VHDL Kurzreferenz

Aufbau eines VHDL Moduls

Ein VHDL Modul besteht im Wesentlichen aus zwei Teilen: Der Schnittstellendefinition (`entity`) und mindestens einer zugehörigen Architekturbeschreibung (`architecture`). In einer Konfiguration (`configuration`) wird u.a. festgelegt, welche `architecture` für eine `entity` benutzt werden soll. VHDL Module greifen oftmals auf bereits bestehende Konstrukte zu. Diese Konstrukte sind in Pakete gegliedert und in Bibliotheken (`library`) abgelegt. Sie müssen vor ihrer Benutzung eingebunden werden.

Eine VHDL Modul ist typischerweise wie folgt gegliedert:

1. Deklaration von Bibliotheken und Auswahl von Paketen
2. Schnittstellendefinition des Moduls
3. Architekturbeschreibung des Moduls

In diesem einfachen Fall benötigt man keine Konfiguration, da zu der `entity` nur eine `architecture` existiert. Die Syntax eines solchen VHDL Moduls sieht wie folgt aus (Strings in „spitzen Klammern“ (`<>`) sind frei wählbare Namen):

```
library <LibraryX>;
use <LibraryX>.<PackageY>.all;
[...]

entity <Modul> is
port(
  <InSignal> : in <Type_of_InSignal>;
  <OutSignal>: out <Type_of_OutSignal>;
   ...
);
end <Modul>;

architecture <Functionality> of <Modul> is
-- Declare Sub-Components and Signals
begin
-- Concurrent Statements
end <Functionality>;
```

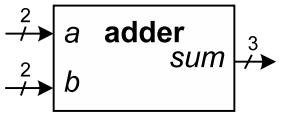
Bibliotheken

`library IEEE;` macht die Bibliothek *IEEE* sichtbar
`use IEEE.STD_LOGIC_1164.ALL;` bindet alle Konstrulte aus dem Paket *STD_LOGIC_1164* der Bibliothek *IEEE* ein
`use IEEE.STD_LOGIC_1164.STD_ULOGIC;` bindet nur den Datentyp *STD_ULOGIC* aus dem Paket *STD_LOGIC_1164* ein

Entities

Die Spezifikation eines VHDL Moduls besteht aus zwei Teilen: Der Beschreibung der externen sowie internen Sicht. Die externe Sicht auf ein VHDL Modul ist die Schnittstellendefiniton oder *Entity Declaration*. Eine Entity ist die Black-Box Beschreibung eines VHDL Moduls. Hier wird (hauptsächlich) spezifiziert, welche Anschlüsse (`ports`) das Modul besitzt.

Im folgenden Beispiel ist ein Block gegeben, der einen Addierer beschreibt: Ohne die Funktionalität zu kennen, kann man die Schnittstellendefiniton in VHDL angeben:




```
entity adder is
 port(
  a,b : in  std_logic_vector(1 downto 0);
  sum : out std_logic_vector(2 downto 0)
 );
end entity adder;
```

Die Ports `a, b` sind Inputs des Moduls `adder`, `sum` ist ein Output Port. Der Datentyp aller Ein- und Ausgangsleitungen ist `std_logic_vector`.

Architectures

Die interne Sicht auf ein VHDL Modul ist die Architektur oder *Architecture Body*. Hier wird spezifizert, was innerhalb der „Black-Box“ passiert. Eine `architecture` ist folgendermaßen aufgebaut:

```
architecture <Architecture> of <Component> is
-- Declare Signals, Components, ...
begin
-- Concurrent Statements
end <Architecture>;
```

Eine mögliche Architektur für die Entity `adder` ist:

```
architecture myAdder of adder is
 signal x,y: std_logic_vector(2 downto 0);
begin
 x <= '0' & a;
 y <= '0' & b;
 sum <= x + y;
end myAdder;
```

Durch Anhängen einer vordern Null (MSB=Most Significant Bit) mit dem &-Operator werden aus den 2-bit Eingangsvektoren `a` und `b` 3-bit Vektoren gemacht und diese den Signalen `x` und `y` zugewiesen. Die *lokalen Signale* `x` und `y` müssen dazu zuvor (vor `begin`) deklariert werden. Durch die *Signalzuweisung* `sum <= x + y;` wird auf den Ausgangsport `sum` die Summe der Werte auf den beiden Signalen `x` und `y` zugewiesen. Das man den +-Operator benutzen kann, ist in VHDL nicht selbstverständlich. Hierzu muss zusätzlich ein Paket eingebunden verden, das den Operator definiert, z.B. mit `IEEE.STD_LOGIC_ARITH.ALL`.

