Mit VHDL ist ein wesentlicher Durchbruch gelungen

Franz J. Rammig

Mit VHDL ist bei Hardwarebeschreibungssprachen ein wesentlicher Durchbruch gelungen. Er besteht darin, daß eine breitbandige Entwurfsprache national (IEEE) standardisiert wurde. Es ist zu erwarten, daß entweder im Rahmen eines evolutionären Prozesses aus der heutigen VHDL oder aber als erfolgreiche Konkurrenzentwicklung eine CHDL entstehen wird, deren technische Qualitäten alle Aspekte befriedigt.

Prof. Dr. *Franz J. Rammig* (44) studierte Mathematik mit Nebenfächern Wirtschaftswissenschaften und Informatik an der Universität Bonn. Nach Studienabschluß wurde er wissenschaftlicher Angestellter am Lehrstuhl Informatik I der Universität Dortmund, wo er 1977 auch promovierte. Seit 1983 hat er eine Professur für Praktische Informatik an der Universität-GH Paderborn inne. Er ist Mitglied des zweiköpfigen Vorstands von Cadlab, dem gemeinsamen Forschungsinstitut der Universität-GH Paderborn und der Siemens Nixdorf Informationssysteme AG, und Vorsitzender des ITG-Fachausschusses 5.2 „Rechnerunterstütztes Entwerfen", ist Mitglied zweier gemeinsamer GI/ITG-Fachgruppen und mehrerer internationaler Fachgremien. (Universität-Gesamthochschule Paderborn, FB 17, Mathematik-Informatik, PF 16 21, 4790 Paderborn, T 0 52 51/60-20 67)

VHDL-Prinzipien

Als die Hardwarebeschreibungssprache VHDL entwickelt wurde, hatten sich bereits eine Reihe von Prinzipien für Entwurfsprachen durchgesetzt. Sie entstammen z. T. aus dem Bereich der allgemeinen Programmiersprachen, z. T. sind sie spezifisch für Hardware-Beschreibungssprachen (Hardware Description Languages, HDL). Da VHDL im Rahmen des vom amerikanischen Verteidigungsministerium (DoD) geförderten VHSIC-Projekts entwickelt wurde, war Ada die Programmiersprache, aus der eine Reihe von Prinzipien entlehnt wurde.

Von Anfang an sahen sich die Entwickler von HDL mit dem Problem konfrontiert, sowohl Verhaltens- als auch Struktursicht beschreiben zu müssen. Orthogonal zur Einteilung in Sichten ist die in Abstraktionsebenen zu sehen. In diesem Beitrag soll von einer Einteilung in Systemebene, algorithmischer Ebene, Registertransferebene, Gatterebene, Schalterebene und elektrischer Ebene ausgegangen werden.

Frühe HDL wie CHL oder DDL waren reine Verhaltenssprachen auf Registertransferebene; die Eingabesprachen für Logiksimulation sind reine Strukturbeschreibungssprachen auf Gatterebene. Mit dem Conlan-Projekt wurde durch ein generatives Konzept versucht, alle Ebenen und Sichten mit Sprachen einer Familie zu überdecken; mit Dacapo [1] ist es erstmals gelungen, das Konzept einer Ebenen- und Sichten-übergreifenden „Breitbandsprache" in die industrielle Praxis einzuführen. Diesem Konzept folgt auch VHDL.

Obwohl VHDL im Grunde zunächst eine Verhaltenssprache war, bietet sie auch reiche Sprachmittel zur strukturellen Beschreibung von Hardware. Die Verhaltenssicht ist dabei leider etwas zu sehr von der Sichtweise einer imperativen Programmiersprache geprägt. Man bemerkt deutlich, daß die Entwerfer sich nicht auf die Fragestellung konzentrieren haben, das Verhalten von Hardware zu beschreiben, sondern stets (möglicherweise unterbewußt) die Simulation dieser Beschreibung auf einem konventionellen Von-Neumann-Rechner im Auge hatten.

