang (bei FE's, deren Schnittstelle in Ports unterteilt ist, gilt dieses pro Port), aber beliebig viele Eingänge zugelassen. Demzufolge wird die Höhe einer FE, also deren Ausdehnung in Y-Richtung, i.d.R. durch die Zahl der Eingänge bestimmt sein. Für den Fall, daß eine FE keine Eingänge trägt, wird eine Standardhöhe angenommen. Die Breite (= Ausdehnung in X-Richtung) kann auf diese Weise nicht bestimmt werden, es wird daher eine Standardbreite angenommen.

C) Für jedes Segment jeder Verbindung muß eine exakte Länge und dessen Anfangs- bzw. Endpunkt festgelegt werden.

D) Zwischen senkrechten Segmenten und FE's, die mit diesen Segmenten verbunden sind, müssen Verbindungs-"stücke" definiert werden, die **Brücken (Strecken** "B" in Abb. 5.5).

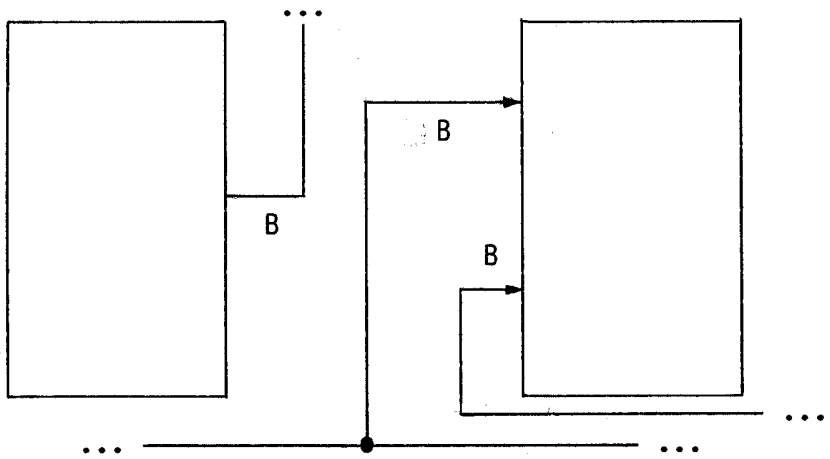


Abb. 5.5: Zum Begriff der Brücken

5.5.2 Rasterungsverfahren

Folgendes Vorgehen liefert eine (vorläufige) Abbildung der logischen Anordnung in das Koordinatensystem:

1. Bilde jede Kette im natürlichen Gitter, d.h. die Teilgraphen mit den Knoten
   $SK_C := \{ (c,i) \mid c = const \in N, i=1,2,3,... \}$
   ('senkrechte Ketten') bzw.
   $WK_C := \{ (i,c) \mid c = const \in N, i=1,2,3,... \}$
   ('waagerechte Ketten'),
   auf entsprechende Teilflächen des Koordinatensystems ab. Diese Teilflächen erstrecken sich in der entsprechenden Richtung.

2. Die in Punkt 1. erwähnten Teilflächen heißen im folgenden **Kanäle** und seien definiert als:
   $SKK_C := WID_C \times LEN_C < R^2$ (zur Kette SK_C gehöriger Kanal)
   $WKK_C := LEN_C \times WID_C < R^2$ (zur Kette WK_C gehöriger Kanal)
   wobei WID_C und LEN_C jeweils Intervalle aus R mit natürlichen Randpunkten sind.

3. Das Intervall 'WID_c' eines Kanals c heißt die **Breite** des Kanals und wird folgendermaßen festgelegt:
   a) Ist c ungerade, so sind Segmente von Kantennetzen Teilketten der Kette c. Im Kanal c sollen daher die Streckenbilder dieser Segmente zu liegen kommen. (Dieses gilt für waagerechte wie für senkrechte Ketten/Kanäle/Segmente.) Die Breite WID eines solchen Kanals sei daher durch die Anzahl der in ihm enthaltenen Segmentstrecken gegeben.
   b) Ist c gerade, dann enthält die Kette SK_c bzw. WK_c Knoten von FE's, aber immer nur einzelne Knoten von Segmenten. Im entsprechenden Kanal sollen daher die Rechtecke der FE's zu liegen kommen (und Teile der Segmentstrecken, die diesen Kanal durchstoßen). In einem waagerechten Kanal sei die FE mit der größten Höhe bestimmt; deren Höhe legt die Breite WID des Kanals fest. In einem senkrechten Kanal wird die Breite WID durch die Standardbreite der FE's bestimmt.

4. Sind 'cmax' und 'rmax' die größte von FE's belegte Spalte bzw. Zeile der gesamten Anordnung, so sind cmax + 1 bzw. rmax + 1 die höchsten Nummern von belegten Kanälen. Jedem waagerechten Kanal w wird damit als **Länge** LEN_w die Summe der Breiten der senkrechten Kanäle zugeordnet. Analog wird die Länge der senkrechten Kanäle definiert.

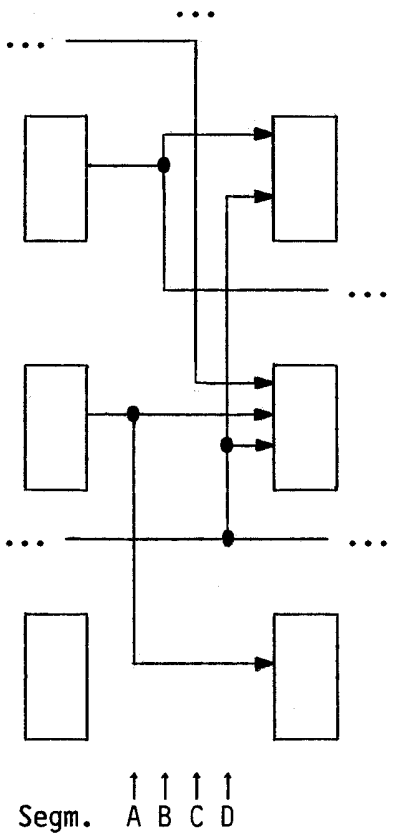
5. Unter der Forderung, daß das endgültige Diagramm 'bündig' zu dem linken und unteren Rand des Koordinatensystems zu liegen kommen soll, liegen die Ränder jedes Kanals in der Breite numerisch fest. (Z.B. beginnt der erste senkrechte Kanal SKKs1 bei x=1 und endet bei WID_{s1}, der zweite (SKK_{s2}) beginnt bei (WID_{s1})+1 und endet bei $WID_{s1} + WID_{s2}$ usw.)

   a) Für jede FE lassen sich nun die Strecken numerisch bestimmen, die das Rechteck der FE bilden sollen. Danach liegen auch die Koordinaten der Anschlüsse an jeder FE fest.

b) Die numerischen Daten der Segmente (d.h. der Streckenbilder der Segmente) sind nur z.T. angebbar. Für senkrechte Segmente liegen nur die Endpunkte der entsprechenden Strecken fest, die durch die Koordinate eines Anschlusses gegeben sind. Die Segmente innerhalb eines Kanals sollen derart permutiert werden, daß die Anzahl der Kreuzungen von Segmenten mit Brücken (in senkrechten Kanälen) bzw. von Segmenten mit Segmenten (in beiden Kanalarten) möglichst klein ist (s. 5.5.3). Deshalb ist eine numerische Festlegung der Segmentspuren noch nicht sinnvoll.

5.5.3 Kanalverdrahtung

Im Abschnitt 5.4 wurde das Routing vorgestellt, das eine logische Zuordnung der Verbindungen zum natürlichen Gitter durchführte. Dort wurde allerdings nicht auf das Problem der Minimalität von Kreuzungen bzgl. der Kantennetze eingegangen. Nach der Zuordnung von Segmenten der Kantennetze zu Kanälen sollen nun die Segmente derart sortiert werden, daß möglichst wenig Überschneidungen resultieren (s. Abb. 5.6, links). Das Sortierverfahren wird dabei auf jeden Kanal separat angewandt und entspricht somit einem 'Kanalverdrahtungsverfahren', wie sie in der VLSI-Technologie Verwendung finden. Da die dort auftretenden Randbedingungen in dem hier interessierenden Zusammenhang jedoch nicht benötigt werden, wurde ein einfaches und schnelles Verfahren für den Blockdiagrammgenerator entwickelt. In diesem Verfahren werden zunächst je zwei Semgente eines Kanals in je zwei Lagen zueinander bewertet. Für jede Lage kann die Anzahl der Kreuzungen, die beide Segmente gegenseitig erzeugen, festgestellt werden (Abb. 5.6, o.r.). Aus diesen Informationen wird ein **paarweise gerichteter Graph** gewonnen, dessen Knoten die Segmente darstellen und dessen Kanten die oben erhaltene Kreuzungsbewertung beschreiben (Abb. 5.6, u.r.). Die Aufgabe, die Segmente eines Kanals nun derart zu sortieren, daß möglichst wenige Kreuzungen entstehen, entspricht somit der Aufgabe, aus dem erhaltenen Graphen einen spannenden Teilgraphen zu extrahieren, der azyklisch ist, aus jedem Kantenpaar genau eine Kante enthält und minimale Kantensumme (Summe der Werte aller verwendeten Kanten) trägt. Dieses wird im o.a. Verfahren durch eine Heuristik annähernd erreicht, wobei ein existierendes Optimum immer gefunden wird.

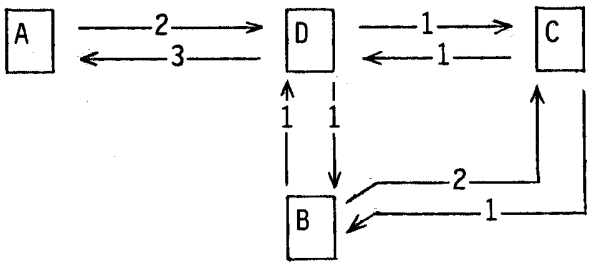


Lagematrix:

x/y	A	B	C	D
A	X	0	0	2
B	0	X	2	1
C	0	1	X	1
D	3	1	1	X

(lies: liegt Segment x links von Segment y, dann entstehen z Kreuzungen)

Zugehöriger Graph (fasse die Lagematrix als bewertete Adjazenzmatrix auf):



Optimaler Teilgraph:

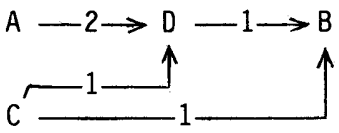


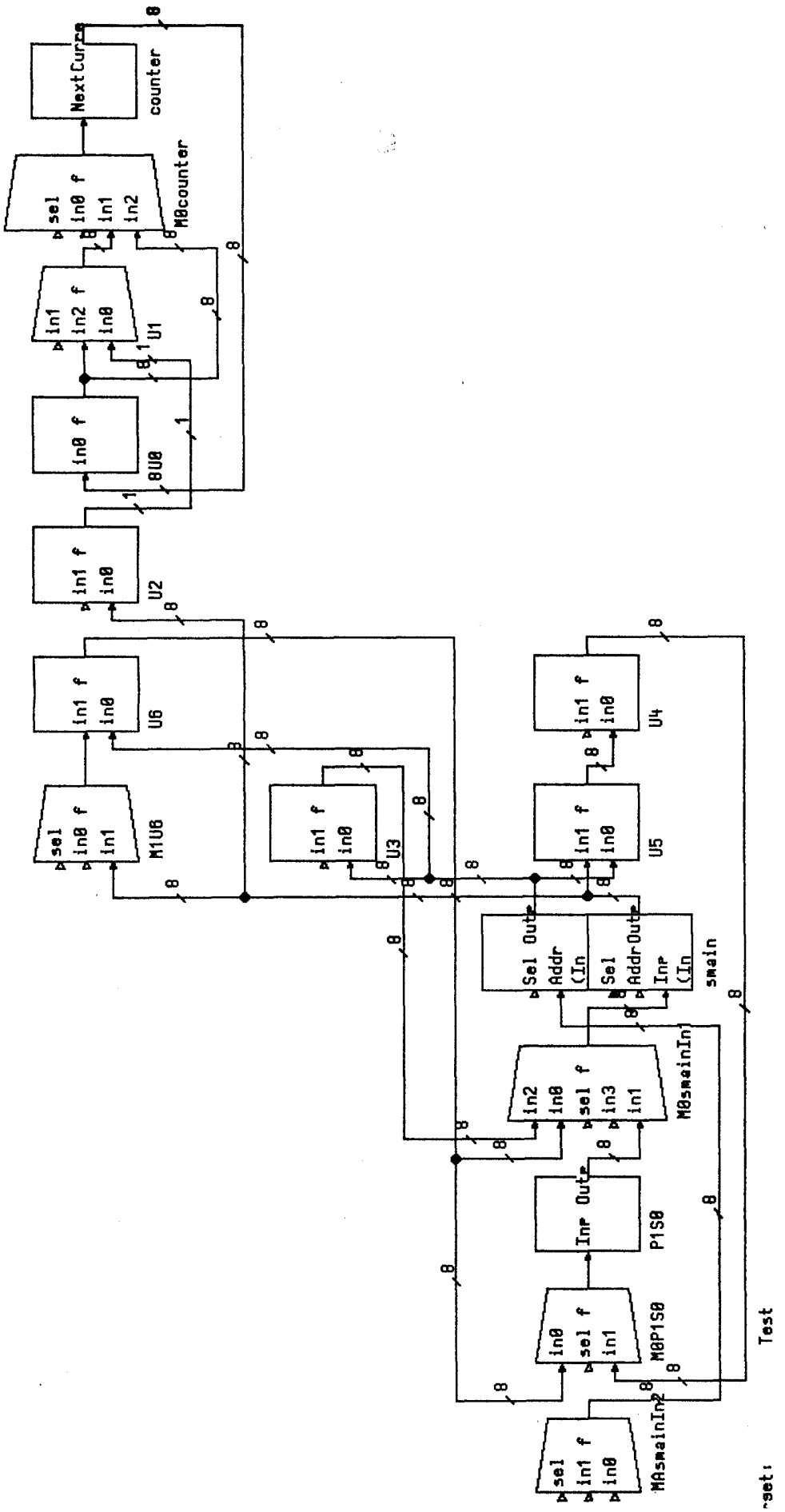
Abb. 5.6: Zur Kanalverdrahtung

5.5.4 Kompaktierung

Durch die Kanalverdrahtung kann es geschehen, daß in einem Kanal zwei oder mehr Segmente dieselbe Spur erhalten. Da bisher die Breite eines Kanals durch die Gesamtzahl der in ihm enthaltenen Segmente bestimmt war, kann in so einem Fall die Breite entsprechend kleiner gewählt werden. Z.B. kann in Abb. 5.6 die Breite des Kanals von 4 auf 3, gemäß dem optimalen Teilgraphen, reduziert werden, da Segment A und C dieselbe Spur erhalten. Um das Gesamtschaubild so klein wie möglich zu erzeugen, muß es in X- bzw. Y-Richtung "gestaucht" werden, d.h. alle Strecken von Modulrändern, Segmenten und Brücken müssen um eine gewisse Anzahl von nicht benutzten "Rasterlinien" verschoben bzw. verkürzt werden.

5.6 Ein Beispiel

Auf der folgenden Seite ist ein Beispiel für eine Schaltung angegeben, wie sie durch den Blockdiagrammgenerator erzeugt und auf einer Apollo-Workstation mit einem entsprechenden Treiber dargestellt wurde. Die Schaltung selbst ist durch


NextCurr
counter
M0counter
sel
in0 f
in1
in2
in1
in2 f
in0
U1
in0 f
8U0
in1 f
in0
U2
in1 f
in0
U6
sel
in0 f
in1
M1U6
in1 f
in0
U3
in1 f
in0
U4
in1 f
in0
U5
Sel Out
Addr (In
Sel
Addr Out
Inp (In
smain
in2
in0
sel f
in3
in1
M0smainIn
Inp Out
P1S0
in0
sel f
in1
M0P1S0
sel
in1 f
in0
M0smainIn2
Test

5.7 Zusammenfassung

In der vorliegenden Arbeit wurde der Versuch unternommen, das Problem der graphischen Darstellung von Hardware-Beschreibungen durch einen graphentheoretischen Ansatz zu lösen. Das sollte einerseits durch eine Abstrahierung der Schaltung auf einen gerichteten Graphen geschehen, der andererseits jedoch auch die Möglichkeiten bieten sollte, funktionelle Gegebenheiten der Schaltung zu berücksichtigen. Dieses geschah durch die Bewertung der Kanten, die den Verbindungen entsprechen. Das Placement wurde zum größten Teil durch diese Kantenbewertung gesteuert, was sich nach dem Test des Blockdiagrammgenerators auf den verschiedensten Schaltungen in den meisten Fällen als positiv herausstellte. Teilweise wurde jedoch deutlich, daß gewisse Optimierungen des erhaltenen Placements sehr viel bessere Ergebnisse bringen können. Das Routing, sowie besonders die Kanalverdrahtung, brachten in allen Fällen gute Ergebnisse. Ebenso hat sich die (portable) Darstellung des Diagramms in der Zwischensprache des MIMOLA-Systems (TREEMOLA) als zweckmäßig erwiesen, da eine Anpassung von verschiedenen Treibern in kurzer Zeit durchgeführt werden konnte.

5.8 Literatur

[Ake72] S.B. Akers: Routing, in: M.A.Breuer: Design Automation of Digital Systems, Vo l.1, Prentice Hall, Englewood Cliffs, 1972

[Aou86] Aoudja, Laborie, Saint-Paul : CASE, Automatic Generation of Electrical Diagrams, Computer-Aided-Design, Vo1.18, No.7, Sep. 1986, S.356-360

[Hei77] 0. Hein: Graphentheorie für Anwender, BI-Hochschultaschenbücher Mannheim 1977

[Kum86] Kumar, Arya, Swaminathan, Misra: Automatic Generation of Digital System Schematics Diagrams, IEEE Design and Test, Feb. 1986 S.58-65

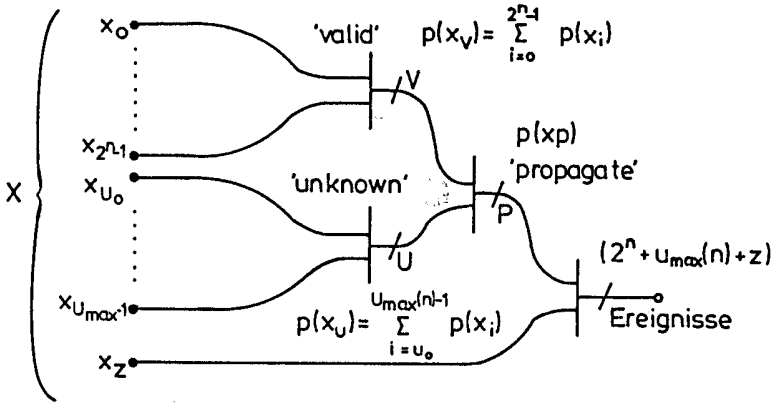


Abb.7.2.2: Aufspaltung des Ereignisraumes X in 3 Gruppen: V, U und P.

Das hier abgeleitete Testbarkeitsmaß basiert auf der Behauptung bzw. Erfahrungstatsache, daß zur Auftrittswahrscheinlichkeit $p(x_i)$ einer Datenvektorkomponente seine Ereignistestbarkeit $t(p(x_i))$ korrespondiert. Die Knotentestbarkeit selbst wird als die auf die Bitbreite bezogene Erwartung aller möglichen 'validen' Knotenereigniszustände definiert, woraus genau dann eine möglichst gute Testbarkeit resultiert, wenn alle validen Ereignisse gleich gut erzeugt und beobachtet werden können.

$$T_n(X) := \frac{1}{n} \cdot E[t(x)] = \frac{1}{n} \cdot \sum_{i=0}^{2^n-1} t(p(x_i)) \cdot p(x_i) \qquad (7.2.3)$$

7.2.1 Ableitung und Interpretationen des eindeutigen Knotenbewertungdmaßes

Ein Knotenbewertungsmaß, welches auf der Basis des Testinformationsflusses durch eine Rechnerarchitektur arbeitet, muß zur quantitativen Festlegung 4 Grundaxiome erfüllen, die es eindeutig und widerspruchsfrei festlegen /11/:

1. Monotonieeigenschaft:

   Der Erwartungswert für Bleichverteilte Ereignistestbarkeiten soll monoton mit der Pfadbreite anwachsen!

$$E[t_o(x)] < E[t_1(x)] \quad \forall\ n_o < n_1 \qquad (7.2.4)$$

   Ein breiter Datenpfad transportiert mehr *Testinformationen als* z.B. eine logische 1-Bit *Signalleitung.*

z. Acruivalenzeigenschaft:

Bei der Verknüpfung mehrerer unabhängiger Datenpfade zu einer gemeinsamen Leitung addieren sich die Ereignistestbarkeiten! (Konkatenationen von Bitleitungen)

$$E\left[\prod_{i=0}^{m} t_i(x)\right] = \sum_{i=0}^{m} E\left[t_i(x)\right] \qquad (7.2.5)$$

3. Gruppeneigenschaft:

Einführung bedingter Testbarkeiten durch Aufspaltung eines zusammenhängenden Ereignisraumes in Untergruppen V, U und P. Dabei ist von besonderem Interesse die Frage nach dem bedingten Erwartungswert der 'validen' Ereignisse!

$$E\left[t(x_p)\right] = E\left[t(V,U)\right] + p(V)\cdot E\left[t(x/V)\right] + p(U)\cdot E\left[t(x/U)\right] \qquad (7.2.6)$$

4. Kontinuitätseigenschaft:

Aus Gründen der mathematischen Praktikabilität sollen kleine Änderungen der Ereigniswahrscheinlichkeiten nur kleine Änderungen der Knotentestbarkeiten bewirken!

$$E\left[t(x)\right] = E\left[t\left(p(x)\right), 1-t\left(p(x)\right)\right] \qquad (7.2.7)$$

Es lässt sich zeigen, das die in G1, (7.2.8) angegebene Gruppentestbarkeit der 'validen' Knotenereignisse eindeutig ist und die Axiome (1...4) ausschließlich erfüllt:

$$T_n(X/V) = \begin{cases} \frac{1}{n}\left\{\sum_{i=0}^{2^n-1} p(x_i)\cdot \mathrm{ld}\,\frac{1}{p(x_i)} - \left[p(x_p)-p(x_u)\right]\cdot \mathrm{ld}\,\frac{1}{p(x_p)-p(x_u)}\right\} \\ \\ T_n(X) \quad - \quad T_U(X) \end{cases} \qquad (7.2.8)$$

Die mittlere Unbestimmtheit oder Testbarkeit eines funktionalen Knotens hängt nicht von den diskreten gültigen Ereignissen x_i selbst ab, sondern nur von deren Wahrscheinlichkeiten $p(x_i)$, sie an dem betrachteten Knoten zu erzeugen. Sie setzt sich zusammen aus der Differenz der gültigen,

informationstragenden Ereignistestbarkeiten und der gewichteten Testbarkeit der Gruppe der validen Zustände. Abb. 7.2.3 rechtfertigt die intuitive Annahme, daß für eine 'unknown' und 'high impedance'-Wahrscheinlichkeit $p(x_u) + p(x_z) = 1$ die bedingte Testbarkeit $T_n(X/V) = 0$ ist und nur für $p(x_u)$, $p(x_z) = 0$ ein Schaltungsknoten überhaupt maximal testbar sein kann.

