VHDL

Ilja Kipermann

Seminar: Formal Methods For Fun and Profit – Universität Koblenz-Landau –
Sommersemester 2005 – Seminarleiter: Jun.-Prof. Beckert

1 Einleitung

Wenn es darum geht, einen elektronischen Baustein wie einen Prozessor zu entwickeln, hat man einen langen Entwicklungsweg vor sich, der aus mehreren komplexen Schritten besteht. Im Folgenden werde ich skizzieren, wie ein solcher Entwicklungsprozess abläuft und wie man die Hardwarebeschreibungssprache VHDL innerhalb dieses Entwicklungsprozesses nutzen kann.

Bei der Entwicklung digitale Systeme sind zwei Herstellungsprozesse relevant. Zum einen die Programmierung von FPGAs (Field programmable gate arrays), sowie die Herstellung spezifischer Schaltkreise, die ASICs (Application Specific Integrated Circuits) genannt werden.

Ein FPGA ist ein Logikschaltkreis, welcher noch nach seiner Fertigung neu programmiert werden kann. Unter ASICs versteht man eine integrierte Schaltung, die für einen speziellen Anwendungsfall, exklusiv für einen bestimmten Kunden gefertigt wird. Dabei hat sich die elektronische Unterstützung für den kompletten Designprozess dieser Bausteine in den letzten Jahrzehnten von spezialisierten Programmen zum manuellen Zeichnen der Belichtungsmasken, hin zu Compilern, die diese Masken aus textuellen Beschreibungen heraus erzeugen, entwickelt. Heutzutage beschreibt der Hardwareentwickler, mit Hilfe von Hardwarebeschreibungssprachen immer mehr, *was* ein Baustein tut und nicht *wie* er es tut.

2 Geschichte von VHDL

VHDL wurde Usprünglich vom amerikanischen Verteidigungsministerium (Department of Defense) entwickelt. Man hat eine Sprache

zur Hardwareentwicklung gebraucht, die maschinen- und menschenlesbar ist und den Entwickler dazu zwingt strukturiert und „leserlich“ zu schreiben. Anfang der 70er Jahre wurde zum ersten mal über eine solche Hardwarebeschreibungssprache diskutiert. Zum ersten mal wurde VHDL dann 1987 standardisiert und 1993 aktualisiert. 1999 entstand VHDL-AMS, was eine Erweiterung von VHDL um analoge und mixed-signal Sprachelemente ist. VHDL-AMS ist damit eine Obermenge zu VHDL. Allerdings ist die Analogsynthese ein zu komplexes Feld und wird von VHDL-AMS ausgespart.[1]

3 Was ist VHDL?

VHDL ist eine Hardwarebeschreibungssprache. Sie beschreibt das Verhalten und die Struktur elektronischer Bausteine. VHDL ist eine Notation, und ist präzise und komplett definiert im dazugehörigen Language Reference Manual. VHDL ist ein internationaler IEEE Standard und es ist nicht propietär. Es ist kein Informationsmodell, kein Simulator und auch keine Methodik, jedoch ist eine Methodik und ein Satz von Software-Werkzeugen notwendig um VHDL effektiv nutzen zu können. Denn wenn man erfolgreich einen Hardwarebaustein entwickeln möchte benötigt man eine Sprache, einen Satz an Werkzeugen und eine geeignete Methodik. Simulation und Synthese (siehe später) sind die zwei wichtigsten Arten von Werkzeugen, die VHDL verarbeiten. Es ist ein wichtiger Vorteil von VHDL, dass man VHDL-Beschreibungen mit Hilfe von Werkzeugen komplett simulieren kann. Das Language Reference Manual definiert zwar keinen Simulator, es definiert aber eindeutig, wie ein Simulator jeden Teil der Sprache zu verarbeiten hat. Obwohl VHDL Ursprünglich nicht dafür konzipiert wurde, werden auch formale Methoden zur Verifikation eines VHDL Designs erforscht um die Richtigkeit des Designs sicher zu stellen.

