Teil C Anwendung von VHDL

1 Simulation

1.1 Überblick

Die Simulation dient im allgemeinen der Verifikation von Entwurfsschritten. Bei einer Designmethodik mit VHDL unter Verwendung von Synthesewerkzeugen werden vorwiegend Verhaltensmodelle auf abstrakten Entwurfsebenen (System-, Algorithmische und Register-Transfer-Ebene) und die entsprechenden strukturalen Modelle auf Logikebene eingesetzt. Die Simulation von VHDL-Modellen hat dabei konkret folgende Aufgaben zu erfüllen:

1.1.1 Simulation von Verhaltensmodellen

In der Regel werden Verhaltensmodelle von Hand erstellt oder durch ein Front-End-Tool generiert. Verhaltensmodelle dienen

- zur frühzeitigen Verifikation des Entwurfs,
- als Eingabe für ein Synthesewerkzeug.

Meist wird für das Verhaltensmodell bereits auf abstrakter Ebene eine Testumgebung ("Testbench") des Modells erstellt, welche die Eingangssignale (Stimuli) für das Modell zur Verfügung stellt und dessen Ausgangssignale (Ist-Antworten) mit den erwarteten Werten (Soll-Antworten) vergleicht. Durch die Angabe von erwarteten Antworten kann ein aufwendiges und fehlerträchtiges, manuelles Überprüfen der Ausgangssignale entfallen.

Eine Simulation von Verhaltensmodellen auf abstraktem Niveau muß folgende Fragen beantworten:

Ist das Modell syntaktisch korrekt?

Manuell erstellte VHDL-Modelle sind in der Regel nicht von vorne herein syntaktisch korrekt. Eine entsprechende Überprüfung kann vom Compiler-Modul des VHDL-Simulators oder von speziellen Syn-

tax-Checkern durchgeführt werden. Wurde das Verhaltensmodell durch ein Front-End-Tool generiert, kann man von der syntaktischen Korrektheit des Modells ausgehen.

Stimmt das Modell mit der Spezifikation überein?

Die Überprüfung der funktionalen Korrektheit des Entwurfsschrittes von der Spezifikation zum abstrakten Verhaltensmodell ist der eigentliche Sinn einer Simulation. Dabei stellt sich das Problem, daß durch die Simulation zwar die Anwesenheit eines Fehlers gezeigt, für größere Schaltungen aber nie die Abwesenheit von Fehlern bewiesen werden kann. Es existieren zur Verifikation des funktionalen Verhaltens zwar andere Verfahren, die dies leisten (formale Verifikation), ausgereifte Werkzeuge zur Handhabung komplexer Schaltungen stehen aber dafür nicht zur Verfügung.

Welche Eigenschaften besitzt das modellierte System?

Neben einer Überprüfung der Funktionalität kann durch die Simulation des Verhaltensmodells beispielsweise die Auslastung von Bussen oder eine geeignete Synchronisation von Submodulen bestimmt werden.

1.1.2 Simulation von strukturalen Modellen

Die VHDL-Gatternetzlisten werden kaum manuell erstellt. Sie werden in der Regel, unter Verwendung von technologiespezifischen Logikmodellen, mit Hilfe von Synthesewerkzeugen generiert. Die Simulation solcher Modelle dient zur Untersuchung der Frage:

Stimmt die Gatternetzliste mit dem Verhaltensmodell funktional überein und erfüllt sie die zeitlichen Anforderungen?

Bei einer Simulation der Gatternetzliste werden nicht nur funktionale Aspekte, sondern auch die zeitlichen Randbedingungen untersucht. Dazu kann die Testbench des Verhaltensmodells, ggf. nach Hinzufügen von weiteren zeitlichen Informationen oder genauerer Überprüfung der Ausgänge, verwendet werden. Sollen auch die Einflüsse des Layouts auf das zeitliche Verhalten der Gatternetzliste untersucht werden, so muß ein Backannotation-Schritt erfolgen, d.h. Informationen aus dem Layout - dabei handelt es sich typischerweise um die Ein-

flüsse der Leitungskapazitäten - werden in das Logikmodell zurückgeführt.

1.2 Simulationstechniken

Eine Kenntnis der unterschiedlichen Simulationstechniken ist u.a. für die Auswahl eines geeigneten Simulators bei gegebenen Schaltungsgrößen wichtig, denn sie beeinflussen die Performance des Simulators erheblich. Prinzipiell unterscheidet man, ähnlich wie bei Interpretern und Compilern für Programmiersprachen, zwei Konzepte für VHDL-Simulatoren, das interpretierende und das compilierende.

1.2.1 Interpretierende Simulationstechnik

Sie kann als die klassische Methode angesehen werden. Der VHDL-Quellcode wird bei der Simulationsvorbereitung in einen Pseudo-Code umgewandelt, der mit einem, meist auf der Programmiersprache "C" basierenden Simulator abgearbeitet werden kann. Kennzeichen der interpretierenden Simulationstechnik sind kurze Zeiten bei der Simulationsvorbereitung und längere Zeiten bei der Simulation selbst.

1.2.2 Compilierende Simulationstechnik

Hierbei wird der VHDL-Quellcode bei der Simulationsvorbereitung zunächst komplett in "C" übersetzt und mit einem C-Compiler in Objektcode umgewandelt. Bei der eigentlichen Simulation kann also direkt ein Maschinenprogramm ausgeführt werden. Längere Zeiten bei der Simulationsvorbereitung stehen einem schnelleren Simulationsablauf gegenüber.

Normalerweise ist die interpretierende Technik eher für kleinere Modelle geeignet, bei denen die Simulationszeit nicht sehr ins Gewicht fällt. Bei großen Modellen und langen Simulationszeiten macht sich jedoch aufgrund einer schnelleren Simulation der Vorteil einer compilierenden Technik bemerkbar.

1.2.3 Native-Compiled Simulationstechnik

Neuartige Simulatoren gehen einen Zwischenweg und versuchen, die Vorteile beider Ansätze zu verknüpfen. Bei der Methode der Native-Compiled-Simulation entfällt die Übersetzung in "C"; statt dessen wird aus dem VHDL-Quellcode direkt der Maschinencode erzeugt. Dadurch wird einerseits die Simulationsvorbereitungszeit im Bereich eines interpretierenden Simulators liegen, während andererseits ein ausgesprochen optimierter Objektcode vorliegt. Die eigentliche Simulation sollte noch deutlich schneller als bei compilierenden Simulatoren ablaufen.

Abb. C-1 stellt die drei Simulationstechniken gegenüber.

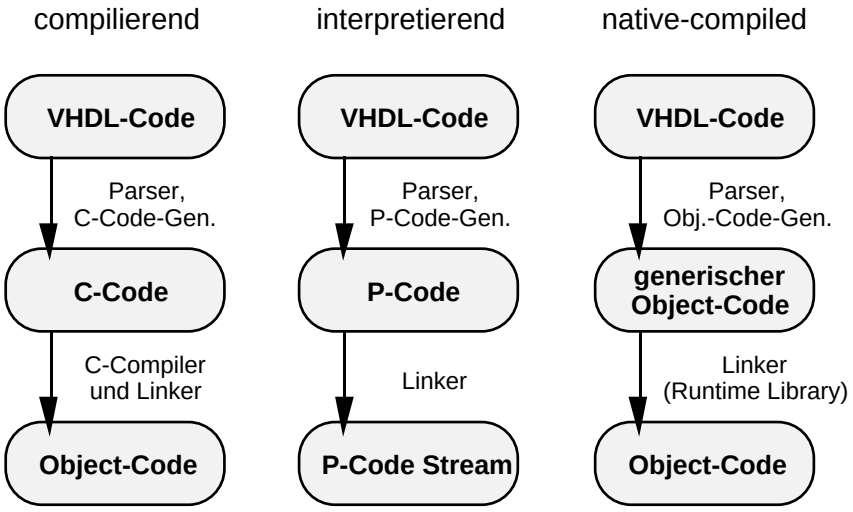

Abb. C-1: Simulationstechniken

1.3 Simulationsphasen

Gemäß dem LRM[1] der Sprache VHDL erfolgt die Simulation einer VHDL-Beschreibung in den drei Schritten "Elaboration", "Initialization" und "Execution".

In der **"Elaboration"**-Phase wird das Netzwerk der Schaltung mit allen Hierarchieebenen, Blöcken und Signalen aus den compilierten VHDL-Modellen aufgebaut. Die Elaboration-Phase ist vergleichbar mit dem "Link"-Vorgang bei der Software-Entwicklung.

In der **"Initialization"**-Phase werden alle Signale, Variablen und Konstanten mit Anfangswerten versehen. Anfangswert ist entweder der Wert, der in der VHDL-Beschreibung durch explizite Angabe bei der Deklaration des Objektes vorgegeben wurde oder der Wert, der durch das `LEFT`-Attribut des entsprechenden Typs spezifiziert ist. Außerdem wird in dieser Phase jeder Prozeß einmal gestartet und bis zur ersten `WAIT`-Anweisung bzw. bis zum Ende ausgeführt.

In der **"Execution"**-Phase wird die eigentliche Simulation durchgeführt. Zu jedem Zeitpunkt der Simulation erfolgen bis zum Eintreten eines stabilen Zustandes ein oder mehrere Delta-Zyklen. Anschließend wird die Simulationszeit bis zum nächsten Eintrag in der Ereignisliste erhöht. Die Simulation ist beendet, wenn eine spezifizierte Simulationsdauer erreicht ist oder wenn keine weiteren Signaländerungen mehr auftreten.

1.4 Testumgebungen

Zu einer kompletten Beschreibung eines elektronischen Systems in VHDL gehört auch eine Testumgebung (im Englischen "Testbench"). Darunter versteht man die Bereitstellung von Eingangssignalen (Stimuli) und die Überprüfung der Ausgangssignale (Ist-Antworten) mit den erwarteten Werten (Soll-Antworten). Testumgebungen können

---

1 LRM = Language Reference Manual

auch dazu verwendet werden, verschiedene Architekturen einer Entity miteinander zu vergleichen: ein Verhaltensmodell auf RT-Ebene kann z.B. mit dessen Syntheseergebnis (Gatternetzliste) verglichen werden.

Für die Bereitstellung der Stimuli und der Soll-Antworten sowie für die Instantiierung des zu testenden Modells ("model under test", MUT) sind verschiedene Strategien denkbar.

Die kompakteste Möglichkeit besteht darin, in der Testbench selbst Stimuli zu beschreiben und die Antworten des Modells zu überprüfen. Dies kann z.B. in getrennten Prozessen erfolgen. In dieser Testbench wird gleichzeitig auch das MUT instantiiert und mit den Stimuli bzw. Antwortsignalen verdrahtet (siehe Abb. C-2).

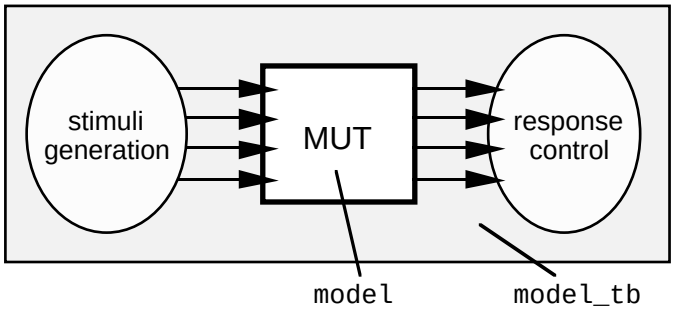

Abb. C-2: Testbenchstrategie mit einem VHDL-Modell

Daneben können Stimulibeschreibung und Antwortkontrolle auch in einem oder zwei unabhängigen VHDL-Modellen erfolgen (siehe Abb. C-3). Die Testbench dient in diesem Fall nur der Zusammenschaltung der zwei bzw. drei Modelle. Sie ist also rein struktural.