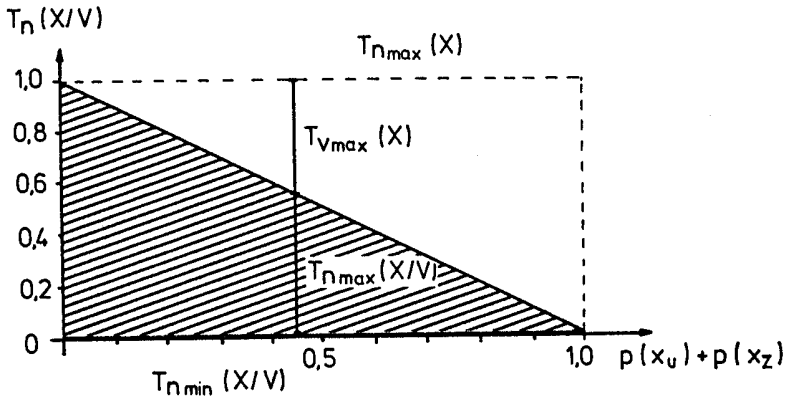


Abb.7.2.3: Bedingtes Testbarkeitsdreieck in Abhängigkeit der 'unknown'- und 'high impedance'-Wahrscheinlichkeit $p(x_u) + p(x_z)$.

Anschaulich kann $T_U(X)$ in Gl. (7.2.8) als eine 'a priori'-Untestbarkeit interpretiert werden, da sie infolge der ungültigen Wahrscheinlichkeiten zu große Ereignistestbarkeiten $t\left(p(x_i)\right)$ der validen Ereignisse vortäuscht.

Die bedingte Testbarkeit $T_n(X/V)$ kann allgemein als die mittlere Menge der Ja/Nein-Entscheidungen aufgefaßt werden, die gefällt werden müssen, um den Zustand einer 1-Bitleitung eines n-Bit Pfades eindeutig zu bestimmen. Erzeugen von außen angelegte Teststimuli an internen funktionalen Schaltungsknoten geringe Erwartungswerte für die Ereignistestbarkeiten, so sind entweder im Mittel die Testvektoren ungeeignet und/oder die Architektur ist nach Testbarkeitsgesichtspunkten unfähig, möglichst alle internen Knotenzustände zu erzeugen.

Eine weitere wichtige Interpretationsmöglichkeit ist die Aussage über die mittlere Anzahl der signifikanten Knotenereignissequenzen in Abhängigkeit der Testbarkeit des betrachteten Datenpfades. Dazu wird die Ereignissignifikanz definiert, die aussagt, daß ein Knoten der Breite n mit der Testbarkeit $T_n(X/V) = 1$ genau 2^n wohlunterscheidbare Bitpattern führt und im umgekehrten, ein Knoten mit $T_n(X/V) = 0$ nur ein einzelnes Haftmuster trägt.

$$E_S = \left\lceil \left(2^n\right)^{T_n(X/V)} \right\rceil = \left\lceil 2^{E[t(x/V)]} \right\rceil \qquad (7.2.9)$$

$$\sigma_X = E_S{}^N \qquad (7.2.10)$$

Die Zahl der typischen Testsequenzen in Abhängigkeit der Testvektoren N gibt Gl. (7.2.10) an und lest sich unmittelbar aus dem Testbarkeitsma8 herleiten, sie stellt eine hilfreiche Brücke dar zur Berechnung der Fehlerüberdeckung und Rückschlußsicherheit interner Knotenereignisse.

7.2.2 Testbarkeit und Information

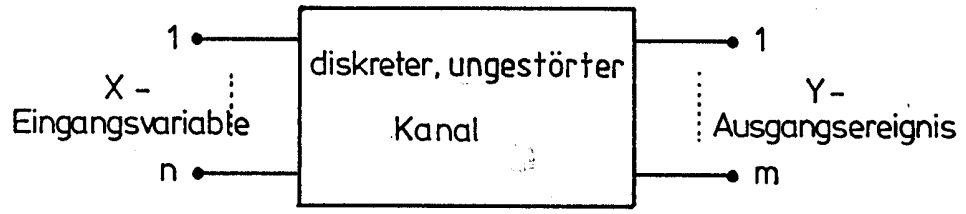
Der Erwartungswert für die Ereignistestbarkeit ist identisch mit der aus der Informationstheorie her bekannten Kommunikationsentropie H(X). Diese angestrebte Übereinstimmung ist für die Handhabung des Testbarkeitsmaßes von besonderer Wichtigkeit, da somit auf Sitze und Eigenschaften der von Shannon begründeten Theorie beim Informationstransport durch Nachrichtenkanäle zurückgegriffen werden kann /12,13/.

$$H(X) := E\left[t\left[p(x)\right]\right] = \sum_{i=0}^{2^n-1} p(x_i) \cdot \mathrm{ld}\, \frac{1}{p(x_i)} \quad \text{bit/pattern} \qquad (7.2.11)$$

$$T_n(X/V) = \frac{1}{n} \cdot \left\{H(X) - H_U(X)\right\} \quad \text{pattern}^{-1} \qquad (7.2.12)$$

Die bedingte oder 'tatsächliche' Knotentestbarkeit Tn(X/V) ist somit die auf die Pfadbreite n bezogene 'wahre) Entropie der validen Knotenereignisse H(X), abzüglich aller ungültigen Ereignisse, vgl. auch Gl. (7.2.8).

Soll ein digitaler Schaltkreis auf der Basis informationstheoretischer Methoden hin untersucht werden, muß jeder funktional beschriebene Modul auf RT-Sprachniveau als ein diskreter Kanal beschreibbar sein, der ein gegebenes Eingangsalphabet oder einen Ereignisraum $X \in \{x_0, \ldots, x_{2^n-1}, x_u\}$ in ein Ausgangsalphabet $Y \in \{y_0 \ldots, y_{2^m-1}, y_u, y_z\}$ überführt, Abb. 7.2.4 verdeutlicht die Abstraktion. Dabei ist nicht mehr von notwendigem Interesse, wie sich die Ereignisräume transformieren, sondern nur, wie sich deren zugehörige diskreten Wahrscheinlichkeitsverteilungen abbilden.



$$X \in \{x_0,\ldots,x_{2^n-1},x_u\} \qquad Y \in \{y_0,\ldots,y_{2^m-1},y_u,y_z\}$$

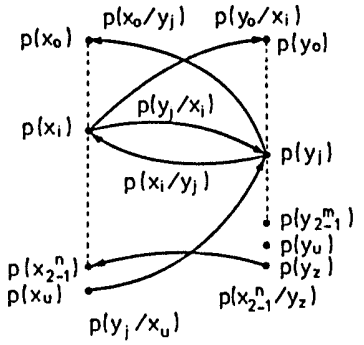
$$p(x_i) = \text{prob}\{X=x_i\}\ \forall\ i\in[0,2^n-1,u]; \quad p(y_j) = \text{prob}\{Y=y_j\}\ \forall\ j\in[0,2^m-1,u,z]$$

Abb.7.2.4: Register-Transfer Modul als diskreter, störungsfreier Übertragungskanal

Der diskrete Kanal selbst wird dargestellt als ein System von Übergangsund Rückschlußwahrscheinlichkeiten, wobei die Frage nach der Auftrittswahrscheinlichkeit für das xi-te und yj-te Ereignis durch die Verbundwahrscheinlichkeiten beantwortet werden:

$$p(x_i,y_j) \stackrel{!}{=} p(y_j,x_i) := \text{prob}\{(X=x_i)\cap(Y=y_j)\}\ \forall \begin{cases} i \in [0,2^n-1,u] \\ j \in [0,2^m-1,u,z] \end{cases} \qquad (7.2.13)$$

Die eingeführten Wahrscheinlichkeiten lassen sich durch den in Abb. 7.2.5 dargestellten Entscheidungsfächer in Verbindung bringen:



$p(y_j/x_i)$:= bedingte Wahrscheinlichkeit

$p(x_i/y_j)$:= Rückschlußwahrscheinlichkeit

Abb. 7.2.5: Entscheidungsfacher für die Ein/Ausgangsereignisse

Ein Modul, dessen interne Schaltungsstruktur dem Designer beim Entwurf einer Rechnerarchitektur auf der funktionalen Ebene verborgen ist, hat als einziges Kriterium zur Bewertung des Informationstransfers die Ubergangs- bzw. Rückschlußwahrscheinlichkeitsmatrizen, die vollständig die Modultransformationseigenschaften wiedergeben.

7.2.3 Kennwerte für Modultestbarkeiten

Formal kann jetzt der Entropiebegriff auf ein paarweise auftretendes Doppelereignis ausgedehnt werden, dabei ist die Verbundtestbarkeit das Maß für die mittlere gleichzeitige Testbarkeit des X_i-ten Ereignisses am Moduleingang und des y_j-ten am Ausgang. Sind die beiden zu testenden funktionalen Knoten über den Modul verknüpft, d.h. besteht eine gegenseitige statistische Bindung, so resultiert für die Gesamttestinformation des Bausteins:

$$T(X,Y) = \frac{n}{n+m} \cdot T_n(X/V) + \frac{m}{n+m} \cdot T(Y/X) \qquad (7.2.14)$$

$$T(Y,X) = \frac{m}{n+m} \cdot T_m(Y/V) + \frac{n}{n+m} \cdot T(X/Y) \qquad (7.2.15)$$

Die in den Gl. (7.2.14/15) neu auftretenden Größen T(Y/X) und T(X/Y) stellen die irrelevanten bzw. äquivokativen Anteile der Gesamttestinformationen bei einem Modultransfer dar. Im einzelnen können beide Größen folgendermaßen anschaulich interpretiert werden, vgl. dazu auch Abb. 7.2.6.

T(Y/X):= ist ein Maß für die Unbestimmtheit der Ereignisse am Modulausgang Y unter der Bedingung, daß die Eingangsereignisse mit ihren Wahrscheinlichkeiten $p(x_i)$ auftreten, also ein Maß für die Vorhersageunsicherheit am Ausgang des Moduls.

T(X/Y):= ist ein Maß für die Unsicherheit der Eingangsereignisse $\{X=x_i\}$ eines Moduls, wenn am Ausgang die Ereignisse mit ihren Wahrscheinlichkeiten $p(y_j)$ beobachtet werden, also ein Maß für die Rückschlußunsicherheit auf den Eingang!

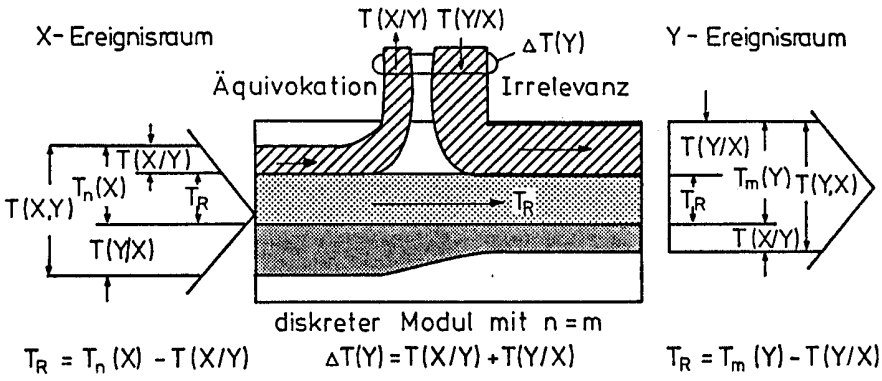


Abb.7.2.6: Testinformationstransport beim Durchgang eines funktionalen Moduls (modifiziertes Berger'sches Diagramm mit n=m)

Die Gl. (7.2.14/15) stellen die Basis dar für einen sukzessive berechenbaren Ereignis- und Informationstransport durch eine modulare Architektur. Subtrahiert man beide Gleichungen voneinander, so ergeben sich zwei dem Modul charakteristische Differenzen, die als Transfertestbarkeit $T_R(X/Y)$ des Moduls bezeichnet werden:

$$T_R(X/Y) := \begin{cases} \left(\frac{n}{m}\right)^{\frac{1}{2}} \cdot \left\{T_n(X/V) - T(X/Y)\right\} \\ \left(\frac{m}{n}\right)^{\frac{1}{2}} \cdot \left\{T_m(Y/V) - T(Y/X)\right\} \end{cases} , \qquad (7.2.16)$$

sie stellt das Maß über die transportierte Testinformation der Testmuster X durch Beobachtung der Ausgangsereignisse Y dar. In Analogie zur Kanalkapazität der Informationstheorie kann die Testkapazität $T_C(X/Y)$ für einen jeden funktionalen Modul definiert und berechnet werden.

$$T_C(X/Y) := \max\left\{T_R(X/Y)\right\} \qquad (7.2.17)$$

sie schwankt im Falle der losesten bzw. engsten Kopplung im Intervall:

$$0 \leq T_C(X/Y) \leq \left(\frac{m}{n}\right)^{\frac{1}{2}} \cdot \max\left\{T_m(Y/V)\right\} \qquad (7.2.18)$$

Die Relation (7.2.18) ist fundamental und kann durch keinen Modul unterbzw. überschritten werden. Das modifizierte Berger'sche Diagramm in Abb. 7.2.6 zeigt skizzenhaft den Testinformationstransport beim Transfer durch einen funktionalen Modul.

Nun ist in den meisten Fällen einer praktischen Testbarkeitsanalyse interessanter zu wissen, wieviel Testbarkeit beim Durchgang durch einen Modul infolge des 'information collapsing' vom Ein- zum Ausgang verlorengeht. Dem Berger'schen Diagramm kann sofort entnommen werden, daß dies die Summe aus der Äquivokatio und der Irrelevanz sein muß:

$$\left(n \cdot m\right)^{\frac{1}{2}} \cdot \Delta T(Y) := \begin{cases} n \cdot T(X/Y) + m\ T(Y/X) \\ n \cdot T_n(X/V) + m\ T_m(Y/V) - 2 \cdot \left(n \cdot m\right)^{\frac{1}{2}} \cdot T_R(X/Y) \end{cases} , \qquad (7.2.19)$$

sie ist ein Maß über die nutzlose und mehrdeutige Testinformation. zur Anbindung des Testbarkeitsmaßes an die Hardware-Beschreibungsebene des MIMOLA-Entwurfssystems ist es notwendig, für alle deklarierbaren Moduln die Transfercharakteristiken für die Symbolwahrscheinlichkeiten zu ermitteln, wobei ein Symbol ein Bitstring des Datenpfades der Breite n-Bit

mit $1 \leq n \leq 32$ darstellt. Ein funktionaler Baustein ordnet nun eindeutig jedem Ereignis der Eingangsvariablen X mit $p(x_i) = \text{prob}\{X = x_i \ \forall\ i\}$ ein Symbol des Ausgangsalphabets Y mit $p(y_j) = \text{prob}\{Y = y_j \ \forall\ j\}$ zu, woraus für die Elemente der bedingten Übergangsmatrix $p(y_j/x_i)$ folgen:

$$p(y_j/x_i) = \begin{cases} 1 & \text{für } \{X=x_i\} \mapsto \{Y=y_j\} \\ 0 & \text{sonst} \end{cases}, \qquad (7.2.20)$$

und somit für die Transfertestbarkeit operationaler Moduln:

$$T_R(X/Y) \stackrel{!}{=} \left[\frac{m}{n}\right]^{\frac{1}{2}} \cdot T_m(Y/V) = \left[\frac{n}{m}\right]^{\frac{1}{2}} \{T_n(X/V) - T(X/Y)\} \qquad (7.2.21)$$

Die Testbarkeit des Ausgangsknotens $T_m(Y/V)$ entspricht der durch den operationalen Modul transportierten Testinformation, solche Bausteine produzieren keine irrelevanten Testinformationen, sie sind beliebig kontrollierbar.

Für eine praktische Testbarkeitsanalyse funktional beschriebener Rechnerarchitekturen reduziert sich das Problem einer geeigneten Quantifizierung auf die Bestimmung der Ereigniswahrscheinlichkeitsverteilung aller möglichen Modulausgangsdatenvektoren, wobei sich allgemein für speicherlose Operatoren der Testbarkeitsverlust gemäß Gl. (7.2.19) ermitteln läßt:

$$\Delta T(Y) = \left[\frac{n}{m}\right]^{\frac{1}{2}} \cdot T_n(X/V) - \left[\frac{m}{n}\right]^{\frac{1}{2}} \cdot T_m(Y/V) \qquad (7.2.22)$$

7.2.4 Abgeleitete Großen

Neben detaillierten Informationen über Schwachstellen des Entwurfs interessieren den Schaltkreisdesigner vor den aufwendigen Fehlersimulationen und Testmustergenerierungen globale Aussagen über die zu erwartende Fehlerüberdeckung in Abhängigkeit der minimal erforderlichen Testmustersequenzlänge. Unter Ausnutzung des Fundamentaltheorems der Informationstheorie wird nun die Rückschluß- sowie die Fehlererkennungswahrscheinlichkeit auf interne Ereignisse für eine determinierte, fehlerfreie und

ungestörte Architektur in Abhängigkeit des oben eingeführten Testbarkeitsmaßes angegeben. Dazu wird ein Partitionierungsschnitt durch eine gegebene Rechnerarchitektur betrachtet und nach der Wahrscheinlichkeit gefragt, ein bestimmtes Ereignis dort kontrollieren und beobachten zu können.

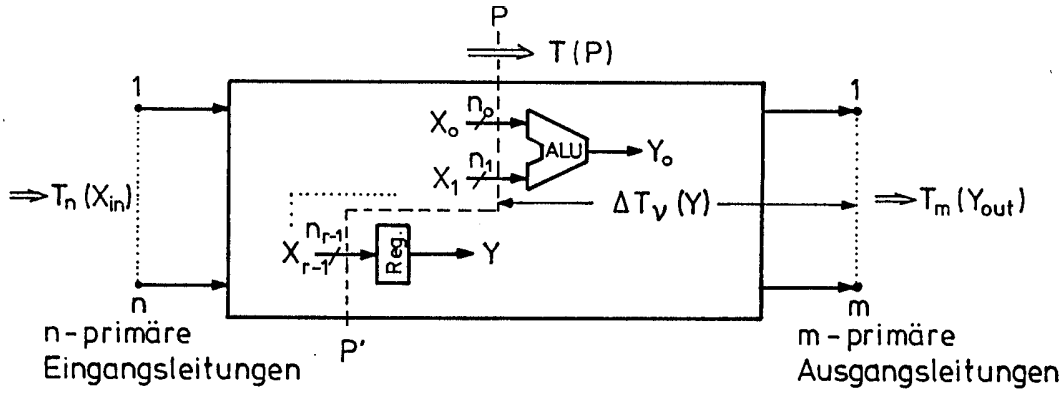


Abb.7.2.7: Interner Partitionierungsschnitt durch eine Rechnerarchitektur. Insgesamt werden r funktionale Leitungsbündel geschnitten

Beträgt die gesamte durch den Schnitt PP' transportierte Testentropie H(P), so überträgt jeder der einzelnen Knoten im günstigsten Fall $H(X_\nu) = \frac{1}{r} \cdot H(P)$ Knotenentropie. D.h., tritt auf dem Weg vom Eingang bis hin zum Schnitt PP' ein funktionaler Fehler z.B. am Knoten X_ν auf, so sind insgesamt $G_{X\nu} = 2^{N \cdot H(P)/r}$-Testvektorsequenzen fehlerhaft, die zwangsläufig auf andere Sequenzen abgebildet sein müssen. Fragt man nach der Fehlernichterkennungswahrscheinlichkeit eines internen Knotens X_ν, so errechnet sich diese aus dem Quotienten der Anzahl der günstigen Knotensequenzen, die die fehlerhafte Sequenz verdecken, zur Anzahl der möglichen Sequenzen des sonst fehlerfreien Knotens. Daraus errechnet sich die mittlere Fehlererkennungswahrscheinlichkeit für einen internen funktionalen Knoten der Architektur:

$$p_{e_\nu}(N) = \left[1 - 2^{-N \cdot T_{n_\nu}(X_\nu/V)}\right] \qquad (7.2.23)$$

Hieraus lassen sich zwei wichtige Resultate zwischen dem zu erwartenden Testgenerierungsaufwand und der Knotentestbarkeit $T_{n\nu}(X_\nu/V)$ ablesen:

1) Für eine geforderte interne Erkennungswahrscheinlichkeit $p_{e\nu}(N)=1$ steigt die notwendige Testvektorsequenzlänge N über alle Schranken hinaus.

2) N fällt umgekehrt proportional mit der Testbarkeit $T_{n\nu}(X_\nu/V)$ am Ausgang des ν-ten Moduls.

Diese Schlüsse stimmen inhaltlich mit dem "Prinzip der maximalen Entropie" überein, welches besagt, daß die Korrekturwahrscheinlichkeit am Ausgang eines gestörten Kanals dann am größten wird, wenn die Testinformationsrate $T_R(X/Y)$ so nahe wie möglich an die Testkapazität $T_C(X/Y)$ gesteigert wird.

Zur Ableitung von Gl. (7.2.23) wird stillschweigend vorausgesetzt, daß funktionale Fehler, die am internen Schnitt PP' in Erscheinung treten, unmittelbar beobachtet werden können. Dieser nur für in diskreter Technik aufgebaute Architekturen gültige Grenzfall gilt für integrierte Schaltkreise im allgemeinen nicht. Hier sind nur solche Fehler erkennbar, die durch Rückschluß von den primären Ausgangstestsequenzen als solche identifiziert werden können.

Gefragt wird nach der Wahrscheinlichkeit für den Eintritt eines sicheren Rückschlusses auf ein internes Ereignis durch Beobachtung der Ausgangstestsequenz $\{Y_{out}\}$, wenn genau N Testvektoren übertragen werden. Unter Verwendung von Gl.(7.2.22) folgt für die Rückschlußwahrscheinlichkeit $p_{RS\nu}(N)$ in Abhängigkeit des Testbarkeitsverlustes auf ein Ereignis am Knoten X_ν:

$$p_{RS_\nu}(N) \geq \left[2^{5/3\cdot(n_\nu\cdot m)^{1/2}/N}\right]^{-\Delta T_\nu(Y)} = \left[\text{const}^{-\Delta T_\nu(Y)}\right]^{N-1} \qquad (7.2.24)$$

Gl. (7.2.24) verdeutlicht, daß mit steigendem Testbarkeitsverlust beim Testdatentransfer durch eine irrelevanzfreie Architektur die Wahrscheinlichkeit stark abnimmt, mit der von einem beobachtbaren Ausgangsereignis $\{Y_{out} = y_{j_{out}}\}$ auf ein bestimmtes Ereignis $\{X_\nu = x_{i\nu}\}$ im Schnitt PP' zurückgeschlossen werden kann. Wird nun die Fehlerüberdeckung ('fault coverage') als das Stichprobenmittel über r interne Fehlererkennbarkeiten in Abhängigkeit der Testvektorsequenzlänge definiert, so folgt unter Beachtung obiger Tatsachen, daß alle erkennbaren funktionalen Fehler in der Regel nicht unmittelbar beobachtbar sind, für den Erwartungswert der 'fault coverage':

$$E[f(N)] = \overline{p_e(N)\cdot p_{RS}(N)} = \sum_{\nu=0}^{r_{ges}-1} p_{e_\nu}(N)\cdot p_{RS_\nu}(N)\cdot p\left[p_{e_\nu}(N)\cdot p_{RS_\nu}(N)\right], \quad (7.2.25)$$

wobei der zweite Produktterm in Gl. (7.2.25) die Wahrscheinlichkeit ausdrückt, mit der durch einen endgültigen Testvektorsatz infolge des 'information collapsing' die v-te Fehlererkennbarkeit zur Fehlerüberdeckung beiträgt. Diese Verteilung spiegelt die statistischen Bindungen der internen Testzustände wider, sie muß also existieren. Im Falle der Rekonvergenz- und Redundanzfreiheit einer Architektur stellt die Fehlerüberdeckung als Stichprobenmittel sämtlicher funktionaler Schaltungsknoten die untere Schranke dar, allgemein kann formuliert werden:

$$f(N) \geq \frac{1}{r_{ges}} \cdot \sum_{\nu=0}^{r_{ges}-1} p_{e_\nu}(N) \cdot p_{RS_\nu}(N) \qquad (7.2.26)$$

Für eine praktische Testbarkeitsanalyse auf dem strukturellen RT-Niveau stellt sich die zu losende Hauptaufgabe, für alle bekannten und üblichen Operationen die Abbildungsvorschriften für die Wahrscheinlichkeiten der Ereignisräume vom Ein- zum Ausgang eines Moduls in algorithmischer Form anzugeben.