4 Konzepte der VHDL Programmierung

Da die Parallelität eine wichtige Eigenschaft von Hardwareeinheiten ist, gibt es zweierlei Arten von Anweisungen bei VHDL. Sequenzielle Anweisungen und Nebenläufige Anweisungen. Sequenzielle Anweisungen werden, ähnlich wie bei Softwareprogrammierung, strikt

hintereinander abgearbeitet. Dabei können nachfolgende Anweiseungen vorhergehende überschreiben. Deswegen ist es wichtig in welcher Reiehnfolge man die Anweisungen angibt. Nebenläufige Anweisungen haben die Eigenschaft, dass sie gleichzeitig wirksam sind. Das bedeutet, dass es unwichtig ist welche Reihenfolge sie haben. Damit ist es möglich die Parallelität echter Hardware nachzubilden.

Es wurden bei VHDL weiterhin drei wichtige Modellierungsmethodiken berücksichtigt. Die Abstraktion, Die Modularität, Die Hierarchie [3].

Die **Abstraktion** erlaubt es dem Programmierer verschiedene Detailstufen für verschiedene Teile eines Models zu verwenden. So werden zum Beispiel Module, die man nur zur Simulation braucht, nicht so detailliert beschrieben wie Module, die zur Synthese gebraucht werden. Es wird also unterschieden zwischen wichtigen und (aktuell)unwichtigen Informationen. Soll ein Modell einen bestimmten Abstraktionsgrad besitzen, so muß jedes Modul des Modells den gleichen Abstraktionsgrad besitzen. Dieser Abstraktionsgrad kann dann im Verlaufe des Entwicklungsprozesses immer weiter verfeinert werden.

Die **Modularität** erlaubt es, große Funktionsblöcke in kleinere abgeschlossene Blöcke zu unterteilen, sprich einzelne Module zu bilden.

Die **Hierarchie** erlaubt, dass aus diesen Modulen, die Untermodule besitzen, die selbst auch wieder Untermodule besitzen können, ein System aufzubauen. Man verwendet hierzu verschiedene Hierarchieebenen. So bilden ein oder mehrere Module mit unterschiedlichem Abstraktionsgrad eine Hierarchiebene und die Untermodule, die in diesen enthalten sind, bilden dann die Hierarchieebene darunter. Die Teilbeschreibungen aus unteren Hierarchieebenen werden als Instanzen an die übergeordneten Module weitergegeben.

4.1 Abstraktionsebenen

VHDL legt keinen Beschreibungsstil fest den man zu verwenden hat. Es erlaubt jedwede Methodik die der Entwickler bevorzugt: top down, bottom up oder middle out, wie der Entwickler es möchte. VHDL kann genutzt werden, um auf drei verschiedenen Abstraktionsebenen Hardware zu beschreiben. Der Hauptunterschied zwischen

diesen Ebenen ist dabei die Berücksichtigung des Timings. Die drei Ebenen sind:

- Verhaltensmodellierung
- Register-Transfer-Ebene (RTL)
- Gatterebene

Wenn man eine informelle Spezifikation vorliegen hat muss diese analysiert und zunächst zu einer Beschreibung verfeinert werden die simuliert werden kann. Dies geschieht auf der Ebene der Verhaltensmodellierung. Nach der ersten Simulation muss das Design dann weiter verfeinert werden, so dass ein Synthese-Werkzeug(siehe Maschinelle Synthese) es verarbeiten kann. Dabei ist die höchste Abstraktionsebene die ein solches Werkzeug akzeptiert die Register-Transfer-Ebene. Wenn es dann darum geht die einzelnen Bausteine zu verbinden und Gesamtschaltungen zu schaffen bewegt man sich dann auf der Gatterebene. Die tatsächlichen physikalischen Begebenheiten, wo die tatsächlichen Verbindungen und Transistoren auf einem Chip spezifiziert werden und somit die tiefste Abstraktionsebene beim Hardwaredesign liegt aber schon ausserhalb des VHDL-Blickwinkels.[5] [6]

Verhaltensmodellierung Basierend auf einer informellen Spezifikation wird eine Verhaltensmodellierung in VHDL erstellt. Eine reine Verhaltensmodellierung besteht aus einem Satz von Anweisungen die ausgeführt werden um zu einem bestimmten Ergebnis zu kommen. Sie berücksichtigt keinen Takt. Diese Verhaltensmodellierung ist bereits simulierbar und der Entwickler kann mit geeigneten Simulationswerkzeugen prüfen ob seine Modellierung performant arbeitet und so funktioniert wie sie soll. Dabei ist wichtig zu berücksichtigen, dass es nicht möglich ist eine Verhaltensmodellierung zu synthetisieren. Die Register-Transfer-Ebene wird als Eingabeformat von Synthese-Werkzeugen genutzt.