Eine Testbenchstrategie, die auf mehreren Modellen basiert, ist aufwendiger zu erstellen, als eine aus einem einzigen Modell bestehende Testbench. Allerdings bietet eine feinere Strukturierung den Vorteil, daß sich die einzelnen Modelle leichter in anderen Entwürfen wiederverwenden lassen und die Stimuli-Datensätze einfacher ausgewechselt werden können.

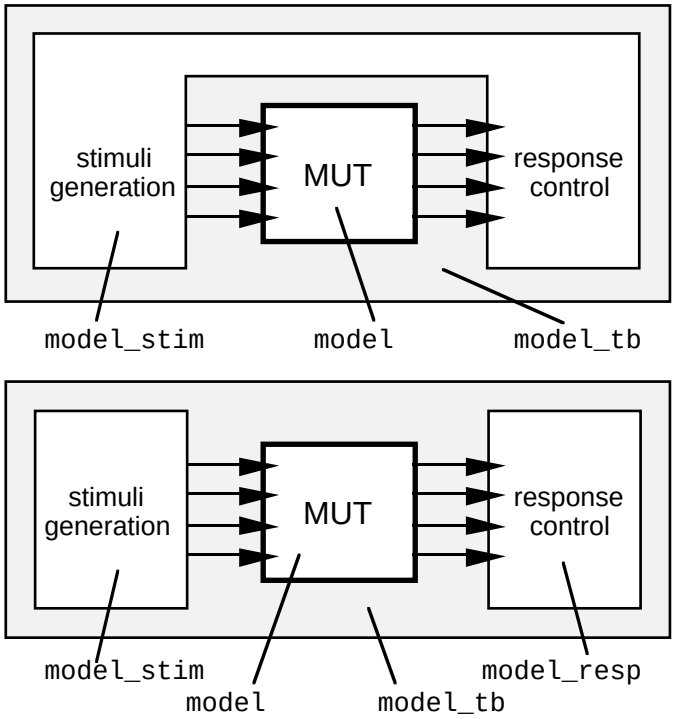

Abb. C-3: Testbenchstrategie mit zwei oder drei VHDL-Modellen

Bei der Beschreibung der Stimuli selbst bietet sich eine kombinierte Zuweisung von mehreren Signalen in einer Signalzuweisung an. Dazu ist i.d.R. ein qualifizierter Ausdruck erforderlich.

Folgendes Beispiel beschreibt die Stimuli für ein NAND2-Modell und überprüft die Antworten jeweils 2 ns danach. Die Testbench ist nach der erstgenannten Strategie angelegt.

```
ENTITY nand2_tb IS                    -- Testbench-Entity
END nand2_tb;                         -- ohne Ports
```



```
ARCHITECTURE strategy_1 OF nand2_tb IS
   COMPONENT nand2_socket
      PORT (in1, in2 : IN bit ; out1 : OUT bit);
   END COMPONENT;
   SIGNAL a,b,c : bit;
   SUBTYPE t2 IS bit_vector (1 TO 2);
BEGIN
----- Instantiierung des Model under Test (mut) ------------
   mut : nand2_socket PORT MAP (a,b,c);
----- Beschreibung der Eingangssignale (Stimuli) -----------
   stimuli_generation: PROCESS
   BEGIN
      -- eine Zuweisung zum Zeitpunkt 0 ns
      (a, b) <= t2'("01") AFTER 10 ns,  -- qualifizierter
                t2'("10") AFTER 20 ns,  -- Ausdruck t2'(...)
                t2'("11") AFTER 30 ns,
                t2'("00") AFTER 40 ns;
      WAIT;
   END PROCESS;
----- Ueberpruefung der Modellantworten (responses) --------
   response_control : PROCESS
   BEGIN
      WAIT FOR 12 ns;                   -- absolut: 12 ns
      ASSERT c='1' REPORT "wrong result" SEVERITY note;
      WAIT FOR 10 ns;                   -- absolut: 22 ns
      ASSERT c='1' REPORT "wrong result" SEVERITY note;
      WAIT FOR 10 ns;                   -- absolut: 32 ns
      ASSERT c='0' REPORT "wrong result" SEVERITY note;
      WAIT FOR 10 ns;                   -- absolut: 42 ns
      ASSERT c='1' REPORT "wrong result" SEVERITY note;
      WAIT;
   END PROCESS;
END strategy_1;
```

Die Signalzuweisung der Stimuli erfolgt im ersten Prozeß komplett zum Zeitnullpunkt. Sie könnte alternativ auch als nebenläufige Anweisung erfolgen. Die Assertions im zweiten Prozeß hingegen müssen, durch `WAIT`-Anweisungen gesteuert, zum entsprechenden Zeitpunkt ausgeführt werden.

Werden bei der Stimulibeschreibung mehrere Einzelanweisungen verwendet, so ist das "Transport"-Verzögerungsmodell einzusetzen, da

sonst durch das "Inertial"-Verzögerungsmodell die vorhergehenden Signalwechsel wieder gelöscht werden:

```
single_step_stimuli_generation : PROCESS
BEGIN
   -- 4 Zuweisungen zum Zeitpunkt 0 ns
   (a, b) <= TRANSPORT t2'("01") AFTER 10 ns;
   (a, b) <= TRANSPORT t2'("10") AFTER 20 ns;
   (a, b) <= TRANSPORT t2'("11") AFTER 30 ns;
   (a, b) <= TRANSPORT t2'("00") AFTER 40 ns;
   WAIT;
END PROCESS;
```

Es können alternativ dazu die Stimuli auch zu den entsprechenden Zeiten erzeugt werden. Dies ist mit `WAIT`-Anweisungen zu steuern. Denkbar wäre in diesem Fall auch eine Kombination mit der Antwortkontrolle:

```
multiple_step_stimuli_generation : PROCESS
BEGIN  -- 4 Zuweisungen zu verschiedenen Zeitpunkten
   WAIT FOR 10 ns; (a, b) <= t2'("01");  -- absolut 10 ns
   WAIT FOR 10 ns; (a, b) <= t2'("10");  -- absolut 20 ns
   WAIT FOR 10 ns; (a, b) <= t2'("11");  -- absolut 30 ns
   WAIT FOR 10 ns; (a, b) <= t2'("00");  -- absolut 40 ns
   WAIT;
END PROCESS;
```

```
stimuli_generation_and_response_control : PROCESS
BEGIN
   WAIT FOR 10 ns; (a, b) <= t2'("01");  -- absolut 10 ns
   WAIT FOR  2 ns;                       -- absolut 12 ns
   ASSERT c='1' REPORT "wrong result" SEVERITY note;
   WAIT FOR  8 ns; (a, b) <= t2'("10");  -- absolut 20 ns
   WAIT FOR  2 ns;                       -- absolut 22 ns
   ASSERT c='1' REPORT "wrong result" SEVERITY note;
   WAIT FOR  8 ns; (a, b) <= t2'("11");  -- absolut 30 ns
   WAIT FOR  2 ns;                       -- absolut 32 ns
   ASSERT c='0' REPORT "wrong result" SEVERITY note;
-------- Fortsetzung auf naechster Seite ------------------
```

```
-------- Fortsetzung von vorhergehender Seite --------------
      WAIT FOR  8 ns; (a, b) <= t2'("00");  -- absolut 40 ns
      WAIT FOR  2 ns;                       -- absolut 42 ns
      ASSERT c='1' REPORT "wrong result" SEVERITY note;
      WAIT;
   END PROCESS;
```

Zur Beschreibung regelmäßiger oder komplexer Stimuli eignen sich die sequentiellen Konstrukte der Sprache VHDL. Es sei hier auf eine der Übungsaufgaben in Teil D zur Erzeugung eines Taktsignals verwiesen. Die Stimuli für einen 8-Bit Decoder können etwa auf folgende Art beschrieben werden:

```
ENTITY dec_stim IS
   PORT (stim : OUT bit_vector(7 DOWNTO 0));
END dec_stim;

ARCHITECTURE behavioral OF dec_stim IS
-- Funktion zur Wandlung einer Integerzahl in e. Bit-Vektor
   FUNCTION integer_to_bit (a: integer) RETURN bit_vector IS
      ...
      ...
   END integer_to_bit;
BEGIN
-- Zyklische Ausgabe von 0 bis 255 -------------------------
   stimuli_generation: PROCESS
      VARIABLE a : integer := 0;
   BEGIN
      WAIT FOR 10 ns;  -- Stimulifrequenz: 100 MHz
      stim <= integer_to_bit(a);
      a := a + 1;
      IF a > 255 THEN a := a - 256; END IF;
   END PROCESS;
END behavioral;
```

1.5 Simulation von VHDL-Gatternetzlisten

Beim Einsatz von VHDL zur Dokumentation und Simulation von elektronischen Schaltungen beschränkt man sich hauptsächlich auf abstrakte Beschreibungsebenen. Typischerweise wird VHDL auf Systemebene, Algorithmischer Ebene und auf der Register-Transfer-Ebene als Eingabeformat für Syntheseprogramme eingesetzt. Bei der Verifikation der generierten Netzliste hingegen setzt man oft noch auf spezielle Digitalsimulatoren. Da VHDL leistungsfähige Konzepte zum Vergleich von Beschreibungen auf RT-Ebene und Logikebene anbietet (Auflösungsfunktionen, Assertions), drängt sich ein Einsatz auch auf dieser Ebene auf. Dies wird momentan jedoch durch zwei Probleme erschwert:

1.5.1 Performance-Nachteile

VHDL-Simulatoren arbeiten auf der Logikebene heute noch wesentlich langsamer als reine Digitalsimulatoren, die speziell für diesen Zweck entwickelt wurden. Führende Softwarehersteller haben jedoch angekündigt, daß die Leistung ihrer VHDL-Simulatoren entweder bald (d.h. bis Mitte 1994) die Leistung der konventionellen Digitalsimulatoren erreichen werde oder daß sie ihren VHDL-Simulator mit dem Digitalsimulator verschmelzen wollen.

1.5.2 Verfügbarkeit von Technologiebibliotheken

Zur Zeit sind kaum verifizierte VHDL-Gattermodelle in technologiespezifischen Bibliotheken verfügbar.

Einen Lösungsansatz zur Beseitigung dieses Engpasses könnte die Initiative VITAL[1] darstellen. Sie dient dem Zweck, ASIC-Bibliotheken für VHDL schneller verfügbar zu machen. Der Grundstein hierfür wurde 1992 beim "VHDL International Users Forum" (VIUF) und auf der "Design Automation Conference" (DAC) gelegt. Im Oktober 1992

---

1 VITAL = VHDL Initiative Towards ASIC Libraries

wurde bereits eine technische Spezifikation vorgelegt. VITAL umfaßt eine Beschreibungsform für das Zeitverhalten in ASIC-Modellen durch ein spezielles Format, SDF ("standard delay format"), und ermöglicht den Zugriff auf Standardbibliotheken der Hersteller.