Das Grundprinzip einer Verhaltensbeschreibung läßt sich wie folgt darstellen (selbsterklärende Pascal-Erweiterung, die VHDL-Notation ist davon verschieden):

```
while TRUE do
    conbegin
       while TRUE do Anweisung_1;
       ...
       while TRUE do Anweisung_n;
    end
```

Die Anweisungen können dabei einfache Zuweisungen sein oder sequentielle Prozesse. All dies geschieht zunächst „zeitlos", es können aber auf einer simulierten Echtzeitachse Ereignisse eingeplant werden. Man bemerkt hier deutlich die Sichtweise der (Betriebs-)Systemprogrammierung, eine Sichtweise, die durch Ada an VHDL vererbt wurde. Es ist offensichtlich, daß damit Hardwareverhalten nicht direkt beschrieben wird, wohl aber ein Ablauf auf einem Universalrechner, der das Hardwareverhalten simuliert.

Für die Strukturbeschreibung werden ebenfalls Prinzipien angewendet, die aus dem Entwurf von Programmiersprachen bekannt sind. Ähnlich wie Dacapo [1] wird hier das Typ-Objekt-Konzept zu „components" ausgedehnt, d. h. auf prozedurähnliche Objekte mit wohldefinierter Schnittstelle. Diese Schnittstelle verfügt über formale „ports", vergleichbar mit formalen Parametern. Werden nun Instanzen derartiger Komponentendefini-

tionen angelegt und die formalen „ports" an aktuelle Signale gebunden, so beschreibt man damit präzise die Struktur eines Hardwaresystems. Man beachte, daß für dieses Konzept eine Unterscheidung zwischen Schnittstelle und Innerem einer Komponente nötig ist. Dieses Konzept wird in VHDL durch die Unterscheidung zwischen `entity`-Deklaration und `architecture`-Rumpf sehr konsequent verfolgt.

VHDL geht in dieser Hinsicht über seine Vorbilder weit hinaus, indem es sehr mächtige Werkzeuge zur Zuordnung verschiedener Rümpfe an eine Schnittstelle, und zwar Zuordnung von aktuellen Signalen an formale „ports", anbietet. Hier wird die Funktionalität von Bindern in bisher nicht bekanntem Umfang in eine Hardwarebeschreibungssprache integriert.

Von Ada übernahm VHDL das Konzept der „packages", d. h. die Möglichkeit, Typen und Funktionen getrennt zu übersetzen und in Bibliotheken abzulegen. Dieses sehr mächtige Konzept erlaubt es u. a., Logiken beliebiger Wertigkeit zu definieren. Der uralte Streit, welches die zur Modellierung am besten geeignete Logik sei, scheint damit zunächst obsolet zu sein: Jeder Benutzer kann sich die Logik seiner Bedürfnisse definieren. Dieser Vorteil wendet sich allerdings zum Nachteil, wenn VHDL eine zentrale, ihr zugedachte Rolle übernehmen soll: die einer Dokumentations- und Modellaustauschsprache. Modelle werden nur verständlich, wenn sie einschließlich des „package", das seine Logik definiert, ausgetauscht werden, und lassen sich mit Modellen anderer Logiken oft nur schwer kombinieren. Konsequenterweise gibt es Standardisierungsbemühungen zur Festlegung einer „Normallogik" (package) in VHDL.

VHDL-Objekte

VHDL kennt drei Arten von (Daten-)Objekten: `variable`, `signal` und `constant`. Eine `variable` bezeichnet dasselbe wie eine Variable in allgemeinen Programmiersprachen, d. h. einen Behälter, der im Laufe der Zeit verschiedene Werte annehmen kann. Wichtig dabei ist, daß zu jedem Zeitpunkt immer nur der aktuelle Wert bekannt ist. Im Gegensatz dazu trägt ein `signal` seine Vorgeschichte und – soweit bekannt – seine zukünftige Entwicklung in sich. Der Wertebereich eines `signal` ist daher eine Folge von Paaren (Wert, Zeitpunkt), wobei der jeweilige Wert, wie bei Variablen, im durch den Typ festgelegten Wertebereich liegen muß.

Beispiel (VHDL-Notation):

```
signal S : integer := 0
   begin
   S <= 1 after 1 ns, 3 after
        3 ns, 5 after 5 ns;
   end
```

Unmittelbar nach Ausführung des „signal assignment" (Zuweisungszeichen `<=`) ist S als Folge $<(1, t + 1\ \text{ns}), (3, t + 3\ \text{ns}), (5, t + 5\ \text{ns})>$ definiert, wobei t den Ausführungszeitpunkt der Zuweisung bezeichne. In diese Folge können im weiteren Verlauf weitere Paare eingefügt werden. In der Zeitspanne zwischen zwei angegebenen Zeitpunkten behält ein `signal` den zuletzt zugewiesenen Wert.