7.2.5 Erweiterung des Entropiebegriffs auf Ereignissequenzen

Der Ausgang eines speicherfähigen Kanals kann als Informationsquelle identifiziert werden, aus dem eine Sequenz von Symbolen oder Ereignissen nach speziellen Regeln ausströmt, wobei allesamt zu einem definierten, endlichen Speicheralphabet gehören /14/. Die hier zu behandelnde Modulklasse der Informationsspeicher läßt sich im allgemeinen beschreiben durch zeitinhomogene, nichtstationäre Markov-Prozesse, deren zugehörige Übergangsmatrizen zu verschiedenen Zeitpunkten t^N, t^{N+1} ungleich sind:

$$\Pi^{N+1} \overset{!}{\neq} \Pi^{N} = \left[p(s_{jk}^{N}/s_{il}^{N-1})\right] = \begin{bmatrix} p(s_{oo}^{N}/s_{oo}^{N-1}) & \cdots & p(s_2 n_2 m^{N}/s_{oo}^{N-1}) \\ \vdots & \ddots & \vdots \\ p(s_{oo}^{N}/s_2 n_2 m^{N-1}) & \cdots & p(s_2 n_2 m^{N}/s_2 n_2 m^{N-1}) \end{bmatrix}. \qquad (7.2.27)$$

Die Elemente der Transitionsmatrix geben die Wahrscheinlichkeit wieder, daß sich zum t^N-ten Zeitpunkt das j-te Ereignis in der k-ten Zelle befindet unter der Bedingung, daß sich zum t^{N-1}-ten Zeitpunkt das i-te Datum in der l-ten Speicherzelle befunden hat.

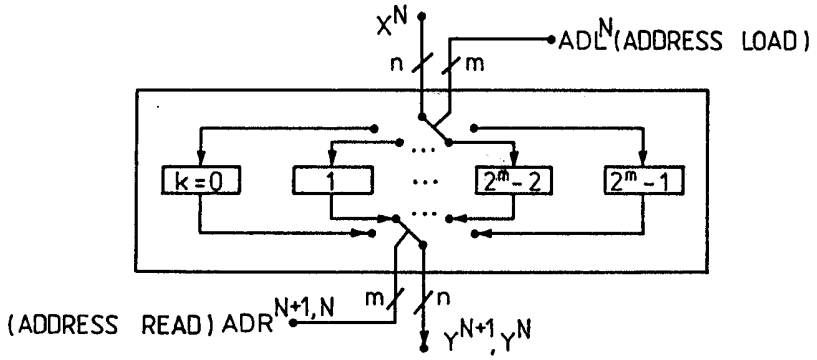


Abb.7.2.8: Speicherfähiger Modul (Kanal) mit 2^m- Speicherzellen

Jedes Ereignis der Variablen Y^N am Ausgang des Speichermoduls bezieht sich auf einen der möglichen internen Speicherzustände S, wobei Y^0 die Startvariable einer beliebigen, eindeutigen Anfangsverteilung am Ausgang des Kanals darstellt. Die Wahrscheinlichkeit für das j-te Ereignis am Ausgang des Speichers zum t^{N+1}-ten Zeitpunkt errechnet sich zu:

$$\text{prob}\{Y^{N+1}=y_j^{N+1}\} = p(y_j^{N+1}) = \bigcup_{k=0}^{2^m-1} \Big[\underbrace{p(x_j^{N+1},adl_k^{N+1})}_{s_{jk}^{N+1}} \cap \text{prob}\{ADR^{N+1} = adr_k^{N+1}\}\Big]. \qquad (7.2.28)$$

Der Ausgangsprozeß wird nun derart konstruiert, daß die Verteilung von Y^{N+1} nur von den Zuständen t^N, t^{N+1} und der Matrix Π^{N+1} abhängt, woraus sich die Markov-Eigenschaft allgemein ausdrücken läßt:

$$\text{prob}\{Y^{N+1} \;\forall\; s_{jk}^{N+1}/s_{jk}^0,\ldots,s_{jk}^N\} = \text{prob}\{Y^{N+1} \;\forall\; s_{jk}^{N+1}/s_{jk}^N\} \qquad (7.2.29)$$

Die zugehörige Transitionsmatrix Π^N, deren Elemente nichtnegativ und deren Zeilensummen gleich 1 sind, wird Markov- oder Stochastische Matrix genannt. Die Testbarkeit einer Ereignissequenz am Ausgang eines Speichermoduls kann nun sinnvoll definiert werden:

$$T_m\{Y\} := \lim_{N\to\infty} T_m(Y^{N+1}/Y^0,\ldots,Y^N) = \lim_{N\to\infty} \frac{1}{N+1} \cdot T_m(Y^0,Y^1,\ldots,Y^N), \qquad (7.2.30)$$

sie ist ein Maß für die Unbestimmtheit der Ausgangsvariablen Y^{N+1} unter der Voraussetzung, daß die gesamte Ereigniskette beobachtet worden war, über die der Zustand zum Zeitpunkt t^{N+1} erreicht wurde. Die Berechnung der Testbarkeit einer solchen Informationsquelle ist i.a. schwierig, jedoch

können für bestimmte Klassen von Markov-Matrizen Vereinfachungen ausgenutzt werden, wobei im Falle der Stationarität eine obere Schranke angegeben werden kann:

$$T_{m_{stat}}\{Y\} \leq T_m(Y^N/Y^0,\ldots,Y^{N-1}) \leq T_m(Y^N) \leq 1\ \text{pattern}^{-1} \quad . \qquad (7.2.31)$$

Die Testbarkeit der k-ten internen Speicherzelle errechnet sich definitionsgemäß zu:

$$T(X/ADL=adl_k) := \frac{1}{n} \cdot \sum_{j=0}^{2^n-1} p_{stat}(s_{jk}) \cdot ld \frac{1}{p_{stat}(s_{jk})} = T_{stat}(X), \qquad (7.2.32)$$

woraus sich die Testbarkeit aller Speicherzellen als Mittelung über sämtliche Adressenlesewahrscheinlichkeiten am Ausgang des Speichers ergibt zu:

$$T_{m_{stat}}\{Y\} = \sum_{k=0}^{2^m-1} p_{ADR}(k) \cdot T(X/ADL=adl_k) \stackrel{!}{=} T_{stat}(X) \qquad (7.2.33)$$

Im Falle der Eingangsereignisstationarität einer Datensequenz $X^0,\ldots,X^N$ ist nach ∞ - vielen Lade/Lesezyklen ein speicherfähiger Kanal transparent, d.h. die Testbarkeit der Ausgangssequenz ist so gut wie diejenige der Eingangsquelle. Diese Tatsache wird bei der Diskussion der abgeleiteten, rekursiv formulierten Transformationsalgorithmen noch anschaulich demonstriert, vergl. dazu auch Kap.7.3.

7.2.6 Testinformationsrate sreicherfähiger Kanäle

In Analogie zur axiomatischen Findung der Transfertestbarkeit (vgl. G1. (7.2.16)) beim übertragen von Testdaten durch kombinatorische Moduln kann desgleichen die Prozeßtransfertestbarkeit für sequentielle Bausteine formuliert werden, welche eine Aussage über die durch den Modul transportierten Informationen des Eingangsprozesses {X} macht durch Beobachtung der Speicherausgangssequenz {Y}. Werden auch hier wieder die Begriffe der bedingten bzw. Prozeßrückschlußtestbarkeit für die Ein/Ausgangssequenzen eingeführt, so ergibt sich formal der gleiche **Zusammenhang wie bei den speicherlosen Kanälen:**

$$T_R\left[\{X\}/\{Y\}\right] = \begin{cases} \lim\limits_{N\to\infty} \frac{1}{N+1} \cdot T_R\left[X^0,\ldots,X^N/Y^0,\ldots,Y^N\right] \\ \\ T_m\left\{Y/V\right\} - T\left[\{Y\}/\{X\}\right] \end{cases} \qquad (7.2.34)$$

Die Frage lautet, besteht auch wie bei den kombinatorischen Netzwerken für die speicherfähigen Moduln die Tatsache, daß die bedingte Prozeßtestbarkeit T({Y}/{X}) für die Ausgangssequenz {Y} bei bekannter Prozeßführung der Eingangssignalquelle {X} wieder zu null wird?

Da es sich bei den hier zu betrachtenden Prozessen um stochastische mit Markov-Eigenschaften handelt, lautet die Definitionsgleichung zur Beantwortung der obigen Frage:

$$T\left[\{Y\}/\{X\}\right] = \lim_{N\to\infty} T(Y^{N+1}/X^{N+1},Y^N) \quad , \qquad (7.2.35)$$

d.h. der Ausgangszustand des Speichers zum Zeitpunkt t^{N+1} hängt neben de aktuellen Eingangsverteilung nur vom vorausgegangenen Speicherzustand zu Zeit t^N ab. Es zeigt sich, daß bei der Berechnung der Elemente de Bedingungsmatrix die Lade/Lesewahrscheinlichkeiten für die Speicherzellen adressen die Transformationsvorschriften liefern, vgl. dazu di Gl. (7.2.36). Für k,i≠u sind die bedingten Wahrscheinlichkeiten gleic Null, für k,i=u sind die Elemente gleich Eins, diese liefern keine Beitrag zur bedingten Prozeßtestbarkeit.

$$p(y_k^{N+1}/x_i^{N+1},y_j^N) = \begin{cases} 1 - \left[1 - \sum\limits_{l=0}^{2^m-1} p_{ADL}(l)\cdot p_{ADR}(l)\right]^{N+1} & k,i=j \\ 0 & k,i\neq j \\ \left[\sum\limits_{l=0}^{2^m-1} p_{ADL}(l) \cdot \sum\limits_{\substack{h=0 \\ h\neq l}}^{2^m-1} p_{ADR}(h)\right]^{N+1} \text{ für} & k,j=u \\ 1 & k,i=u \end{cases} \qquad (7.2.36)$$

Somit ergibt sich für die bedingte Prozeßtestbarkeit unabhängig von den Lade/Lesewahrscheinlichkeiten der Speicherzellen der stationäre Endwert:

$$\lim_{N\to\infty} T\left[Y^{N+1}/X^{N+1},Y^N\right] = T_{stat}\left[\{Y\}/\{X\}\right] = 0 \qquad (7.2.37)$$

Nach hinreichend häufigem Laden/Lesen eines Speicherbausteins mit einer stationären Eingangsdatenquelle ist die Transfertestbarkeit $T_R\{[X]/[Y]\}$ durch den Modul gleich der Ausgangssequenztestbarkeit $T_m\{Y/V\}$, d.h. speicherfähige RT-Bausteine sind nach Anlegen von $N \to \infty$ unabhängigen Testvektoren beliebig gut kontrollier- und beobachtbar.

$$T_{R_{stat}}\{[X]/[Y]\} = T_{m_{stat}}[Y/V] = T_n[X/V] \quad ; \quad N \to \infty \qquad (7.2.38)$$

Werden die gefundenen Ergebnisse in die Gleichung für den Testbarkeitsverlust eingesetzt, folgt für den stationären Grenzwert mit m = n das Ergebnis:

$$\Delta T_{stat}[Y] = T_n[X/V] - T_{m_{stat}}[Y/V] = 0 \quad , \qquad (7.2.39)$$

wodurch die obigen Aussagen bestätigt werden.

In Kapitel 7.3 werden beispielhaft die Algorithmen für speicherfähige Bausteine dargelegt und interpretiert.

7.2.7 Stationäres Verhalten und Grenzwerte für die Testbarkeit

Während in den vergangenen Abschnitten die grundsätzliche Existenz eines stationären Speicherausgangszustands nachgewiesen wurde, soll in diesem Kapitel dargelegt werden, wie diese Ereignisverteilung dem transparenten, Grenzwert zustrebt. Dazu wird von einem stationären Eingangsalphabet eines allgemeinen Speicherbausteins ausgegangen, der einen festen, beliebigen internen Speicheranfangszustand aufweisen möge.

Die Speichertransformationsalgorithmen können für diesen Fall in Form der geometrischen Reihe aufsummiert und in geschlossener Form angegeben werden. Wird der Grenzübergang für $N \to \infty$ durchgeführt, ergibt sich das nun erwartete Ergebnis:

$$\lim_{N\to\infty} p(Y_j^N) = \begin{cases} p_{FCT}(READ)\cdot p(x_j) & \text{für } 0 \leq j \leq 2^n-1, u \\ 1-p_{FCT}(READ) & \text{für } j=z \end{cases} \qquad (7.2.40)$$

Interessieren nur die fortpflanzungsfähigen Ausgangszustände, dies sind sämtliche ohne den 'high impendance'-Zustand so kann G1. (7.2.40) durch folgenden Satz zusammengefast werden:

> Im Falle der Eingangsstationarität speicherfähiger Moduln werden solche nach hinreichend vielen Lade/Lesezyklen transparent, unabhängig von der internen Startzustandsverteilung. Für diesen Grenzfall geht der Testbarkeitsverlust beim Testinformationstransfer gegen Null!

In Abb. 7.2.9 ist die Ausgangstestbarkeit $T_m\{Y^{N+1}/V\}$ eines Speicherbausteins in Abhängigkeit seiner Kapazität dargestellt. Die Eingangstestbarkeit $T_n\{X/V\}$ liegt bei 100 % pattern^{-1}, was besagt, daß sämtliche validen Eingangsereignisse gleichwahrscheinlich auftreten.

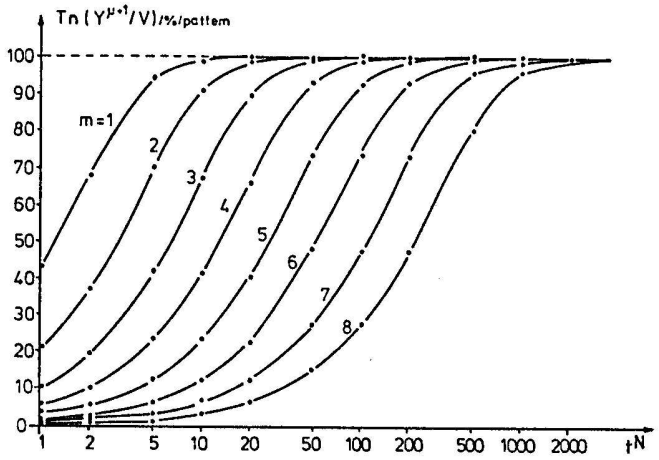


Abb.7.2.9: Speicherausgangstestbarkeit als Funktion der Anzahl der unabhängigen Testvektoren, die sequentiell an den Baustein angelegt werden. Parameter ist die Speicherzellengröße 2^m (m = Adreßbreite)

In der Praxis sind schon bei relativ kleinen Speicherkapazitäten von $\approx 2^8$-Speicherzellen notwendige Testzyklen von $N > 10^3$ erforderlich, um stationäre, d.h. auch optimale Testbarkeitswerte an den Ausgängen zu erzielen. Dies führt zu dem Schluß, daß Testbarkeitspfade, in denen adressierbare Speicherbausteine eingebettet sind, einen hohen Testzeitaufwand erwarten lassen. In dem Maße, wie sich nun die Testbarkeitswerte ändern und sich den für die analysierte Architektur charakteristischen Grenzwerten annähern, kann die sequentielle Tiefe des Schaltwerkes abgeschätzt werden.

7.2.8 Quellen und Senken statistischer Bindungen

Im weiteren werden auf die besonderen Methoden der Informationsverteilung und -umverteilung innerhalb einer Rechnerarchitektur und deren Auswirkungen auf die Gesamttestbarkeit eingegangen. Im Gegensatz zur logischen Beschreibungsebene, bei der eine Informationsleitung tatsächlich nur aus einem physikalischen Leiter besteht, beinhaltet die RT-Ebene komplexere Verteilungsmechanismen und Rekonvergenzeigenschaften, die statistische Abhängigkeiten auf dem RT-Niveau erzeugen. Sie treten dort auf, wo Ereignisräume einer Informationsquelle über ein Fan-out > 1 in mehrere Funktionspfade aufgespaltet werden, die dann allesamt voneinander abhängen. Erfolgt eine rekonvergente Schleife innerhalb der Rechnerarchitektur derart, daß zwei statistisch abhängige Ereignisverteilungen durch einen Operator verknüpft werden, müssen über eine entsprechende Verbundoperation die Eingangsräume in eine Bedingungsform gebracht werden, die die

Berechnung des Ausgangsereignisraumes gestattet. Die Hauptaufgabe bei der Behandlung statistisch gebundener Eingangsereignisse liegt in der Aufstellung der bedingten Eingangsdatenmatrix $[p(Z_1/Z_0]$. Die Berechnung dieser bedingten Wahrscheinlichkeiten gelingt nur dann, wenn sie auf die sie erzeugende Variable der statistischen Bindung zurückgeführt werden kann. Dazu wird eine allgemeine RT-Struktur betrachtet, an der die Verhältnisse konkret diskutiert werden können.

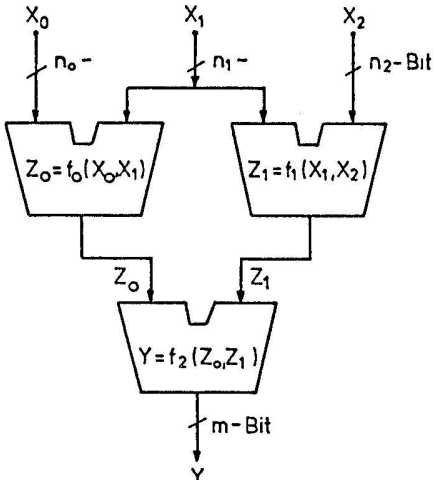


Abb.7.2.10: Kombinatorischer Netzwerkbaum zur Konstruktion einer allgemeinen statistischen Bindung zwischen den Variablen Z_0 und Z_1.

Allgemein kann zur Berechnung der Wahrscheinlichkeit des j-ten Ausgangsereignisses $\{Y=y_j\}$ folgender Zusammenhang angegeben werden:

$$p(y_j) = \sum_{\forall\, j=f(i,k)} p_o(z_i)\cdot p(z_{1k}/z_{oi}) \qquad (7.2.41)$$

Das Problem der Berechnung dieser Matrixelemente wird nun reduziert auf dasjenige der Ermittlung der bedingten Übergangswahrscheinlichkeiten der Operationen f_o und f_1. Somit erhält man die algorithmische Berechnungsvorschrift zur Bestimmung des Ereignisraumes des Ausgangsknotens Y eines rekonvergenten kombinatorischen Netzwerks:

$$p(y_j) = \sum_{\forall\, j=f(i,k)} \left[\sum_{\nu=0}^{2^{n_1-1},u} p_1(x_\nu)\cdot p(z_{oi}/x_{1\nu})\cdot p(z_{1k}/x_{1\nu}) \right] \qquad (7.2.42)$$

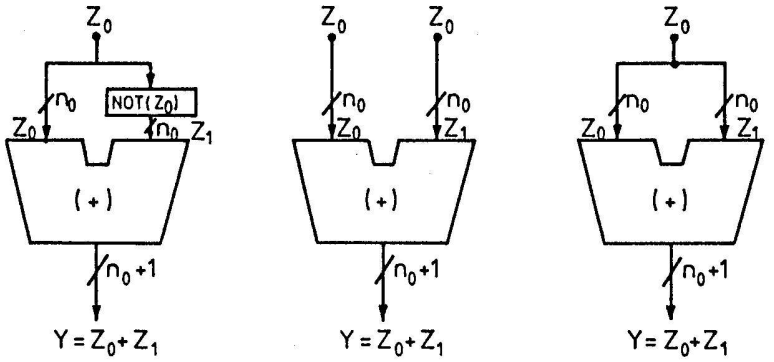
Die Gl. (7.2.42) ist für verschiedene, arithmetische und logische, rekonvergente Netzwerkbäume ausgewertet worden. Dabei zeigt sich, das die unter Nichtbeachtung der statistischen Bindungen ermittelten Testbarkeitswerte unter denen der exakt berechneten liegen, für einen weiten Bereich der Eingangsereignisverteilungen, unabhängig von den verwendeten Operatoren. Diese Resultate decken sich mit der Erfahrung, das redundante kombinatorische Netzwerke besser testbar erscheinen als solche mit logisch minimiertem strukturellen Aufbau. Im nächsten Abschnitt wird ein geeignetes Abschätzungsverfahren für die zu erwartende Abweichung vorgestellt. Die Rekonvergenzstruktur aus Abb. 7.2.10 ist ebenfalls anwendbar für Bäume, die mehrere horizontale Lagen enthält. Zunächst wird

dann die resultierende Funktion f_o und f_1 ermittelt, bevor die bedingten Übergangsmatrizen zur abhängigkeitserzeugenden Eingangsvariablen bestimmt werden. Dieses Verfahren erfordert einen enorm hohen Aufwand zur programmtechnischen Realisierung, wobei dessen Informationsgewinn gegenüber den Testbarkeitswerten, die ohne Beachtung der statistischen Bindungen ermittelt wurden, noch nachgewiesen werden muß.