Register-Transfer-Ebene Die Register-Transfer-Ebene-Beschreibung ist eine Beschreibung des Datenflusses zwischen Registern, in welchen die Informationen zwischen einzelnen Zeitschritten gespeichert

werden. Eine RTL-Beschreibung spezifiziert wie und wo die Information gespeichert wird und wie sie während einer Operation weitergeleitet wird. Diese Ebene der Hardwarebeschreibung berücksichtigt bereits explizit den Takt. Alle Operationen sind getimet, um zu bestimmten Taktzyklen ausgeführt zu werden, jedoch werden noch keine Verzögerungen die durch die einzelnen Hardwareelemente des Bausteins entstehen, berücksichtigt. Da dieser Schritt ebenfalls simulierbar ist, werden Verhaltes- wie RTL-Beschreibungen simuliert und gegengeprüft und mit Hilfe eines Synthesewerkzeugs weiterverarbeitet. Die Gatterebene ist dabei das Ausgabeformat.

Maschinelle Synthese Nachdem die formale Spezifikation ausreichend konkretisiert wurde haben wir eine Modellbeschreibung die mit Hilfe automatisierter Werkzeuge aus dem RTL-Modell eine Schaltungsdarstellung auf der Gatterebene erzeugen. Der VHDL-Code der mit dem Synthese-Werkzeug verarbeitet wird muss speziell für dieses Werkzeug entwickelt worden sein, da verschiedene Werkzeuge unterschiedliche ansprüche an den Eingabecode stellen. Manchmal wird die Synthese auch für maschinell synthesefähige Beschreibungen manuell durchgeführt. Diese Handoptimierung wird angewandt um zum Beispiel den Flächenbedarf und/oder den Energiebedarf, sowie die Maximierung der erzielbaren Taktrate zu verbessern. Man muss hierbei jedoch den Zeitbedarf für das manuelle Design und die Gewinne durch einen solchen Vorgang mit einander in Verhältnis setzen, um festzustellen, ob sich dieses Vorgehen auch lohnt.Nach der maschinellen Synthese haben wir nun eine Beschreibung der Schaltung bezüglich ihrer tatsächlicher Komponenten und die logische Struktur zwischen ihnen.

Gatterebene Eine Gatterebenen-Beschreibung besteht aus einem Netzwerk von Gattern und Registern die über eine Technologiebibliothek instanziiert werden, in der technologiespezifische Verzögerungen für jedes Gatter enthalten sind. Damit haben wie eine Beschreibung des Hardwarebausteins in seiner Funktionsweise mit einem expliziten Takt und Verzögerungsinformationen.

5 Aufbau von VHDL

An dieser stelle ein ganz einfaches Beispiel für ein VHDL Programm: ein 8-bit Komparator. Dieser akzeptiert zwei 8-bit Inputs, vergleicht sie, und gibt ein 1-Bit Ergebniss aus. Eine 1 bei übereinstimmung und eine 0 bei einer Differenz.

Eine VHDL Verhaltensmodellierung dazu würde so aussehen:

```
ENTITY compare IS
    PORT( A, B: IN bit_vector(0 TO 7);
            EQ: OUT bit);
END compare;

ARCHITECTURE compare1 OF compare IS
BEGIN

    EQ <= 1 WHEN (A = B) ELSE 0;

END compare1;
```

Jede VHDL Beschreibung besitzt zumindest eine ENTITY und eine ARCHITECTURE. In größeren Designs hat man meistens mehrere Entity/Achitecture Paare die eine Gesamtschaltung darstellen.

Die ENTITY beschreibt die Schaltung wie man sie von „Ausserhalb“ sieht. Aus der Perspektive der Eingabe-/Ausgeb Schnittstellen. Zu jeder Entity muss es eine ARCHITECTURE geben bevor man eine Simulation oder gar Synthese starten will, denn in der Architecture steht die tatsächliche Verhaltensbeschreibung drin.