VITAL erfreut sich einer starken Unterstützung durch CAE- und ASIC-Hersteller. Die Softwarehersteller wollen unmittelbar nach Festlegung des technologieunabhängigen Standards ihre Werkzeuge anpassen. Falls zu diesem Zeitpunkt auch leistungsfähigere Simulatoren zur Verfügung stehen, dürfte die VHDL-Simulation auf Logikebene keine Nachteile gegenüber der Simulation mit speziellen Digitalsimulatoren mehr aufweisen.

2 Synthese

2.1 Synthesearten

Unter Synthese versteht man allgemein den Übergang von der formalen Beschreibung eines Verhaltens zu einer dieses Verhalten realisierenden Struktur. Abhängig vom Abstraktionsgrad der Beschreibung, die als Eingabe für die Synthese dient, spricht man von Logiksynthese, Register-Transfer-Synthese, Algorithmischer Synthese und Systemsynthese. Während die Systemsynthese gegenwärtig noch vom Entwickler von Hand durchzuführen ist, stehen für die übrigen Synthesearten bereits Programme zur Verfügung. Die meisten beschränken sich dabei jedoch auf die Register-Transfer-Ebene oder Logikebene.

2.1.1 Systemsynthese

Auf der Systemebene wird ein Modul global durch seine Leistung und Funktion beschrieben. Die Systemsynthese entwickelt aus einer formalen Spezifikation einzelne Teilprozesse und entscheidet aufgrund der Vorgaben über einen günstigen Parallelitätsgrad in der Abarbeitung der Prozesse. Es ergibt sich eine Grobstruktur aus mehreren Subsystemen.

2.1.2 Algorithmische Synthese

Die Algorithmische Synthese transformiert ein Verhaltensmodell in eine Struktur auf Register-Transfer-Ebene. Das Verhaltensmodell enthält dabei lediglich den Algorithmus, der die Eingabedaten in Ausgabedaten überführt. Die Darstellung erfolgt mittels einer Hardwarebeschreibung, die Sequenzen, Iterationen und Verzweigungen enthält.

Auf der resultierenden Register-Transfer-Ebene wird die Schaltung durch eine Struktur aus Registern, Funktionseinheiten (z.B. Addierer, Multiplizierer, Komparatoren, etc.), Multiplexern und Verbindungs-

strukturen beschrieben. Zur Ansteuerung der Hardwaremodule wird die Zustandsübergangstabelle eines endlichen Zustandsautomaten (FSM = Finite State Machine) generiert.

Grundprinzip der algorithmischen Synthese ist meistens die Umsetzung der algorithmischen Beschreibung in einen Datenfluß- und einen Kontrollflußgraphen (Abb. C-4).

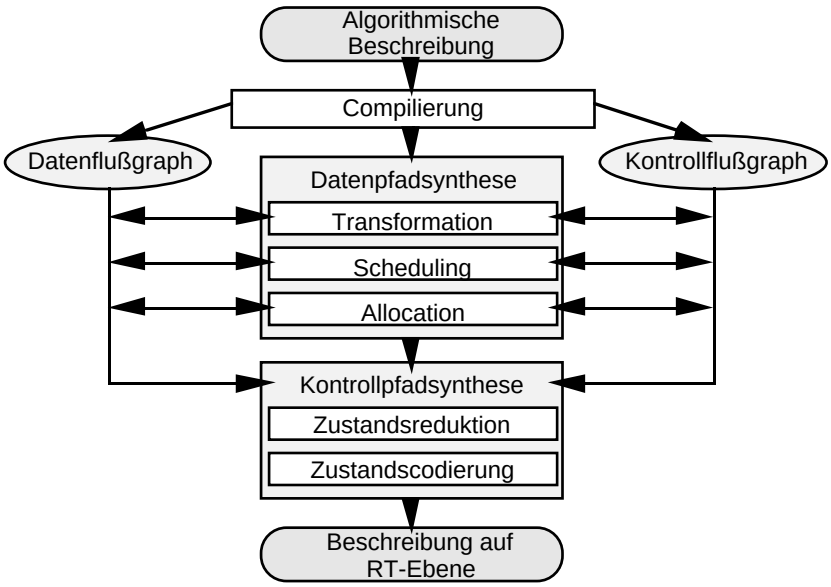

Abb. C-4: Ablauf der Algorithmischen Synthese [BIT 92]

Mit dem Datenflußgraphen werden die einzelnen Operationen, die die Eingangssignale in Ausgangssignale überführen, beschrieben. Die Knoten des Datenflußgraphen repräsentieren die verschiedenen Operationen; Kanten geben Variablen oder Konstanten wieder und definieren die Abhängigkeiten der Operatoren. Der Graph muß nicht zusammenhängend sein, parallele Abläufe sind möglich. Der zeitliche Ablauf der einzelnen Operationen wird dagegen im Kontrollflußgraphen abgebildet. Die Knoten dieses Graphen sind die Zustände des endlichen Automaten, die Kanten die Zustandsübergänge.

Die Synthese des Datenpfades besteht in der Realisierung des spezifizierten Algorithmus mit einer geeigneten Auswahl und Anzahl von Hardwaremodulen (z.B. Addierer, Register, Speicher sowie Multiplexer

und Busse zur Verbindung der Funktionseinheiten). Zur Ansteuerung dieser Module generiert die Kontrollpfadsynthese einen endlichen Automaten, der die Datenpfadregister lädt, Multiplexer und Busse schaltet und die auszuführenden Operationen auswählt.

2.1.3 Register-Transfer-Synthese

Auf Register-Transfer-Ebene wird der Datenfluß einer Schaltung dargestellt. Die Beschreibung enthält neben Funktions- auch Strukturangaben des Entwurfs. Die dabei verwendeten Signale und Zuweisungen an Signale können nahezu direkt in eine Struktur aus Registern und Verarbeitungseinheiten ("Transfers") zwischen den Registern übertragen werden.

Während auf Algorithmischer Ebene eine Schaltung aus **imperativer** Sicht, d.h. der Sicht des Steuerwerks, beschrieben wird, das die einzelnen Aktionen sequentiell zu vorhergehenden Aktionen anstößt, wird das Verhalten auf RT-Ebene aus der **reaktiven** Sicht der Elemente dargestellt. Das bedeutet, daß die einzelnen Objekte "beobachten", ob eine bestimmte Triggerbedingung wahr wird, und dann entsprechende Aktionen ausführen.

Die einzelnen Komponenten der Register-Transfer-Ebene sind keiner Ordnung unterworfen. Als typische Sprachelemente zur Definition der Objekte dienen sog. "guarded commands". Sie repräsentieren Verarbeitungen, die dann aktiviert werden, wenn bestimmte Ereignisse eintreten. Die Register-Transfer-Ebene enthält außerdem ein bestimmtes Synchronisationsschema, das durch die Triggerbedingungen definiert ist.

Ein Beispiel:

> Ausgegangen wird von einer Schaltung, die unter anderem zwei Register enthält, welche bidirektional mit zwei Bussen verbunden sind. Auf einer ALU können Additionen, Subtraktionen und UND-Verknüpfungen durchgeführt werden. Das Operationsergebnis wird in einem dritten Register gespeichert, welches auf die beiden Busse schreiben kann. Die steigende Flanke des Taktes triggert die Operatoren und Register.

Die Register-Transfer-Synthese setzt das Verhaltensmodell eins-zu-eins in eine äquivalente Blockstruktur auf gleicher Ebene um. Ähnlich der algorithmischen Synthese werden Kontroll- und Datenpfad getrennt behandelt.

Bei der Datenpfadsynthese wird zunächst festgestellt, welche Signale gespeichert werden müssen. Dafür werden Flip-Flops bzw. Register angelegt. Die Art der Flip-Flops (D-Flip-Flop, J-K-Flip-Flop, getriggert durch steigende oder fallende Flanke, etc.) gibt die Triggerbedingung und Blockbeschreibung vor. Für die Verarbeitung der nicht zu speichernden Signale werden nur kombinatorische Logik und Verbindungsleitungen vorgesehen.

Durch Analyse der Signalabhängigkeiten können die Verbindungsstrukturen ermittelt werden. Diese bestehen aus dedizierten Leitungen und Bussen sowie aus Treibern und Multiplexern zwischen den Registern und zwischen Registern und Operationseinheiten.

Die Kontrollpfadsynthese erzeugt die Steuerung der Register-Transfers aus den Triggerbedingungen. Wie in der Algorithmischen Synthese wird dazu ein Automat angelegt, um Treiber und Multiplexer anzusteuern. Das Übergangsnetzwerk des Automaten, das durch Boolesche Gleichungen repräsentiert wird, kann mit der anschließenden Logiksynthese implementiert werden.

Aus den Ergebnissen der Daten- und Kontrollpfadsynthese wird meist eine generische Netzliste aus den Elementen einer technologieunabhängigen Bibliothek erzeugt. In dieser Bibliothek sind die Gatterstrukturen der einzelnen Elemente hinterlegt. Die Abbildung der Gatter auf eine technologiespezifische Bibliothek geschieht im Technology Mapping der Logiksynthese.

2.1.4 Logiksynthese

Bei der Logiksynthese werden die realisierungsunabhängigen Booleschen Beschreibungen der kombinatorischen Netzwerke (Multiplexer, Operatoren, Übergangsnetzwerke der Automaten, etc.) optimiert und anschließend mit den Elementen der gewählten Zieltechnologie aufgebaut. Diese technologiespezifische Netzliste wird wiederum optimiert, um die Benutzervorgaben zu erreichen.

Folgende Einzelschritte laufen bei der Logiksynthese mit kommerziellen Werkzeugen im allgemeinen ab:

Flattening

Alle Zwischenvariablen der Booleschen Ausdrücke werden zunächst entfernt und alle Klammern aufgelöst. Damit erhält man beispielsweise eine zweistufige AND-OR-INVERT-Darstellung:

- Vor dem Flattening:
  $f = f1 \wedge f2;$ mit: $f1 = a \vee (e \wedge (c \vee d));$ $f2 = c \vee b$
- Nach dem Flattening:
  $f = (a \wedge c) \vee (a \wedge b) \vee (c \wedge e) \vee (e \wedge d \wedge c) \vee (e \wedge c \wedge b) \vee (e \wedge d \wedge b)$

Das Flattening löst also die vorgegebene Struktur der Logik auf. Die Auflösung eines vorher strukturierten Blockes in seine Produktterme kann eventuell bzgl. Geschwindigkeit und Fläche schlechtere Ergebnisse liefern. Bei unstrukturierter, krauser Logik ist es aber möglich, durch die anschließende Minimierung sowohl schnellere als auch kleinere Schaltungen zu erzeugen. Weil durch das Auflösen der Zwischenterme große Datenmengen entstehen, kann in den meisten Synthesesystemen der Grad des Flattenings vorgegeben werden.

Logikminimierung

Die Darstellung aus Produkttermen wird mit Minimierungsverfahren, wie z.B. dem Nelson-Verfahren, weiterverarbeitet. Jede Funktion kann dabei einzeln oder innerhalb eines Funktionsbündels minimiert werden. Die Anzahl der Produktterme reduziert sich dadurch und redundante Logik wird entfernt.