7.2.9 Modulausgangstestbarkeit in Abhängigkeit des Korrelationskoeffizienten

Obwohl in der Praxis auf der Register-Transfer-Ebene statistisch abhängige Informationsquellen kombinatorisch nicht verknüpft werden, soll dieser Grenzfall dazu herangezogen werden, hier speziell für die analytisch handhabbare Additionsoperation, den Korrelationskoeffizienten in Beziehung

zur Modulausgangstestbarkeit zu setzen. Dazu werden drei signifikante Fälle betrachtet, statistische Unabhängigkeit, starke statistische und starke inverse statistische Abhängigkeit der Dateneingangsvariablen Z_0 und Z_1. Es kann gezeigt werden, daß die Testbarkeit T(Z/V) des Knotens Z und die Var[Z] des zugehörigen Ereignisraumes monoton verlaufen, wenn die Verteilungsdichte auf eine Normalverteilung rückführbar ist. Weiterhin ist die Varianz Var[Y] der Ausgangsereigniswahrscheinlichkeiten stark abhängig von der Konvarianzfunktion der Eingangsvariablen Z_0 und Z_1, und da T(Z) ~ Var[Z] ist, stellt diese Funktion somit einen Indikator dar für die zu erwartende Ausgangsknotentestbarkeit in Abhängigkeit der statistischen Bindung der Eingangsvariablen im Vergleich zur statistischen Unabhängigkeit. In Abb. 7.2.10 sind die Register-Transfer Realisationen wiedergegeben, in der zugehörigen Tabelle sind die Eingangsvariablen-eigenschaften für den Fall einer Exponentialverteilung mit $p_o(z) = \lambda_o/C_o \cdot \exp\{-\lambda_o \cdot z\}$ aufgelistet.



starke inverse- statistische Un- starke Abhängigkeit

Eingangsvariablen-eigenschaften	starke inverse Abhängigkeit	statistische Unabhängigkeit	starke statistische Abhängigkeit
bedingte Wahrscheinlichkeitsdichtefunktion: $p(z_{1k}/z_{0i})=$	$1\cdot\delta(z_{0i}\cdot z_{1k}-(2^{n_0}-1))$	$p_1(z_k)=\frac{\lambda_0}{C_0}\cdot e^{-\lambda_0\cdot z_{1k}}$	$1\cdot\delta(z_{0i}-z_{1k})$
Eingangstestbarkeit: $T(Z_0), T(Z_1)=$	$\frac{1}{n_0}\cdot \mathrm{ld}\ \frac{e}{\lambda_0}$	$\frac{1}{n_0}\cdot \mathrm{ld}\ \frac{e}{\lambda_0}$	$\frac{1}{n_0}\cdot \mathrm{ld}\ \frac{e}{\lambda_0}$
Varianz der Ereignisse: $Var[Z_0]=$	$\frac{1}{\lambda_0^2}$	$\frac{1}{\lambda_0^2}$	$\frac{1}{\lambda_0^2}$
Kovarianz: $Cov[Z_0,Z_1)=$	$-\frac{1}{\lambda_0^2}$	0	$\frac{1}{\lambda_0^2}$
Korrelationskoeffizient: $\rho(Z_0,Z_1)=$	-1	0	+1
Verbundtestbarkeit: $T(Z_0,Z_1)=$	$\frac{1}{2}T(Z_0)$	$T(Z_0)$	$\frac{1}{2}T(Z_0)$

Abb.7.2.11: Dyadischer Operator, der die Additionsoperation auf die statistisch gebundenen Operanden Z_0 und Z_1 anwendet.

Aus den Eigenschaften der Eingangsvaribalen bestimmen sich die charakteristischen Merkmale des Ausgangsereignisraumes Y zu:

Ausgangsereignis - eigenschaften	starke inverse Abhängigkeit		statistische Unabhängigkeit		starke statistische Abhängigkeit
Varianz: $Var[Y] = Var[Z_0 + Z_1] =$	0	<	$\frac{2}{\lambda_0^2}$	<	$2 \cdot \frac{2}{\lambda_0^2}$
Knotentestbarkeit: $T(Y) =$	0	<	$\frac{1}{n_0} \cdot (ld \frac{e}{\lambda_0} + \frac{C}{ln2})$	<	$\frac{1}{n} \cdot (ld \frac{e}{\lambda_0} + 1)$
Korrelationskoeffizient: $\varrho (Z_0, Z_1)$	-1	<	0	<	+1

Der Tabelle der Ausgangsereigniseigenschaften kann entnommen werden, daß die Knotentestbarkeit T(Y/V) proportional zum Korrelationskoeffizienten verläuft, für den Fall im weiten Sinne normalverteilter Eingangsereignisverteilungen. Aus den durchgeführten Studien über die Testbarkeit statistisch abhängiger kombinatorischer Netzwerke lassen sich zwei Erfahrungstatsachen unmittelbar ableiten:

1. Stark inverse statistische Abhängigkeiten treten in digitalen Rechnerarchitekturen praktisch nicht auf, da sie eine Kollabierung des Ausgangsereignisraumes bewirken, der, falls er erforderlich ist, hardwaremäßig wesentlich einfacher realisiert werden kann. Der Korrelationskoeffizient p zweier abhängiger Ereignisräume ist üblicherweise größer als Null; somit stellen die Testbarkeitsmaßzahlen, die unter Nichtbeachtung statistischer Bindungen ermittelt werden, in einer Architekturbewertung die obere Schranke für den zu erwartenden Testaufwand dar!

z. Beinhalten Rechnernetzwerke statistisch abhängige Rekonvergenzschleifen, so resultieren aus dieser Art Schaltungsredundanz stark erhöhte Modulausgangstestbarkeiten der betrachteten kombinatorischen Zweige. Anders formuliert heißt dies: Werden redundante Schaltungsressourcen wegoptimiert, so er**zielt man durch diese Maßnahme dieselbe Schaltkreistestbarkeit**.

7.2.10 Tragfähigkeit der Testvorbereitungsaussagen im Vergleich zu Fehlersimulationen

Das in Kap. 7.2 abgeleitete Testbarkeitsanalysemaß für funktional beschriebene Rechnerarchitekturen auf RT-Niveau wurde speziell als Bewertungsmaß zur Ableitung von Partitionierungskriterien entwickelt, welches kompatibel zum Hardware-Sprachumfang des MIMOLA-Entwurfssystems (MSS2) arbeitet, vgl. dazu Kap. 7.3.3. Das Maß erhebt den Anspruch, ohne auf die rechenzeitintensiven Logik- oder sogar Fehlersimulationen zurückgreifen zu müssen, verschiedene Hardware-Entwürfe untereinander bezüglich ihrer zu erwartenden Testbarkeit hin zu bewerten. Alle internen Ereigniswahrscheinlichkeiten der verschiedenen Knoten werden zur Beurteilung der Testbarkeit mit herangezogen. Obwohl das auf Auftrittswahrscheinlichkeiten basierende Maß unter Zuhilfenahme stochastischer Verfahren definiert ist, erfolgt die Fortschaltung und Manipulation der Knotenereignisse nach genauen deterministischen Regeln und Abbildungsvorschriften, es stellt kein Schätzverfahren im allgemeinen Sinne dar. Nach diesem Verfahren wird die Spur eines gesamten Eingangsereignisraumes quasi parallel durch die Architektur verfolgt, wobei seine Veränderungen beobachtet und bewertet werden.

Im Rahmen dieses Projektes konnte zur Verifikation die Hardware-Struktur eines asynchronen, gemischt kombinatorisch/sequentiellen Schaltwerks eines mittelständischen Halbleiterherstellers untersucht werden. Im einzelnen bewegt sich die Schaltkreiskomplexität bei ca. 60 Registern, 50 Invertierern, 100 Boole'schen und Arithmetischen Operatoren, 70 Multiplexerfunktionen sowie 500 funktionalen Testknoten mit unterschiedlichen Bitbreiten von (1...8)-Bit. Die CPU-Zeit auf einem Minicomputer DG-Eclipse MV/10000 beträgt für die Ermittlung der stationären Testbarkeitswerte ca. 1,5 Std. Dabei können alle potentiell schwer bzw. nicht testbaren Schaltkreisgebiete eindeutig lokalisiert werden. Durch Variation der Struktur sowie durch Hinzunahme eines zusätzlichen primären Eingangsknotens kann die Fehlerüberdeckung des gesamten Chip um 22 % auf insgesamt 87 ö gesteigert werden.

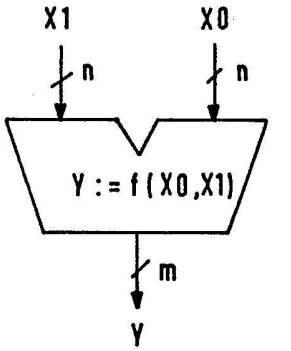
In den beiden folgenden Kapiteln werden nun speziell die Testbarkeitsalgorithmen dargelegt sowie theoretische Grundlagen für die **MIMOLAspezifische, iterative Testbarkeitsmethodik abgeleitet und abschließend einige praktische Beispiele aufgeführt.**

7.3 Testbarkeitsanalysealgorithmen

Die zentrale Aufgabe einer Testbarkeitsanalyse auf funktionaler Ebene besteht in der Ermittlung der Verteilungen der Musterwahrscheinlichkeiten aller internen Daten-, Adress- und Instruktionsknoten, beginnend an den primären Eingängen der Architektur mit fest vorgegebenen, beliebigen Verteilungen sukzessive fortschreitend bis hin zu den primär beobachtbaren Ausgängen. In diesem Kapitel werden exemplarisch für ausgewählte MIMOLA Registertransferoperatoren und -bausteine die Abbildungsvorschriften für die Eingangsmusterwahrscheinlichkeiten auf die Ausgangsmusterwahrscheinlichkeiten in algorithmischer Form angegeben.

7.3.1 Byte-Slice-Technik zur Komplexitätsreduktion

Allgemein kann dazu von einem Operationsmodul mit zwei Eingangsvariablen XO und X1 und einem Ausgangsereignis Y ausgegangen werden. Der Modul berechne die Funktion Y:=f(XO,X1). Der zugehörige Abbildungsalgorithmus für die Transformation der Ereigniswahrscheinlichkeiten lautet dann:



$$p(Y=j) = \sum_{i,k \ni j=f(i,k)} p(X0=i) \cdot p(X1=k)$$

Abb.7.3.1.1: Allgemeiner dyadischer Operationsmodul

Für die Berechnung der Ausgangsverteilung $\{p(Y=j)/j=0(1)2^m-1\}$ ergibt sich bei n-Bit breiten Eingangspfaden eine Berechnungskomplexität von der Ordnung $O(2^{2n})$. Solche Algorithmen lassen sich bis zu einer Pfadbreite von n=8-Bit auf herkömmlichen von Neumann-Rechnern effizient auswerten. Für Architekturen mit Pfadbreiten größer als 8 Bit, vorzugsweise 16 und 32 Bit, wurde ein Approximationsverfahren entwickelt und programmtechnisch implementiert, welches die Modultransferverteilungen praktisch gleichwirksam berechnet, das sogenannte Byte-Slice-Verfahren.

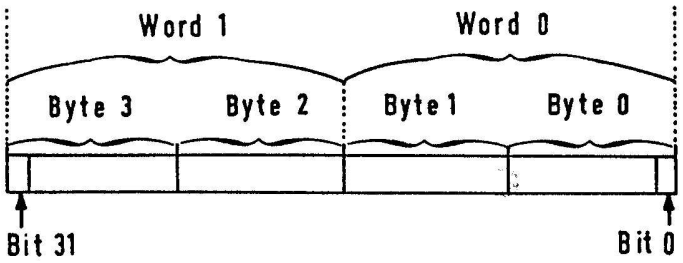
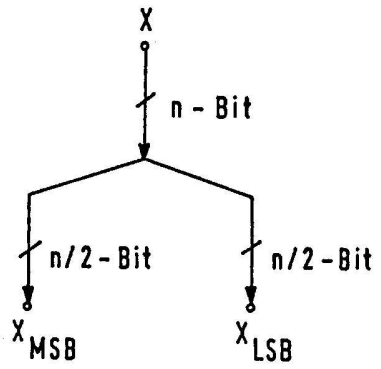


Abb.7.3.1.2: Byte/Word Slices für einen 32-Bit breiten Dateneingang

Benachbarte Bytes/Words kommunizieren über Carry/Borrow- beziehungsweise Overflow/Underflow-Wahrscheinlichkeiten; außer bei logischen Operatoren, bei denen generell keine Kommunikation zwischen benachbarten Bitzellen stattfindet. Bei Aufspaltung eines 16-Bit breiten Datenpfades mittels der FORK-Operation (Kap.7.3.3.3) ergeben sich die Wahrscheinlichkeitsverteilungen der Most und Least Significant Bytes (MSB und LSB) formal zu:



$$p(X_{MSB}=j) = \sum_{i \ni j=iDIV2^{n/2}} p(X=i);\ i=0(1)2^n-1$$

$$p(X_{LSB}=j) = \sum_{i \ni j=iMOD2^{n/2}} p(X=i);\ i=0(1)2^n-1$$

Abb.7.3.1.3: MSB/LSB-Verteilungen bei Aufspaltung eines n=16-Bit breiten Datenpfades

Die Inputverteilungen des Moduls liegen nun in laufgespaltener, Form vor. Dabei wird angenommen, daß sämtliche MSB's und LSB's untereinander statistisch unabhängig sind.

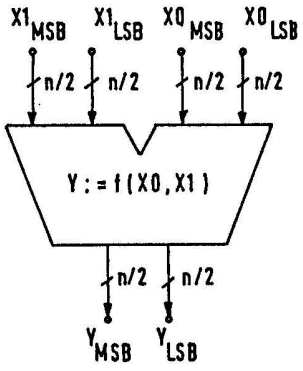
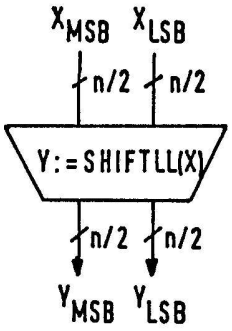


Abb.7.3.1.4: Aufspaltung der Modul-operationen in Teiloperationen für die einzelnen Segmente

Die Ausgangsverteilung Y kann jetzt sukzessive für die einzelnen Segmente Y_{LSB} und Y_{MSB} berechnet werden, wobei die Art der Lösung stark von der Abbildungsfunktion abhängt. Der Fehler, der bei dieser Form von Stichprobenverfahren gemacht wird, ist die Nichtbeachtung der statistischen Bindungen zwischen dem Most und dem Least Significant Byte bei 16-Bit Datenpfaden sowie dem Most und dem Least Significant Word bei 32-Bit Pfaden.

Beispielhaft soll der Byte-Slice-Mechanismus für die monadische "Shift Left Logical One Position"-Operation SHIFTLL(.) vorgestellt werden. Der zugehörige Modul schiebt einen n=16-Bit breiten String um eine Position nach links, unter Vernachlässigung des Übertrags. Der gesamte Abbildungsvorgang zerfällt in drei Teilschritte, wobei zunächst die LSB-Ausgangsverteilung, dann die interne overflowwahrscheinlichkeit und zum Schluß die MSB-Ausgangsverteilung ermittelt wird:



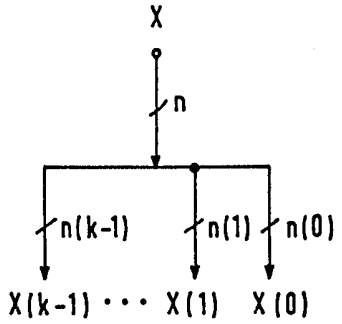
$$1)\quad p(Y_{LSB}=j) = \sum_{i \ni j=2i \bmod 2^n} p(X_{LSB}=i)$$

$$2)\quad p(OVR) = \sum_{i=2^{n-1}}^{2^n-1} p(X_{LSB}=i)$$

$$3)\quad p(Y_{MSB}=j) = \sum_{i \ni j=(2i \bmod 2^n)+1} p(X_{MSB}=i)\cdot p(OVR) + \sum_{i \ni j=2i \bmod 2^n} p(X_{MSB}=i)\cdot(1-p(OVR))$$

Abb.7.3.1.5: Byte-Slice-Technik für einen monadischen SHIFTLL-Operator

Treten in einer vorgegebenen Architektur ganz beliebige Datenpfade breiter als 8 Bits auf, so werden mittels der SPLIT-Operation (Kap.7.3.3.3) von links nach rechts in Datenflußrichtung jeweils ein Byte breite Teilstrings abgespalten, die in den nachfolgenden Testbarkeitsanalysealgorithmen für MIMOLA-Standardoperatoren und MIMOLA-Hardwarestrukturen als Komponenten eines Vektors aufgefaßt werden:



$$n = \sum_{i=0}^{k-1} n(i)$$

$$n(0)=n(1)= \ldots =n(k-2)= 8\text{-Bit},\ n(k-1) \leq 8\text{-Bit}$$

$$Vec(X,k):=[X(k-1), \ldots ,X(1),X(0)]$$

Abb.7.3.1.6: Komponentenzerlegung beliebiger Datenpfade breiter als 8 Bits

Ein allgemeiner dyadischer Modul in Komponentendarstellung erhält dann folgendes Aussehen:

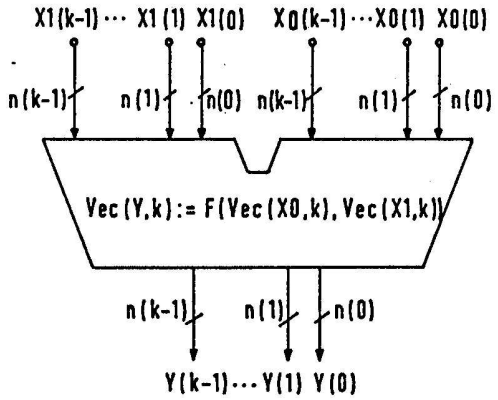


Abb. 7.3.1.7: Allgemeiner dyadischer Modul in Komponentendarstellung

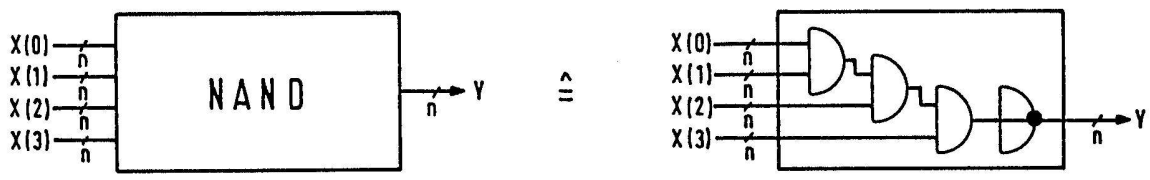
Somit ist es möglich, kompatibel zum Hardwarebeschreibungsumfang der Sprache MIMOLA, auch Rechnerarchitekturen größerer Bitbreite auf ihre funktional zu erwartende Testbarkeit hin zu analysieren und zu bewerten.

7.3.2 Algorithmen für MIMOLA-Standardoperatoren

In MIMOLA gibt es über fünfzig standardmäßig vordeklarierte Operatoren. Ein Operator steht für ein beliebiges kombinatorisches Netzwerk, das die bezeichnete Funktion realisiert. Die MIMOLA-Basisoperatoren, die Grundmenge solcher Netzwerke, lassen sich in fünf Funktionsgruppen aufteilen: arithmetische Operatoren, logische Operatoren, Vergleichsoperatoren, Schiebeoperatoren und selektive Operatoren. Je nach Grad des Operators unterscheiden wir einstellige (monadische), zweistellige (dyadische) und N-stellige (N-adische, N33) Operatoren.
Beispiele für monadische arithmetische Schaltfunktionen sind Inkrementieren und Absolutbetragbildung, für monadische logische Operatoren die bitweise Negation und für monadische Vergleichsoperatoren der Vergleich eines Operanden mit Null. Monadische Schiebeoperatoren schieben das Argument um genau eine Stelle (1-bit shifts). Dyadische arithmetische Operationen sind die vier Grundrechenarten -Addition, Subtraktion, Multiplikation und Division - , jeweils auf vorzeichenlosen und vorzeichenbehafteten Dualzahlen. Konjunktion und Disjunktion - beide auch negiert - , Äquivalenz und Antivalenz repräsentieren dyadische logische Operationen. Dyadische Vergleichsoperationen testen die Beziehung zwischen ihren Argumenten, zum Beispiel auf Gleichheit. Dyadische Schiebeoperationen, als Barrel-Shifter implementiert, schieben den Operanden um eine über den zweiten Eingang festgelegte Anzahl von Stellen (multibit shifts). Selektive Operatoren letztlich stellen Multiplexerfunktionen dar.

Beispiel 7.3.2.1: Interne Berechnung eines Vierfach-NAND-Operators

```
MODULE Nand4 (OUT res:(7:0); IN op1, op2, op3, op4:(7:0));
 PARBEGIN
  res <- "NAND"(op1, op2, op3, op4);
 PAREND;
```

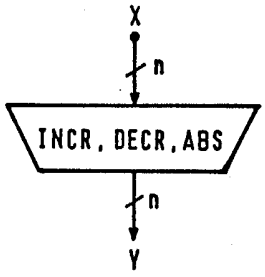


$$Y := \overline{X(0) \cdot \ldots \cdot X(3)} = \overline{X(0)} + \ldots + \overline{X(3)}$$

Assoziative arithmetische und logische Operationen - Addition und Multiplikation, Konjunktion und Disjunktion - lassen sich durch N-adische Operatoren realisieren, indem - die entsprechenden dyadischen Grundoperationen kaskadiert werden. Programmtechnisch läßt sich das Problem einfach so lösen, daß man die jeweilige Grundoperation als Prozedurparameter zusammen mit den N Eingangsbelegungen an eine globale Prozedur übergibt, die das Kaskadieren besorgt. N-adische Sheffer- und Piercefunktionen werden - funktional äquivalent - als kaskadierte dyadische Konjunktionen respektive Disjunktionen mit anschließender Negation implementiert, wie in obigem Beispiel dargestellt.
Im folgenden wird auf die arithmetischen und logischen MIMOLA-Operatoren ausführlich eingegangen. Auf eine Präsentation der Algorithmen für die Vergleichs- und Schiebeoperatoren soll hier wegen ihres Umfangs verzichtet werden.

7.3.2.1 Arithmetische Operatoren

Monadische arithmetische MIMOLA-Operatoren sind das **Vermehren um Eins ("INCR"), das Vermindern** um Eins ("DECR") und die **Bildung des Absolutbetrages** ("ABS")



$$INCR(X) := (X+1) \bmod 2^n$$

$$DECR(X) := (X-1) \bmod 2^n$$

$$ABS(X) := \text{if } X < 2^{n-1} \text{ then } X \text{ else } INCR(NOT(X))$$

Die Verteilungen an den Ausgängen der zugehörigen Operation Boxes' berechnen sich zu:

$$p(INCR(X)=j) = \begin{cases} p(X=2^n-1) & j=0 \\ \text{für} & \\ p(X=j-1) & j=1(1)2^n-1 \end{cases}$$

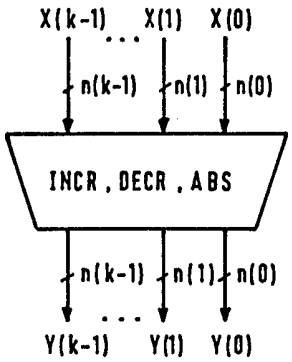
$$p(DECR(X)=j) = \begin{cases} p(X=j+1) & j=0(1)2^n-2 \\ \text{für} & \\ p(X=0) & j=2^n-1 \end{cases}$$

$$p(ABS(X)=j) = \begin{cases} p(X=j) & j=0;-2^{n-1} \\ p(X=j) + p(X=-j) \quad \text{für} & j=1(1)2^{n-1}-1 \\ 0 & j=-2^{n-1}+1(1)-1 \end{cases}$$

$$p(FCT(X)=u) = p(X=u) \quad \text{für} \quad FCT \in \{INCR,DECR,ABS\}$$

Der Parameter 'u=unknown' in der letzten Zeile bezeichnet einen unbekannten Zustand. Inkrementieren und Dekrementieren arbeiten mit positiven Dualzahlen; die Absolutbetragbildung ist für Zweierkomplementzahlen ausgelegt.

