Synopsys & Votan

Thorsten Kukuk

19. November 1994

Seminar
der
Projektgruppe Systementwurf
WS94/95 und SS 95

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

VHDL ist eine Sprache zur Beschreibung digitaler elektronischer Schaltungen. Sie wurde vom Institute of Electrical and Electronic Engineers (IEEE) als Standard akzeptiert. Diese Sprache beschreibt die einzelnen Bauteile sehr abstrakt. Um die von der Sprache beschriebenen Schaltungen als Hardware zu verwirklichen, muß die abstrakte Sprache auf Register-Ebene transformiert werden. Auf dieser Ebene können dann auch Optimierungen durchgeführt werden.

Für diese Aufgabe gibt es nun mehrere Programme. Zwei davon sind die Programmpakete Votan und Synopsys. Votan ist eine Art Precompiler. Es übersetzt Standart VHDL Code in vereinfachten Code für kommerzielle Programme. Synopsys ist ein Analyse Tool. Mit ihm können graphisch einzelne Bausteine und Schaltungen erzeugt werden, die auch als VHDL Code abgespeichert werden können. Diese Schaltungen können nun mit einem Debugger auf Fehler oder mit dem Analyser auf kritische Stellen untersucht werden. Beim Compilieren werden bestimmte Ziele beachtet, wie kleinste oder schnellste Schaltung.

2 Votan

VOTAN steht für VHDL Optimization, Transformation and Analysis. VOTAN wurde als einfach zu bedienendes Toolpaket geschrieben, um High-level Synthese in einem festgesetzten VHDL Fluß zu integrieren. Es verfolgt zwei Ziele: erstens, es kann als Precompiler für ein RT-Synthese Tool dienen und dafür Scheduling Transformationen durchführen. D.h. VOTAN ändert in einem gegebenen VHDL Text Konstrukte in equivalente Ausdrücke um. Dies ist nötig, da die meisten kommerziellen Tools nur einen Subscript von VHDL verstehen. Die zweite Intention von VOTAN ist den Abstraktionslevel von Beschreibungen zu erhöhen. Die Benutzung von Hardware-Beschreibungssprachen hat einen starken Einfluß auf den Design Prozeß. VOTAN ermöglicht einen strukturierten Programmierstil, indem es Prozeduren mit Wait Statements, Funktionen und willkürliche Schleifen und Sprünge erlaubt. Diese hierarchische Beschreibung, die einfacher zu schreiben und besser zu warten ist, kann in VHDL Code umgewandelt werden, der von kommerziellen RT-Level Synthese Tools synthesierbar ist.

Das Softwarepaket besteht aus der Ascii-Shell VOTAN und der aufgesetzten graphischen Oberfläche VOX zur interaktiven Design-Transformation. Das Interface VOX dient zur schnelleren Navigation durch die VHDL Files und Design Units und ermöglicht eine schnellere Erprobung unterschiedlicher Design Alternativen.

VHDL wird sowohl bei der Spezifikation als auch beim Design von elektronischen Systemen akzeptiert. Aber es ist noch eine große Lücke zwischen der VHDL Beschreibung, die zur Spezifikation von Systemen benutzt wird, und der Beschreibung, die von kommerziellen Tools verstanden wird.

VOTAN ist eine erweiterbare Plattform für kommerzielle RT-Level Systeme. Es ist als offene Synthese Plattform implementiert worden, in der neue Applikations Transformationen schnell und effektiv implementiert werden können. Dies macht VHDL Code unabhängig von dem Synthese Tool.

2.1 High-Level Synthese Design Fluß

In der aktuellen Version manipuliert VOTAN jeden einzelnen Prozeß individuell. Ein Prozeß ist ein Teil einer VHDL Architektur, der sich auf andere VHDL Strukturen beziehen kann. Der Anwender wählt den zu optimierenden Prozeß aus und führt die Transformationen entweder interaktiv oder im Batch Mode durch die Abarbeitung von Kommandoscripts aus.