Jedes Paar (Wert, Zeitpunkt) bedeutet ein Ereignis. Auf derartige Ereignisse reagieren Anweisungen der VHDL-Verhaltensbeschreibung.

Für beide Objektarten, `variable` und `signal`, ist eine Typisierung möglich, wie man sie z. B. aus Ada kennt. Es gibt dabei skalare (nicht weiter zerlegbare) und zusammengesetzte Typen. Bei den skalaren Typen wird der übliche Umfang angeboten: bit (zweiwertig), integer, (sub)range, Aufzählungstyp. Der Wertebereich mehrwertiger Logiken läßt sich als Aufzählungstypen leicht vereinbaren.

VHDL bietet zwei Arten zusammengesetzter Typen: `arrays` und `records`. Hier folgt VHDL den aus Sprachen wie Pascal oder Ada bekannten Konventionen. Da die Bereichsangaben bei `arrays` offen bleiben können (Symbol `<>`), können universell einsetzbare Objekte vereinbart werden. Als Beispiele mögen die beiden vordefinierten Typen `Bit_vector` und `String` dienen:

```
type Bit_vector is array (Natural
 range <>) of Bit;
type String is array (Positive
 range <>)of Character;
```

Zu den verschiedenen Typen lassen sich Attribute vereinbaren, wobei die wichtigsten vordefiniert sind.

Mit diesem Konzept ist es möglich, in einfacher und kompakter Form Signalverläufe zu überwachen. Die Existenz derartiger Attribute bedingt aber auch, daß bei Signalen nicht nur die Ereignisse in der bekannten Zukunft, sondern auch über eine gewisse Vergangenheit gehalten werden müssen.

Die Verhaltenssicht in VHDL

VHDL unterscheidet konsequent zwischen Schnittstelle (`entity`-Deklaration) und Rumpf (`architecture`-Vereinbarung). Für einen Rumpf sind mehrere Beschreibungsstile möglich, wobei in diesem Abschnitt nur die reine Verhaltenssicht betrachtet werden soll.

Syntaktisch hat dieser Rumpf die Form

```
architecture Rumpfid of Entitätsid
is
  begin
      Anweisung_1;
      ...
      Anweisung_n;
  end Rumpfid;
```

Die Bedeutung ist, daß die n Anweisungen nebenläufig ständig aktiv sind. Dies wiederum bedeutet, daß eine Anweisung aufgrund bestimmter Ereignisse aktiviert wird und ohne Zeitverbrauch ausgeführt wird, um danach auf das nächste Eintreffen eines sie betreffenden Ereignisses zu warten. Merkwürdigerweise unterstellt die VHDL-Semantik zusätzlich noch einen globalen Ausführungszyklus über das gesamte Modell. Dieser Zyklus wird ebenfalls als keine Zeit verbrauchend angenommen, definiert jedoch „microsteps", die der VHDL-Semantik unterliegen. Hier stößt man wieder auf eine spezielle Sichtweise einer speziellen Simulatortechnik, die in der Definition einer HDL eigentlich nichts zu suchen hat. Hardwaresynthese aus VHDL-Beschreibungen wäre bei exakter Respektierung der semantischen Definition beispielsweise kaum möglich, sie müßte stets in einer programmierten sequentiellen Maschine resultieren.

Die einfachste Form einer nebenläufigen Anweisung ist eine Signalzuweisung. Eine derartige Anweisung wird immer dann ausgeführt, wenn für den aktuellen (simulierten) Zeitpunkt für eines der Argumente (rechte Seite der Zuweisung) ein Ereignis vorliegt. Eine Signalzuweisung enthält üblicherweise eine Verzögerungsangabe. Dies bewirkt dann das Einfügen von zukünftigen Ereignissen für das Zielsignal (linke Seite der Zuweisung).

```
architecture Verhalten of DFF is
   begin
       NQ <= not D after 5 ns;
   Q<= D after 5 ns;
   end
```

Beide Anweisungen werden immer dann ausgeführt, wenn für den aktuellen Zeit-

punkt ein Ereignis für *D* eingetragen ist. Dies wird für die beiden Anweisungen unabhängig festgestellt, auch wenn sie hier zufälligerweise vom selben Argument abhängen. Die sich ergebenden Werte werden für einen um 5 ns in der Zukunft liegenden Zeitpunkt für *NQ* bzw. *Q* eingetragen.