Bei Segmentierung des Eingangs und des Ausgangs in k Komponenten



$$Vec(Y,k) := FCT(Vec(X,k)); \quad FCT \in \{INCR,DECR,ABS\}$$

$$\sum_{j=0}^{k-1} n(j) = n$$

gilt definitionsgemäß:

$$INCR(Vec(X,k))_j := (X(j) + I(j)) \bmod 2^{n(j)}$$

wobei $I(j) :=$ if $j=0$ then 1 else $(X(j-1) + I(j-1)) \operatorname{div} 2^{n(j-1)}$ für $j=0(1)k-1$

$$DECR(Vec(X,k))_j := (X(j) - D(j)) \bmod 2^{n(j)}$$

wobei $D(j) :=$ if $j=0$ then 1 else if $X(j-1)<D(j-1)$ then 1 else 0 für $j=0(1)k-1$

$$ABS(Vec(X,k)) := \text{if } X(k-1)\geq 0 \text{ then } Vec(X,k) \text{ else } INCR(NOT(Vec(X,k)))$$

Inkrementieren und Dekrementieren sind durch ihre j-ten Teiloperationen definiert. Die Output-Wahrscheinlichkeitsverteilungen der einzelnen Operatoren lauten dann wie umseitig dargestellt:

$$p(INCR(Vec(X,k))_0=i) = p(X(0)=(i-1)\bmod 2^{n(0)})$$

$$p(INCR(Vec(X,k))_j=i) = p(X(j)=(i-1)\bmod 2^{n(j)}) \cdot \prod_{l=0}^{j-1} p(X(l)=2^{n(l)}-1) +$$

$$p(X(j)=i) \cdot \left(1 - \prod_{l=0}^{j-1} p(X(l)=2^{n(l)}-1)\right)$$

$$\text{für } i=0(1)2^{n(j)}-1,\ j=1(1)k-1$$

$$p(DECR(Vec(X,k))_0=i) = p(X(0)=(i+1)\bmod 2^{n(0)})$$

$$p(DECR(Vec(X,k))_j=i) = p(X(j)=(i+1)\bmod 2^{n(j)}) \cdot \prod_{l=0}^{j-1} p(X(l)=0) +$$

$$p(X(j)=i) \cdot \left(1 - \prod_{l=0}^{j-1} p(X(l)=0)\right)$$

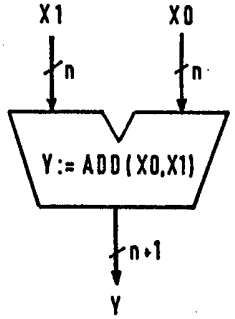
$$\text{für } i=0(1)2^{n(j)}-1,\ j=1(1)k-1$$

$$p(ABS(Vec(X,k))=i) = p(X(k-1)<0) \cdot p(INCR(NOT(Vec(X,k)))=i) + p(X(k-1)\geq 0) \cdot p(Vec(X,k)=i)$$

$$\text{für } i=0(1)2^{n}-1$$

$$p(FCT(Vec(X,k))=u) = p(Vec(X,k)=u) \text{ gilt allgemein für } FCT\in\{INCR,DECR,ABS\}.$$

Von den vier Grundrechenarten als Repräsentanten dyadischer arithmetischer MIMOLA-Operatoren soll hier stellvertretend die Addition vorzeichenlosen Dualzahlen behandelt werden. Die Algorithmen für die Subtraktion lauter ähnlich. Für die Multiplikation müssen die Eingangsbitbreiten au: Komplexitätsgründen auf sechzehn Bits begrenzt werden.



$$ADD(X0,X1) := X0 + X1$$

$$ADD: S^n \times S^n \rightarrow S^{n+1} \qquad \text{mit } S^n:=\{0,\ldots,2^n-1\}$$

"Sample Space"

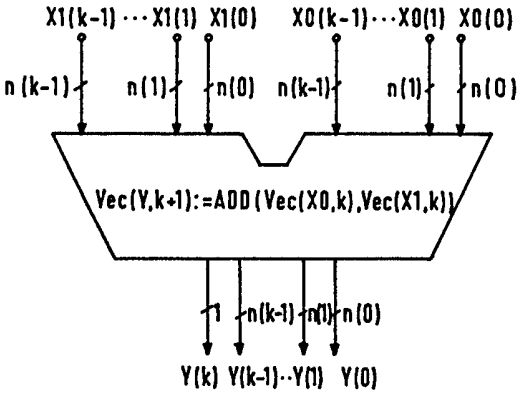
Die Eingangsereignisräume sind n-dimensional, der Ausgangsereignisraum (n+1)-dimensional. Zusätzlich wird noch gesondert die Fortpflanzung der Unknown-Wahrscheinlichkeit betrachtet.
Die Transformationsalgorithmen für die Wahrscheinlichkeitsverteilungen ergeben sich zu

$$p(ADD(X0,X1)=j) = \sum_{i=0}^{\min[2^n-1,\,j]} p(X0=i) \cdot p(X1=k); \quad \begin{array}{l} j=0(1)2^{n+1}-2, \\ k:=j-i=0(1)2^n-1 \end{array}$$

Für die Unknown-Wahrscheinlichkeit gilt

$$p(ADD(X0,X1)=u) = p(X0=u) + p(X1=u) - p(X0=u) \cdot p(X1=u)$$

Bei Addierern mit Portbreiten größer als acht Bits kommt die nachfolgende Komponentenzerlegung zur Anwendung. Die zusätzliche k-te Ausgangskomponente der Bitbreite Eins dient der Speicherung des Gesamtübertrages der Operation.



komponentenweises ADD:

$$\left(\bigtimes_{i=0}^{k-1} S^{n(i)}\right) \times \left(\bigtimes_{i=0}^{k-1} S^{n(i)}\right) \rightarrow \bigtimes_{i=0}^{k} S^{n(i)}$$

mit $S^{n(i)} := [0,\ldots,2^{n(i)}-1]$ und $n(k)=1$

Die Teiloperationen für die einzelnen Komponenten lassen sich folgendermaßen definieren

$$ADD(Vec(X0,k),Vec(X1,k))_j := (ADD(X0(j),X1(j))+C(j-1)) \bmod 2^{n(j)} \quad \text{für } j=0(1)k-1$$

$$ADD(Vec(X0,k),Vec(X1,k))_k := C(k-1)$$

wobei $C(j) :=$ if $j=-1$ then 0
else $(ADD(X0(j),X1(j))+C(j-1)) \operatorname{div} 2^{n(j)}$

Die zugehörigen Transformationsalgorithmen für die Verteilungen lauten dann

$$p(ADD(Vec(X0,k),Vec(X1,k))_j=i) = \sum_{(r,s)\in SET1} p(X0(j)=r)\cdot p(X1(j)=s)\cdot p(C(j-1)=0) + \sum_{(r,s)\in SET2} p(X0(j)=r)\cdot p(X1(j)=s)\cdot p(C(j-1)=1)$$

$$\text{für } i=0(1)2^{n(j)}-1, j=0(1)k-1$$

$$p(ADD(Vec(X0,k),Vec(X1,k))_k=i) = p(C(k-1)=i) \text{ für } i=0;1$$

wobei

$$SET1:=\{(r,s)/ADD(r,s)\bmod 2^{n(j)}=i\},\quad SET2:=\{(r,s)/ADD(r,s)\bmod 2^{n(j)}=i-1\}$$

Die darin auftretenden Wahrscheinlichkeiten für die internen Überträge berechnen sich zu

$$p(C(j)=1) = \text{if } j=-1 \text{ then } 0 \text{ else } p(ADD(X0(j),X1(j))\geq 2^{n(j)}) + p(ADD(X0(j),X1(j))=2^{n(j)}-1)\cdot p(C(j-1)=1)$$

$$p(C(j)=0) = 1 - p(C(j)=1)$$

Die Unknown-Wahrscheinlichkeit lautet für komponentenweises Addieren vorzeichenloser Dualzahlen

$$p(ADD(Vec(X0,k),Vec(X1,k))=u) = p(Vec(X0,k)=u) + p(Vec(X1,k)=u) - p(Vec(X0,k)=u)\cdot p(Vec(X1,k)=u)$$

Der Zustand eines Eingangsvektors gilt als unbekannt, wenn eine seiner Komponenten unbekannt ist. Abschließend sei noch angemerkt, das N-adische (N33) Addition durch Kaskadierung der entsprechenden dyadischen Operatoren unter Berücksichtigung der internen Oberträge implementiert wurde.

7.3.2.2 Logische Operatoren

Die standardmäßig vorgegebenen logischen MIMOLA-Operatoren sind in folgender Tabelle aufgeführt:

Operator	Operation	Bedeutung
"NOT"	Y := ¬ X0	Negation
"AND"	Y := X0 ∧ X1	Konjunktion
"OR"	Y := X0 ∨ X1	Disjunktion
"EQU"	Y := X0 ↔ X1	Äquivalenz
"XOR"	Y := X0 Ⓥ X1	Antivalenz
"NAND"	Y := X0 ⊼ X1	Shefferfunktion
"NOR"	Y := X0 ⊽ X1	Peircefunktion

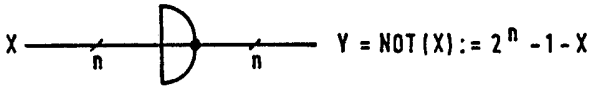
Tab. 7.3.2.2.1: Gruppe der logischen Operatoren in MIMOLA

Zwischen den logischen Operatoren gelten die algebraischen Beziehungen:

```
NEG(X0)      = NOT(X0) + 1
OR(X0,X1)    = NOT(AND(NOT(X0),NOT(X1)))
EQU(X0,X1)   = OR(AND(NOT(X0),NOT(X1)),AND(X0,X1))
XOR(X0,X1)   = OR(AND(NOT(X0),X1),AND(X0,NOT(X1)))
NAND(X0,X1)  = NOT(AND(X0,X1))
NOR(X0,X1)   = NOT(OR(X0,X1))
```

Die logischen Operatoren NEG, OR, EQU, XOR, NAND und NOR konnten somit äquivalent auf den monadischen Booleschen Operator NOT und den dyadischen Operator AND zurückgeführt werden. Die {NOT,AND}-Operatorenfamilie ist algebraisch in sich abgeschlossen. Der NOT-Operator bedeutet die Bildung des Einerkomplements, Neg die Bildung des Zweierkomplements. EQU steht für die logische Identität /15/.

Der monadische **NOT-Operator** ist definiert als die bitweise Negation seines Arguments



Seine Ausgangswahrscheinlichkeitsverteilung berechnet sich zu

$$p(NOT(X)=j) = p(X=2^n-(j+1));\ j=0(1)2^n-1$$
$$p(NOT(X)=u) = p(X=u)$$

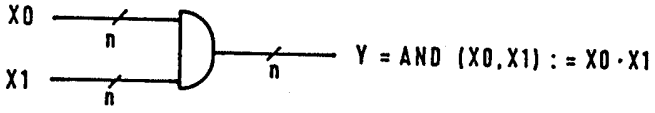
Da sie bitweise wirken, lassen sich die Booleschen Operatoren bei vorgegebener Segmentierung direkt auf die einzelnen Komponenten anwenden.,

Definition: $NOT(Vec(X,k))_j := NOT(X(j));\ j=0(1)k-1$

Verteilung: $p(NOT(Vec(X,k))_j=i) = p(NOT(X(j))=i);\ i=0(1)2^{n(j)}-1, j=0(1)k-1$

$$p(NOT(Vec(X,k))=u) = p(Vec(X,k)=u)$$

Der dyadische AND-Operator ist definiert als logisches Produkt seiner Argumente



Die zugehörige Verteilung für geringe Bitbreiten lautet

$$p(AND(X0,X1)=j) = p(X0=j)\cdot p(X1=j) + \sum_{i=j}^{[(2^n-1)+j]div2\ 2^n-1} \sum_{k=i+1} [p(X0=i)\cdot p(X1=k)+p(X0=k)\cdot p(X1=i)]$$

$$\text{für } [j/j\in[0,2^n-1] \text{ und } j=\sum_{v=0}^{n-1}(i_v\cdot k_v)\cdot 2^v],$$

$$\text{wobei } i:=\sum_{v=0}^{n-1} i_v\cdot 2^v,\ k:=\sum_{v=0}^{n-1} k_v\cdot 2^v \text{ gilt.}$$

Die unbekannten Zustände pflanzen sich wie folgt fort

$$p(AND(X0,X1)=u) = p(X0=u) + p(X1=u) - [p(X0=u)\cdot p(X1=u)+p(X0=0)\cdot p(X1=u)+ p(X1=0)\cdot p(X0=u)]$$

Für die Konjunktion in Komponentendarstellung gelten folgende Beziehungen:

$$\textbf{Definition: } AND(Vec(X0,k),Vec(X1,k))_j := AND(X0(j),X1(j))$$

$$\textbf{Verteilung: } p(AND(Vec(X0,k),Vec(X1,k))_j=i) = p(AND(X0(j),X1(j))=i)$$
$$\text{für } i=0(1)2^{n(j)}-1,\ j=0(1)k-1$$
$$p(AND(Vec(X0,k),Vec(X1,k))=u) = p(Vec(X0,k)=u)+p(Vec(X1,k)=u)-$$
$$[p(Vec(X0,k)=u)\cdot p(Vec(X1,k)=u)+ \prod_{j=0}^{k-1} p(X0(j)=0)\cdot p(Vec(X1,k)=u)+$$
$$\prod_{j=0}^{k-1} p(X1(j)=0)\cdot p(Vec(X0,k)=u)]$$

7.3.3 Algorithmen für MIMOLA-Hardwarestrukturen

Ein Registertransferbaustein mit einer oder mehreren Schaltfunktionen kann abstrakt als ein **schwarzer Kasten** (black box) realisiert werden. Das Verhalten eines schwarzen Kastens wird spezifiziert durch die Inputvariablen, die Outputvariablen und die Funktionen, die auf den Inputs ausgeführt werden müssen, um die Outputs zu produzieren. Ein schwarzer Kasten läßt sich also als Codeumsetzer mit n-stelligen Binärwerten am Eingang, m-stelligen Binärwerten am Ausgang auffassen:

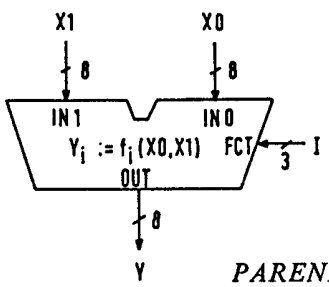
Abb. 7.3.3.1: RTL-Baustein als Codeumwandler

Jedes Objekt, das eine Information (Menge von Werten) erhält, manipuliert und weitergibt, benötigt dazu eine gewisse Menge von Bits. Bei MIMOLA-Hardwarestrukturen treten diese explizit als Feldbreite im Hardware Interface in Erscheinung.

Sei X die Input-Schnittstelle, Y die Output-Schnittstelle eines beliebigen RTL-Bausteins. X kann Werte aus der Menge {0,1,...,2^n-1,u}, Y Werte aus der Menge {0,1,,...,2^{m-1},u,z} annehmen. Der Parameter 'u' bezeichne einen unbekannten (unknown), 'z' einen hochohmigen (high impedance) Ausgangszustand. Die Verteilung am Eingang wird durch den Wahrscheinlichkeitsvektor {p(X=i);i=0,1,...,2^n-1,u} charakterisiert, die Verteilung am Ausgang durch den Wahrscheinlichkeitsvektor {p(Y=j);j=0,1,...,2^m-1,u,z}. Gesucht sind im folgenden Transformationsalgorithmen für sämtliche Bausteine der Registertransfer-Strukturebene: Operationsmoduln, Speicher und Kanäle, die das Abbildungsverhalten der Eingangsverteilungen (Eingangsinformation) auf die Ausgangsverteilungen (Ausgangsinformation) bei den Bausteinen beschreiben. Für die Multifunktionsbausteine kann dabei auf die bereits hergeleiteten Algorithmen für die einzelnen Operationen zurückgegriffen werden. Der wesentliche Unterschied von speichernden zu operationalen Moduln liegt in der Tatsache, daß bei speichernden Moduln die zeitlich nächste Ausgangsverteilung außer von der augenblicklichen Eingangsverteilung noch vom Modulzustand abhängt, was zu rekursiven Algorithmen führt. Die durch die Topologie einer Architektur bedingten Informationsumverteilungen kommen beispielsweise in den Algorithmen für die Aufspaltung oder Verzweigung von Datenpfaden zum Ausdruck.

7.3.3.1 Multifunktionsbausteine

Bausteine, die pro Programmschritt eine oder mehrere Operationen aus einer vorgegebenen Liste von Moduloperationen - Basisoperationen oder zusammengesetzte Operationen - ausführen können, heißen **Multifunktionsbausteine.** Multifunktionsbausteine sind kombinatorische Netzwerke (Schaltnetze) der Art:




```
MODULE Balu (OUT res:(7:0); IN op1, op2:(7:0); FCT ctr:(2:0));
  PARBEGIN
    res <- CASE ctr OF
             %000 :  op2 "-" op1;        %001 : op1 "-" op2;
             %010 :  op1 "AND" op2;      %011 : op1 "XOR" op2;
             %10X : TRISTATE  (* hochohmig *);
             ELSE : X         (* undefiniert *);
           END;
  PAREND;
```

Das arithmetisch/logische Netzwerk aus obigem Beispiel verfügt über zwei Dateneingänge und einen Datenausgang der Bitbreite acht sowie einen drei Bit breiten Instruktionseingang. Ober den Instruktionseingang lassen sich maximal acht verschiedene Instruktionen auswählen. Pro Instruktion wird bei diesem Baustein jeweils nur eine Operation ausgeführt.

Grundsätzlich lassen sich die Instruktionscodes in drei Kategorien einteilen:

1. Codes, die ausführbare Operationen anwählen.
2. Codes, die "high impedance" am Ausgang verursachen. Der Tri-State-Ausgangszustand ist nicht propagierbar.
3. Codes, die ins Leere gehen. 'X' steht für "no operation". Der Ausgangszustand ist unbekannt.

Sei 'oplist(m)' die Menge aller von einem beliebigen kombinatorischen Modultyp 'm' bereitgestellten ausführbaren Operationen. Dann gilt für die Instruktionswahrscheinlichkeitsverteilung:

$$\sum_{f_i \in oplist(m)} p(FCT=f_i) + p(FCT=TRISTATE) + p(FCT=NOOP) \overset{!}{=} 1$$

Kann ein und dieselbe Operation, einschließlich 'TRISTATE' und 'NOOP', unter mehreren Instruktionscodes ausgeführt werden, so werden deren Wahrscheinlichkeiten vorher aufaddiert. Nur die lesbaren Ausgangszustände tragen zur Fortpflanzung der Testbarkeitsinformation bei. Die Lesewahrscheinlichkeit am Ausgang ergibt sich zu:

$$p(FCT=READ) = 1 - p(FCT=TRISTATE)$$

Die auf die lesbaren Zustände normierte, propagierbare Wahrscheinlichkeitsverteilung am Ausgang Y eines beliebigen Multifunktionsbausteins mit den Eingängen X0, X1, ... lautet dann:

$$p(Y=j) = \frac{1}{p(FCT=READ)} \cdot \sum_{f_i \in oplist(m)} p(FCT=f_i) \cdot p(f_i(X0,X1,...)=j); \quad j=0(1)2^n-1$$

$$p(Y=u) = \frac{1}{p(FCT=READ)} \cdot \left\{ \sum_{f_i \in oplist(m)} p(FCT=f_i) \cdot p(f_i(X0,X1,...)=u) + p(FCT=NOOP) \right\}$$

Die Verteilungen der einzelnen Funktionen aus der Operationsliste des Moduls werden mit Hilfe der in den beiden vorangehenden Kapiteln hergeleiteten Algorithmen für MIMOLA-Standardoperatoren berechnet. Bei Komponentenzerlegung der Ein- und Ausgänge eines Multifunktionsbausteins läßt sich obiger Algorithmus direkt auf die einzelnen Komponenten übertragen.

7.3.3.2 Schreib-Lese-Speicher

Speicheralgorithmen für Schreib-Lese-Speicher (Random Access Memories oder kurz RAMs) beziehen sich grundsätzlich auf Speicherausgangs- oder Leseports, für die in jedem Iterationsschritt neue Verteilungen berechnet werden müssen. Je nach Anzahl der zugehörigen Speichereingangs- oder Schreibports lassen sich die Algorithmen klassifizieren in

Algorithmen für Speicher mit einem Eingangsport (uni- oder bidirektional).

Algorithmen für Speicher mit zwei und mehr Eingangsports.

Speicher mit einem Eingangsport

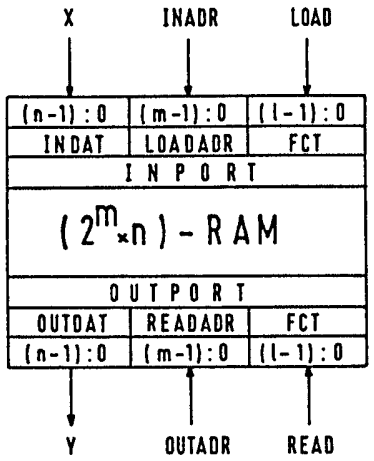


Abb. 7.3.3.2.1:
Speichermodul mit einem Schreib- und einem Leseport

```
MODULE Sram
 PORT inport  (IN   dat:(31:0); ADR addr:(7:0); FCT ctr:(0));
 PORT outport (OUT dat:(31:0); ADR addr:(7:0); FCT ctr:(0));
  PARBEGIN
   WITH inport DO
    CASE ctr OF
      0 :    Sram(addr) := dat;
      1 :    (* noload *);
    END;
   WITH outport DO
    CASE ctr OF
      0 :    dat <- Sram(addr);
      1 :    dat <- TRISTATE;  (* noread *)
    END;
  PAREND;
```

Bei Anlegen einer Null an den jeweiligen Kontrolleingängen des obigen Beispiels erfolgt über den Dateneingang *inport_dat* das Schreiben eines Wertes in den Speicher, über den Datenausgang *outport_dat* das Lesen eines Wertes aus dem Speicher. Eine Eins an den Kontrolleingängen unterdrückt jede Aktion.

In MIMOLA werden folgende vier **Speicheroperationen** unterschieden:

LOAD -lade den Inputwert in eine Speicherzelle,
READ -weise den Speicherzelleninhalt dem Outputport zu,
NOLOAD - lade nicht (deklariert durch ein leeres Statement),
NOREAD - setze den Outputport zu 'tri-state' (high impendance').