Notiz: Wenn man in einer typischen SW-Programmiersprache eine +-Operation benutzt, wird diese durch den Compiler in eine (oder mehrere) Addier-Instruktion(en) überführt. Fast jeder Prozessor unterstützt mindestens einen Addierbefehl. Bei der HW-Beschreibungssprache VHDL befindet man sich auf einer tieferen Abstraktionsebene! Hier kann vielmehr spezifiziert werden, wie die Addition als Schaltung funktioniert.

Angenommen, wir hätten die Bibliothek `STD_LOGIC_ARITH` nicht zur Verfügung, dafür aber ein VHDL Modul namens `fulladder` welches die Summe aus den Eingangs-Bits `A, B` und `Cin` auf den Ausgang `S` schreibt und einen etwaigen Übertrag über die ebenfalls 1-bit breite Ausgangsleitung `Cout` ausgibt. In der Architektur `myOtherAdder` werden zwei dieser Volladierer benutzt um die 3-bit Summe der beiden 2-bit Eingangswerte `a` und `b` zu berechnen. Einzelne Bitstellen eines Vektors kann man selektieren, indem man den Index der gewünschten Position in Runden Klammern nach dem Signalnamen angibt. `a(0)` etwa selektiert das Bit mit dem niedrigsten Wert (hier Index 0) aus dem Vektor `a`. Ist `a` vom Typ `std_logic_vector`, so ist `a(0)` vom Typ `std_logic`.

```
architecture myOtherAdder of adder is
 component fulladder is
  port(A, B, Cin: in std_logic;
       S, Cout : out std_logic );
 end component fulladder;
 signal c0, c1: std_logic;
begin
 c0 <= '0';
 add0: fulladder port map( A => a(0), B => b(0),
 Cin => c0, S => sum(0), Cout => c1 );
 add1: fulladder port map( A => a(1), B => b(1),
 Cin => c1, S => sum(1), Cout => sum(2) );
end myOtherAdder;
```

Instanziierung von Komponenten

Wie Instanzen einer Sub-Komponente gebildet werden, kann man im obigen *fulladder*-Beispiel sehen. Zunächst muss die Komponente (mit `component`) eingebundenen werden. Die Instanziierung einer Sub-Komponente kann man sich als „einsetzen“ eines bestehenden Moduls in die aktuelle Schaltung vorstellen. VHDL basiert stark auf der hierarchischen Verschachtelung von Modulen. Einmal implementierte Lösungen können so vielfach wiederverwendet werden. Möchte man eine Instanz bilden, so gibt man zuerst den Namen der Instanz an und dann, abgetrennt durch einen Doppelpunkt den Namen der Sub-Komponente und damit den Typ der Instanz. Anschließend folgt das sogenannte *port mapping*. Dabei werden die „Anschlüsse“ (*ports*) der gebildeten Instanz mit Signalen verbunden. Diese Signal können entweder lokale Signale sein, oder auch die *ports* des Moduls in das die Sub-Komponente eingesetzt wird. Die Syntax für das verbinden eines *ports* ist: `<Port der Sub-Komponente> => <Port/Lokales Signal des aktuellen Moduls>`.

Prozesse

Prozesse stellen die wichtigste (komplexe) Parallele Anweisung (*concurrent statement*) in VHDL dar. Innerhalb einer `architecture` können mehrere Prozesse spezifiziert werden. Wichtig ist: Alle Prozesse laufen zueinander parallel! Generell gilt für die concurrent statements einer Architektur, dass es völlig egal ist, in welcher Reihenfolge sie angegeben werden. Für VHDL (wie für HW allgemein) gilt: **Alles passiert parallel zueinander!** Natürlich gibt es auch in HW Abläufe von Ereignissen, die einzelnen HW-Elemente sind aber ständig aktiv. Es ist daher nicht trivial, das zeitliche Verhalten von in VHDL beschrieben Schaltungen zu verstehen.

Prozesse haben im Allgemeinen folgende Struktur:

```
<label>: process [(sensitivity_list)]
 --Local Declarations
begin
 --Sequential Statements
end process;
```

So kann etwa die Signalzuweisung `a<=b xor c;` auch als Prozess spezifiziert werden:

```
compute_xor: process (b,c)
begin
  a<=b xor c;
end process;
```

Prozesse werden häufig eingesetzt, um zusammenhänge Teile eines Moduls zu kapseln. Außerdem benötigt man Prozesse um getaktete, sogenannte sequentielle Logik zu beschreiben.