VHDL unterscheidet zwischen zwei verschiedenen Verzögerungsmodellen: trägheitslosem „transport delay", ausgedrückt durch das Schlüsselwort `transport` nach dem Zuweisungssymbol, und trägem „inertial delay", ausgedrückt durch die alleinige Verwendung des Zuweisungssymbols. Auf Feinheiten dieser Konzepte soll hier nicht eingegangen werden.

Neben der einfachen Zuweisung gibt es zwei Formen einer bedingten Zuweisung. Die `when`-Anweisung hat die syntaktische Form:

```
Ziel <= transport
     Ereignisfolge_1 when Bedin-
      gung_1 else
     ...
     Ereignisfolge_(n-1) when Be-
      dingung_(n-1) else
     Ereignisfolge_n;
```

Hier werden die verschiedenen Bedingungen der Reihe nach versucht, bis die erste angegebene Bedingung wahr ist. Die zugeordnete Ereignisfolge („waveform") wird dem Zielsignal zugewiesen. Die Anweisung wird immer dann ausgeführt, wenn ein Ereignis eines Signals in irgendeiner Bedingung vorliegt.

```
architecture when_Stil of
 NAND_Gatter is
   begin
     C <= transport
       '0' after 7 ns when
         A = '1' and B = '1' else
       '1' after 5 ns;
   end when_Stil;
```

Die select-Anweisung hat die Form

```
with Ausdruck select
     Ziel <= transport
       Ereignisfolge_1 when Be-
        dingung_1;
       ...
       Ereignisfolge_n when Be-
        dingung_n;
```

Hier wird diejenige Ereignisfolge dem Ziel zugewiesen, deren zugehörige Bedingung gleich dem aktuellen Ausdruck ist. Wieder wird die Anweisung immer dann ausgeführt, wenn ein Ereignis für ein im Ausdruck vorkommendes Signal vorliegt.

Die allgemeinste Form einer nebenläufigen Anweisung ist die `process`-Anweisung. Sie hat die syntaktische Form

```
process (Sensitivitätsliste);
Deklarationen
begin
    Anweisungen;
end process;
```

Ein `process` hat damit gewisse Ähnlichkeiten mit einer sequentiellen Prozedur üblicher Programmiersprachen. Vor allem kann er über lokale Variable verfügen, auch über lokale `signals`. Die Semantik eines `process` ist wie folgt:

Ein `process` beschreibt einen streng sequentiellen Ablauf, der immer dann angestoßen wird, wenn für ein `signal`, das in der Sensitivitätsliste aufgeführt ist, ein Ereignis eintritt. Nach Anstoß läuft der `process` ohne Zeitverbrauch ab, kann aber für `signals` Ereignisse einplanen. Beispiel:

```
architecture Verhalten1 of
 NAND_Gatter is
   begin
     process (a,b); --  a, b in-
      put-, c outputport
     variable zwischenwert: Bit
      := '1';
     begin
       zwischenwert := a NAND b;
       c <= zwischenwert after
        5 ns;
     end process;
   end Verhalten1;
```

Ein `process` mit einer Sensitivitätsliste, die ständig ein Ereignis enthält, läuft natürlich zyklisch ohne Unterbrechung unendlich lange ab. Dies gilt auch für einen `process` ohne Sensitivitätsliste. Neben der Steuerung (Synchronisation mit externen Ereignissen) über die Sensitivitätsliste ist auch eine gezielte Suspendierung über eine `wait`-Anweisungen möglich. Das obige Beispiel ist gleichwertig mit:

```
architecture Verhalten2 of
 NAND_Gatter is
   begin
     process
       variable Zwischenwert: Bit
        := '1';
       begin
         Zwischenwert:= a NAND b;
         c <= Zwischenwert after
          5 ns;
         wait on a, b;
     end process;
   end Verhalten2;
```

Daß die `wait`-Anweisung hier am Ende des Prozesses steht, mag verblüffen. Es liegt daran, daß laut VHDL-Semantik zu Beginn einer Simulation alle Prozesse zunächst einmal gestartet werden. Nach dieser Initialisierung ist es natürlich irrelevant, ob die `wait`-Anweisung am Anfang oder am Ende eines Prozesses steht.