Structuring

Beim Structuring oder Factoring werden gemeinsame Unterausdrücke ausgeklammert und als temporäre Variablen verwendet. Die Schaltung erhält erneut eine Struktur. Dabei wird zunächst ein Liste angelegt, die die möglichen Faktoren enthält. Die Bewertung der Faktoren (benötigte Halbleiterfläche, Anzahl ihrer Verwendung) wird so oft wiederholt, bis kein neuer Faktor ermittelt werden kann, der die Schaltung verbessert. Die Faktoren, die die Logik am stärksten reduzieren, werden zu temporären Variablen. Ein Beispiel zum Structuring:

- Vor dem Structuring:
  $f = (a \wedge d) \vee (a \wedge e) \vee (b \wedge c \wedge d) \vee (b \wedge c \wedge e)$
- Nach dem Structuring:
  $f = t0 \wedge t1;$ mit: $t0 = a \vee (b \wedge c);$ $t1 = d \vee e$

Auswirkungen der Optimierungen

Die richtige Anwendung der verwendeten Strategien beim Einsatz eines Synthesewerkzeuges hat entscheidenden Einfluß auf das Syntheseergebnis. Spaltet man ein Design durch Flattening vollständig in Produktterme auf, minimiert anschließend die einzelnen Funktionen getrennt und verzichtet auf Structuring, so erhält man eine große, aber sehr schnelle Schaltung, da nur wenige Logikstufen zwischen den Ein- und Ausgängen liegen. Wird die ursprüngliche Struktur allerdings beibehalten und zusätzlich das Structuring verwendet, so kann eine kleine, aber langsame Schaltung entstehen.

Die Strategie, mit der eine Schaltung optimiert werden kann, hängt von verschiedenen Faktoren ab, wie ihre Komplexität, die Anzahl der Ein- und Ausgänge oder die Güte der vorgegebenen Struktur. Dadurch bietet es sich an, mit Synthesewerkzeugen verschiedene Möglichkeiten auszuprobieren.

Technology Mapping

Vor dem Technology Mapping ist die synthetisierte Schaltung noch technologieunabhängig. Das Technology Mapping setzt die optimierte Logik und die Flip-Flops in die Gatter einer bestimmten Technologiebibliothek um.

Zunächst wird die Schaltung vollständig mit technologiespezifischen Gattern abgebildet. Durch lokales Neuarrangieren von Komponenten oder Verwendung von Bausteinen mit unterschiedlicher Anzahl an Eingängen wird versucht, die "constraints" des Entwicklers zu erfüllen. Die benutzerdefinierten Einschränkungen beziehen sich neben einer maximal zulässigen Fläche, maximaler Laufzeit oder Taktrate auch auf die Setup- und Hold-Zeiten für die Flip-Flops. Für diese Parameter wird eine Kostenfunktion erstellt, die durch geeignete Partitionierung und lokale Substitutionen von Gatterkonfigurationen minimiert wird. Hierbei werden heuristische Techniken angewandt.

Ein wichtiges Leistungsmerkmal eines Mapping-Algorithmus ist seine universelle Anwendbarkeit auf verschiedene Technologiebibliotheken, die sehr unterschiedliche Komplexgatter und Makrofunktionen enthalten können. Ein Problem ist die schnelle Zunahme an benötigter Rechenzeit bei großen Schaltungen mit vielen Gattern.

Abb. C-5 zeigt zwei Beispiele zum Technology Mapping. Beispielsweise muß bei diesem Vorgang eine logische Verknüpfung mit vier Eingangsvariablen durch eine funktional äquivalente Verknüpfung aus drei Gattern mit jeweils zwei Eingangsvariablen ersetzt werden, da nur diese in der Technologiebibliothek verfügbar sind (oberes Beispiel). Eine Einsparung von Gattern beim Mapping ist unter anderem möglich, wenn in der Bibliothek Module mit negierten Ausgängen vorliegen (unteres Beispiel).

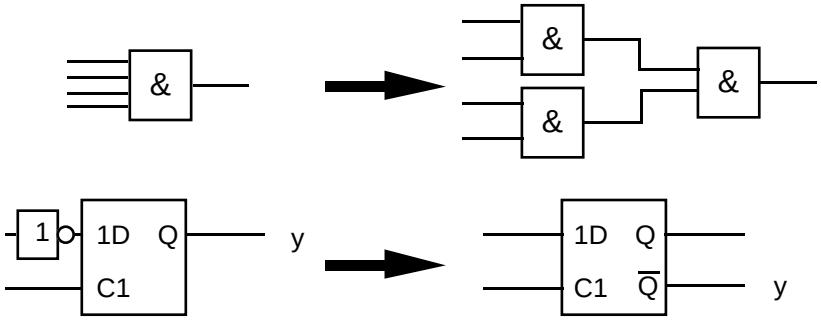

Abb. C-5: Beispiele zum Technology Mapping

Auch manuell erstellte Netzlisten können mit den Logiksynthese-Werkzeugen optimiert oder von einer Technologiebibliothek auf eine andere umgesetzt werden. Das Ergebnis der Synthese kann als technologiespezifische Netzliste (z.B. im EDIF oder VHDL-Format) ausgegeben werden.

2.2 Einsatz der Syntheseprogramme

In diesem und in allen weiteren Abschnitten zur Synthese von VHDL-Beschreibungen wird nur noch auf Werkzeuge eingegangen, die eine Umsetzung von einer RT-Beschreibung in eine Gatternetzliste unter-

stützen, da bei der Anwendung von VHDL z.Zt. diese Werkzeuge die größte Bedeutung haben.

2.2.1 Umstellung von Schematic Entry

Der Einsatz von Syntheseprogrammen bedeutet nicht, daß die komplette bisherige Entwurfsumgebung durch eine neue ersetzt wird. Nach [SYN 92] kann eine Umstellung vom gewöhnlichen Schematic-Entry (Graphische Schaltungseingabe auf Logikebene) auf die Synthesewerkzeuge in mehreren Schritten erfolgen:

① Nachträgliche Optimierung der manuell erstellten Schaltungen mit Syntheseprogrammen,

② nachträgliche Optimierung und Verbesserung der Testbarkeit,

③ teilweise Erfassung des Entwurfs mit VHDL und Synthese,

④ volle Beschreibung des Entwurfs mit VHDL und Synthese.

Es ist zu beachten, daß auch durch die Verwendung von Synthesewerkzeugen Iterationszyklen nicht vermieden werden können. Diese spielen sich lediglich auf einer anderen Ebene ab (*"We´ve got still the same problems, but on a higher level"*). Die Synthesewerkzeuge bieten dem Entwickler aber die Möglichkeit, die Ebene des Entwurfs von der Logikebene auf die abstraktere Register-Transfer-Ebene zu verlagern, wodurch Komplexitäten besser beherrscht werden. Ganz losgelöst von der "Hardware" (der Logikebene) kann sich jedoch kein Entwickler bewegen, weil nur über die Analyse des Syntheseergebnisses auf Logikebene auf eine geeignete VHDL-Modellierungstechnik auf RT-Ebene geschlossen werden kann.

2.2.2 Zielsetzung

Eine der wichtigsten Überlegungen beim Einsatz eines Syntheseprogramms gilt dem Ziel, Einfluß auf die Optimierung der Schaltung zu nehmen.

2.2.2.1 Optionen und Randbedingungen

Jedes Syntheseprogramm bietet die Möglichkeit, durch Einstellung von bestimmten Optionen Randbedingungen bzw. Optimierungskriterien bei der Synthese vorzugeben (Setzen von "constraints"). Dies können neben der Auswahl von optimaler Fläche oder optimaler Laufzeit meist noch Laufzeitgrenzen für einzelne Pfade und weitere Randbedingungen, wie z.B. die Vorgabe einer Zustandscodierung, sein.

Bei vielen Syntheseprogrammen zeigt sich jedoch, daß mit den jeweiligen Einstellungen der Randbedingungen (Fläche, Laufzeit) nicht optimal auf die Schaltung Einfluß genommen werden kann. Für die Generierung der Schaltung in Abhängigkeit von den Randbedingungen werden nämlich teilweise Heuristiken verwendet, so daß man nie sicher sein kann, wirklich die optimale Schaltung erzeugt zu haben. Eine Schaltung, die beispielsweise eine Laufzeitvorgabe von 0 ns (also möglichst schnell) hatte, kann langsamer sein als eine mit der Laufzeitvorgabe 30 ns.

2.2.2.2 Modellierungsstil

Durch geschickte Verwendung von VHDL-Sprachelementen kann bereits bei der Modellierung eine Entscheidung über eine mehr oder weniger geeignete Schaltungsarchitektur getroffen werden. Dazu ist erstens eine detaillierte Kenntnis von VHDL und zweitens das Wissen über die spätere Umsetzung der VHDL-Sprachkonstrukte durch das eingesetzte Synthesewerkzeug erforderlich.

Beispiel:

> Einen Zähler kann man beispielsweise als Zustandsautomaten oder als sequentiellen Zähler modellieren, wobei nicht immer eine Beschreibungsart die optimale ist. In vielen Fällen, insbesondere bei 2er-Potenzen als Zähllängen, ergibt sich bei der sequentiellen Beschreibung ein besseres Ergebnis, während bei Zähllängen, die knapp über einer 2er-Potenz liegen (z.B. 129), die Realisierung als Automat aufgrund der umfangreichen Minimierung der Übergangslogik (viele freie Zustände) günstiger ist.

Im folgenden soll deshalb anhand einiger VHDL-Beispiele illustriert werden, welchen Einfluß der Modellierungsstil und die gewählten Randbedingungen ("Constraints") auf das Syntheseergebnis haben.

Für diese Betrachtungen wurden mehrere kommerzielle Syntheseprogramme herangezogen, um programmspezifische Besonderheiten ausmitteln zu können. Einschränkungen hinsichtlich der Verwendbarkeit von VHDL-Anweisungen und des Beschreibungsstils werden mit zukünftigen Programmversionen zunehmend geringer werden.

2.3 Synthese von kombinatorischen Schaltungen

In diesem Abschnitt soll dargestellt werden, wie VHDL-Modelle von kombinatorischen Funktionen in Schaltungsarchitekturen umgesetzt werden. Nähere Angaben finden sich in den Dokumentationen zu den jeweiligen Synthesewerkzeugen.

2.3.1 Einführung

In VHDL gibt es zwei Möglichkeiten, kombinatorische Schaltungen zu beschreiben: die Modellierung mit Hilfe nebenläufiger Anweisungen und mit Hilfe sequentieller Anweisungen (innerhalb von Prozessen und Unterprogrammen).

Am einfachen Beispiel eines achtfachen NAND-Gatters (siehe Abb. C-6) sollen vier verschiedene Beschreibungsarten gezeigt werden.

Dieses und alle weiteren VHDL-Beispiele verwenden dabei das IEEE-Package `std_logic_1164` mit dem 9-wertigen Logiktyp `std_ulogic`.

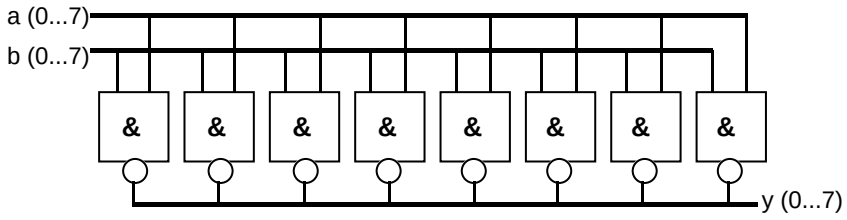

Abb. C-6: Schaltbild des 8-fach NAND-Gatters

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY nand2 IS
   PORT (a,b: IN  std_ulogic_vector (0 TO 7);
         y:   OUT std_ulogic_vector (0 TO 7));
END ENTITY;
```

Die Architekturen `one` und `two` verwenden die überladene Funktion `NAND` aus dem Package `std_logic_1164` in vektorieller und Einzelbitversion.