Datentypen

Der einfachste **binäre Datentyp** in VHDL ist `bit`. Ein Signal vom Typ `bit` kann die Werte '0' und '1' annehmen. Da man mit VHDL digitale Systeme (vollständig) beschreiben möchte, reicht dieser Datentyp für normalerweise nicht aus, denn Signale einer Schaltung können weitere Werte annehmen. Der Datentyp `std_logic` ist das „physikalische" Pendant zu `bit`. Signale vom Typ `std_logic` haben folgende mögliche Werte:

'U'	nicht initialisiert	'Z'	hochohmig	
'X'	treibend *unbekannt*	'W'	schwach *unbekannt*	
'0'	treibend *logische 0*	'L'	schwach *logische 0*	Oft
'1'	treibend *logische 1*	'H'	schwach *logische 1*	
'-'	*don't care*			

werden mehrere binäre Signale zu einem **Bus** zusammen gefasst. In VHDL heißen solche Busse Vektoren. Am häufigsten wird der Datentyp `std_logic_vector` für Busse mit dem Grundtyp `std_logic` verwendet. (Vektoren sind als **Array** (vor-)definiert: `TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF std_logic;`)

```
signal c:std_logic_vector(3 downto 0):="0001";
```

Hier wird ein Signal namens *c* erzeugt, das einen 4-Bit Vektor aus `std_logic` Signalen implementiert und mit dem Wert "0001" initialisiert ist. In Klammern ist die Indizierung des Vektors angegeben, d.h., das Bit ganz links hat den Index 3, das Bit ganz rechts den Index 0, u.s.w. Alternativ könnte man die Reihenfolge der Nummerierung ändern, indem man den Vektor mit `(0 to 6)` indiziert. Die Gewichtung der Bits (wenn man den Vektor als Zahl auffasst) wird durch die Indizierung nicht beeinflusst. In VHDL steht das Bit mit niedrigsten Wert (LSB=least significant bit) immer ganz rechts!

Der **ganzzahlige Datentyp** in VHDL ist `integer`. Ein `integer`-Signal deklariert man wie folgt:

```
signal count : integer range 0 to 10 := 1;
```

Hier wird ein Signal namens *count* erzeugt, das ganze Zahlen im Wertebereich [0..10] speichern soll. Zusätzlich wird count mit 1 initialisiert. Zu `integer` existieren noch zwei Untertypen mit vordefinierten Wertebereichen. `positive` speichert positive Werte ($[1..n]$), `natural` besitzt den Wertebereich $[0..n]$. n ist dabei der maximale Integer-Wert (dieser Wert ist systemabhängig).

Neben den vordefinierten Typen können in VHDL auch eigene Typen definiert werden. Im einfachsten Fall sind dies **Aufzählungstypen**.

```
type AMPEL is (ROT,GELB,GRUEN);
```

Dieser Ausdruck definiert den Typ AMPEL, der die Werte ROT, GELB und GRUEN annehmen kann. Man kann im dann Signale dieses Typs, z.B. wie folgt, erzeugen und initilisieren:

```
signal myAmpel : AMPEL := ROT;
```

Attribute

In VHDL gibt es Typ-bezogene, Feld-bezogene (ein Feld ist ein Vektor bzw. ein Array) und Signal-bezogene Attribute. Die Syntax für Attribute ist in jedem Fall `<Typ/Objekt>'<Attribut>`.

Typ-bezogene Attribute geben Informationen über einen Datentyp. So liefert etwa `AMPEL'pos(GELB)` den Integerwert 1 zurück und damit die Position des Wertes GELB in dem Aufzählungstyp AMPEL. `AMPEL'succ(GELB)` liefert mit GRUEN einen Wert vom Typ AMPEL.

Feld-bezogene Attribute werden auf Arrays bzw. Vektoren angewendet. Angenommen, der Vektor `c` vom Typ `std_logic_vector(7 downto 4)` trägt den Wert "0001", dann gilt z.B.: `a'left=7`, `a(a'left)='0'`, `a'right=4`, `a(a'right)='1'`.

Signal-bezogene Attribute geben Informationen zum dynamischen Signalverhalten. Als ein wichtiges Beispiel kann das Signalattribut `event` angegeben. Der Ausdruck `a'event` ist immer dann wahr, wenn sich der Wert des Signals `a` ändert. Dieses Attribut wird in der Logiksynthese (Übersetzung einer VHDL-Beschreibung in eine Schaltung) benutzt um sequentielle Logik zu beschreiben. Ist `clk` das Clock-Signal der Schaltung, so ist der Ausdruck `(clk'event and clk='1')` immer an den *steigenden Taktflanken* wahr. Innerhalb von Prozessen wird dieser Ausdruck benutzt, um bestimmte Signalzuweisungen nur bei steigenden Taktflanken auszuwerten. Die gesetzten Werte sind dann für den Zeitraum der folgenden *Taktperiode* stabil. Mit einer solchen Spezifikation kann also das Verhalten eines Speicherelements beschrieben werden (siehe Prozesse).