Bei einem Speicher mit einem Eingangsport lautet der Algorithmus für die Transformation der Verteilung zwischen Eingang und Ausgang:

$$p(Y_{T+1}=i) = (1 - p(FCT=LOAD)\cdot p(EQ(INADR,OUTADR)=1)) \cdot p(Y_T=i) + p(FCT=LOAD)\cdot p(EQ(INADR,OUTADR)=1) \cdot p(X_{T+1}=i)$$

$$\text{für } i=0(1)2^n-1 \text{ und } i=u$$

$$p(Y_{T+1}=z) = 1 - p(FCT=READ)$$

Nach Definition des EQUAL-Operators gilt für die Gleichheit der Lade- und Leseadressen:

$$p(EQ(INADR,OUTADR)=1) = \sum_{k=0}^{2^m-1} p(INADR=k)\cdot p(OUTADR=k)$$

Die aktuelle Wahrscheinlichkeitsverteilung am Ausgang eines Speichers mit einem Eingangsport zum Zeitpunkt T+1 setzt sich zusammen aus der aktuellen Verteilung am Dateneingang, unter der Annahme, daß der Inputwert direkt zum Ausgang durchgeschaltet wird, und der Verteilung, die zum vorangegangenen Zeitpunkt T vorlag. Schon relativ kleine Speicher werden im Falle stationärer Eingangsereignisverteilungen erst nach einigen tausend Lade/Lesezyklen transparent; d.h. die Speicherausgangsverteilung entspricht dann der Eingangsdatenverteilung, unabhängig von der internen Startzustandsverteilung des Speichers. Die iterative Testbarkeitsanalyse **geht von unbekannten** Speicheranfangszuständen aus. Daraus läßt sich folgern, daß Architekturpfade, die Speichermodule enthalten, einen hohen zeitlichen Testaufwand erfordern werden. Ein Ziel der iterativen Testbarkeitsanalyse wird es deshalb sein, wenn möglich, solche Pfade durch deterministische Belegungen an den primären Eingängen oder den Mikroprogrammspeicherausgangsfeldern zu isolieren.

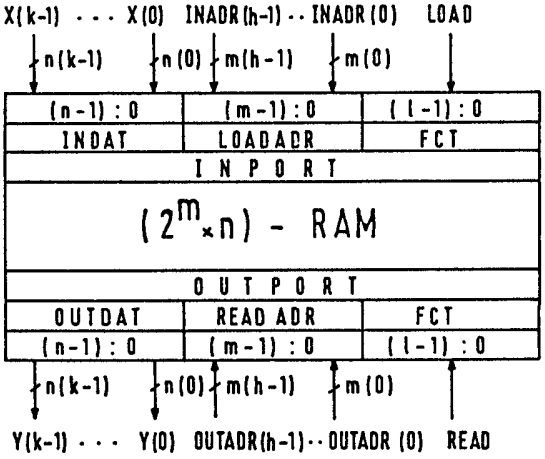


Abb. 7.3.3.2.2:
Mehr-Komponenten-Darstellung eines Speichermoduls mit einem Schreib- und einem Leseport

Bei Segmentierung der Daten- und Adreßanschlüsse von Eingangs- und **Ausgangsport läßt sich obiger Algorithmus** direkt übertragen:

$$p(Y_{T+1}(j)=i) = (1 - p(FCT=LOAD)\cdot\prod_{v=0}^{h-1} p(EQ(INADR(v),OUTADR(v))=1))\cdot p(Y_T(j)=i)$$
$$+ p(FCT=LOAD)\cdot\prod_{v=0}^{h-1} p(EQ(INADR(v),OUTADR(v))=1)\cdot p(X_{T+1}(j)=i)$$
$$\text{für } i=0(1)2^{n(j)}-1 \text{ und } i=u,\ j=0(1)k-1$$

$$\text{wobei}\quad p(EQ(INADR(v),OUTADR(v))=1) = \sum_{k=0}^{2^{m(v)}-1} p(INADR(v)=k)\cdot p(OUTADR(v)=k)$$

Speicher mit zwei und mehr Eingangsports

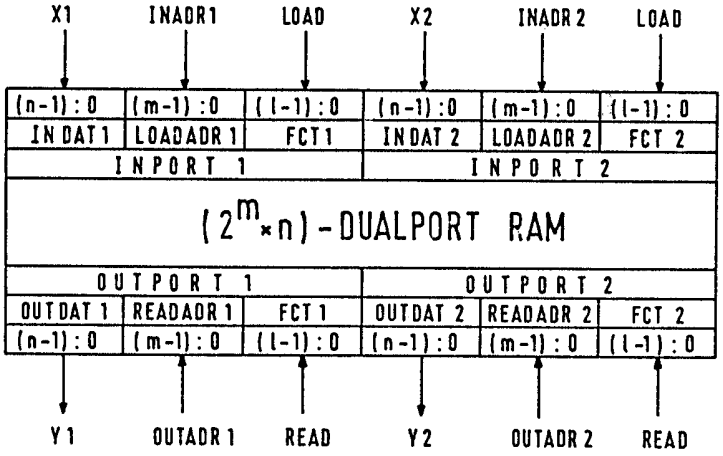


Abb. 7.3,3.2,3:
Speichermodul mit zwei Schreib- und zwei Leseports

```
MODULE S2x2portram
 PORT p1, p2 (IN    x:(7:0); ADR ref:(3:0); FCT ctr:(0));
 PORT p3, p4 (OUT y:(7:0); ADR ref:(3:0));
  PARBEGIN
   WITH p1, p2 DO
    CASE ctr OF
      0 : S2x2portram(ref) := x;
      1 : (* no_op *);
    END;
   WITH p3, p4 DO
    y <- S2x2portram(ref);
  PAREND;
```

Analog zum gewöhnlichen RAM mit einem Eingangs- und einem Ausgangsport berechnet sich die Verteilung an einem beliebigen Ausgangsport Y eines Multiportspeichers, in Abhängigkeit von den Verteilungen an allen Eingangsports und der Verteilung aus dem letzten Zeitschritt, rekursiv zu:

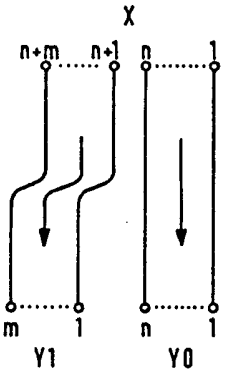
$$p(Y_{\tau+1}=i) = \left(1 - \sum_{j=1}^{\#inports} p(FCT(j)=LOAD)\cdot p(EQ(INADR(j),OUTADR)=1)\right)\cdot p(Y_\tau=i) + \sum_{j=1}^{\#inports} p(FCT(j)=LOAD)\cdot p(EQ(INADR(j),OUTADR)=1)\cdot p(X(j)_{\tau+1}=i)$$

$$\text{für } i=0(1)2^n-1 \text{ und } i=u$$

Der Parameter '#inports' bezeichnet die Anzahl der Eingangsports. Eine ausführliche theoretische Ableitung der Speicheralgorithmen befindet sich in /16/,

7.3.3.3 Leitungsbündel und Busse

Adress-, Daten-, und Kontrollbusse, d.h. Leitungsbündel allgemein, stellen auf Registertransferniveau funktionale Datenpfade dar. Für eine Architekturbeschreibung müssen Teile solcher Datenpfade abgespalten oder miteinander verschmolzen werden können. Die SPLITLEFT-Operation berechnet die Wahrscheinlichkeitsverteilung auf dem (in Datenflußrichtung gesehen) linken abgespaltenen Ast eines Datenpfades, SPLITRIGHT auf dem rechten Ast, wenn dieser oder jener laut MIMOLA-Verbindungsliste weiter betrachtet werden soll:

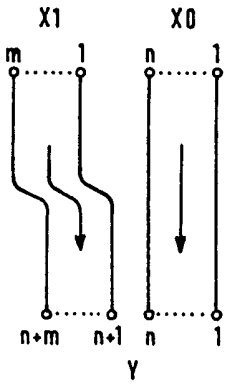


$$p(Y0=j) = \sum_{i \ni j=i \bmod 2^n} p(X=i);\ i=0(1)2^{n+m}-1 \quad \text{"SPLITLEFT"}$$

$$p(Y1=j) = \sum_{i \ni j=i \operatorname{DIV} 2^n} p(X=i);\ i=0(1)2^{n+m}-1 \quad \text{"SPLITRIGHT"}$$

$$p(Y0=u) = p(Y1=u) = p(X=u)$$

Die Operationen werden häufig für das Abtrennen von Statusbits an Modulausgängen verwendet. SPLITLEFT und SPLITRIGHT zusammengefaßt entsprechen der **FORK-Operation** /17/ für das Aufspalten funktionaler Datenpfade. Die komplementäre Operation dazu, die JOIN-Operation, verschmilzt zwei Teilpfade zu einem einzigen. Nach einem solchen Prozeß ergibt sich als Ausgangsverteilung:



$$p(Y=k) = \sum_{i,j \ni k=i \cdot 2^m+j} p(X0=i) \cdot p(X1=j);\ i=0(1)2^n-1,\ j=0(1)2^m-1$$

$$p(Y=u) = p(X0=u) + p(X1=u) - p(X0=u) \cdot p(X1=u) \quad \text{"JOIN-Operation"}$$

Die JOIN-Operation kommt besonders oft am Eingang monadischer Operationsmoduln vor, wo zwei Datenworte zunächst vereinigt und dann der Operation zugeführt werden müssen. MIMOLA-Anweisungen wie die Aneinanderkettung **(Konkatenation) zweier Bitbereiche lassen sich damit bearbeiten.**

7.4 Das iterative Testbarkeitsanalyseverfahren

7.4.1 Theoretische Grundlagen

Allgemein gesprochen beschreibt das MIMOLA System mikroprogrammgesteuerte, synchron arbeitende Rechnerstrukturen auf einem Systemniveau, wo Zustandsübergänge ('state transitions') zwischen speichernden Bauelementen stattfinden - dem **Registertransferniveau** (RTL ^-_ register transfer level). Ein Datenpfad auf RT-Ebene hat prinzipiell folgendes Aussehen:

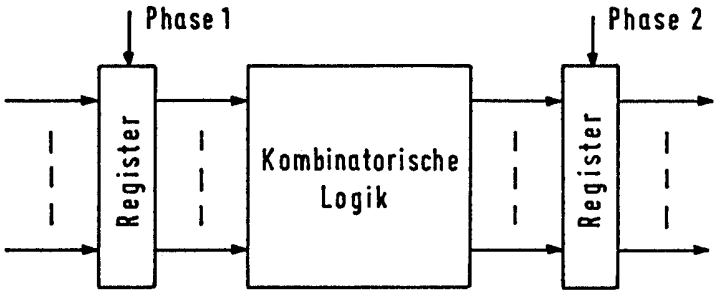


Abb. 7.4.1.1: Datenpfad auf Registertransferniveau

Daten, die sich während Phase 1 im Inputregister befinden, pflanzen sich dann in und durch das kombinatorische Schaltwerk fort und werden während Phase 2 im Outputregister gespeichert. Alle hardwaremäßig möglichen Registertransferoperationen dieser Art, d.h. alle Zustandsübergänge innerhalb einer Architektur, werden programmintern auf Hardwarebäume abgebildet. Hardwarebäume repräsentieren den Datenfluß pro Taktzyklus. Blätter und Wurzeln der Bäume bilden die oben erwähnten Zustandsfelder. **Testbarkeitsbäume** sind erweiterte Hardwarebäume, deren Knoteninhalte den Erfordernissen der iterativen Testbarkeitsberechnung angepaßt wurden (Stichwort **"Testbarkeitsknoten"**, Kap.7.4.1.1).

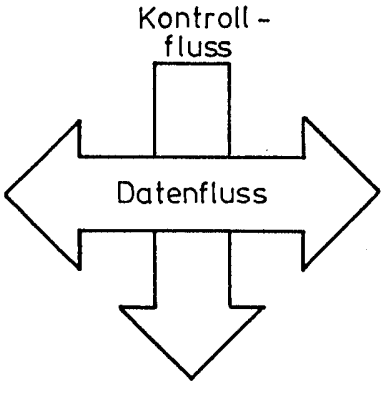


Abb. 7.4.1.2:
Informationsfluß ≙ Daten- und Kontrollfluß

Idealisiert kann ein Architekturblock als ein Kreuzungspunkt zweier Arten von Informationsfluß aufgefaßt werden /18/: erstens des Datenflusses, der sich längs der Hardwarebäume dynamisch fortpflanzt - und zweitens, orthogonal dazu, des Kontrollflusses, der starr vom Mikroprogrammspeicher zu den einzelnen Hardwaremoduln geht.
Die Größe des Datenflusses wird durch die Auftrittswahrscheinlichkeiten interner Knotenereignisse repräsentiert. In Relation zu den gewählten Eingangsmustern und des den Datenfluß steuernden Instruktionsspeicherinhaltes lassen sich die Testbarkeitsverluste an den internen Knoten bestimmen. Als Meßvorschrift dient die Berechnung des mittleren Informationsgehaltes jedes funktionalen Knotens. Die Testbarkeit eines Schaltkreises ergibt sich insgesamt als eine Funktion der Schaltkreistopologie, des operationellen Verhaltens der einzelnen Moduln, der Firmware und der Eingangsvektorverteilung.
Bei der Initialisierung, vor dem ersten Durchlauf der iterativen Testbarkeitsanalyse, werden alle Register zurückgesetzt, sämtliche festverdrahteten Konstanten zugewiesen und die Speicherausgangsports, ebenso wie die primären Ausgänge der Architektur, mit dem Zustand UNKNOWN belegt. Die Verteilungen für die an den primären Eingängen anliegenden **Testvektoren** sind frei wählbar. Beliebige Gleich-, Gauss-, Wortschatz-, Pseudozufalls- und Zufallsverteilungen oder aber deterministische Werte können dort wahlweise angelegt werden. Gleichverteilung an einem Knoten bedeutet definitionsgemäß eine maximale Testbarkeit von einhundert Prozent pro Testmuster, die Abwesenheit unbekannter und hochohmiger Zustände vorausgesetzt. Die Gauss- oder Normalverteilung kann mittels ihrer Standardabweichung und ihres Mittelwertes so variiert werden, daß sich praktisch alle Verteilungsformen, von der singulären Verteilung oder Ereignissicherheit mit einer Testbarkeit von null Prozent pro Muster bis zur Gleichverteilung einstellen lassen. Die Wortschatzverteilung mag als Testmusterverteilung für mit MIMOLA projektierte Sprachprozessoren zur Anwendung kommen. Pseudozufallsverteilungen werden durch softwaremäßig nachgebildete Signaturregister /19/ erzeugt. Die Koeffizienten der zugehörigen irreduziblen Generatorpolynome /20/ lassen sich insbesondere auch manuell setzen. Zusätzlich liefert ein vom System bereitgestellter Zufallsgenerator Werte, die über dem Intervall (0,1) gleichverteilt sind.

Aufgrund der auf RT-Niveau beschreibbaren Speicherfähigkeit einzelner **Moduln kann bei einem** einmaligen 'Scan' über alle internen Knoten keine endgültige Knotentestbarkeitsaussage gemacht werden. Je nach sequentieller

Tiefe einer Schaltung sind die zu berechnenden Knotenentropien mehr oder weniger stark zeitabhängig. Dieser Tatsache wird in Form einer **iterativen Analyse** Rechnung getragen. Die Iteration beginnt jeweils an den Ausgängen von Zustandsmoduln mit entsprechend gewählten Startverteilungen, zum Beispiel mit dem Unknown-Zustand für adressierbare Speicher, und endet mit der Berechnung von Eingangsverteilungen für speicherfähige Kanäle. Aus den rekursiven Speicheralgorithmen können dann die Verteilungen für, den nächstfolgenden Iterationsschritt bestimmt werden. Speicherbausteine werden nach einer gewissen Testzyklenzahl, abhängig von der Topologie und den Speicherkapazitäten der Gesamtarchitektur, beliebig gut kontrollierbar und beobachtbar, mit anderen Worten "transparent". Die Knotenentropien konvergieren hin zu stationären Endwerten ('steady state'). Architekturen mit hohen Ausgangsentropien und kurzer Iterationszeit gelten generell als gut testbar. Die Testbarkeitsmaßzahlen geben Anhaltspunkte für Testflaschenhälse und für eventuell notwendige Architekturpartitionierungen. Inhaltlich stimmt die dargestellte Teststrategie mit dem "Prinzip der maximalen Entropien überein, welches besagt, daß die Kontrollierbarkeit und **die Beobachtbarkeit interner Schaltungsknoten** genau dann maximiert wird, wenn der Informationsfluß durch das Schaltwerk optimiert wird.

In den folgenden Kapiteln soll auf zwei grundlegende Begriffe der **iterativen** Testbarkeitsanalyse - den Testbarkeitsknoten und den Testbarkeitsbaum - näher eingegangen werden.

7.4.1.1 Der Testbarkeitsknoten

Knotenpunkte einer Architektur unter Testbarkeitsgesichtspunkten stellen die Ausgänge der einzelnen Hardwaremoduln dar. Diese müssen bei der iterativen Testbarkeitsanalyse nach jedem Zustandsübergang mit Testbarkeitsmaßzahlen belegt werden können. Das interne Datenformat eines Testbarkeitsknotens muß daher so ausgelegt sein, daß es sämtliche zur Berechnung dieser Maßzahlen notwendigen Großen umfaßt. Zudem soll es sich an der Syntax der MIMOLA-Hardwarebeschreibung orientieren. Das nachfolgende Bild gibt die Datenstruktur der Hardwarekomponenten in MIMOLA am Beispiel eines Speichers wieder:

	MIMOLA-Identifikatoren	Strukturbaum
Gruppe	*S*	
Modultyp/Port	*Smain-port2*	
Exemplar	*Smain-port2_ram1*	
Subport	*Smain-port2_ram1&output*	
Bitbereich	*Smain-port2_ram1&output.(7:4)*	

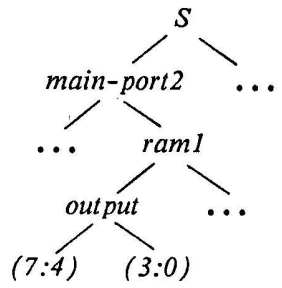


Abb. 7.4.1.1.1: MIMOLA-Hardwaredatenstruktur

Eine zugehörige Testknotenbeschreibung besteht aus dem Tupel:

(old_pdv, new_pdv, p_unknown, p_tristate, entropy, significance, testability, op_entropy, module_id).

Old_pdv bezeichnet den Wahrscheinlichkeitsvektor der bekannten Ereignisse am Modulausgang aus dem vorhergehenden Iterationsschritt, **new_pdv** denjenigen des aktuellen Iterationsschrittes.

P_unknown := p(READ)·p(X=u) speichert die Wahrscheinlichkeit für bausteininterne unbekannte Zustände, **p_tristate** := 1 - p(READ) die Wahrscheinlichkeit für nichtlesbare Ausgangszustände. Dabei sei X der für den Ausgang relevante Testbarkeitsknoten.

Entropy := $\sum_{i=0}^{2^n-1} p(X=i)\cdot ld(1/p(X=i))$ [bit/pattern] bedeutet die Entropie der bekannten Zustände am Modulausgang, **significance** := 2 ×× entropy bestimmt die Anzahl der dort anliegenden signifikanten Vektoren.

Testability := $p(READ)\cdot\{T(Y) - \tilde{T}(Y)\}\cdot 100$ [%/pattern] mit

$$T(Y) := \frac{1}{n} H(Y) = \frac{1}{n} \cdot \sum_{i=0}^{2^n-1} p(Y=i)\cdot ld(1/p(Y=i)) \text{ und}$$

$$\tilde{T}(Y) := \frac{1}{n} \tilde{H}(Y) = \frac{1}{n} \cdot \{(1-p(Y=u))\cdot ld(1/(1-p(Y=u)))\}$$

entspricht der Testbarkeit der validen Ereignisse.

Das Triple "testability, p unknown, p tristate" zusammen bildet den Maßstab, an dem die Testbarkeit der Knotenpunkte einer Architektur bewertet werden soll.

Op _entropy steht für ein Feld, in dem die Entropien der einzelnen Operatoren eines Moduls hinterlegt werden. Durch Vergleich der Werte besteht die Möglichkeit, diejenige Operation durchzuschalten, die am Modulausgang maximale Entropie liefert. **Moduleid**, die Bausteinbeschreibung, definiert ein weiteres Tupel aus den Komponenten:

(maxval_out, module_type, op_list).

Maxval out := 2^{xx} bitwidth legt den maximalen Wert fest, der am Modulausgang auftreten kann. **Module type** bestimmt die Art des Bausteins, nämlich ob es sich um einen arithmetisch/logischen (Multifunktions-) Baustein, einen Speicher (mit uni/bidirektionalen Ports), ein Register, ein Instruktions-ROM, einen Bus oder um eine festverdrahtete Konstante handelt. Op list repräsentiert die dynamisch verkettete Liste der von einem Modultyp bereitgestellten Operationen.

Jede dieser Operationen wird wiederum durch ein Tupel beschrieben:

(op_code, maxval_in, instr_codes, inports).

Op code enthält den jeder Operation programmintern eindeutig zugewiesenen Operationscode. **Maxval in** ist der maximale Wert, der am breitesten Eingang des zugehörigen Bausteins auftreten kann. Instr codes stellt diejenige Menge der Instruktionscodes dar, die, am Kontrolleingang eines Moduls angelegt, die Ausführung der Operation bewirken. Die Tupelkomponente **inports** schließlich enthält die Liste derjenigen Inputports, an denen die für die Operation relevanten Operanden anliegen. Programmintern existiert eine eineindeutige Abbildung der Menge der Ausgangsbitbereiche auf die natürlichen Zahlen, m.a.W. eine globale Numerierung der Testbarkeitsknoten. Die Testbarkeitsknoten werden in einem Testdatenarray hinterlegt, der als Hintergrundspeicher fungiert, um Redundanzen bei der Abarbeitung der Testbarkeitsbäume (siehe folgendes Kapitel) zu vermeiden.

7.4.1.2 Der Testbarkeitsbaum

Aus den von MIMOLA bereitgestellten internen Listen - der Modultypliste, der Exemplaroder Inkarnationsliste, der Verbindungsliste, der Liste der Instruktionsfelder sowie den Operationsbeschreibungen wird der sogenannte **Testbarkeitsbaum aufgebaut**. Testbarkeitsbäume sind Hardwarestrukturbäume, die den Datenfluß pro Taktzyklus repräsentieren. Die Wurzeln von Testbarkeitsbäumen bilden "Zustandsfelder", d.h. Register und Speicherausgangsports. Blätter von Testbarkeitsbäumen können noch zusätzlich festverdrahtete Konstanten und Instruktionsfelder sein. Intern existiert der Testbarkeitsbaurn als dynamischer binärer Baum, realisiert durch PASCAL-Records und -Pointer, für den es schnelle Such- und Sortieralgorithmen gibt; extern als Textfile, in Klammernotation, zur eventuellen Weiterverarbeitung. Die ersten Teilbäume des vollständigen Testbarkeitsbaumes einer Hardware dienen zur Berechnung von Busknoten, die Zwischenergebnisse speichern. Eine weitere Sonderrolle spielen Speicherteilbäume, deren Struktur den jeweiligen Berechnungsalgorithmen angepaßt werden muß.