```
ARCHITECTURE one OF nand2 IS
BEGIN
   y <= a NAND b;
END one;
```

```
ARCHITECTURE two OF nand2 IS
BEGIN
   PROCESS (a,b)
   BEGIN
      FOR i IN a'RANGE LOOP
         y(i) <= a(i) NAND b(i);
      END LOOP;
   END PROCESS;
END two;
```

Die folgenden beiden Architekturen (`three` und `four`) verwenden logische Gleichungen zur Beschreibung der Funktion. Sie haben den Nachteil, daß im Rahmen einer funktionalen Simulation beim Auftreten von Signalen wie `'X'`, `'Z'` oder `'H'` an einem Eingang das Verhalten des Gatters nicht korrekt modelliert ist, da in diesem Fall der Ausgang den Wert `'1'` annehmen würde. Infolgedessen ist es hier zu empfehlen, den vordefinierten `NAND`-Operator aus dem IEEE-Package (wie in Architektur `one` oder `two`) zu verwenden.

```
ARCHITECTURE three OF nand2 IS
BEGIN
   y(0) <= '0' WHEN a(0)='1' AND b(0)='1' ELSE '1';
   ...
   y(7) <= '0' WHEN a(7)='1' AND b(7)='1' ELSE '1';
END three;
```

```
ARCHITECTURE four OF nand2 IS
BEGIN
   PROCESS (a,b)
   BEGIN
      FOR i IN a'RANGE LOOP
         CASE a(i) & b(i) IS
            WHEN "11"   => y(i) <= '0';
            WHEN OTHERS => y(i) <= '1';
         END CASE;
      END LOOP;
   END PROCESS;
END four;
```

Bei der Synthese einer derart einfachen Beschreibung bestehen keine Unterschiede in der Implementierung der verschiedenen Beschreibungsarten. Für die Architekturen `three` und `four` wird zwar zunächst eine recht komplizierte Schaltungsarchitektur generiert, diese dann aber bei der Optimierung wieder in acht einfache NAND-Gatter aufgelöst. Man benötigt mit diesen Versionen lediglich größere Rechenzeiten bei der Synthese.

2.3.2 Verzweigungen

Anhand des folgenden Modells wird betrachtet, wie die Verzweigungsanweisungen `IF` und `CASE` in Hardware umgesetzt werden:

```
ENTITY if_und_case IS
  PORT (i:     IN  integer RANGE 0 TO 9;
        a,b,c: IN  std_ulogic_vector (7 DOWNTO 0);
        z:     OUT std_ulogic_vector (7 DOWNTO 0) );
END if_und_case;
```

```
ARCHITECTURE if_variante OF if_und_case IS
BEGIN
  p1: PROCESS (i,a,b,c)
  BEGIN
    IF    (i = 3) THEN z <= a;
    ELSIF (i < 3) THEN z <= b;
    ELSE               z <= c;
    END IF;
  END PROCESS p1;
END if_variante;
```

```
ARCHITECTURE case_variante OF if_und_case IS
BEGIN
  p1: PROCESS (i,a,b,c)
  BEGIN
    CASE i IS
      WHEN 3      => z <= a;
      WHEN 0 TO 2 => z <= b;
      WHEN OTHERS => z <= c;
    END CASE;
  END PROCESS p1;
END case_variante;
```

Die beiden Architekturen `if_variante` und `case_variante` sind funktional identisch. Bei der Synthese jedoch werden unterschiedliche Schaltungen erzeugt. Das liegt daran, daß eine `IF`-Anweisung grundsätzlich eine Priorität beinhaltet, nämlich die bevorzugte Abfrage des ersten Zweiges (hier: `i=3`). Bei der `CASE`-Anweisung sind dagegen alle Zweige gleichberechtigt. Nachstehende Schaltungen

ergeben sich zuerst aufgrund der obigen Beschreibungen. Bei der anschließenden Optimierung werden aber dann in der Regel wieder identische Schaltungen erzeugt.

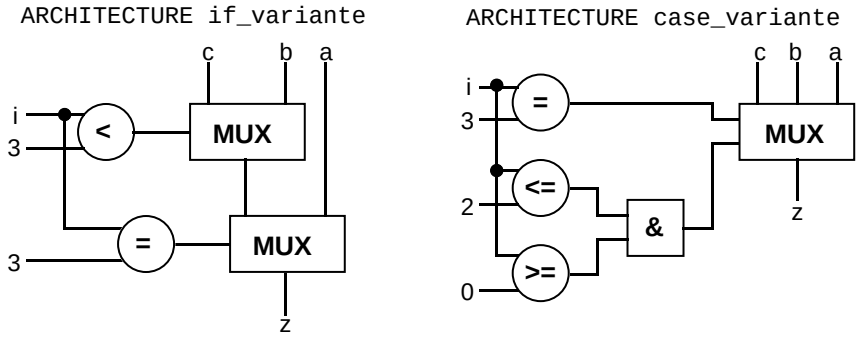

Abb. C-7: Schaltbild der IF- und der CASE-Variante

2.3.3 Signale und Variablen

Bei der Beschreibung von Algorithmen ist normalerweise die Speicherung von Zwischenergebnissen notwendig. Dazu könnten prinzipiell Signale oder Variablen verwendet werden. Da Zuweisungen an Signale immer erst ein "Delta" später wirksam werden, führt der Einsatz von Signalen als Zwischenspeicher in Algorithmen jedoch häufig zu Modellierungsfehlern und damit zu unerwarteten Syntheseergebnissen.

Zur Illustration soll hier ein Beispiel gezeigt werden, bei dem mit Hilfe einer Schleife eine regelmäßige Schaltungsstruktur (XOR-Kette) beschrieben werden soll:

```
ENTITY kette IS
  PORT ( hbyte: IN  std_ulogic_vector (0 TO 3) := "0000";
         value: OUT std_ulogic );
END kette;
```

```
ARCHITECTURE richtig OF kette IS
BEGIN
  PROCESS (hbyte)
    VARIABLE merker: std_ulogic := '0';
  BEGIN
    merker := '0';
    FOR i IN hbyte'RANGE LOOP
      merker := merker XOR hbyte(i);
    END LOOP;
    value <= merker;
  END PROCESS;
END richtig;
```

```
ARCHITECTURE falsch OF kette IS
  SIGNAL merker: std_ulogic := '0';
BEGIN
  PROCESS (hbyte)
  BEGIN
    FOR i IN hbyte'RANGE LOOP
      merker <= merker XOR hbyte(i);
    END LOOP;
  END PROCESS;
  value <= merker;
END falsch;
```

Bei der Architektur `richtig` wird die Variable `merker` in der Schleife der Reihe nach mit allen Bits des Signals `hbyte` verknüpft, so daß primär bei der Synthese eine Kette von XOR-Gattern entsteht (siehe Abb. C-8).

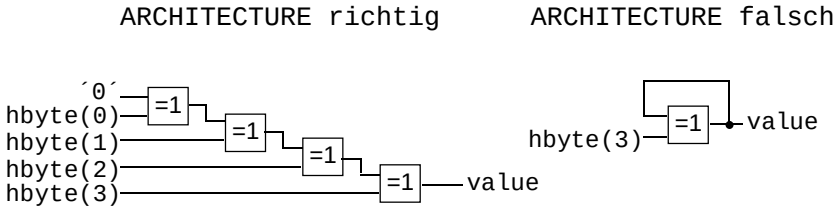

Abb. C-8: Syntheseergebnisse des VHDL-Modells kette

Bei der Architektur `falsch` hingegen werden Signale eingesetzt. Da Signalzuweisungen im Prozeß nicht sofort ausgeführt werden, werden die Zuweisungen an `merker` nicht wirksam, so daß nur noch eine Zuweisung, nämlich "`merker <= merker XOR hbyte(3)`" verbleibt, also ein einzelnes rückgekoppeltes XOR-Gatter.

2.3.4 Arithmetische Operatoren

Um die Probleme bei der Umsetzung von arithmetischen Operatoren näher zu beleuchten, wird das Beispiel eines 4-Bit-Volladdierers aufgegriffen. Dessen Schnittstellenbeschreibung lautet:

```
ENTITY addierer IS
  PORT ( a,b:  IN  std_logic_vector (3 DOWNTO 0);
         cin:  IN  std_logic;
         s:    OUT std_logic_vector (3 DOWNTO 0);
         cout: OUT std_logic );
END addierer;
```

Die Eingangssignale `a` und `b` stellen die Summanden des Addierers dar. Die Ports `cin` und `cout` entsprechen dem Übertragsbit ("carry") auf der Ein- bzw. Ausgangsseite. Das vier Bit breite Ausgangssignal `s` schließlich steht für die Summe.

Im folgenden werden verschiedene VHDL-Beschreibungen des Addierer-Verhaltens betrachtet. Die Architektur `zwei_plus` beschreibt den Addierer sehr einfach, indem die beiden Eingangssignale und der Carry-Eingang durch Hinzufügen von `"0"`-Stellen mit zwei Pluszeichen verknüpft werden:

```
ARCHITECTURE zwei_plus OF addierer IS
  SIGNAL temp: std_logic_vector (4 DOWNTO 0);      -- 5 Bit
BEGIN
  temp <= ("0" & a) + ("0" & b) + ("0000" & cin);
  cout <= temp(4);                              -- Uebertrag
  s <= temp(3 DOWNTO 0);                        -- Summe
END zwei_plus;
```

Da zu erwarten ist, daß manche Syntheseprogramme aus zwei Pluszeichen auch zwei Addierer aufbauen, wird die Beschreibung in der Architektur `ein_plus` auf ein Pluszeichen reduziert:

```
ARCHITECTURE ein_plus OF addierer IS
  SIGNAL temp: std_logic_vector (5 DOWNTO 0);  -- 6 Bit
BEGIN
  temp <= ("0" & a & cin) + ("0" & b & "1");
  cout <= temp(5);                              -- Uebertrag
  s <= temp(4 DOWNTO 1);                        -- Summe
END ein_plus;
```

Als Alternative zu diesen beiden funktionalen Beschreibungen kann man aber auch "hardware-orientiert" modellieren und beispielsweise direkt eine "Ripple-Carry-Struktur" vorgeben:

```
ARCHITECTURE ripple OF addierer IS
  SIGNAL c: std_logic_vector (3 DOWNTO 0);
BEGIN
  s <= (a XOR b) XOR (c(2 DOWNTO 0) & cin);
  c <= ((a XOR b) AND (c(2 DOWNTO 0) & cin)) OR (a AND b);
  cout <= c(3);
END ripple;
```

Natürlich kann man alternativ auch eine "Carry-Look-Ahead-Struktur" beschreiben:

```
ARCHITECTURE cla OF addierer IS
  SIGNAL c:   std_logic_vector (2 DOWNTO 0);
  SIGNAL p,g: std_logic_vector (3 DOWNTO 0);
BEGIN
  p <= a XOR b;
  g <= a AND b;
  s <= p XOR (c & cin);
  c(0) <= g(0) OR (p(0) AND cin);
  c(1) <= g(1) OR (p(1) AND c(0));
  c(2) <= g(2) OR (p(2) AND c(1));
  cout <= g(3) OR (p(3) AND c(2));
END cla;
```

Bei der Synthese der verschiedenen Architekturen ergibt sich, daß mit `ein_plus` meistens bessere Ergebnisse erreicht werden als mit `zwei_plus`. Abb. C-9 zeigt die Ergebnisse (Fläche in Gatteräquivalenten, GÄ; Laufzeit in ns) bei der Synthese von 4-Bit- und 32-Bit-Addierern mit den vier unterschiedlichen Architekturen, jeweils auf minimale Fläche (4F, 32F) bzw. minimale Laufzeit (4L, 32L) optimiert:

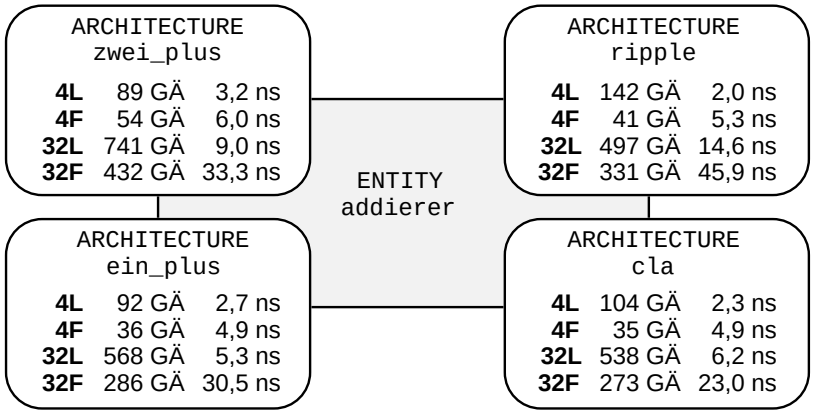

Abb. C-9: Syntheseergebnis verschiedener Addiererarchitekturen

Abb. C-9 zeigt, daß sich das Syntheseergebnis sowohl durch die "constraints" (Optimierungsbedingungen) als auch durch den Modellierungsstil erheblich beeinflussen läßt. Dies gilt für alle betrachteten

Syntheseprogramme. Besonders fällt hier auf, daß die Architektur `cla` (Carry-Look-Ahead) meist auf eine kleinere Schaltung als die Ripple-Carry-Architektur führt. Beim manuellen Entwurf hätte man durch Einsatz einer Ripple-Carry-Architektur eine kleinere Schaltung erzielt als mit der Carry-Look-Ahead-Architektur. Dies zeigt, daß die Syntheseprogramme die gegebenen Gleichungen nicht als Addierer erkennen können und in diesem Fall die Gleichungen für den Carry-Look-Ahead-Addierer offensichtlich mehr Spielraum für die Flächenoptimierung bieten.

Bei neueren Versionen der Syntheseprogramme ist die Verwendung von arithmetischen Operatoren sinnvoller ist als die Angabe von logischen Gleichungen, da die Programme oft optimierte Module für die Synthese von arithmetischen Operatoren zur Verfügung stellen.

2.3.5 Schleifen

Von den drei Schleifenarten, die in VHDL möglich sind (`FOR`-Schleife, `WHILE`-Schleife und Endlosschleife) wird nur die `FOR`-Schleife von den Syntheseprogrammen uneingeschränkt unterstützt, weil hier die Anzahl der Schleifendurchläufe vor der Ausführung der Schleife bekannt ist.

Das folgende Beispiel zeigt, wie durch einen für die Simulation korrekten, aber für die Synthese ungünstigen Einsatz von Schleifen ein hoher Flächenaufwand impliziert wird. Dazu soll das Modell eines Barrel-Shifters betrachtet werden, der 8-Bit breite Daten (`data_in`) in Abhängigkeit von einem Steuersignal (`adresse`) um bis zu sieben Stellen rotieren kann.

```
ENTITY barrel IS
   PORT (data_in  : IN  std_ulogic_vector (7 DOWNTO 0);
         adresse  : IN  std_logic_vector  (2 DOWNTO 0);
         data_out : OUT std_ulogic_vector (7 DOWNTO 0));
END barrel;
```

```
ARCHITECTURE eins OF barrel IS
   SIGNAL adr : integer RANGE 0 TO 7 := 0;
BEGIN
   adr <= conv_integer(adresse); -- Konvertierung in integer
   PROCESS (data_in, adr)
   BEGIN
      FOR i IN 7 DOWNTO 0 LOOP
         data_out((i + adr) mod 8) <= data_in(i);
      END LOOP;
   END PROCESS;
END eins;
```

```
ARCHITECTURE zwei OF barrel IS
   SIGNAL adr : integer RANGE 0 TO 7;
BEGIN
   PROCESS (data_in, adresse)
      VARIABLE puffer : std_ulogic_vector(7 DOWNTO 0);
   BEGIN
      puffer := data_in;
      FOR i IN 0 TO 2 LOOP
         IF (adresse(i) = '1') THEN
            puffer := puffer(7-2**i DOWNTO 0) &
                      puffer(7 DOWNTO 7-2**i+1);
         END IF;
      END LOOP;
      data_out <= puffer;
   END PROCESS;
END zwei;
```

Bei der Synthese der Architektur `eins` ist zu erkennen, daß für jeden Schleifendurchlauf ein eigener Addierer generiert wird, also insgesamt acht Addierer entstehen. Zwar können diese Addierer bei entsprechender Optimierungseinstellung zum Teil wieder reduziert werden, aber die Schaltung bleibt für immer größer und langsamer als bei geschickter Modellierung unter Verwendung einer Puffervariablen (Architekur `zwei`). Eine Untersuchung der Synthese mit fünf verschiedenen Beschreibungsvarianten für einen 32-Bit-Linksrotierer ergab Gatteräquivalente zwischen 480 und 3096 und Laufzeiten zwischen 9,5 und 72,5 ns. Hieraus wird ersichtlich, daß man, zumindest in einigen extre-

men Fällen, durch ungeschickte Modellierung einige 100% an zusätzlicher Logik erzeugen kann.

Die Ergebnisse können dahingehend verallgemeinert werden, daß Schleifen zur Verarbeitung von einzelnen Signalen vermieden werden sollten, da dies zu einer Vervielfachung der Hardware führt.

2.3.6 Zusammenfassung

Nach Untersuchung einer Vielzahl von kombinatorischen Schaltungen unter Verwendung von diversen Modellierungsarten zeichnen sich folgende Ergebnisse ab:

- Bei **kleinen Schaltungen** hängen die Syntheseergebnisse **nicht** von der **Art der Beschreibung** ab. Es spielt also keine Rolle, ob eine bestimmte Funktion mit `IF`, `CASE`, `SELECT` oder durch direkte Angabe der logischen Funktion mit arithmetischen und Booleschen Operatoren beschrieben wird.
- Bei **großen Schaltungen** hingegen ist eine tabellarische Beschreibung der Funktion mit Hilfe von sequentiellen Anweisungen (`IF`, `CASE` etc.) kaum noch möglich. Hier müssen möglichst **einfache und kompakte Operatoren** gewählt werden. Gegenüber eigenen Beschreibungen der Funktionalität haben diese Operatoren darüberhinaus den Vorteil, daß sie vom Syntheseprogramm besser interpretiert werden können, d.h. durch geeignetes Setzen von "constraints" kann man i.d.R. die Optimierungsziele leichter erreichen.
- Innerhalb algorithmischer Beschreibungen sollten **Variablen** verwendet werden. Das Ergebnis des Algorithmus wird dann den **Signalen** oder Ports zugewiesen (vgl. Beispiel "`kette`").
- In einigen Fällen kann man durch eine komplexere, **hardwarenähere Modellierung** auch ein besseres Syntheseergebnis erzielen (sieht man vom Beispiel des "Ripple-Carry-Addierers" einmal ab).
- Beim Einsatz von **Schleifen** zur Abarbeitung der einzelnen Elemente eines Vektors wird oft vielfache Logik erzeugt (vgl. Bsp. "`barrel`"). Schleifen sollten deshalb nur dann eingesetzt werden, wenn genau dieses erwünscht ist (vgl. Bsp. "`kette`").

2.4 Synthese von sequentiellen Schaltungen

2.4.1 Latches

Gegeben sei folgende VHDL-Beschreibung:

```
ENTITY was_ist_das IS
  PORT ( a,b: IN  bit;
         c:   OUT bit );
END was_ist_das;

ARCHITECTURE behave OF was_ist_das IS
BEGIN
  PROCESS (a,b)
  BEGIN
    IF (a = '0') THEN c <= b;
    END IF;
  END PROCESS;
END behave;
```

Wenn das Signal `a` auf `'0'` liegt, wird dem Ausgang `c` der Wert des Eingangs `b` zugewiesen. Im Falle `a = '1'` erfolgt keine explizite Zuweisung des Ausgangs `c`, so daß der vorhergehende Wert beibehalten wird. Dies bedeutet aber wiederum, daß der Wert gespeichert werden muß. Folglich beschreibt das obige Modell ein Speicherelement, ein D-Latch (Abb. C-10). Bei diesem Latch ist `a` das Enable-Signal, `b` der Dateneingang und `c` der Ausgang. Bei `a = '0'` ist das Latch transparent, so daß alle Änderungen des Eingangs `b` sofort am Ausgang `c` erscheinen. Bei `a = '1'` ist das Latch gesperrt (Halten), so daß Änderungen des Eingangs `b` sich nicht auf den Ausgang `c` auswirken.

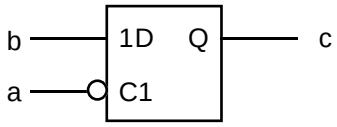

Abb. C-10: Schaltbild eines D-Latch

Aus dieser Beschreibung kann man nicht nur ableiten, wie man prinzipiell ein Latch modelliert, sondern auch die Gefahr der Synthese unerwünschter Speicherelemente bei unvollständigen IF-Anweisungen in VHDL-Modellen erkennen. Immer dann, wenn in einer IF-Anweisung bestimmte Signale nur in einem Teil der Zweige auf der linken Seite von Signalzuweisungen stehen, muß ein Speicherelement erzeugt werden. Dies gilt auch für unvollständige Zuweisungen in CASE-Anweisungen.

2.4.2 Flip-Flops

Flip-Flops unterscheiden sich von Latches durch ihre **Taktflankensteuerung**. Zur Modellierung muß ein Pegelübergang an einem Taktsignal erkannt werden, wozu sich das VHDL-Attribut EVENT eignet. Dieses Attribut bezieht man auf das Taktsignal und plaziert es in einem Prozeß entweder in einer WAIT- oder in einer IF-Anweisung:

```
ENTITY dff IS
  PORT (clk,d: IN  std_ulogic;
        q:     OUT std_ulogic);
END dff;
```

```
ARCHITECTURE variante1 OF dff IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1';
    q <= d;
  END PROCESS;
END variante1;
```

```
ARCHITECTURE variante2 OF dff IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk'EVENT AND clk = '1' THEN
      q <= d;
    END IF;
  END PROCESS;
END variante2;
```

Beide Architekturen beschreiben ein D-Flip-Flop. Streng genommen sind sie jedoch nicht ganz korrekt, denn der Fall eines Wechsels am Signal `clk` von `'X'` nach `'1'` ist nicht erfaßt. Die Beschreibungen würden dann fälschlich eine steigende Taktflanke erkennen. Eine zusätzliche Überprüfung, ob das Taktsignal vorher `'0'` war, kann mit Hilfe des Attributs `LAST_VALUE` geschehen:

```
ARCHITECTURE variante3 OF dff IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1'
                         AND clk'LAST_VALUE = '0';
    q <= d;
  END PROCESS;
END variante3;
```

```
ARCHITECTURE variante4 OF dff IS
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk'EVENT AND clk = '1' AND clk'LAST_VALUE = '0'
      THEN q <= d;
    END IF;
  END PROCESS;
END variante4;
```

Da eine Erkennung von steigenden oder fallenden Flanken eines Signals häufig benötigt wird, sind die zwei Funktionen `RISING_EDGE` und `FALLING_EDGE` in das IEEE-Package integriert worden. Sie enthalten die Beschreibung gemäß `variante4` und geben ein Signal vom Typ `boolean` zurück, welches `true` ist, wenn eine steigende bzw. fallende Flanke erkannt wurde. Allerdings wird das Attribut `LAST_VALUE`, und damit auch diese beiden Funktionen, nicht von allen Syntheseprogrammen unterstützt.

Wenn man statt des D-Flip-Flops ein T-Flip-Flop (Toggle-Flip-Flop) beschreiben möchte, kann man die folgende Architektur verwenden.