Mit der `wait`-Anweisung kann man Prozesse auch gezielter aktivieren, d. h. als Reaktion auf bestimmte Ereignisse. In diesem Fall hat die Anweisung die Form `wait until Bedingung`. Außerdem kann man auch für eine bestimmte Zeit warten: `wait for Zeitraum`.

Neben der Zuweisung an `variable` (Zuweisungssymbol :=) und `signals` (Zuweisungssymbol <=) gibt es die aus imperativen Sprachen bekannten Konstrukte Fallunterscheidung und Schleife.

Fallunterscheidungen gibt es in mannigfaltigen Variationen, die hier nicht diskutiert werden sollen. Auch an Schleifenkonstruktionen gibt es eine Vielzahl: Endlosschleife, Schleife über Bereich und Schleife mit Bedingung.

Eine wichtige Diskussion wurde bisher zurückgestellt: die der konkurrierenden Zuweisung an gemeinsame Signale. VHDL stellt sich hier auf den Standpunkt, daß für Zuweisungen an ein `signal` aus verschiedenen nebenläufigen Anweisungen heraus eine ständig wirksame Konfliktauflösung („resolution function") vorliegen muß („strukturelle Resolution"). Eine Konfliktauflösung durch Sicherstellen, daß keine zeitgleichen Zugriffe stattfinden („temporale Resolution"), wie sie beispielsweise Dacapo zusätzlich anbietet, kennt VHDL nur innerhalb eines Prozesses. Streng genommen liegt in diesem Fall gar keine Resolution vor, da durch die strikt sequentielle Natur eines Prozesses in diesem Fall Konfliktsituationen ausgeschlossen sind.

Die Struktursicht in VHDL

In der bisherigen Diskussion wurde nur auf den Rumpf von Entwurfobjekten eingegangen, nicht auf deren Schnittstelle. Nach außen ist eine derartige Entität aber gerade durch ihre Schnittstelle sichtbar, so daß es sinnvoll erscheint, deren Definition in VHDL in einer `entity`-Deklaration anzugeben. Die `entity`-Deklara-

tion muß stets verfügbar sein, darauf wird für alle Bindevorgänge zugegriffen. Es ist dann ohne weiteres erlaubt, daß es für eine entity-Deklaration mehrere verschiedene architecture-Beschreibungen gibt, wovon dann jeweils eine bestimmte ausgewählt werden kann. So wird beispielsweise in VHDL die Entwurfmethodik der schrittweisen Verfeinerung unterstützt, indem man – ausgehend von einer groben Verhaltensbeschreibung – im Rumpf zu immer feineren, strukturell aufschlüsselnden Beschreibungen übergeht, die externe Schnittstelle jedoch beibehält. Das Zusammenspiel von entity und architecture soll anhand des folgenden, sehr einfachen Beispiels illustriert werden:

```
entity NAND_gate is
    port (a, b : in bit; c : out
     bit);
end NAND_gate;
architecture simple_behaviour of
 NAND_gate is
    begin
       c <= a nand b after 5 ns;
end simple_behaviour;
```