Bild 7.4.1.2.1 übernächste Seite zeigt als Beispiel den Testbarkeitsteilbaum für das Register *R3 zeroflag*. Seine Struktur wird aus dem abgebildeten Architekturausschnitt deutlich. Die Testhardware enthält einen kleinen Speicher SI *mainram* (256x4 Bit) mit einem unidirektionalen I/0-Port, ein Instruktions-ROM I der Dimension 256x21 Bit, den 4 Bit breiten Datenbus *W3 databus*, die beiden Akkumulatoren *R4 accua/accub*, die ALU mit Namen *Bi slice*, zwei Statusflipflops *R3 carry/zero* für einen Overflow-Test und einen Test auf "0", zwei Tristate-Buffer *A2 busdrive* und *A3 status*, einen Negierer *A1 invert*, einen Tester auf "=0" *A4 equa10* und **einen 3-fach Multiplexer** *A6 setstat. Const0, Constl* und *Const00* bezeichnen festverdrahtete Konstanten. An den Eingangsports der Bausteine *BI slice* und *A3 status* finden Konkatenationen, an den Ausgangsports der Bausteine *Bl slice* und *A1 invert* Abspaltungen bzw. Verzweigungen von Datenpfaden (siehe Kap.7.3.3.3) statt.

Betrachtet werde im folgenden der dyadische Multifunktionsbaustein BI *slice* (*'BI'* ist der Modultyp, *'slice'* seine Inkarnation), der laut MIMOLAQuellcode acht Operationen zur Verfügung stellt:

```
MODULE B1 (OUT result:(5:0); IN op1, op2:(5:0); FCT ctr:(2:0));
 PARBEGIN
   result <- CASE ctr OF
               %000 :  op1 "+" op2;        %001 :  op1 "-" op2;
               %010 :  op1 "OR" op2;       %011 :  op1 "AND" op2;
               %100 :  op1 "a'ORb" op2;    %101 :  op2 "a'ORb" op1;
               %110 :  op2;                %111 :  #1;
             END
 PAREND;
```

Inputs von *B1_slice* sind an Eingangsports &1 eine festverdrahtete binäre Null (*F%0*), das Register *R4_accua* und der Multiplexer *A6_setstat*; an Eingangsport &2 eine festverdrahtete binäre Null, das Register *R4_accub* und eine festverdrahtete binäre Eins (*F%1*). Der Instruktionseingang &101 ist mit Ausgangsfeld (5:3) des Instruktionsspeichers *I* verbunden. *FB1* ist der zum Modultyp *B1* gehörige Funktionsbaum. Für eine binäre Null am Instruktionseingang wird die Addition (Operationssymbol "+") angewählt. Das Ergebnis der Operation wird am Modulausgangsport *result* der Bitbreite (5:0) ausgegeben; die Operanden stehen an den Moduleingangsports *op1* und *op2* an. Für eine binäre Vier als Funktionscode wird der negierte Operand an Port *op1* über die OR-Funktion mit dem Operanden an Port *op2* verknüpft. Funktionscode Sechs schaltet den Operanden an Port *op2* durch; Funktionscode Sieben schließlich setzt den Ausgangsport auf Eins. Das erste Zeichen jeden Knotens kennzeichnet den Knotentyp: '_' eine Hardwarequelle, '.' eine Bausteinfunktion, '=' eine Konstante und 'F' die Wurzel des Funktionsbaumes.

Der vollständige Testbarkeitsbaum einer Architektur liefert intern sämtliche Information, die zur iterativen Berechnung der Wahrscheinlichkeitsverteilungsvektoren, respektive der Testbarkeitsmaßzahlen, an den Modulausgängen notwendig ist. Optional läßt sich der Testbarkeitsbaum als reiner Strukturbaum, gestaucht ohne Ausgangsbitbereiche oder auch in Klammernotation ausdrucken.

```
_R3_zeroflag — _A4_equal0@1.(0) — _A1_invert@1.(3:0) — _B1_slice@1.(5:1) — _F%0_const0@1.(0)
                                                                          _R4_accua@1.(3:0)
                                                                          _A6_setstat@1.(0) — _R3_carryflag@1.(0)
                                                                                              _I@101.(1:0)
                                                                                              FA6 — =%00@101.SET(1:0) — _outp.(0) — .%0@101.(1:0)
                                                                                                    =%01@101.SET(1:0) — _outp.(0) — .%1@101.(1:0)
                                                                                                    =%10@101.DAT(1:0) — _outp.(0) — _dat@1.(0)
                                                                                                                                    .DAT@101.(1:0)
                                                                          _F%0_const0@2.(0)
                                                                          _R4_accub@2.(3:0)
                                                                          _F%1_const1@2.(0)
                                                                          _I@101.(5:3)
                                                                          FB1 — =%000@101.+(2:0) — _result.(5:0) — _op1@1.(5:0)
                                                                                                                   _op2@2.(5:0)
                                                                                                                   .+@101.(2:0)
                                                                                =%001@101.-(2:0) — _result.(5:0) — _op1@1.(5:0)
                                                                                                                   _op2@2.(5:0)
                                                                                                                   .-@101.(2:0)
                                                                                =%010@101.OR(2:0) — _result.(5:0) — _op1@1.(5:0)
                                                                                                                    _op2@2.(5:0)
                                                                                                                    .OR@101.(2:0)
                                                                                =%011@101.AND(2:0) — _result.(5:0) — _op1@1.(5:0)
                                                                                                                     _op2@2.(5:0)
                                                                                                                     .AND@101.(2:0)
                                                                                =%100@101.a'ORb(2:0) — _result.(5:0) — _op1@1.(5:0)
                                                                                                                      _op2@2.(5:0)
                                                                                                                      .a'ORb@101.(2:0)
                                                                                =%101@101.a'ORb(2:0) — _result.(5:0) — _op2@2.(5:0)
                                                                                                                      _op1@1.(5:0)
                                                                                                                      .a'ORb@101.(2:0)
                                                                                =%110@101.DAT(2:0) — _result.(5:0) — _op2@2.(5:0)
                                                                                                                     .DAT@101.(2:0)
                                                                                =%111@101.SET(2:0) — _result.(5:0) — .#1@101.(2:0)
                                                                  _I@101.(6).
                                                                  FA1 — =%0@101.NOT(0) — _result.(4:0) — _dat@1.(4:0)
                                                                                                         .NOT@101.(0)
                                                                        =%1@101.DAT(0) — _result.(4:0) — _dat@1.(4:0)
                                                                                                         .DAT@101.(0)
                                                  FA4 — =%X.=0 — _compare.(0) — _dat@1.(3:0).
                                                                                 .=0@101
                                  _I@101.(2)
                                  FR3 — =%X@101.READ(0) — _outp.(0) — .READ@101.(0)
                                        =%1@101.LOAD(0) — _R3.(0) — _dat@1.(0)
                                                                    .LOAD@101.(0)
                                        =%0@101.NOLOAD(0) — _R3.(0) — .NOLOAD@101.(0)
```

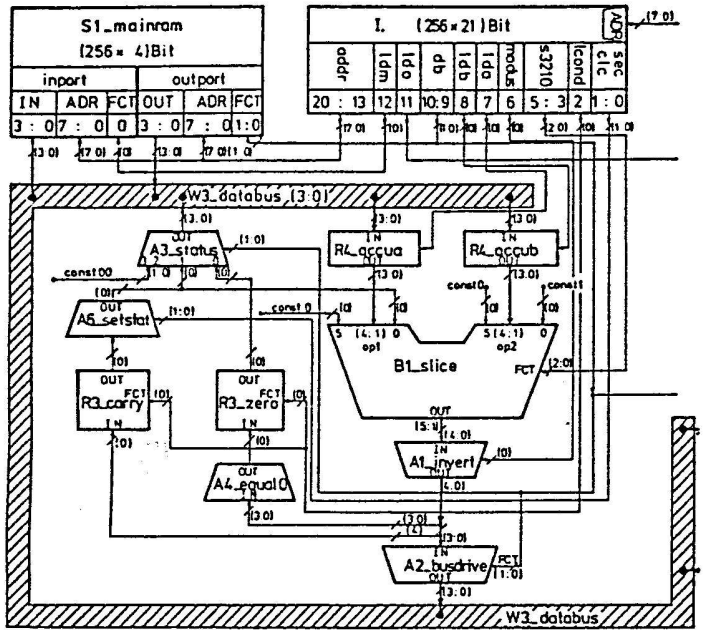


Abb. 7.4.1.2.1: Testbarkeitsteilbaum für das Register *R3_zeroflag* (mit Architekturausschnitt)

7.4.2 Praktische Beispiele

7.4.2.1 Eine Prozessorstruktur auf Register-Transfer-Ebene

Die Rechnerarchitektur, an der das Prinzip einer Testbarkeitsanalyse auf Register-Transfer-Niveau demonstriert werden soll, ist der SPDM ≙ Subprozessor mit dynamischer Mikroprogrammierung /21/, ein experimenteller Spezialprozessor, der unter Verwendung der MIMOLA-Designmethode den architekturellen Erfordernissen zur Berechnung eines umfangreichen wissenschaftlichen Softwarepaketes angepaßt wurde. Bild 7.4.2.1.1 nächste Seite gibt die Grundstruktur des Prozessors - eine unter Laufzeitgesichtspunkten optimierte Rechnerkonfiguration - wieder. Seine Datenpfadbreite wurde für Testzwecke auf acht Bit festgelegt. Der SPDM ('slave') ist über einen schnellen DMA-Kanal mit einem universellen Minicomputer ('master') so gekoppelt, daß die Möglichkeit besteht, den Hauptspeicher des Spezialrechners vom Wirtsrechner aus über den Kanal direkt mit Testvektoren zu laden (DMA ≙ direct memory access). Neben dem Hauptspeicher enthält der Prozessor noch zwei Dualport-RAMs zur Zwischenspeicherung lokaler Variablen. Auf diese Weise können zeitintensive Ladevorgänge des Hauptspeichers erheblich reduziert werden. Weitere Kernstücke des Prozessors sind fünf parallel arbeitende arithmetisch/logische Einheiten, die verschiedene, softwaremäßig notwendige, nach ihren statistischen Auftrittshäufigkeiten auf die einzelnen Moduln verteilte Operatoren zur Verfügung stellen. Andere gewünschte Funktionen müssen programmtechnisch implementiert werden. Ein Mikrobefehlswort des SPDM hat eine Breite von 103 Bits, aufgeteilt in 29 separate Kontrollfelder, deren Bedeutung hier nicht näher spezifiziert werden soll. Andere Inplementierungsdetails können in diesem Zusammenhang ebenfalls entfallen.

Insgesamt enthält die Architektur 37 komplexe Registertransfermodule (inclusive 5 Dekoder), äquivalent mit 39 internen funktionalen Schaltungsknotenpunkten (wegen der Multiportspeicher), deren Testbarkeit mit dem beschriebenen Analyseverfahren näher untersucht wurde /22/.

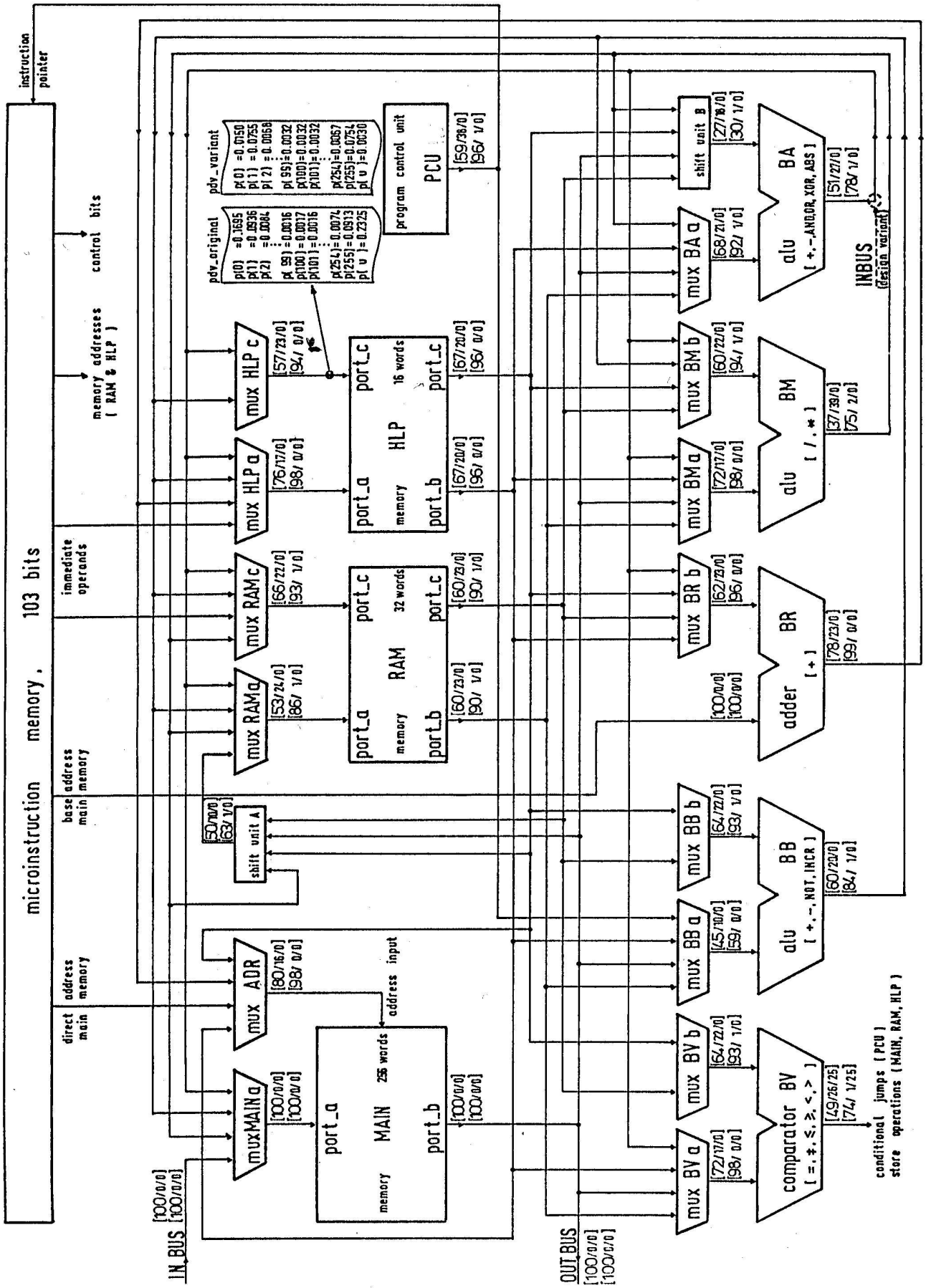


Abb. 7.4.2.1.1: Vergleichende Testbarkeitsbewertung von Originalarchitektur und einer Designalternative des SPDM

Die Ergebnisse einer vergleichenden iterativen Testbarkeitsanalyse sollen nachfolgend an zwei Designvarianten des SPDM demonstriert werden. Wie im Blockdiagramm Bild 7.4.2.1.1 angedeutet, unterscheidet sich die modifizierte Architektur des SPDM vom Original lediglich dadurch, daß der Dateneingangsbus *INBUS* am Ausgang des ärithmetisch/logischen Bausteins *BA* über einen zusätzlichen 2-fach Multiplexer *muxBAc* (fehlt in Zeichnung!) zugeschaltet wird. Andernfalls führt er exklusiv via Multiplexer *muxMAINa* auf den Dateneingang des Hauptspeichers. Die Grundstruktur des Prozessors bleibt erhalten; die Anzahl der externen Anschlußpins wird nicht verändert; nur das Mikroinstruktionswort wächst um ein Bit, das Steuerbit des hinzugefügten Multiplexers. Es stellt sich die Frage: Welchen Einfluß hat die architekturelle Änderung auf das Testbarkeitsverhalten der Hardware?

Als erste Voraussetzung der durchzuführenden Analyse muß gelten, daß der SPDM vom Universalrechner über den DMA-Kanal direkt mit Testmustern versorgt werden kann. D.h. der Multiplexer *muxMAINa* für die Originalhardware, die beiden Multiplexer muxBAc und *muxMAINa* zusammen für die Alternativhardware sind programmintern per Mikrocode so geschaltet, das an *INBUS* anliegende Signale direkt in den Hauptspeicher durchgeschaltet werden. Als weitere Voraussetzung sei an *INBUS,* dem primären Eingang der Schaltung, eine Gleichverteilung aller möglichen Eingangsereignisse, gleichbedeutend mit einer Testbarkeit von einhundert Prozent, vorgegeben. Die Testbarkeitsmaßzahlen der Architekturvarianten sind genau dann konvergiert, wenn sich am Ausgang des Hauptspeichers ebenfalls eine Testbarkeit von einhundert Prozent eingestellt hat, m.a.W. wenn Transparenz des Speicherbausteins vorliegt. Bei einer Hauptspeichergroße von 256 Wörtern ist das nach zirka 2000 Testzyklen der Fall. Dafür braucht **man auf einer 32-Bit Eclipse** MV/10000 von Data General etwas weniger als 1.5 CPU-h Rechenzeit, ein vertretbarer Zeitaufwand für eine derart komplexe Hardwarestruktur. Den MIMOLA-Quellcode des SPDM von ca. 500 Zeilen in die baumartige Zwischensprache, auf der die Analyse stattfindet, zu kompilieren, ist eine Sache von wenigen Minuten.

Mal abgesehen von den Shifteinheiten, findet durch die vorgenommene Designmodifikation generell eine Testbarkeitsverbesserung von 20-30% in der Hardware statt. Einher mit einer Testbarkeitsverbesserung der Modulausgangsports geht eine Verringerung der lunknown'-Ausgangszustände, **wie ein Blick auf das Schaltbild bestätigt. Das Testzahlentripel aus**

[Knotentestbarkeit/Ilunknown°-Wlkeit/Ithigh impedancell-Wlkeit]

in der Zeichnung kennzeichnet das Testbarkeitsverhalten der internen Schaltungsknoten im stationären Endzustand. Die Werte in eckigen Klammern sind prozentuale Angaben. Zu jedem beliebigen Iterationszeitpunkt können die propagierbaren Wahrscheinlichkeitsverteilungen auf den internen Datenpfaden der Architektur abgefragt werden. Der Verteilungsvektor der Designvariante im stationären Endwert 'pdv_vatiant' auf der Verbindung *"muxHLPc -> HLP-port c"* kommt einer Gleichverteilung (p(i)=1/256=0,.0039 di) viel näher als der Verteilungsvektor der Originalarchitektur 'pdv original . Daher der Testbarkeitsunterschied von 94 zu 57 Prozent! Tristate-Zustände treten nur am Ausgang des Vergleichers BV auf, und zwar in 25 Prozent aller Fälle.

Die zu erwartende Fehlerüberdeckung UI in Abhängigkeit der Testvektormenge INS liefert eine globale Testbarkeitsbewertung für die Rechnerarchitekturvarianten als Ganzes. Alle 39 (40) funktionalen Testbarkeitsknoten der Originalarchitektur (modifizierten Architektur) tragen zur Berechnung der Fehlerüberdeckung bei. Für eine Kanalbreite von acht Bit entspricht ein Testvektor 256 Testmustern pro Iteration. Wie aus der nachfolgenden Zeichnung ersichtlich, kann eine vorgegebene Fehlerüberdeckung bei der modifizierten Architektur mit wesentlich weniger Testaufwand erzielt werden!

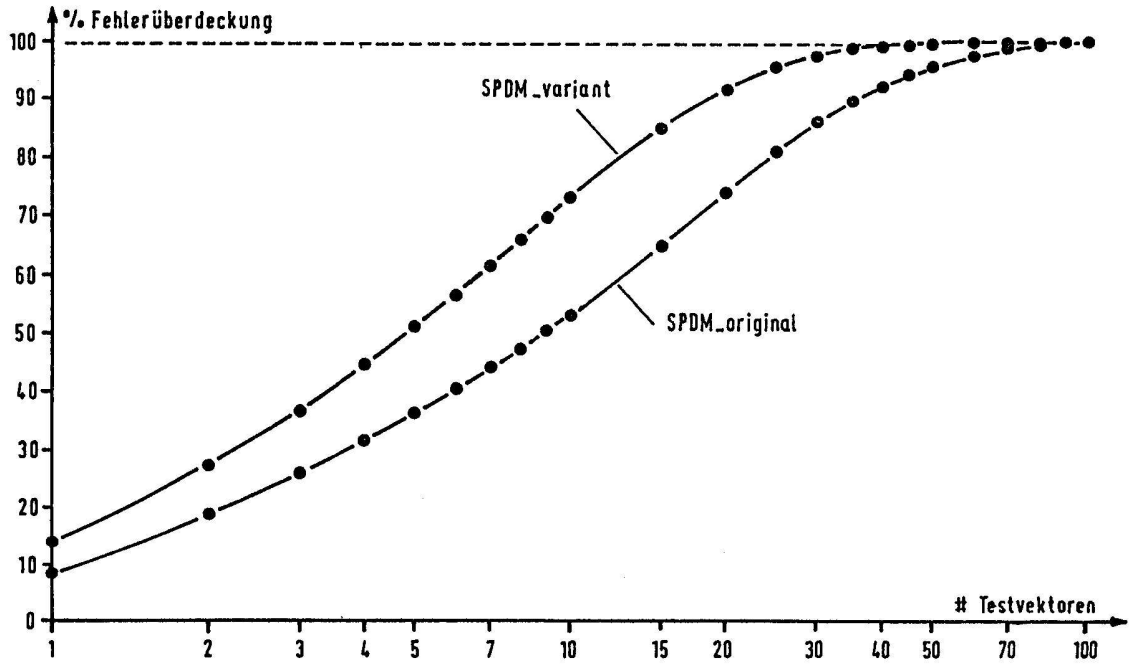


Abb. 7.4.2.1.2: Fehlerüberdeckung in Abhängigkeit der Testvektorsequenzlänge

Am Schluß der Untersuchung steht das allgemeine Fazit, daß nach allen zur **Verfügung gestellten Testkriterien die Designvariante** des SPDM signifikant besser testbar ist als die Originalversion.

7.4.2.2 Eine Zustandsmaschine auf Logik-Ebene

Um die Spannweite des beschriebenen Testbarkeitsanalyseverfahrens zu unterstreichen, wird als ein weiteres Beispiel ein DRAM-Controller vorgestellt, der als Zustandsmaschine ('state machinel) unter Zuhilfenahme von PAL-Bausteinen (PAL ^-_ 'programable array logic') entworfen und realisiert worden ist. Die zu diesem Schaltwerk durchgeführten Logik- und Fehlersimulationen stimmen in hohem Maße mit den Testbarkeitsanalyseaussagen, Bilder 7.4.2.2.1 bis 7.4.2.2.4 auf den folgenden Seiten, überein. Die aus dem vorhergehenden Beispiel bekannten Testbarkeitstripel reduzieren sich für den hier vorliegenden Fall auf die prozentuale Angabe der Knotentestbarkeiten, da weder unbekannte noch hochohmige Zustände in dem Schaltwerk auftreten. Hohe Testbarkeitswerte deuten unmittelbar auf eine schnelle Testvektorgenerierung für den betreffenden Knoten hin und umgekehrt. Die primären Eingangstestvektorverteilungen sind in einem interaktiven Prozeß derart optimiert worden, daß der daraus resultierende Testinformationsfluß nach ca. 50 Testzyklen maximal und stationär wird.