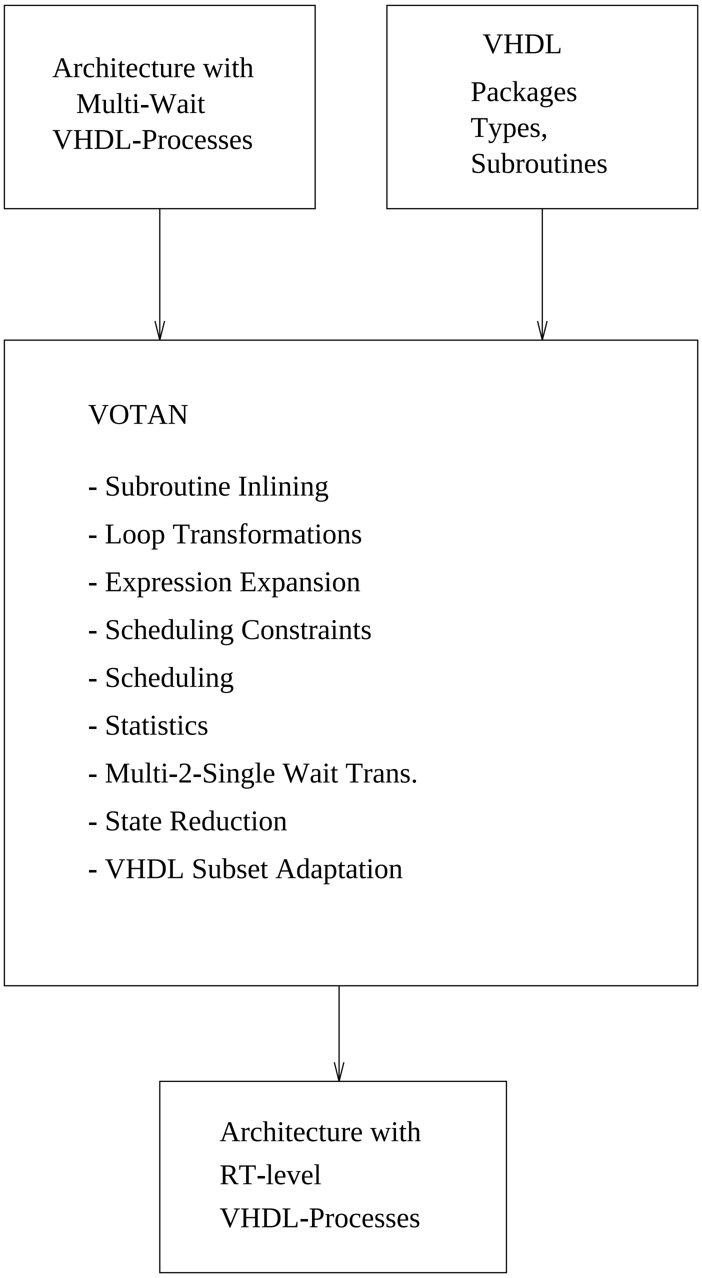


Abbildung 1: High Level Synthesis Design Fluß

Abbildung 1 zeigt so einen Synthese Fluß. Als Eingabe wird eine Architektur mit Multi-Wait Statements erwartet. Ausserdem werden eine Bibliothek mit den Typen und Subroutinen benötigt. Auf diese Eingabe werden nun die einzelnen Funktionen von VOTAN angewendet. Die Ausgabe besteht dann aus einer RT-Level Architektur, die sich einfach weiterverarbeiten läßt.

2.2 „Multiple-Wait“ Description Style

In VHDL sind die sequentiellen Befehle in einzelne Prozesse zusammengefaßt. Alle Strukturen wie Schleifen und Sprünge sind möglich, auch Prozeduren und Funktionen sind erlaubt.

Dies läßt einen Pascal ähnlichen Beschreibungsstil vom Schaltungsverhalten zu. Damit ist in VHDL ein gut strukturierter und hierarchischer Aufbau möglich, der in der Software Entwicklung inzwischen unerläßlich geworden ist. Das Ziel von VOTAN ist, diesen Stil auch für die Hardware Entwicklung zu ermöglichen.

Diese Beschreibung wird kombiniert mit der RT-Level Spezifikation vom Timing Verhalten. In der momentanen Version von VOTAN ist das I/O Timing durch ein Clock Signal beschränkt. D.h. Read-Write Operationen zwischen verschiedenen Bausteinen werden erst nach *wait until clk = '1'* Statements durchgeführt. Diese werden Wait-Statemants genannt.

Ein Prozeß kommuniziert mit seinem Environment über Signale. Eine Lese-Operation ist z.B. die Benutzung eines Input Signals auf der rechten Seite einer Zuweisung. Die einzigen Schreib-Operationen sind Signal Zuweisungen. Es ist zu beachten, daß das High-level VHDL-Subset kein „after clause“ in Signalzuweisungs Statements erlaubt. Dies bedeutet, daß, wenn ein Prozeß wegen der Ausführung eines *wait until clk ='1'* Statements suspendiert ist, die Ausgabe-Signalzuweisung erst mit einiger Verspätung außen sichtbar wird. Ein zweiter Prozeß, der vom gleichen Clock-Signal abhängt, kann diese Werte erst beim nächsten Clock-Signal lesen.

Die Benutzung von Wait-Statements und die Benutzung von „Sensitiv Lists“ schliessen sich gegenseitig aus. Ein Prozeß mit Wait-Statements ist implizit zu allen Signalen, die in einer „sensitive“ oder „condition clause“ des Wait-Statements benutzt werden, sensitiv. Im Falle des *wait until clk = '1'* Statements ist der Prozeß lediglich zum clk-Signal sensitiv.

Jedes Wait-Statement kann auch als Zustand einer Finite State Machine (FSM) angesehen werden. Alle Statements auf dem Weg zwischen zwei benachbarten Wait-Statements bestimmen den nächsten Zustand und die Output-Funktion der entsprechenden FSM. Jede Schleife, die bei der Synthese nicht aufgelöst wird, muß mind. einmal durch so ein „Wait“-Satement unterbrochen sein, um Rückkopplungen zu vermeiden. Dann existiert auch eine equivalente FSM, welche leicht durch die VHDL-Transformationen **Wait-Folding** oder **Multiple-2-Singel Wait Transformation** erzeugt werden kann. Das Ergebnis ist ein einem RT-Level entsprechender VHDL Prozeß in „goto“-Semantik, der von fast allen kommerziellen Synthese Produkten bearbeitet werden kann.

2.3 Einschränkungen in VHDL durch die Tools

Die momentane VOTAN Version ist hauptsächlich als Precompiler für den Synopsys VHDL Compiler gedacht, um dessen Einschränkungen und Schwierigkeiten zu umgehen.

Synopsys Einschränkungen und Lösungen mit VOTAN:

- „alias“ Definitionen werden nicht unterstützt
- „deferred“ Konstanten werden nicht unterstützt
- „guarded“ Signale und „guardes“ Signalzuweissungen werden nicht unterstützt
- „physical“ und „floating“ Typen werden nicht unterstützt
- „access“ und „file“ Typen werdern nicht unterstützt
- Nur die folgenden Formen von Wait-Statements werden unterstützt:
  - **wait until** clk = *value*;
  - **wait until** clk'event **and** clk = *value*;
  - **wait unitl not** clk'stable **and** clk = value;

wobei *value* entweder '0', '1' oder ein Enumeration Literal sein kann.

Durch diese Einschränkung ist es nicht möglich, das verhalten einer Hardware Komponente mit einen *asynchronous reset* Prozeß, der einen oder mehrere Wait-Statements enthält, zu beschreiben. Mit VOTAN können aber mit Hilfe der *fold -wait* Transformation für Multi-Wait Prozesse bzw. *transform -wait* für Single-Wait Prozesse solche Prozesse für Synopsys transformiert werden.