5.1 ENTITY

Die Entity bietet das komplette Interface für die Schaltung. Mit den Informationen, die man aus der Entity hat (Namen, Datentypen und Ports) hat man alle Information, die man braucht um diesen Teil der Schaltung mit anderen zu verbinden oder um Input Stimuli zu entwickeln für eine Simulation. Die tatsächliche Funktionsweise ist jedoch nicht enthalten.

```
ENTITY compare IS

    PORT( A, B: IN bit_vector(0 TO 7);

              EQ: OUT bit);

END compare;
```

Diese Entity deklaration besitzt einen Namen, *compare*, und eine PORT-Anweisung, in der alle Inputs und Outputs der Entity definiert werden. Es gibt drei Ports: A, B und EQ. Jedes der Ports hat eine Richtung (in oder out) und einen Typ (hier bit vector und bit). Die fundamentalen Datentypen bei VHDL sind Bit, Bit vector, Boolean, Integer, Real, Time, Character und String.

5.2 ARCHITECTURE

Zu jeder Entity gehört mindestens eine Architecture.

```
ARCHITECTURE compare1 OF compare IS

begin

    EQ <= 1 WHEN (A = B) ELSE 0;

END compare1;
```

Die Deklaration beginnt mit dem Namen, der notwendig ist, da es mehrere Architectures zu einer Entity geben kann. Und es folgt die Funktionsbeschreibung des Komparators. Dieser gibt eine 1 aus wenn die Eingangsvariablen gleich sind bzw. eine 0, wenn nicht.

Durch Hierarchien und Unterprogramm-Features von VHDL ist es möglich in der Architecture Komponenten von tieferen Abstraktionsebenen, subroutinen und Funktionen einzufügen.

Wenn es dann daran geht, eine solche Verhaltensmodellierung zu der nächsten Abstraktionsebene zu verfeinern, wird der bestehende Code entsprechend den Anforderungen des Synthesewerkzeugs zum RTL-Level umgeschrieben und erweitert, dabei werden zusätzliche Konstrukte mit neuen Informationen hinzugefügt.

6 Formale Methoden und VHDL

Die gängige Praxis ein VHDL-Modell auf seine Korrektheit zu überprüfen, ist zur Zeit die Simulation. Bugs in einem Hardwaredesign können extrem teuer werden, falls sie nicht rechtzeitig oder gar gar nicht erkannt werden. Bei sicherheitskritischen Hardwarebausteinen, die zum Beispiel in Atomkraftwerken oder in der Luftfahrt eingesetzt werden, reicht es nicht aus diese einfach durch Simulationen auf Korrektheit zu überprüfen. In diesen Fällen ist es enorm wichtig sicher zu stellen, dass der fertige Baustein seiner Spezifikation entspricht. In den letzten Jahren haben sich formale Methoden zu einem alternativen Ansatz bei der Korrektheits- und Qualitätssicherung von Hardwaredesigns entwickelt. Auf Grund der fehlenden Semantik ist eine Verifikation von VHDL der Verhaltens-Ebene jedoch kaum durchführbar. Formale Verifikation benutzt im Gegensatz zum einfachen testen, strenge mathematische Argumente um zu zeigen, dass ein Hardwaredesign seiner Spezifikation entspricht. Derzeit schreiben Entwickler im Normallfall *Testbenches*, die mit verschiedenen zufallsgenerierten oder manuellen Inputs das Hardwaredesign testen. Das reicht jedoch nicht aus, um Fehler komplett ausschließen zu können. Es gibt einige Ansätze um formale Methoden auf VHDL-Beschreibungen anzuwenden, Allerdings sind diese momentan noch eher ein Forschungsgebiet und man ist noch entfernt von einer optimalen Lösung.[7] [8]