```
ENTITY t_ff IS
  PORT (clk, enable: IN     std_ulogic;
        q:           BUFFER std_ulogic := '0');
END t_ff;
```

```
ARCHITECTURE behavioral OF t_ff IS
BEGIN
  PROCESS
  BEGIN
    WAIT UNTIL clk'EVENT AND clk = '1' AND
      clk'LAST_VALUE = '0';
    IF enable = '1' THEN q <= not q;
    END IF;
  END PROCESS;
END behavioral;
```

Mit dieser Beschreibung wird von den Syntheseprogrammen entweder ein Toggle-Flip-Flop oder, falls ein solches in der Technologiebibliothek nicht vorhanden ist, ein D-Flip-Flop mit vorgeschaltetem XOR-Gatter eingesetzt.

Zu beachten ist, daß gewünschte **asynchrone Eingänge** von Speicherelementen nicht automatisch von einem Syntheseprogramm erzeugt werden, sondern in VHDL beschrieben werden müssen. Am Beispiel eines D-Flip-Flops mit asynchronem Rücksetzeingang sei dies gezeigt:

```
ENTITY dff IS
  PORT (clk,d,reset: IN  std_ulogic;
        q:           OUT std_ulogic );
END dff;
```

```
ARCHITECTURE async_reset OF dff IS
BEGIN
  PROCESS (clk,reset)
  BEGIN
   IF reset = '0' THEN               -- low-aktives Reset
     q <= '0';                       -- hat erste Prioritaet
   ELSIF clk'EVENT AND clk = '1' AND -- Abfrage auf Taktfl.
     clk'LAST_VALUE = '0' THEN       -- nur, wenn kein reset
     q <= d;
   END IF;
  END PROCESS;
END async_reset;
```

2.4.3 Zustandsautomaten

Bei der Modellierung von endlichen Zustandsautomaten (FSM) muß man zwischen Mealy-, Moore- und Medvedev-Automaten unterscheiden. Bei einem Mealy-Automaten hängt der Ausgangsvektor vom momentanen Zustand und vom Eingangsvektor ab, beim Moore-Automaten dagegen nur vom Zustand. Ein Medvedev-Automat ist dadurch gekennzeichnet, daß jeder Ausgang des Automaten mit dem Ausgang eines Zustands-Flip-Flops identisch ist. Abb. C-11 beschreibt ein Blockschaltbild für den Mealy-Automatentyp.

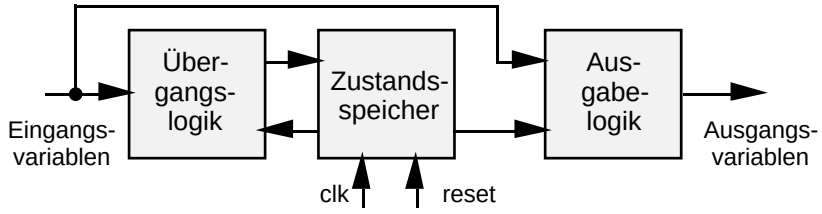

Abb. C-11: Blockschaltbild eines Zustandsautomaten (Mealy)

Die folgenden drei VHDL-Prozesse zeigen die prinzipielle, synthesegerechte Modellierung eines Mealy-Automaten. Die Blöcke aus Abb. C-11 sind hier in getrennten Prozessen realisiert.

```
zustandsspeicher: PROCESS (clk, reset)
BEGIN
  IF (reset = '1') THEN
    zustand <= reset_zustand;
  ELSIF (clk'event AND clk='1' AND clk'LAST_VALUE = '0')
    THEN zustand <= folge_zustand;
  END IF;
END PROCESS zustandsspeicher;
```

```
uebergangslogik: PROCESS (zustand, in1, in2, ...)
BEGIN
  CASE zustand IS
    WHEN zustand1 =>
      IF (in1 = ... AND in2 = ... AND ...)
         THEN folge_zustand <= ...;
      ELSIF ...
      ...
    WHEN zustand2 =>
    ...
  END CASE;
END PROCESS uebergangslogik;
```