- Wait-Statements sind in „For Loops" und Prozeduren nicht erlaubt. VOTAN kann solche Konstrukte in equivalente Schleifen umwandeln bzw. Prozeduren werden Inlinemässig an der Aufrufstelle eingefügt.
- Die folgenden vordefinierten Attribute werden nicht unterstützt: 'leftof(x), 'rightof(x), 'pos(x), 'val(x), 'succ(x), 'pred(x)

2.4 Einschränkungen durch Transformation

2.4.1 Singe-Wait Prozesse

Prozesse, bei dennen nur genau das erste Statement ein Wait-Statement ist, werden Single-Wait Prozesse genannt. Sie können zu Prozessen mit einer „Sensitive List" anstelle des Wait-Statements transformiert werden. Prozesse mit einem „synchronous" oder „asynchronous reset" bestehen aus nur zwei Statements:

- dem Wait-Statement
- Einer if Abfrage mit einem einfachen Vergleich, der entweder zu den Reset- oder den Prozeß-Anweisungen verzweigt.

2.4.2 Multi-Wait Prozesse

Multi-Wait Prozesse werden benutzt, um eine Hardware Komponente auf einem höheren Abstraktionslevel zu beschreiben. Durch das Einfügen von zusätzlichen Zustandsvariablen und einer Case Anweisung sowie der Verdopplung einiger Anweisungen kann aus so einem Prozess ein Single-Wait Prozess erzeugt werden. Abbildung 2 zeigt ein Beispiel vor und nach der Transformation mit VOTAN. Der linke Teil ist ein Prozeß vor der Transformation, der rechte danach. Der orginal Code ist kürzer, übersichtlicher und enthält keine doppelten Anwseisungen. Der rechte ist unübersichtlicher und schwieriger zu warten. Wenn z.B. Statement 5 verändert wird, braucht die Änderung im orginal Code nur einmal vorgenommen werden. Im Single Wait Prozeß dagegen muß die Änderung auch für jede Kopie des Statements durchgeführt werden. Dieses führt viel schneller zu Fehlern. Damit die Transformation überhaupt möglich ist, muß der Multi-Wait Prozeß zwei Kriterien erfüllen:

1. Auf jedem Weg vom Prozeß Beginn zum Prozeß Ende muß mindestens ein Wait Statement stehen.
2. Auf jedem Weg von einem Schleifen Beginn zur letzten Anweisung in dieser Schleife oder zur ersten Anweisung nach der Schleife muß mindestens ein Wait Statement stehen.

```
p : process
begin
  wait until clk = 1;
  statement_1;
  if(condition) then
    statement_2;
    wait until clk = 1;
    statement_3;
  else
    statement_4
  end if;
  statement_5;
  wait until clk = 1;
  statement_6
end process p
```

```
p : process
  type state_type is (s1, s2, s3);
  variable state : state_type;
begin
  wait until clk = 1;
  case state is
    when s1 =>
      statement_1;
      if (condition) then
        statement_2;
        state := s2;
      else
        statement_4;
        statement_5; -- duplicated
        state := s3;
      end if;
    when s2 =>
      statement_3;
      statement_5; -- duplicated
      state := s3;
    when s3 =>
      statement_6;
      state := s1;
  end case;
end process p
```

Abbildung 2: Multiple Wait Prozeß ohne Reset, vor und nach der Transformation

2.5 Higl-Level VHDL Optimierung

2.5.1 Scheduling

Der Scheduling Schritt ermöglicht es, Zuweissungsoperationen auf verschiedene Zeitpunkte zu verteilen. Dieses unterscheidet High-Level Synthese von RT-Level Synthese. VOTAN bietet eine flexible Scheduling Technik an, die nicht wie viele andere Tools primär versucht, die Operationen eines Prozesses in Abhängigkeit der Daten zu parallelisieren. Sondern der Scheduler betrachtet den Kontrollfluß und versucht, diesen effizient zu realisieren.

In der aktuellen VOTAN Implementation ist das Scheduling auf einzelne Prozesse beschränkt, in der alle Wait Statements vom gleichen Signal abhängen müssen. In diesem Prozeß sind vorher alle Prozeduren und Funktionen zu expandieren, d.h. der Code der Funktionen ist an Stelle des Aufrufes einzusetzen.

VOTAN bietet fünf verschiedene Möglichkeiten an, Prozesse während des Scheduling zu minimieren:

- **ASAP Scheduling**: Diese Funktion modifiziert die VHDL Beschreibung so, daß die Operationen zwischen Signalzugriffen so schnell wie möglich ausgeführt werden. Dies hat zur Folge, das „leere“ Kontroll Sequenzen generiert werden. Dies ist ein einfacher Übergang, der noch viel Spielraum für zukünftige Optimierungen läßt.
- **ALAP Scheduling**: Diese Funktion ist ähnlich wie die ASAP, nur das die Operationen so spät wie möglich ausgeführt werden.
- **Minimierung von Wait Statements**: Es wird versucht, so viele Wait Statements zu verschmelzen wie es nur geht, so das ihre Anzahl minimal ist.
- **Maximierung von Wait Statements**: Die Anzahl der Wait Statements wird maximiert, indem D-select Implementierungen von Verzweigungen zu C-select Implementierungen verändert werden.
- **Minimierung von Register**: VOTAN bietet dazu 3 Funktionen an. Die Erste versucht, die Anzahl der gesamten Register zu minimieren. Die Zweite minimiert die Anzahl der aktiven Werte über alle Clock Zyklen. Die dritte Mögliche ist eine vereinfachte Version der ersten, die sehr viel weniger Rechenzeit braucht. Ein für die erste Funktion equivalentes Mathematisches Problem ist das ILP-Problem, dessen Algorithmus exponentielle Laufzeit haben kann. Daher sollte diese Funktion lediglich für kleinere Beschreibungen verwendet werden. Für grössere Beschreibungen sollte die zweite Funktion benutzt werden.

Wenn es zu den Problemen eine Lösung gibt, wird sie von dem Scheduler gefunden. Die Komplexität des Problems wächst linear mit der Anzahl der Statements in einem VHDL Prozeß und nicht, wie in vielen anderen Programmen, mit der Anzahl der möglichen Pfade in dem Prozeß. Die Qualität des Ergebnisses ist allerdings abhängig von der Reihenfolge der Statements und der Position von Signalzugriffen.

VOTAN bietet einen Analyser an, der die Anzahl der benötigten Register für einen Prozeß nach der RT-Synthese abschätzt. Er gibt zwei Werte aus: Die maximale Anzahl über alle Clock Zyklen und die Anzahl der Register, welche ohne Register Sharing notwendig sind.