Operatoren

Bei **logischen Ausdrücken** der Form `a` *Operator* `b` müssen beide Operanden vom gleichen Datentyp (`std_logic` oder `std_logic_vector`) sein. Folgende Operatoren sind definiert:
`and`, `or`, `nand`, `nor`, `xor`, `xnor`, `not` (nur der `b`-Operand!)

Neben den logischen Basisoperatoren sind weitere Operatoren Paketen (wie etwa `std_logic_arith` definiert. Die Operanden von **relationalen Ausdrücken** können (fast) beliebigen müssen aber auch gleichen Typs sein. Mögliche Operatoren sind dabei:
`=` ($a = b$), `/=` ($a \neq b$), `<` ($a < b$), `>` ($a > b$), `<=` ($a \leq b$), `>=` ($a \geq b$)

Desweiteren sind **shift Ausdrücke** auf Vektoren in der Form `a` *ShiftOp* `b` definiert. Hierbei muss `a` ein Vektor und `b` ein Integer sein. Mögliche shift-Operatoren sind:
`sll` (shift left logical), `srl` (_ right logical), `sla` (_ left arithmetic), `sra` (_ right arith.), `rol` (rotate left), `ror` (rotate right)

Weitere Operatoren sind:

additive:	+, −, & (Konkatenation)
multiplikative:	∗, /, *mod* (Modulo), *rem* (Divisionsrest)
sonstige:	∗∗ (Potenz), *abs* (Betragsfunktion)

Concurrent Statements

Durch die **Signalzuweisung** `a <= b;` erhält das Signal `a` den Wert von Signal `b`. `a` und `b` müssen gleichen Typs sein und in der `architecture` definiert sein, entweder als lokales Signal oder als Port der `entity`. Eine Signalzuweisung kann eine (beliebige) kombinatorische Verknüpfungen von Signalen sein, wie etwa: `a <= b xor (c and d);` Wichtig ist, beide Seiten einer Signalzuweisung müssen gleichen Typs sein! Ist a vom Typ `std_logic` und b vom Typ `std_logic_vector(6 downto 0)` so ist die Zuweisung `a <= b;` ungültig. Man kann allerdings einzelne Leitungen eines Vektors selektieren. Somit wäre z.B. die Zuweisung `a <= b(0);` korrekt.

Ein weiteres *concurrent statement* ist die **Bedingte Signalzuweisung**. Hierbei werden verschieden Zuweisungsalternativen durch Bedingungen gesteuert.

```
a <= b0 when sel0 = '1' else
     b1 when sel1 = "00" else
     b2;
```

Ziel der Zuweisung beim obigen Beispiel ist in allen Fällen das Signal `a`. Der zugewiesen Wert liegt, abhängig von den Bedingungen, an den Signalen `b0`, `b1` oder `b2` an, welche alle vom selben Typ wie `a` sind. Die Bedingung nach dem `when` kann alle (lesbaren) Signale beinhalten. Eine zweite Form der bedingten Signalzuweisung ist die **Selektive Signalzuweisung**.

```
with sel1 select
a <= b0 when "00",
     b1 when "11",
     b2 when others;
```

Im Beispiel wird ein Signal (das *select Signal*, hier *sel1*) überprüft und anhand der Belegung dem zu setzenden Signal (*a*) ein Wert zugeordnet. Es ist wichtig, dass immer alle Alternativen des select Signals überprüft werden. Im Beispiel ist dies durch den `when others` Fall abgedeckt, der alle nicht-aufgeführten Alternativen beinhaltet. Die selektive Signalzuweisung ist ideal, um Multiplexer direkt zu modellieren.

Sequential Statements

Neben den verschiedenen Signalzuweisungen sind Prozesse wichtige *concurrent statements*. Innerhalb von Prozessen kann eine zweite Form, die sogenannten *sequential Statements*, benutzt werden. Das `if`-Statement erlaubt in Prozessen die bedingte Ausführung von weiteren sequentiellen Statements.

```
if <condition0> then
 <seq. statements>
elsif <condition1> then
 <seq. statements>
else
 <seq. statements>
end if;
```

```
if sel1="00" then
  a <= b0;
  y <= x0;
elsif sel0='1' then
  a <= b1;
else
  a <= b2;
end if;
```

VHDL Prozesse unterstützen auch ein `case`-Statement. Dabei sollte darauf geachtet werden, einen *default* Fall (`when others`) anzugeben.

```
case <selectSig> is
 when <choice0> =>
  <seq. statements>
 when <choice1> =>
  <seq. statements>
  ...
end case;
```

```
case sel1 is
 when "00" =>
  a <= b0;
  y <= x0;
 when others =>
  a <= b1;
end case;
```

Diese Referenz soll als Hilfe zum Einstieg in VHDL aufgefasst werden. Sie deckt nur einen sehr kleinen Teil des Sprachumfangs ab.
Revision: 0.1, 18. Juni 2009, Heiner Giefers