```
ausgabelogik: PROCESS (zustand, in1, in2, ...)
BEGIN
  CASE zustand IS
    WHEN zustand1 =>
      IF (in1 = ... AND in2 = ... AND ...)
        THEN out1 <= ...; out2 <= ...; ...
      ELSIF ...
      ...
    WHEN zustand2 =>
      ...
  END CASE;
END PROCESS ausgabelogik;
```

Da die Prozesse zur Beschreibung der Übergangslogik und der Ausgabelogik sehr ähnlich sind, können sie auch zusammengefaßt werden. Eine mögliche Fehlerquelle hierbei ist, daß Latches für die Ausgänge erzeugt werden, wenn diese nicht in jedem Zustand und bei jeder Kombination der Eingangssignale einen Wert zugewiesen bekommen.

Wenn man versucht, die FSM komplett in einem Prozeß zu modellieren, der lediglich vom Takt und dem Rücksetzsignal getriggert wird, werden Flip-Flops für die Ausgänge erzeugt. Als Folgerung daraus kann man empfehlen, bei Mealy- und Moore-Automaten einen Prozeß für die Zustandsspeicherung und einen weiteren Prozeß für den rein kombinatorischen Teil zu verwenden.

2.5 Optimierung der "Constraints"

Neben der Optimierung des VHDL-Modells können zur Verbesserung der Syntheseresultate bei jedem Syntheseprogramm eine Reihe von Optionen angegeben werden, mit denen man die gewünschte Zielsetzung bei der Synthese näher spezifizieren kann (sog. "constraints").

2.5.1 Ziele und Randbedingungen

Grundlegende Ziele bei der Synthese sind neben einer funktionierenden Schaltung:

- geringer Flächenbedarf,
- hohe Geschwindigkeit,
- geringe Verlustleistung.

Daneben lassen sich auch viel detailliertere Angaben machen:

- Festlegung der Treiberstärken an den primären Eingängen,
- Festlegung der Lastkapazitäten an den primären Ausgängen,
- Angabe von Schätzwerten bzw. Modellen zur Berücksichtigung von Leitungskapazitäten,
- Angaben zur Schwankungsbreite der Laufzeiten durch Variation von Temperatur, Versorgungsspannung und Prozeß.

Die angegebenen "constraints" beziehen sich auf die gesamte Schaltung. Meist ist es aber wünschenswert, sie speziell auf einen Teil der Schaltung zu beziehen. Denkbar wäre die Angabe einer maximalen Verzögerungszeit von einem Eingang zu einem Ausgang bei sonst möglichst kleiner Schaltung.

2.5.2 Constraint-Strategien

Es ist i.d.R. nicht sinnvoll, "constraints" für die Synthese intuitiv zu setzen. Die Erfahrung zeigt, daß man bessere Schaltungsergebnisse erhält, wenn man Randbedingungen nicht auf unmögliche Werte setzt (z.B. Laufzeit 0 ns), sondern Werte nahe des Machbaren verwendet. Dies liegt u.a. daran, daß bei unmöglichen Angaben die Gewichtungen bei den heuristischen Methoden der Synthesewerkzeuge ungünstig gesetzt werden.

Durch den Trade-Off Fläche-Geschwindigkeit liegen die optimalen Lösungen einer gegebenen Schaltung, aufgetragen in einem Diagramm Laufzeit über Fläche, auf einer Hyperbel. Da in der Praxis eine endliche Anzahl von nicht immer optimalen Syntheseergebnissen vorliegt, erhält man eine Reihe von diskreten Lösungen im sog. Entwurfsraum.

Abb. C-12 zeigt den Entwurfsraum eines Zustandsautomaten. Die Punkte im Entwurfsraum wurden durch viele Syntheseläufe einer VHDL-Beschreibung unter Verwendung verschiedenster "constraints" generiert. Leider steht ein solcher Überblick über die möglichen Lösungen zu Beginn der Synthese nicht zur Verfügung.

Das Ziel beim Einsatz von "constraints" ist es, dem Syntheseprogramm mitzuteilen, welche Realisierung gewünscht wird. Man kann allerdings kaum vorhersagen, welche "constraints" zu welcher Schaltung führen, so daß ein iteratives Vorgehen notwendig wird.

Man geht dabei sinnvollerweise so vor, daß man mit realen, d.h. leicht verwirklichbaren "constraints" beginnt und diese so lange verschärft, bis das Syntheseprogramm keine bessere Schaltung mehr liefert.

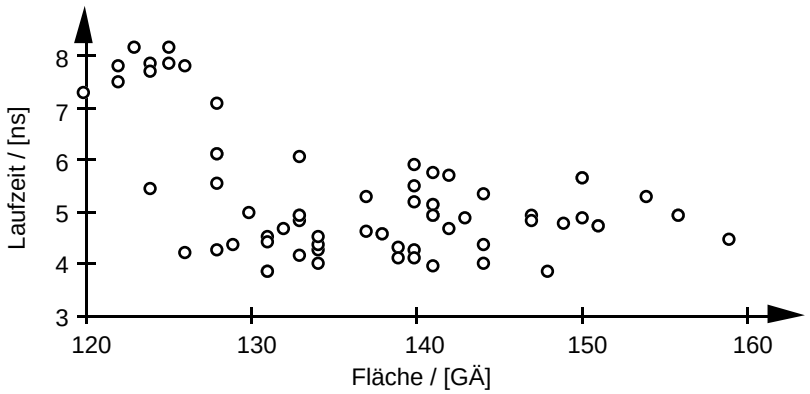

Abb. C-12: Entwurfsraum eines Zustandsautomaten

Am Beispiel des Zustandsautomaten sollen einige mögliche Strategien dargestellt und bewertet werden. Die Ergebnisse haben keine allgemeine Gültigkeit, sollen jedoch die grundsätzliche Problematik aufzeigen.

2.5.2.1 Laufzeit Null

Eine einfache Strategie besteht darin, die gewünschte Laufzeit der Schaltung, ohne Abschätzung der tatsächlich machbaren Laufzeit, auf "Null" zu setzen. Abb. C-13 zeigt das Ergebnis dieser Strategie im Entwurfsraum des erwähnten Beispiels (schwarz ausgefüllter Kreis).

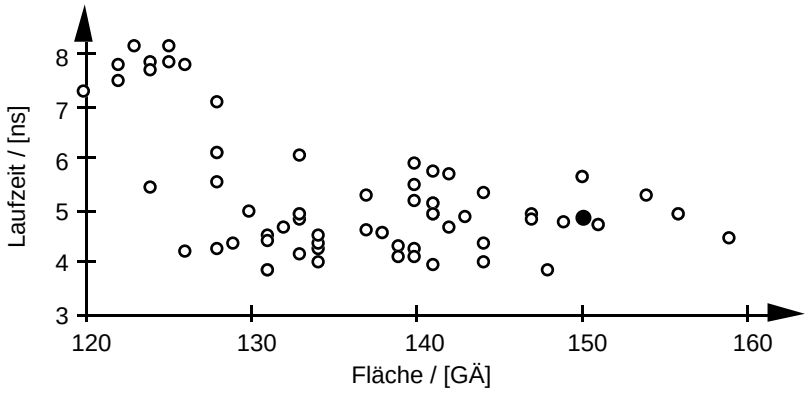

Abb. C-13: Ergebnis der Strategie "Laufzeit Null"

Wie aus der Abbildung ersichtlich ist, führt diese Strategie keineswegs zu einem guten Resultat.

2.5.2.2 Laufzeit "in der Nähe des Machbaren"

Eine andere Strategie besteht darin, ein Laufzeit-Constraint "in der Nähe des Machbaren" zu setzen. Hintergrund ist, daß bei der Verfolgung dieser Strategie die Gewichtungsfaktoren bei der Optimierung eine bessere Wirkung zeigen als bei der zu starken Gewichtung der Null-Laufzeit. Diese Strategie wird auch von vielen Syntheseprogrammherstellern empfohlen.

Für den vorliegenden Fall wurde das Laufzeit-Constraint einmal auf 3 ns und einmal auf 4 ns gesetzt. Die Ergebnisse sind als schwarz ausgefüllte Kreise in Abb. C-14 aufgetragen.

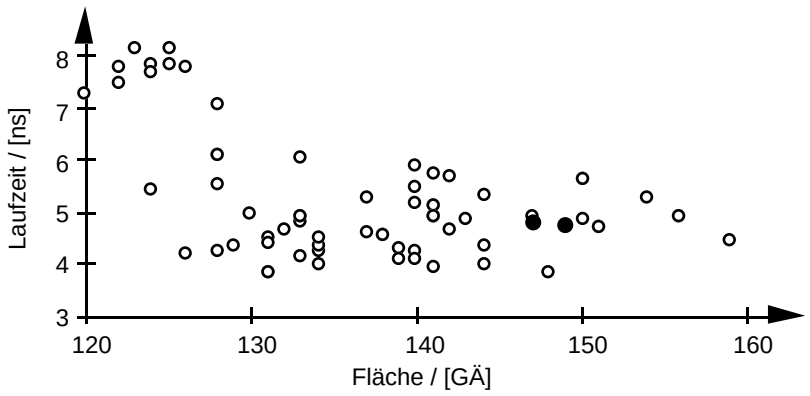

Abb. C-14: Ergebnisse der Strategie "machbare Laufzeit"

Auch damit werden nicht die schnellsten Schaltungen generiert. Vielmehr erhält man ähnliche Ergebnisse wie bei der ersten Strategie.

2.5.2.3 Mehrfache Optimierung

Eine weitere Strategie besteht darin, das Ergebnis obiger Strategien durch mehrere aufeinanderfolgende Syntheseläufe zu optimieren.

Eine neunfache Optimierung des Zustandsautomaten, auf der Basis der Laufzeit 3 ns, führt auf die in Abb. C-15 dargestellten Ergebnisse innerhalb des Entwurfsraums (schwarz ausgefüllte Kreise).

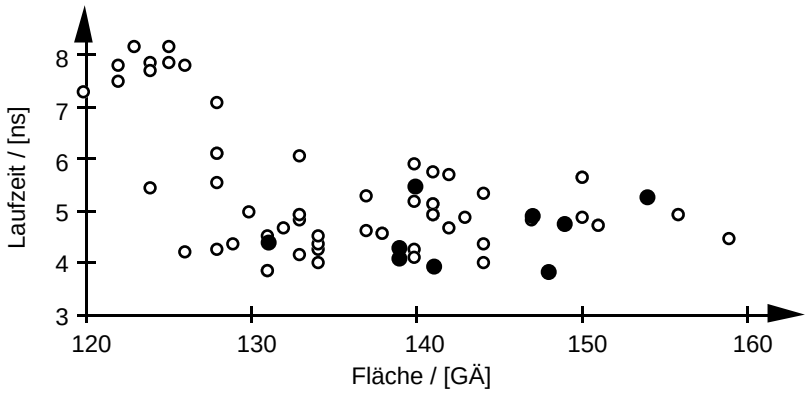

Abb. C-15: Ergebnisse der Strategie "mehrfache Optimierung"

Mit dieser Strategie erhält man eine der schnellsten Schaltungen im Feld mit ca. 4 ns Laufzeit. Aufgrund der mehrfachen Optimierung ergeben sich jedoch deutlich höhere Rechenzeiten. Außerdem läßt sich nicht vorhersagen, nach welcher Iteration das beste Ergebnis, d.h. ein globales Minimum, erreicht wird.

Abb. C-16 zeigt die nach jedem Iterationsschritt erreichte Laufzeit:

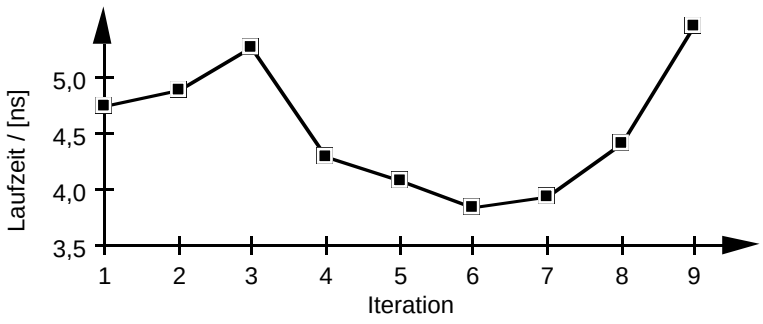

Abb. C-16: Laufzeit bei mehrfacher Zeitoptimierung

An den Ergebnissen des Beispiels läßt sich erkennen, daß eine wiederholte Optimierung nicht immer den gewünschten Erfolg mit sich bringt. Eine mehrfache Optimierung ist jedoch trotz erhöhter Rechenzeiten dem blinden Vertrauen auf ein gutes Ergebnis nach nur einem Syntheselauf vorzuziehen.

2.6 Ressourcenbedarf bei der Synthese

Neben den Performancedaten der von einem Syntheseprogramm erzeugten Schaltung spielt auch der Zeitbedarf für die Synthese eine wichtige Rolle. Kann bei einzelnen Syntheseläufen eine hohe Rechenzeit vielleicht noch akzeptiert werden, so wird bei größeren Entwürfen und mehreren Synthesedurchläufen der Faktor Rechenzeit unmittelbar bestimmend für die gesamte Entwurfszeit. Daneben spielen auch Anforderungen an die benötigte Hardwareplattform, der erforderliche Arbeits- und Swap-Speicher und der vom Programm benötigte Speicherplatz eine Rolle.

Verfügbare Syntheseprogramme bieten leider keine Möglichkeit, die Rechenzeit abzuschätzen oder gar ein Limit dafür zu setzen. Es ist lediglich möglich, die Anzahl der Optimierungszyklen durch Optionen auf niedrig, mittel oder hoch einzustellen. Der Anwender sollte also vor dem Start des Syntheseprozesses eine Vorstellung von der für die Synthese benötigten Rechenzeit haben. Die wichtigsten Einflußfaktoren dafür sind:

- Leistungsfähigkeit der Rechnerumgebung,
- Art des Syntheseprogramms,
- Art der VHDL-Beschreibung,
- gesetzte "constraints",
- verwendete Technologiebibliothek.

Die folgenden beiden Abbildungen zeigen die benötigte Rechenzeit (reine CPU-Zeit) für drei verschiedene Schaltungen, abhängig von der Größe der Schaltung, einmal für Flächenoptimierung und einmal für Laufzeitoptimierung. Zur Synthese wurde ein Rechner vom Typ SUN Sparc IPX mit 32 MB Arbeitsspeicher verwendet.

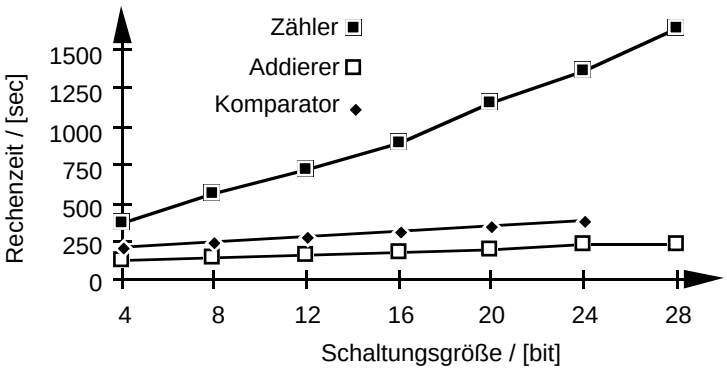

Abb. C-17: CPU-Zeit bei Flächenoptimierung

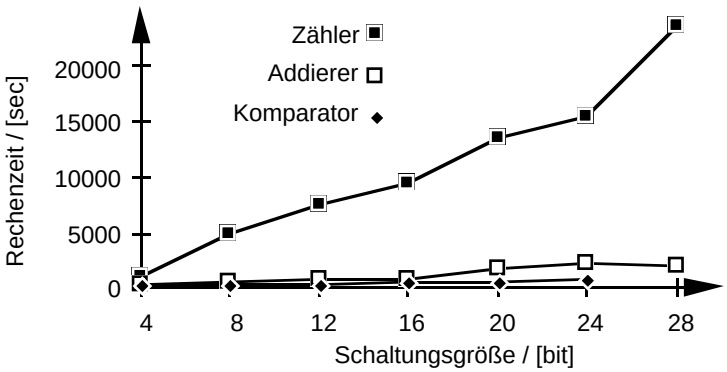

Abb. C-18: CPU-Zeit bei Laufzeitoptimierung

Die beiden Abbildungen zeigen deutlich, daß die Laufzeitoptimierung sehr viel mehr CPU-Zeit erfordert als die Optimierung auf geringste Fläche. Bei den dargestellten Schaltungen, die sich im Bereich von einigen hundert Gatteräquivalenten bewegen, ist ein Unterschied bis etwa zum Faktor 10 festzustellen, während bei größeren Schaltungen (mehrere tausend Gatteräquivalente) Unterschiede bis zum Faktor 100 auftreten.

Außerdem zeigen die letzten beiden Abbildungen, daß die Rechenzeiten für die Addierer- und Komparatorschaltung weit weniger von der Schaltungsgröße abhängig sind als die der Zählerschaltung. Ursache hierfür ist, daß das Synthesewerkzeug bei Addierern und Komparatoren auf programminterne Makros zurückgreifen kann.