Formal equivalence checking Bei dieser Methode wird die RTL-Ebene des Designs als Referenz genommen. Die RTL-Beschreibung wird danach mit einem Synthesewerkzeug zu einer Gatter-Netlist weiterverarbeitet. In der Theorie muss ein Synthesewerkzeug garantieren dass die von ihm erzeugte Netlist logisch äquivalent ist zu dem RTL-source-code. In der Praxis haben Programme jedoch Bugs und es ist auch nicht unüblich, dass der Entwickler manuell optimierungen vornimmt, wodurch Fehlerquellen entstehen. Die Idee hinter formal equivalence checking ist, dass man anstatt zu beweisen, dass zwei einzelne Ausdrücke äquivalent sind, was ein hoffnunfsloses Unterfangen bei größeren Bausteinen wäre, sich der symbolischen Logik [12] bedient, bei der einzelne Ausdrücke die ganze Bandbreite an Inputs representieren. Wenn man dann die Äquivalenz nachgewiesen

hat, hat man es für alle In- und Outputs getan. Die Algorithmen dieses Verfahrens sind jedoch sehr komplex und funktionieren deswegen nicht so gut bei größeren Designs, so dass es nötig ist das Design zu segmentieren, was problematisch sein kann. Ein weiteres großes Problem des Verfahrens ist auch dass sich auf der RTL-Ebene eventuell bereits schwerwiegende Bugs im Design befinden die dann unentdeckt bleiben.

Model Checking Das Model Checking läuft im Prinzip wie folgt ab. Zunächst wird aus einem System durch Abstraktion ein Modell und durch Formalisierung aus einer Spezifikation eine logische Formel gewonnen. Modell und Formel werden dann dem eigentlichen Model Checker übergeben der überprüft, ob das Modell der Spezifikation genügt. Im Idealfall wird dieser ein positives Ergebnis zurückliefern. Typischerweise liefert der Modelchecker aber entweder einen Speicherüberlauf, da das System zu komplex ist oder ein Gegenbeispiel. Im zweiten Fall liegt entweder ein Fehler im System, Modell oder der Formel vor, der korrigiert werden muss bevor Modell und Formel erneut dem Model Checker zur Verifiakton übergeben werden. Die Rückgabe eines explizten Gegenbeispiels vereinfacht die Fehlerkorrektur.[13]

Leider ist es nicht möglich Model Checking mit einem VHDL Code zu nutzen, denn das Design muss in der Form eines endlichen Automats vorliegen. Der gängigste Ansatz ist VHDL in Binary Decision Diagrams [14] zu übersetzen und darauf basierend das Model Checking Verfahren anzuwenden.[11] Dieser Vorgang ist jedoch sehr komplex und daher kosten- und zeitaufwendig. Ausserdem besteht die Gefahr, durch die Fehler bei der Übersetzung, falsche Fehler im Design zu finden. [9]

Theorem proving Unter Theorem Proving versteht man ganz allgemein die Vorgehensweise, Behauptungen auf der Grundlage von Beweissystemen und mit Hilfe von Programmen zu verifizieren. Die Sprache, in der Vermutungen, Hypothesen und Axiome verfasst werden, ist eine logische Sprache, oft basierend auf der klassischen Logik.[10] Hier steht man wieder vor dem Problem dass man mit dem VHDL-Code an sich nicht arbeiten kann. Dieser muss erst für das

Verfahren umgeschrieben werden. Es muss bei diesem Verfahren sehr vieles manuell gemacht werden. Das Verfahren kann genutzt werden um bestimmte verwendete Algorithmen zu verifizieren, erweist sich jedoch bislang als zu hart und kompliziert für die komplette Verifikation.[7]

7 Vorteile von VHDL

7.1 Vielseitigkeit

Ein großer Vorteil von VHDL ist dass es für viele Zwecke einsetzbar ist. Man kann es für die Spezifikation und Simulation einsetzen, aber auch als Ein- und Ausgabesprache für die Synthese. Des weiteren ist die sehr gut lesbare Form von VHDL sehr gut geeignet um eine Dokumentation damit zu erstellen. Schließlich ist durch die firmenunabhängige Normierung der Sprache ein Datenaustausch zwischen verschiedenen Programmen, zwischen verschiedenen Entwurfsebenen, zwischen verschiedenen Projektteams und zwischen Entwickler und Hersteller möglich.[4]