2.5.2 Expression Expansion

Um das Potential zur Optmimierung für den Scheduler zu erhöhen, müssen komplexe boolsche Ausdrücke vereinfacht werden. Dazu werden eindeutige, temporäre Variablen erzeugt. Die neuen Statements werden vor dem Original Ausdruck eingefügt. Dieses geht aber nicht immer. Wegen dem Aufbau von while-loops kann die Methode auf diese Art von Schleifen nicht direkt angewendet werden. Daher werden while-loops erst in equivalente exit-loops umgewandelt:

```
...                                 ...

WHILE (a AND b AND c) LOOP          LOOP
                                       EXIT WHEN NOT (a AND b AND c);
   ...                                 ...

END LOOP;                           END LOOP;

...                                 ...
```

Der Expression Expander braucht Informationen über die Bitbreite von Funktionen mit unconstrained Results, etwa bei Typen wie *SIGNED, UNSIGNED* oder *BIT_VECTOR*. Für die build-in Operatoren von VOTAN sowie für die IEEE- und Synopsys-Bibliotheken gibt es Konfigurationsfiles mit vordefinierten Regeln.

2.5.3 Prozedur- und Funktionsexpansion

Um die Komplexität von realistischen Designs einfacher zu meistern, sind hierarchische Beschreibungen wichtig. Dazu gehören auch Prozeduren und Funktionen (Subprogramme). Da so ein Prozeß mit Funktionen aber nicht mehr vernünftig optimiert werden kann, müssen sie an der Aufrufstelle eingefügt werden.

Normalerweise expandiert der VOTAN Subprogramm Expander alle Prozedur- bzw. Funktionsaufrufe direkt. Dabei geht er rekursiv vor, als erstes werden die Unterprogramme expandiert, dann der Prozeß selber. Durch Flags beim Expansionsaufruf können aber Subprogramme davon ausgenommen werden. Rekursive Subprogramme werden ebenfalls nicht expandiert.

Der Expander unterstützt sowohl Parameter als auch lokale Variablen. Die Parameter werden durch die aktuellen Werte ersetzt. Die lokalen Deklarationen werden mit einem eindeutigen Suffix versehen am Anfang des aktuellen Prozesses eingefügt. Der Suffix besteht normalerweise aus dem Namen des expandierten Unterprogramms.

Beim Expandieren werden Funktionen mit Multiple-Return's erst in Single-Return Beschreibungen umgeformt. Das erledigt VOTAN, indem er ein dummy LOOP-EXIT Konstrukt einfügt, bei dem das einzige Return-Statement dann an letzter Stelle steht. In der Mitte von Prozessen sind Return-Statements aber nicht erlaubt. Sie werden deswegen entfernt und der Funktionsaufruf wird direkt durch den Rückgabewert der Funktion ersetzt.

Die Ausführungsreihenfolge von komplexen Operationen wie *f1(x)+f2(x)-f3(x);* ist in VHDL nicht festgelegt. Sie kann nur mit Hilfe von Klammern festgelegt werden. Wenn nun z.B. *f1* und *f2* globale Variablen benutzen und verändern, kann es zu unangenehmen Seiteneffekten kommen. Um dies zu verhindern, überprüft VOTAN nach dem Expandieren den Code automatisch auf solche möglichen Fehler.

3 Synopsys Design Compiler

Der Synopsys Design Compiler ist ein Logik-Synthese-Tool, das Logik-Designs übersetzt und für Geschwindigkeit sowie Chipgröße optimiert. Logik-Synthese ist als die Integration von zwei Prozessen definiert: Übersetzen und Optimieren. Während des Übersetzungsprozesses werden Designs von verschiedenen Formaten, wie Hardware-Beschreibungs-Sprachen, Netlists und ähnlichem in Gate-Level-Circuit Netlists umgeformt. Beim Optimieren werden die Designs solange verfeinert, bis sie das vorgegebene Ziel bzgl. Größe und Geschwindigkeit erfüllen.

Die Fähigkeiten des Design Compilers umfaßt neben dem Einlesen von Designs verschiedenster Formate die Transformation in das Netlist-Format, die Optimierung und das Remappen der Netlist in die am meisten effiziente Gate Implementierung. Die Designs können hierarchisch, kombinatorisch, sequentiell oder unabhängig sein.

Die Aufteilung von Designs ist sinnvoll, um sie einfacher manipulieren zu können. Dazu werden große, komplexe Designs in einzelne, kleine aufgeteilt. Jede Komponente enthält eine Funktion des Designs und wird auch *Subdesign* genannt. Ein Design, das ein oder mehrere Subdesigns enthält, wird auch *Hierarchisches Design* genannt. Hierarchische Designs können mehrere Level haben, die wiederum mehrere Subdesigns enthalten können. Das Design, welches an der Spitze der Hierarchie steht, wird auch Top-Level-Design genannt.

Der Design Compiler optimiert hierarchische, kombinatorische oder sequentielle Designs und erzeugt daraus neue Libraries. Ein Design Compiler Synthese Prozeß ist als Flußdiagramm in Abb. 3 zu sehen. Am Anfang wird das Design eingelesen. Es kann aus mehreren verschiedenen Hardware Beschreibungs Sprachen bestehen. Anschliessend werden die Attribute und Constraints gesetzt. Beim Design Checken wird überprüft, ob die Vorgaben überhaupt machbar und ob sie vollständig sind. Danach wird dann Optimiert. Abhängig vom Ergebnis, können dann die Vorgaben verandert werden. Dies wird solange wiederholt, bis das Ergebnis zufriedenstellend ist.

3.1 Attribute

Ein Design ist ein Teilstück eines größeren Systems. Attribute sind Eigenschaften, die die interne Design Umgebung und ihr Verhalten zur Außenwelt beschreiben. Die Attribute geben dem Compiler zusätzliches Wissen über die Design Umgebung. Sie haben auch großen Einfluß auf die Kompenenten, die beim Optimieren aus einer Bibliothek benutzt werden. Daher ist es wichtig, die Informationen wie „drive strength“ der Ports, die Zeitverzögerung der Signale, die „load drive“ der Output Ports sowie die Leitungsart dem Compiler mitzuteilen. Die Leitungsart wirkt insbesondere sehr stark auf die Größe des Designs ein.