Abschließend sollen noch einige wesentliche allgemeine Merkmale des dargestellten Verfahrens herausgehoben werden:

1. Eine Testbarkeitsanalyse während der Entwurfsphase ist für gemischt kombinatorisch/sequentielle Schaltwerke mit nahezu beliebiger kombinatorischer Tiefe (nur beschränkt durch den adressierbaren Speicherbereich des Wirtsrechners) und beliebigen Pfadbreiten kleiner als 32 Bit möglich, kompatibel zum Sprachumfang des MIMOLA-Entwurfssystems.
2. Die Maßzahlen, im Bereich zwischen Null und Eins angesiedelt, machen eine unmittelbare Aussage über die Unsicherheit, auf einer Leitung des betrachteten Pfades einen bestimmten logischen Zustand zu erzeugen und zu beobachten.
3. Weder Logik- noch Fehlersimulationen sind zur Bestimmung der Ereigniswahrscheinlichkeiten, auf denen die Testbarkeitswerte beruhen, notwendig. Die Ereignisräume werden quasiparallel durch die gesamte Architektur berechnet.

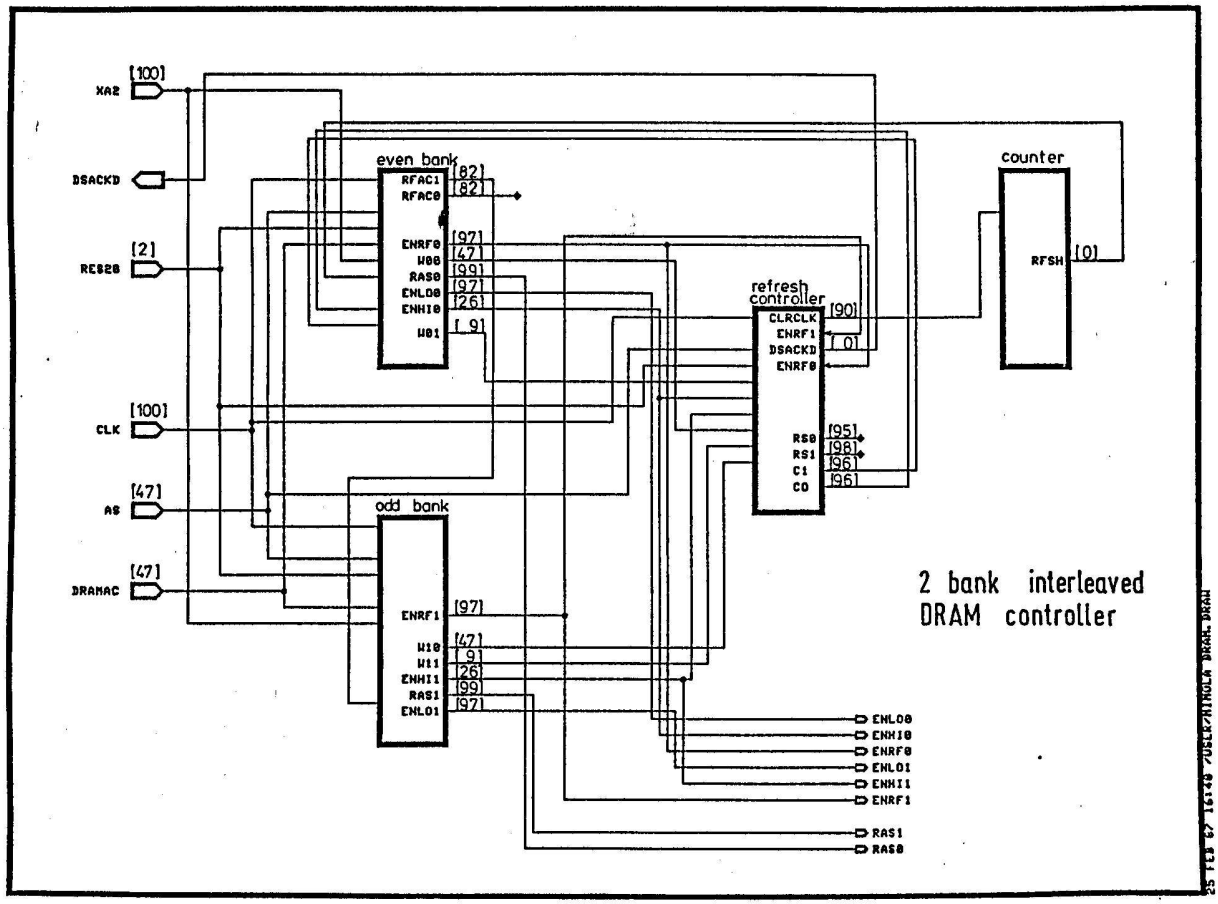


Abb. 7.4.2.2.1: Blockschaltbild des analysierten DRAM-Controllers für eine Doppelspeicherbank mit zyklischer Schreib/Lese- sowie Refresh-Ansteuerung (Die Zahlen in Klammern geben die Knotentestbarkeiten in Prozent wieder)

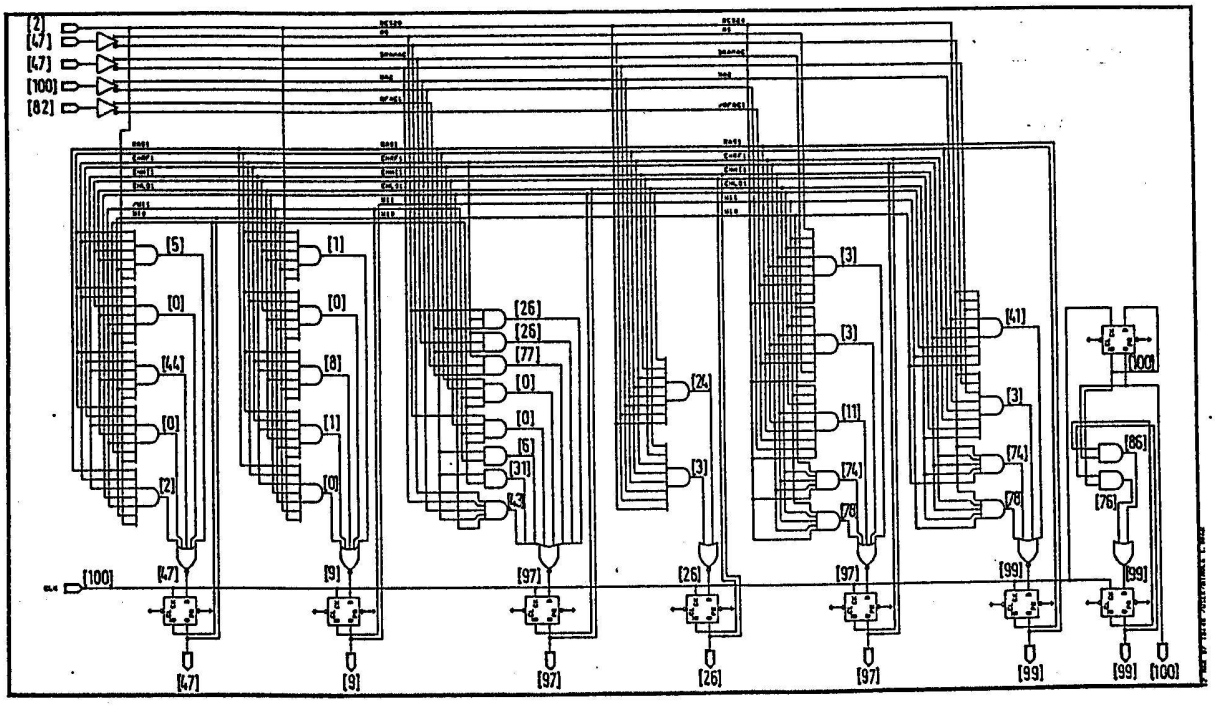


Abb. 7.4.2.2.2: Logikplan zur Ansteuerung der *'odd bank'*

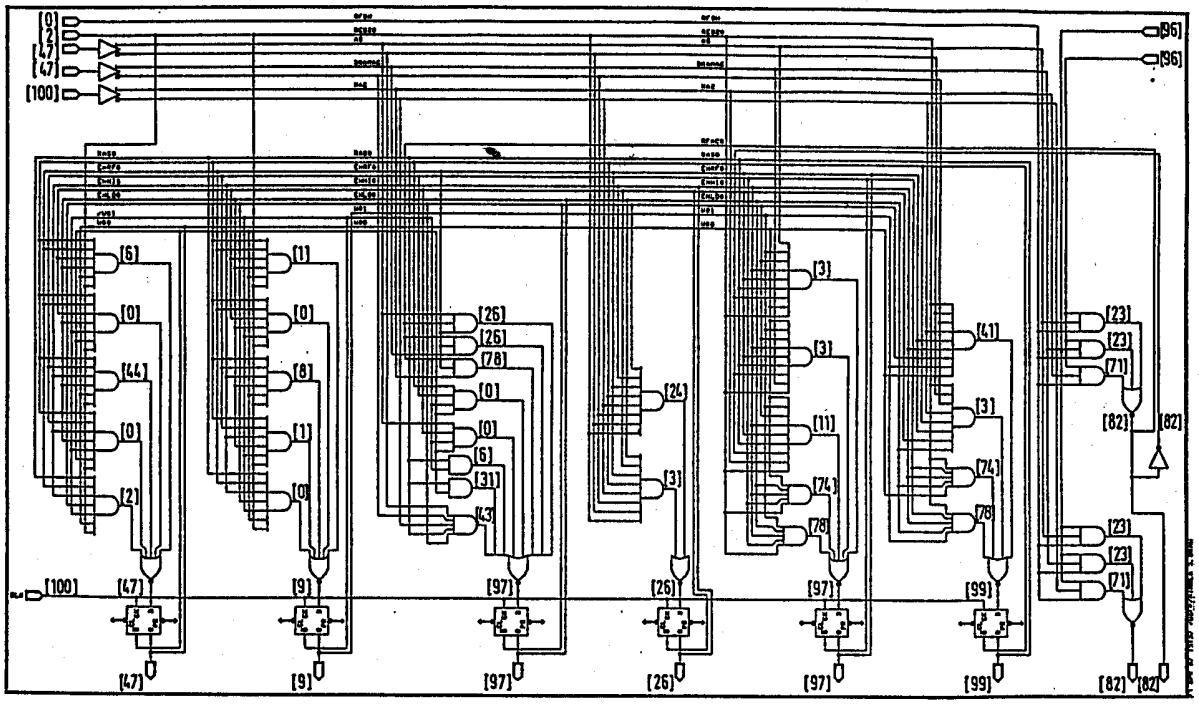


Abb. 7.4.2.2.3: Logikplan zur Ansteuerung *der,'even bank'*

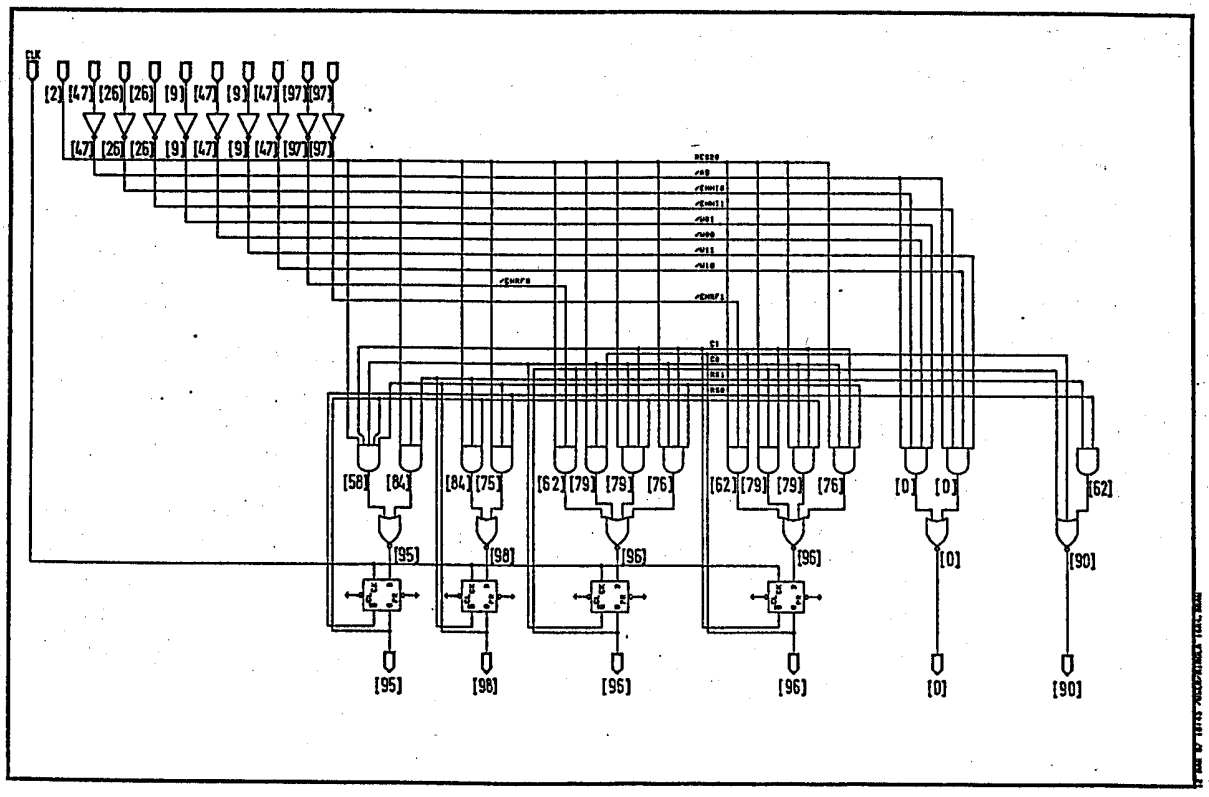


Abb. 7.4.2.2.4: Logikplan des *'refresh controller'*

7.5 Zusammenfassung

Kapitel 7 beschreibt ein neuartiges Verfahren für die Testbarkeitsanalyse mikroprogrammierbarer Rechnerstrukturen. Das dazu entwickelte Testbarkeitsanalysewerkzeug stellt eine Programmkomponente des Hardwaresynthese und Mikrocodegenerierungssystems MSS2 (MIMOLA Software System Version 2) dar. Hardwarebeschreibungsebene ist die Registertransfer-Strukturebene, deren Bausteine -arithmetisch/logische Operationsmoduln, Multiplexer, Register, Zähler, Speicher und Codeumsetzer - eine natürliche architekturelle Partitionierung vorgeben. Aktionsbeschreibungsebene ist die Mikroprogrammierungsebene, wo Zustandsübergänge in den abstrakten funktionalen Blöcken der RT-Strukturebene durch Mikroinstruktionen gesteuert werden.

Grundlage für eine Testbarkeitsaussage bildet die Bestimmung der Ereigniswahrscheinlichkeitsverteilungen an allen internen funktionalen Knoten einer Schaltung. Funktionale Schaltungsknoten stehen dabei synonym für Modulausgangsdatenports. Algorithmen beschreiben das Abbildungsverhalten der Eingangsereignisräume in die Ausgangsereignisräume beim Datentransfer durch die Module, und zwar sowohl für kombinatorische als auch für sequentielle (!) Registertransferbausteine. Mehr als fünfzig Algorithmen für alle standardmäßig vorkommenden arithmetischen, logischen, relationalen, selektiven und Schiebeoperatoren stehen programmintern zur Verfügung. Je nach Bitbreite der Moduldatenports wird der gesamte Ereignisraum oder nur eine Stichprobe desselben algorithmisch fortgepflanzt. 8-Bit-Architekturen bilden das Maximum an Berechnungskomplexität für das exakte Verfahren. Als Stichprobenverfahren für Architekturen größerer Bitbreite, vorzugsweise 16- und 32-Bit, bietet sich deshalb die sogenannte Byte-Slice-Technik an, die weiterhin vertrauenswürdige Testbarkeitsaussagen liefert, ohne die Berechnungszeiten nennenswert zu erhöhen.

Gewählter Maßstab für die Testbarkeit eines internen Schaltungsknotens ist der mittlere Informationsgehalt der dort auftretenden Ereignisse, für die Testbarkeit einer gesamten Schaltung die mittlere Fehlererkennungswahrscheinlichkeit sämtlicher Schaltungsknoten. Das TestbarkeitsmaB, axiomatisch aus der Informationstheorie hergeleitet, stellt eine Erweiterung bekannter Maße für die Logikebene dar, die auf Kontrollierund Beobachtbarkeitsuntersuchungen logischer Zustände basieren. Entlang sogenannter Testbarkeitsbäume, erweiterter Hardwarebäume, deren Knoteninhalte die Testhardware vollständig beschreiben und ihr Testbarkeits

verhalten quantitativ festhalten, wird der Testdatenfluß durch eine Architektur zyklisch simuliert. Blätter und Wurzeln der Bäume, gleichbedeutend mit Quellen und Senken der iterativen Testbarkeitsberechnung, bilden sogenannte Zustandsfelder, d.h. im wesentlichen Speicher- oder Registerdatenausgangsbitbereiche. Die Anzahl der Testzyklen bis zum Erreichen stationärer Endwerte für die Testdaten dient als Maß für die sequentielle Tiefe einer Schaltung.

Das Maßzahlentripel aus Knotentestbarkeit, Unknown-Wahrscheinlichkeit und Tristate-Wahrscheinlichkeit im stationären Endzustand schätzt die Übertragungsqualität - d.h. die Schwierigkeit, anliegende Testmuster durchzuschalten - der einzelnen Bausteine einer Hardware ab. Die Testbarkeitsanalyseergebnisse erweisen sich als hochsensibel gegenüber Änderungen bei den Eingangstestmusterverteilungen, der Firmware, der Topologie oder Verdrahtung, den Bausteinoperatoren und den Speicherfunktionen einer Hardware. Problemzonen im Entwurf - zum Beispiel Testflaschenhälse, Rekonvergenzschleifen, Redundanzen, TestpatternCollapsing, etc. - lassen sich auch ohne Kenntnis des genauen

Funktionsprinzips einer Schaltung zuverlässig identifitieren.

Die Testphilosophie, die hinter dem programmtechnisch realisierten Konzept steht, zielt auf die Bewertung einer Rechnerarchitektur schon während der Entwurfsphase, noch bei der Festlegung des Struktur- und Operationsprinzips des zu entwerfenden Rechners. Nur solche Rechnerkonfigurationen, die von vornherein eine gute Testbarkeit gewährleisten, d.h. für die eine effiziente Testmustergenerierung garantiert ist, sind zur weiteren Verarbeitung zugelassen. Beabsichtigt wird, zeit- und kostenintensive Fehlersimulationsläufe in der Testvorbereitungsphase durch das hier dargestellte Verfahren überflüssig zu machen.

In seiner derzeitigen Ausbauphase umfaßt das Testbarkeitsanalyseprogramm ca. 10000 Zeilen Standard-PASCAL-Code. Die Portabilität des Programms wurde durch Implementation auf Rechnern verschiedener Typen nachgewiesen.

7.6 Literatur

/1/ Seth, S.C:;Agraval,V.D.; "An Exact Analysis for Efficient Computation of Random-Pattern Testability in Combinational Circuits", Proc. FTCS-16, 1986, Digest of Papers, pp.318-323.

/2/ Wunderlich,H.J.; "PROTEST: A Tool for Probabilistic Testability Analysis°, Proc. 22nd Design Automation Conference, Las Vegas, 1985, pp.204-211.

/3/ Agraval,V.D.;Seth,S.C.; "Probabilistic Testability", Proc. ICCD185, New York, 1985, pp.562-565.

/4/ Savir,J.;Ditlow,G.S.;Bardell,P.H.; "Random Pattern Testability", IEEE Trans. Computers, Vol. C-33, 1984, pp.79-90.

/5/ Savir,J.; "Good Controllability and Observability Do Not Guarantee Good Testability", IEEE Trans. an Computers, Vol. C-32, 1983, pp.1198-1200.

/6/ Brglez, F.;Pownall,Ph.; "Applications of Testability Analysis: From ATPG to Critical Delay Path Tracing", 1984 IEEE Int. Test Conference, Cherry Hill, New Jersey, Digest of Papers, pp.705-712.

/7/ Krishnamurthy,B.;Li-Cheng Sheng,R.; "A New Approach to the Use of Testability Analysis in Test Generation", Proc. IEEE Int. Test Conference, 1985, pp.769-778.

/8/ Jain,S.K.;Agrawal,V.D.; "STAFAN: An Alternative to Fault Simulation", 21st Design Automation Conference, 1984, pp.18-23.

/9/ Seth,S.C.; "Predicting Fault Coverage from Probabilistic Testability" Proc. IEEE Int. Test Conference, 1985, pp.803-805.

/10/ Agrawal,V.D.; "Information Theory in Digital Testing - A New Approach to Functional Test Pattern Generation°, Proc. IEEE Int. Conference an Circuits and Computers, Port Chester, NY, Oct.1980, pp.928-931.

/11/ Ash,R.; "Information Theory", in "Interscience Tracts in Pure and Applied Mathematics", No. 19, Interscience Publishers, 1965.

/12/ Shannon,C.E.; "A Mathematical Theory of Communication", in "Key Papers in the Development of Information Theory", edited by David Slepian, IEEE Press, 1973, pp.5-29.

/13/ Mc.Millan,B.; "The Basic Theorems of Information Theory", reprinted with permission from Ann. Math. Stat., Vol. 24, June 1953, pp.196-219, in "Key Papers in the Development of Information Theory", edited by David Slepian, IEEE Press, 1973.

/14/ Brillouin,L., "Science and Information Theory", 2nd Edition, Academic Press Inc., New York, 1962.

/15/ Gilio,W.;Liebig,H.; "Logischer Entwurf digitale Systeme", Springer-Verlag, Berlin-Heidelberg-New York, 1980.

/16/ Wosnitza,F.; "Ein Testbarkeitsanalysemaß zur Architekturbewertung programmierbarer Schaltnetzen, Dissertation am Institut für Theor. Elektrotechnik, RWTH Aachen, Dez. 1986.

/17/ Fung,H.S.;Fong,J.Y.O.; "An Information Flow Approach to Functional Testability Measures", Proc. IEEE Int. Conference an Circuits and Computers, New York, Sept. 1982, pp.460-463.

/18/ Anceau,F.; "The Architecture of Microprocessors", Addison-Wesley Publishing Company, Inc., 1986.

/19/ Leisengang,D.;Wagner,M.; "Signaturanalyse in der Datenverarbeitung", Elektronik, Franzis-Verlag, München, Okt. 1983, pp.67-72.

/20/ Peterson,W.W.;Weldon,E.J.; "Error-Correcting Codes", 2nd Edition, MIT Press, Cambridge, Massachussetts, 1972.

/21/ Marwedel,P.; "The Design of a Subprocessor with Dynamic Microprogramming with MIMOLA", Informatik-Fachberichte 27, Springer-Verlag Berlin-Heidelberg-New York, 1980, pp.164-177.

/22/ Ke11e,K.; 1lEin Testbarkeitsanalyseprogramm für mikroprogrammierbare Rechnerstrukturen - integraler Teil des MIMOLA-CAD-Systemsll, nicht veröffentlichte Dissertation am Institut für Theor. Elektrotechnik, RWTH Aachen, Mai 1987.