Alle im bisherigen Text als Beispiele angegebenen architectures muß man sich durch eine entsprechende entity-Deklaration vervollständigt vorstellen. Für eine Strukturbeschreibung benötigt man nun Komponententypen, mit Anschlußpunkten, muß von derartigen Komponententypen Instantiierungen bilden (die enthalten natürlich implizit auch eine Instantiierung von Anschlußpunkten) und muß schließlich diese instantiierten Anschlußpunkte verbinden. Komponententypen werden durch component-Deklarationen vereinbart. Sie sehen syntaktisch wie entity-Deklarationen aus, stellen aber nichts weiter dar als einen Verweis auf eine Entität. Von derart vereinbarten Komponenten können dann durch Instantiierungsanweisungen beliebig viele Instanzen (Kopien) angelegt werden. Die Verbindung untereinander geschieht dann durch signals. Konsequenterweise sind ports stets Anschlußpunkte für signals. Welche signals an welche ports anzuschließen sind, wird in einer port-map beschrieben. Dieses Konzept kann anhand eines kleinen Beispiels illustriert werden. Beschrieben wird der strukturelle Aufbau eines Einbitvolladdierers. Sein Verhalten ergibt sich dann implizit aus dem Verhalten der Komponenten (Halbaddierer und OR-Gatter):

```
entity full_adder is
    port (a1, a2, cin : in bit;
     plus, carry_out : out bit);
end full_adder
architecture structure of
 full_adder is
 --Deklaration der verbindenden
   Signale
signal Zwischensumme : bit;
signal Zwischencarry_1 : bit;
signal Zwischencarry_2 : bit;
 --Deklarationen der zu benutzen-
   den Komponententypen
component half_adder
    port (x, y : in bit; sum,
     carry : out bit);
end component;
component OR_gate
    port (a, b : in bit; c : out
     bit)
end component
 --man beachte, daß die Entitä-
   ten, auf die Bezug genommen
   wird, getrennt übersetzt sein
   können, durch Vergleich der
   Port-Typen kann geprüft wer-
   den, ob die Referenz legal
   ist.
begin
 --Instantiierung von Komponen-
   ten und Beschreibung der Ver-
   schaltung
H0 : half_adder
port map (x => a, y => b,
 sum => Zwischensumme, cout =>
 Zwischencarry_1);
H1: half_adder
port map (x => Zwischensumme,
 y =>cin, sum => plus,
 cout => Zwischencar-ry_2);
O0: OR_gate
port map (a => Zwischencarry_1,
 b=> Zwischencarry_2, c => car-
 ry_out);
end structure;
```

Dieses kleine Beispiel sollte die Grundprinzipien einer Strukturspräche darstellen. Zu beachten ist, daß die Beschreibung der Entitäten half_adder und OR_gate vorliegen müssen (in der Regel als vorübersetzte Einheiten in einer Bibliothek), falls das Modell simuliert werden soll.

Neben den dargestellten Grundprinzipien gibt es sehr elaborierte Konstrukte. Zu nennen sind hier vor allem generative Konzepte, mit denen man Generatoren für regelmäßige Strukturen angeben kann, und ein generic-Konzept, mit dessen Hilfe man verschiedene Varianten einer Objektklasse mit wenig Aufwand beschreiben kann (Anlehnung an Ada) [2, 3].

Schlußbemerkungen

Mit VHDL ist bei Hardwarebeschreibungssprachen ein wesentlicher Durchbruch gelungen. Dieser besteht nicht unbedingt in der Qualität der Sprache, hier gibt es eine Reihe von Mängeln, die hauptsächlich durch die zu einseitige Sicht einer speziellen Simulatortechnologie und einem zu konservativen Ansatz der Sprachentwerfer entstanden sind. Der Durchbruch besteht darin, daß es gelungen ist, eine breitbandige Hardwarebeschreibungssprache national (IEEE) zu standardisieren und in Richtung eines internationalen Standards entscheidende Fakten zu schaffen. Ist dieser Schritt getan und als Folge davon die Anwendung einer breitbandigen CHDL (eben VHDL) in der Entwurfspraxis allgemein akzeptiert, so wird – entweder im Rahmen eines evolutionären Prozesses aus der heutigen VHDL oder als erfolgreiche Konkurrenzentwicklung – eine CHDL entstehen und benutzt werden, deren technische Qualität in allen Aspekten befriedigt. Und in vielen Aspekten befriedigt eben heute schon der Ansatz von VHDL.

Literatur

[1] Dacapo III System User Manual. Dosis GmbH, Dortmund, 1987

[2] *Coelbo, D. R.*: The VHDL handbook. Kluwer Academic Publ., 1989

[3] *Lipsett, R.; Schaefer, C.; Ussery, C.*: VHDL: Hardware description and design. Kluwer Academic Publ., 1989

[4] IEEE Standard: VHDL Language Reference Manual. Std 1076-1987. IEEE, 1988