Da die Standardwerte unrealistisch sind, können die Ergebnisse dieser Circuits beim Optimieren nicht optimal werden. Z.B. haben Input Ports einen Standart „Drive“ Wert von Null oder eine unbegrenzte „drive strength“, bei den Output Ports ist es dasselbe.

Dies kann zu grossen Problemen führen. Wenn eine starke Auslastung der Verbindungsleitungen existiert und für die Input Ports keine „drive strength“ gesetzt sind, nimmt der Design Compiler eine Unbegrenzte an. Daher werden dann auch keine Puffer für den Support der Last hinzugefügt.

Auch für die Zeitoptimierung werden die Attribute benötigt. Mit ihnen wird die benötigte Zeit berechnet. Fehlen sie, werden die Ergebnisse zu optimistisch.

Neben den Attributen haben auch noch die eingesetzten Bibliotheken, die die Standartbausteine enthalten, sowie die Umgebung, in der die Schaltung arbeiten soll, auf die Optimierung Einfluß. Mit Umgebung sind dabei die Temparatur (z.B. 25 Grad), die Anzahl der Prozesse (z.B. 1) und die Stromstärke (z.B. 5V) gemeint. Das Ziel kann auch auf Best-, Nominal-, oder

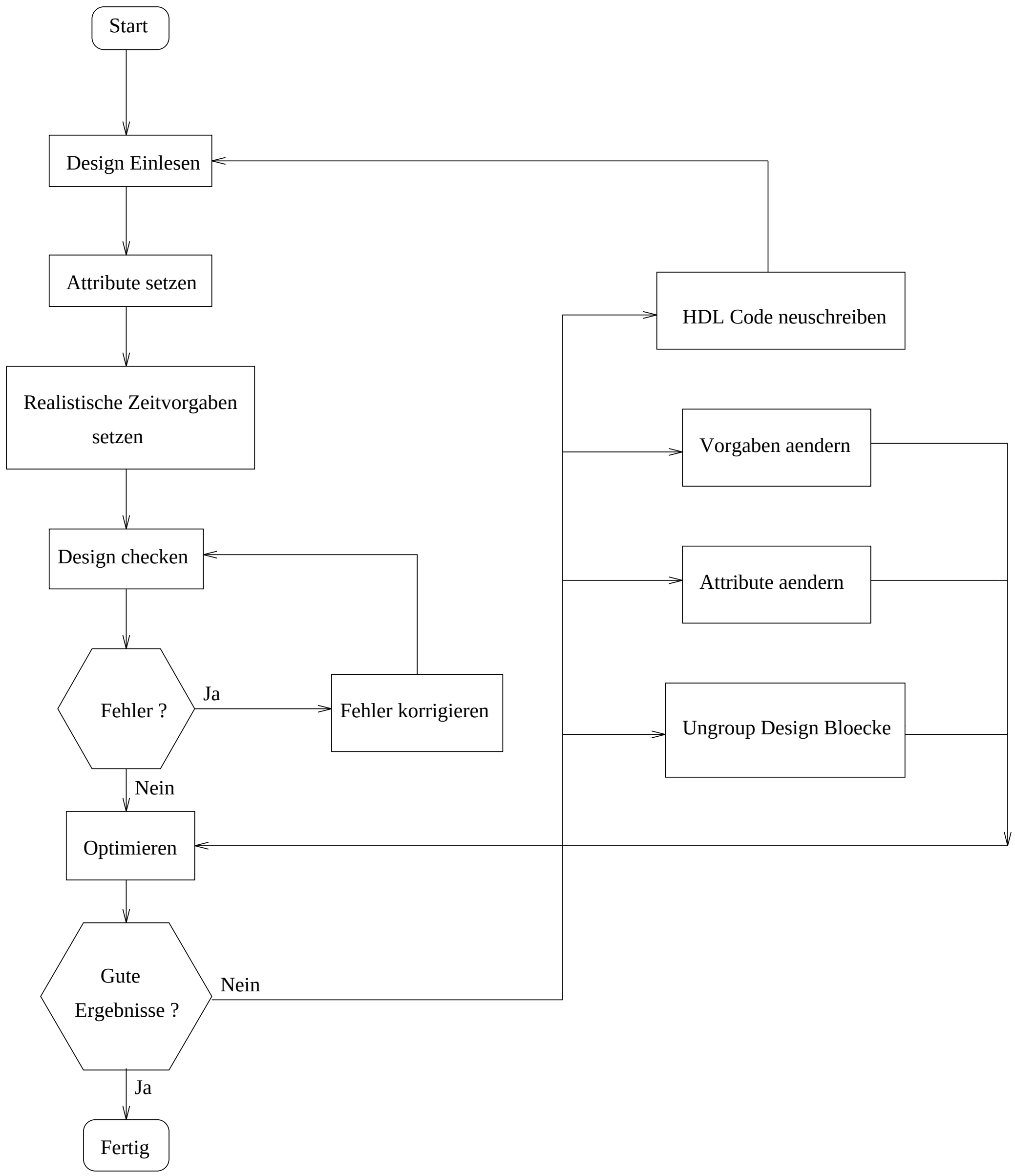


Abbildung 3: Synopsys Synthese Fluß

Worstcase Performance und Operationsbedingungen festgelegt werden.

3.2 Multiple Design Instanzen

Der Design Compiler erlaubt nur eine Instanz eines Subdesigns. Daher müssen solche Subdesigns, deren Instanzen mehr als einmal in einem Design benutzt werden, aufgelöst werden.

Es gibt drei Methoden, um Multiple Design Instanzen aufzulösen:

- Kopien des Subdesigns erzeugen (*uniquify*)
- Subdesign auflösen und verschmelzen (*ungroup*)
- Subdesign vom Compilier- und Optimiervorgang ausnehmen (*dont_touch*)

Beim Kopieren eines Subdesigns wird für jede Referenz im Design eine Kopie erzeugt. Dieser wird ein eindeutiger Name zugeordnet. Die Referenzen zu dem Subdesign werden so geändert, das sie auf den neuen Namen zeigen, unabhängig von der Position in der Hierarchie. Während der Optimierung wird jede Kopie einzeln im Kontext ihres Environments optimiert.