7.2 Programmunabhängigkeit

VHDL ist eine standardisierte Sprache, die an kein Programm eines bestimmten Softwareherstellers gebunden ist und wird von sehr vielen Herstellern unterstützt. Es ist komplett programmunabhängig und es existieren diverse Software-Lösungen für die vielen Entwurfsschritte die beim Einsatz von VHDL möglich sind. Ausserdem wurde bei der Entwicklung von VHDL stark auf die Unabhängigkeit von einem bestimmten Rechnersystem geachtet. Die Systemabhängigen teile sind in Packages gekapselt, das VHDL-Modell an sich ist unabhängig von der eingesetzten Zielarchitektur und damit fast immer portierbar.[4]

7.3 Technologieunabhängigkeit

VHDL ist technologieunabhängig. Erst zu einem relativ späten Zeitpunkt des Entwurfs muss man sich für eine Technologie entscheiden. Wenn man später auf eine andere Technologie wechseln will wird kein komplettes Redesign benötigt.[4]

7.4 Modellierungsmöglichkeiten

´ Beschreibungen auf abstraktem Niveau wie bei VHDL haben verschiedene Vorteile. Sie sind kompakt und überschaubar und man hat den Überblick über den gesamten Entwurf, die Entwicklungszeiten sind kürzer, es wird weniger Zeit bei der Simulation benötigt und es ist eine frühere Verifikation möglich. In Kombination mit detaillierteren Beschreibungen ist eine ebenenübergreifende Simulation möglich, dadurch können einzelne Projektteams unabhängig von einander arbeiten, da eine Gesamtsimulation unterschiedlich weit verfeinerter Modelle durchführbar ist. Eine bessere Arbeitskontrolle ist dadurch auch möglich, da man die Simulationsergebnisse vor und nach einem Entwurfsschritt miteinander vergleichen kann. [4]

7.5 Unterstützung des Entwurfs komplexer Schaltungen

Die durch VHDL verstärkte Verbreitung von Synthesewerkzeugen erlaubt ein Umschwenken auf eine neue und produktivere Entwurfsmethodik. Dadurch, dass die Spezifikation eine simulierbare Beschreibung darstellt, kann man den Entwurf frühzeitig überprüfen. Verhaltensbeschreibungen in VHDL können synthetisiert werden. Die Entwicklung wiederverwendbarer Modelle wird unterstützt, da bereits entwickelte VHDL Beschreibungen in anderen Code eingefügt werden können und es bestehen Umsetzungsmöglichkeiten auf verschiedene Technologien. Alles zusammen führt zur Beherrschung von komplexeren Schaltungen und zu einer wesentlich verkürzten Entwicklungszeit.[4]

8 Nachteile von VHDL

8.1 Modellierung analoger Systeme

Zwar gibt es bei VHDL umfangreiche Beschreibungsmittel für digitale, elektronische Systeme, Konstrukte zur Modellierung analoger, elektronischer Systeme gibt es aber (noch) nicht. Dies gilt auch für komponente mit mechanischen, optischen, thermischen, akustischen oder hydraulischen Eigenschaften. Das bedeutet, dass VHDL keine vollständige Verhaltensmodellierung eines kompletten technischen Systems bietet.[4]

8.2 Komplexität

Die Komplexität der Sprache und damit die vielen Modellierungsmöglichkeiten werden als Vorteil gesehen, bringen aber auch Nachteile mit sich. Es wird ein enormer Einarbeitungsaufwand benötigt, weitaus länger als bei anderen Sprachen wie Verilog. Das verhalten eines komplexen VHDL-Modells in der Simluation ist für einen Neulig so gut wie nicht nachvollziehbar, da viele der Mechanismen nicht von gängigen Programmiersprachen abgeleitet werden können. Das ganze wird auch dadurch erschwert dass das Nachschlagewerk für VHDL, das „Language Reference Manual“, als eines der am schwersten zu lesenden Bücher der Welt gilt. („The base type of a type is the type itself “). Es gibt mittlerweile jedoch andere, leichter zu lesende Referenzen.[4]