Beim Auflösen der Subdesigns wird der entsprechende Level der Hierarchie aufgelöst und die Bausteine mit der umgebenden Logik verschmolzen. Dies kann auch benutzt werden, um „combinational blocks“ zu vereinigen, damit redundante Logik minimiert wird.

Das Herausnehmen aus dem Compilier- und Optimiervorgang bewirkt, das es während der Optimierung in keiner Weise verändert wird. Dazu wird ein *dont_ touch* Attribut bei allen Instanzen gesetzt. Dies ist bei bereits Optimierten oder nur sehr wenig optimierbaren Subdesigns sinnvoll, da es nur einmal in die Bibliothek aufgenommen wird. Ansonsten würde es mehrmals kopiert und es ständen viele identische Subdesigns in der Bibliothek.

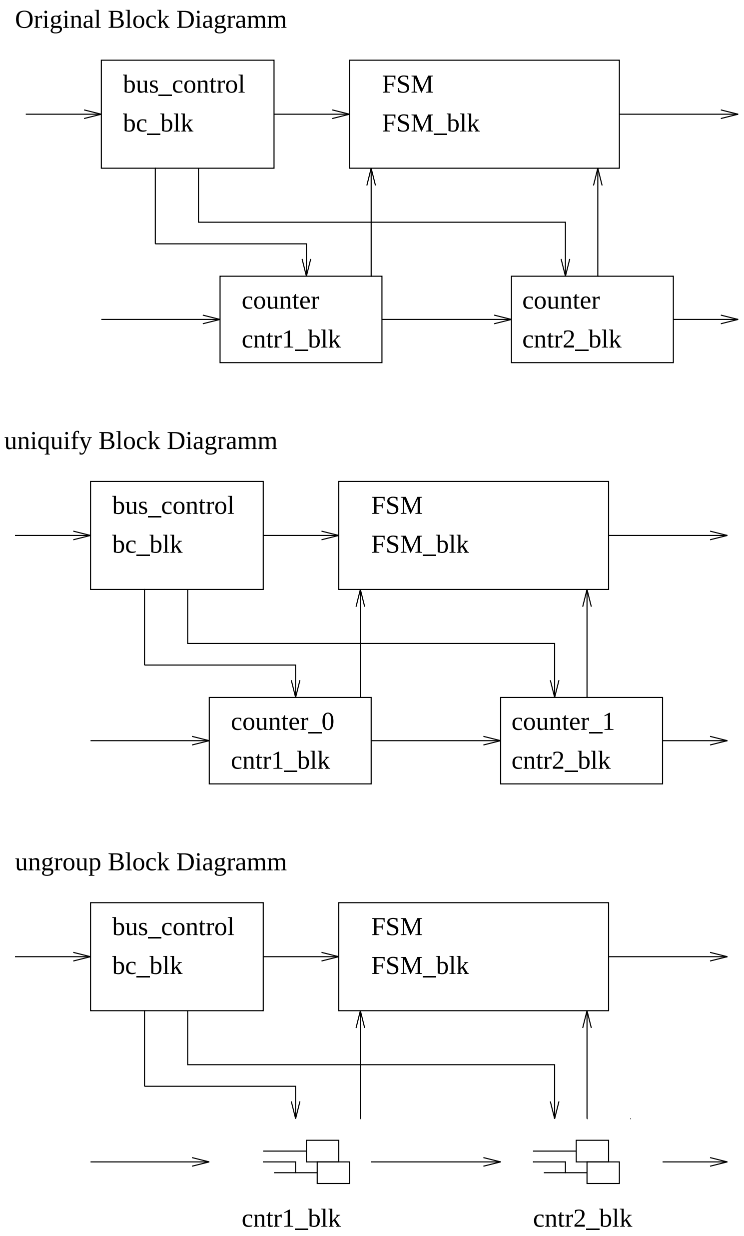


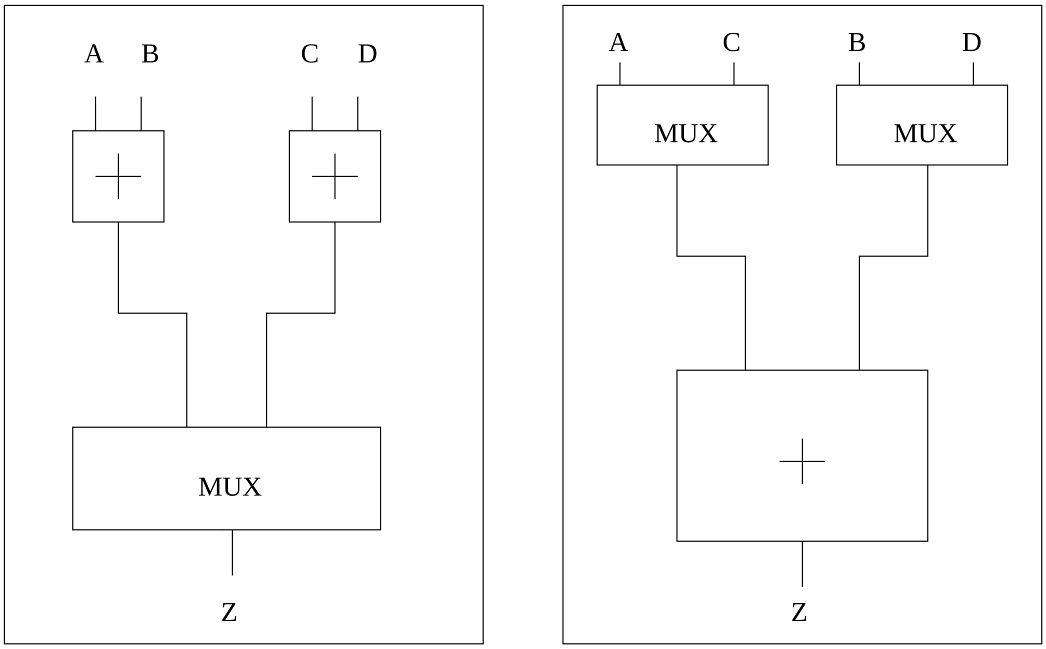
Abbildung 4: Vor und nach der Optimierung mit uniquify bzw. ungroup

3.3 Resource Sharing

Resource Sharing wird eingesetzt, um die Design Größe zu verkleinern. Dabei wird eine Hardware Komponente für mehrere Operationen benutzt.

```
if(cond) then
Z<=A+B
else
Z<=C+D;
end if;
```

VHDL Code



Circuit

Optimized Circuit

Abbildung 5: Optimierung mit Resource Sharing

Für jede Addition mit variablen Operatoren wird normalerweise ein neuer Addierer erzeugt. Diese Wiederholung von Komponenten führt zu einem Anwachsen der Design Größe. Mit Resource Sharing werden die Additionen so implementiert, das immer derselbe Addierer benutzt wird. Da damit weniger Hardware erzeugt wird, ist die benötigte Größe des Designs natürlich auch kleiner (Beispiel siehe Abbildung 5).

Jedes Mal, wenn in dem Design ein komplexer Operator benutzt wird, wird dafür genau eine Unit erzeugt. Komplexe Operatoren sind z.B.:

- Addition (+)
- Subtraktion (−)

- Größer als (>)
- Größer oder gleich wie (>=)
- Kleiner als (<)
- Kleiner oder gleich wie (<=)
- die meisten Benutzer definierten Funktionen

Einige der in der Synopsys Library vorhandenen komplexen Operatoren sind:

- add (Addierer)
- sub (Subrahierer)
- addsub (Addierer/Subrahierer)
- inc (Incrementor)
- dec (Decrementor)
- incdec (Incrementor/Decrementor)
- cmp (Comparator)

Das Resource Sharing kann auf folgendes Eingeschränkt werden:

- Kein Resource Sharing
- Nur Größe, ignoriert Zeitvorgaben
- Zeitvorgabe auf der kleinsten möglichen Fläche. Dies wird defaultmässig benutzt.

3.4 Optimierung

Der Design Compiler compiliert Hierarchien automatisch, ohne sie durcheinander zu bringen. Dabei wird jedes Subdesign einzeln compiliert, die nötigen Informationen werden aus dem Environment geholt. Nachdem jedes Modul in dem Design abgearbeitet ist, wird die Optimierung solange wiederholt, bis die vorgegebenen Werte erreicht sind. Dieses führt manchmal zu re-compilieren von Subdesigns auf kritischen Pfaden. Der Prozeß stopt, wenn die vorgegebene Performanz erreicht wird oder nicht erreicht werden kann.

Abbildung 6 zeigt die Abhängigkeit von Größe und Verzögerung eines Designs. Ein schnelles Design wird größer als ein langsameres. Mit Hilfe der Attribute und Art der Optimierung kann auf diese Komponenten beim Optimieren Einfluß genommen werden.

Die Optimierung kann grob in drei Schritte eingeteilt werden:

- Flattening
- Structuring
- Mapping

Flattening und Structuring operieren auf dem Logik-Level, Mapping auf dem Gate-Level. Beide Stukturen sind wichtig für die Qualität des Designs.

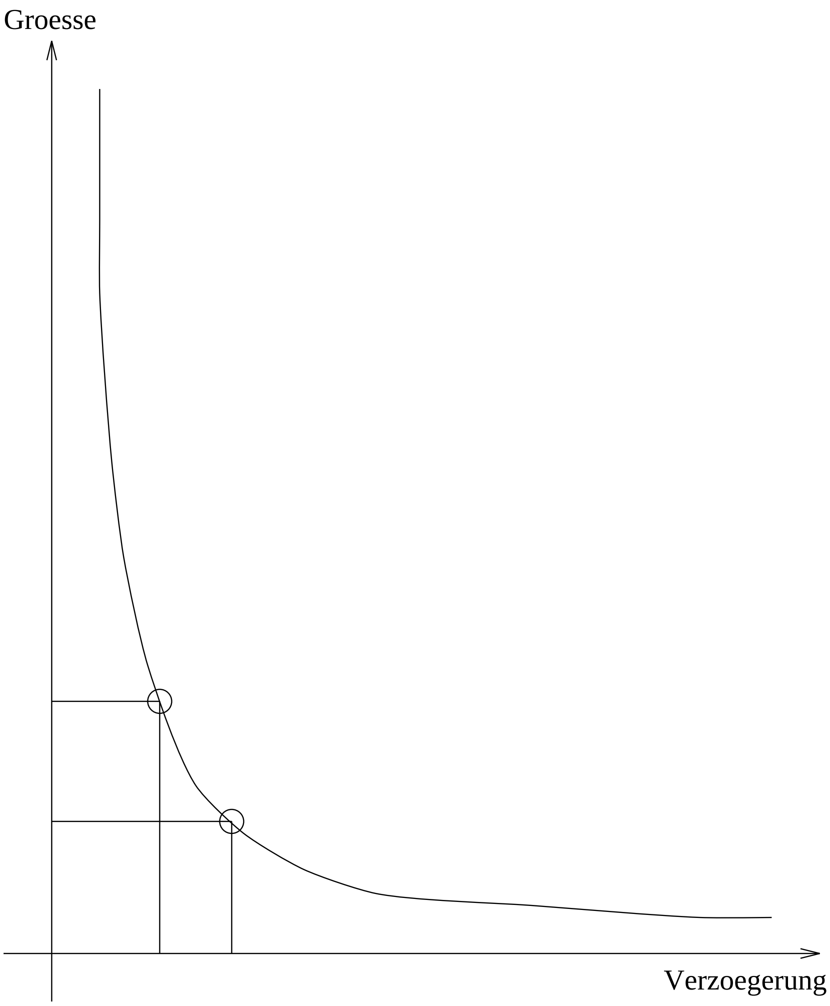


Abbildung 6: Abhängigkeit von Größe und Verzögerung eines Designs

3.4.1 Flattening

Beim Flattening werden alle logischen Strukturen des Designs entfernt, danach alle Zwischenvariblen. Dies erlaubt dem Compiler effektivere Subfunktionen auszuwählen.