8.3 Synthese-Sprachumfang

VHDL ist als Sprache in der Syntax und Simulationssemantik normiert, als Eingabe für Synthesewerkzeuge jedoch nicht. Das Bedeutet, dass jedes Synthesewerkzeug einen etwas anderen VHDL-Sprachumfang unterstützt und einen spezifischen Modellierungsstil erfordert. Hinzu kommt, dass VHDL Konstrukte enthält, die zwar für die Simulation nötig sind, sich aber nicht in eine Hardware-Realisierung umsetzen lassen. Aufgrund dessen müssen VHDL-Modelle auf ein spezielles Synthesewerkzeug zugeschnitten sein, was die Abhängigkeit vom Werkzeughersteller erhöht. Ausserdem muss der Entwickler die Anforderungen des gewählten Werkzeuges kennen und von Anfang an berücksichtigen.[4]

9 Alternativen zu VHDL und Ausblick

VHDL und Verilog sind zur Zeit die am weitesten verbreiteten Hardwarebeschreibungssprachen. Verilog bietet den Vorteil der leichteren Erlernbarkeit, da es sehr C-ähnlich ist. VHDL bietet den Vorteil ausgereifter und abstrakter zu sein, so dass man sich weniger um die direkte Implementierung von Hardware kümmern muss. Beide Sprachen bieten prinzipiell ähnliche Möglichkeiten, obwohl vom Grundsatz her VHDL eine allgemeinere Basis zur Verfügung stellt. In Verilog müssen viele Datentypen erst erzeugt werden, während diese

in VHDL bereits Standard sind. VHDL eignet sich bei Systemen und komplexen Designs wesentlich besser als Verilog. Es ist flexibler einsetzbar, aber auch um einiges komplizierter. VHDL ist in Europa sehr verbreitet, wogegen in Nordamerika und Japan eher Verilog verwendet wird. Ein Vorteil von Verilog ist, dass die Simulation um einiges schneller abläuft als unter VHDL.[4]

Neu auf dem Markt ist SystemVerilog, eine Weiterentwicklung von Verilog. Es ist eine Sprache für die Systemmodellierung und Verifikation auf einer hohen Abstraktionsebene. Der Verilog-Standard wird damit durch eine Vielzahl von Spracherweiterungen ergänzt und es bietet die direkte Verbindung von Verilog mit der Programmiersprache C. Desweiteren werden bei Systemverilog auch viele Sprachelemente aus VHDL eingebunden. Aus diesen Gründen ist es nicht unwahrscheinlich, dass Systemverilog der neue Standard bei Hardwarebeschreibungssprachen werden könnte.[4]

Es existiert auch eine Arbeitsgruppe unter dem Namen VHDL-200X die an einer Weiterentwicklung von VHDL Arbeitet. Allerdings ist das Projekt noch nicht Abgeschlossen und es gibt auch noch keine konkreten Ergebnisse.

Literatur

1. Siemers, Christian - Hardwaremodellierung, Hanser,2001
2. Chang,K.C., Digital design and modeling with VHDL and Synthesis, IEEE Computer Society Press,1997
3. Heinkel, Padeffke, Haas, Buerner, Braisz, Gentner, Grassmann - The VHDL Reference, Wiley, 2002
4. Gunther Lehmann, Bernhard Wunder, Manfred Selz - Schaltungsdesign mit VHDL, Franzis, 1994, Seite 44-50.
5. The Designer's Guide to VHDL, URL: http://www.doulos.com/knowhow/, 2005
6. VHDL Language Guide, URL: http://www.acc-eda.com/vhdlref/index.html
7. D. Dill and S. Tasiran, Simulation meets Formal Verification, slides from a presentation at ICCAD 1999
8. C. Kern and M. Greenstreet, Formal Verification in Hardware Design: A Survey, ACM Transactions on Design Automation of E. Systems, Vol. 4, April 1999, Seiten 123-193
9. Ron Wilson Issues lurk behind formal equivalence checking, EE Times Online, 13.02.2003
10. Software-Engineering-Wissensdatenbank URL: http://www.software-kompetenz.de/?2879
11. Jörg Lohse, Jörg Bormann, Michael Payer, Gerd Venzl - VHDL-Translation for BDD-based Formal Verification, Siemens internal report, 1994
12. Wikipedia 2005, URL: http://en.wikipedia.org/wiki/Symboliclogic

13. Wikipedia 2005, URL: http://de.wikipedia.org/wiki/ModelChecking
14. Wikipedia 2005, URL: http://de.wikipedia.org/wiki/BinaryDecisionDiagram