Flattening ist ein Weg, ineffiziente logische Strukturen zu entfernen. Es sollte trotzdem nur äußerst vorsichtig angewendet werden, da auch gute Strukturen entfernt werden können. Wenn es z.B. auf einem carry-look-ahead adder angewendet wird, ist der Design Compiler nicht in der Lage, diese Struktur zu restaurieren.

Das Ergebnis von Flattening ist eine „two-level sum-of-products“ Form, ähnlich der bei PLA. Die Geschwindigkeit der Gate-Level Darstellung von dem „flatten“ Design kann sehr schnell sein. Zwischen Input und Oput sind meistens nur sehr wenige Bauteile, im Extremfall nur eine AND- und eine OR-Plane. Flattening greift nicht in Delay- und Größenvorgaben ein, trotzdem kann die Größe extrem Anwachsen und die Verzögerung sinken.

Defaultmäßig wird kein Flattening angwendet.

3.4.2 Structuring

Structuring fügt einem Design eine Struktur hinzu, indem gemeinsame Sub-Expressions in „intermediate“ Variablen umgewandelt werden.

Während der Strukturierung versucht der Design Compiler Unterfunktionen zu finden, die Logikgleichungen reduzieren. Diese Unterfunktionen werden aus dem Design entfernt und in intermediate Variablen umgeformt. Structuring veringert die Größe eines Designs.

Synopsys kennt zwei Stucturing Optionen:

- Boolean
- Zeitverhalten

Boolean Strukturierung ist eine Optimierungs Methode, deren Algorithmen zur Reduzierung des Designs auf den grundlegenden Regeln der Boolschen Algebra beruhen.

Boolean Optimierung ist sinnvoll für „Random Control Logic Circuits“ oder PLA's. Diese Methode sollte nur bei Designs verwendet werden, deren Zeitvorgaben einfach zu erreichen sind, aber sehr klein werden sollen. Für reguläre Strukturen wie Addierer, ALU's und Multiplizierer sollte es nicht benutzt werden.

Diese Technik benutzt sehr effektiv *dont_care* Attribute. Diese Attribute können in HDL, FSM und PLA spezifiziert werden.

Beim Zeitverhalten werden bei der Strukurierung die Geschwindigkeitsvorgaben sehr stark beachtet. Das Ergebnis ist ein kleineres Design, bei dem auf den zeitkritischen Pfaden die Logik minimiert wird.

Defaultmässig benutzt Synopsys Structuring mit Zeitverhalten.

3.4.3 Mapping

Während der Mapping Phase versucht der Compiler mit Komponenten aus den Technologie Bibliotheken die Logische Struktur zu implementieren. Wenn z.B. ein 4-fach Oder benutzt wird, in der Bibliothek aber nur 2-fach Oder enthalten sind, muß Synopsys aus den 2-fach Odern ein 4-fach Oder bauen. Das Ziel des Mapping Prozesses ist es, eine Gate-Level Implementierung eines Designs zu synthesieren, welches die Zeit- und Grössenvorgaben einhält. Dabei ist das Ergebnis nicht eindeutig, es kann mit verschiedenen Bausteinen das gleiche Schaltungsverhalten erreicht werden.

4 Literaturverzeichnis

Synopsys Design Compiler Tutorial, Version 3.0 Preliminary Draft
High-Level VHDL Optimization, Transformation, and Analysis, 1993
The VHDL Cookbook, Peter J. Ahsenden, 1990

5 Anhang

5.1 Lizenzserver

Synopsys läuft auf allen Rechnern mit SunOS 4.1.x.

Lizenzen für VOTAN/VOX sind auf folgenden Rechnern eingetragen: Miro, Rubens, Pollux, Themis und Bob.

5.2 .synopsys_dc.setup

Folgende Variablen sollten inder .synopsys_dc.setup Datei im Home-Verzeichnis gesetzt werden:

```
designer = "User Name";
company = "UGH-Paderborn";
view_background = "black";
search_path = {. /homes2/zeus/synopsys/libraries/syn}
link_library = lsi_10k.db
target_library = lsi_10k.db
symbol_library = lsi_10k.sdb
synthetic_library = standard.sldb
edifout_no_array = true
edifout_netlist_only = true
```

5.3 Installation

Dieses Script sollte von einer Shell mit *source* eingelesen werden. Es sollte aber nicht in die *.cshrc* geschrieben werden, da sonst bei jedem tcsh Aufruf die Datei unnötiger Weise gesourcet wird. Da dieses mehrere Minuten dauern kann, geht einmal unnötige Zeit verloren, wenn z.B. ein Shellscript gestartet wird. Wenn eine Datei mal nicht verfügbar ist, wird man unter Umständen nach dem einloggen sofort wieder aus dem System geworfen.

```
if( `uname -r | cut -c1` == 4 ) then

### Das ist fuer SYNOPSYS

   setenv SYNOPSYS /homes2/zeus/synopsys
   set path=($path $SYNOPSYS/sparc/syn/bin )
   set path=($path $SYNOPSYS/iview/bin )
   source $SYNOPSYS/admin/install/sim/environ.csh


### Das ist fuer VOTAN
```

```
   (cd ~hardt)
   setenv VOTAN /homes/mallorca/hotte/Votan
   setenv dls_callas $VOTAN/lib/callas
   setenv CLSI /tmp_mnt/homes/kairo/cadtools/clsi
   setenv VTIP $CLSI/vtip
   setenv LM_LICENSE_FILE $VTIP/license/license.dat
   source $VTIP/com/setupvtip
   set path= ( $path $VOTAN/bin )
   setenv MANPATH "$VOTAN/man:${MANPATH}"
   setenv dls_ieee /homes1/europa/baruch/synthese/Libraries/IEEE
   setenv dls_callas $VOTAN/lib/callas
   setenv dls_synopsys /homes1/europa/baruch/synthese/Libraries/synopsys
   setenv dls_work "./Work"

else

   echo " "
   echo "Synopsys und Votan laufen nur unter SunOS 4.1.3 !"
   echo -n "Dies ist "
   uname -s -r
   echo " "

endif
```