ertis embedded real-time systems ulm university institute

Einführung in VHDL

VHDL ist eine Hardwarebeschreibungssprache, die sich im Gegensatz zu Softwaresprachen dadurch auszeichnet, dass Abarbeitungen parallel ablaufen können. Dieses beutetet wiederum, dass eine sequenzielle Beschreibung, wie sie in den meisten Software- Programmiersprachen stattfindet, nicht direkt in VHDL umgesetzt werden kann. Hierzu muss selbst eine Taktung eingeführt werden. Wie dieses im Einzelnen aussieht, hängt von der Situation ab. Typische Konzepte sind die Zustandsautomaten oder die Benutzung des Taktes in Zusammenhang mit einem Zähler. Als Grundregel können Sie zuerst einmal davon ausgehen, dass alle innerhalb eines Prozesses beschriebenen Aktionen parallel ausgeführt werden und die Belegung der Signale sowie Ein- und Ausgänge erst am Ende des Prozesses stattfindet. Somit ist folgende Beschreibung in VHDL möglich:

```
1  ARCHITECTURE Tauschen OF Beispiel IS
2     SIGNAL a , b : STD_LOGIC ;
3
4     BEGIN
5     PROCESS( a , b )
6        BEGIN
7        a <= b;
8        b <= a ;
9     END PROCESS;
10 END Tauschen;
```

VHDL bietet die Möglichkeit sowohl eine strukturelle Beschreibung als auch eine Verhaltensbeschreibung eines Bausteins zu implementieren, um dessen Ein- und Ausgabeverhalten zu spezifizieren. Die strukturelle Beschreibung eines Bausteines gibt an, wie viele Instanzen der einzelnen Komponenten zur Realisierung dieses Bausteines benutzt werden, wie diese Komponenten untereinander verdrahtet sind und wie diese mit den Ein- und Ausgängen des Bausteines verdrahtet sind. Die Verhaltensbeschreibung dagegen beschreibt einen Baustein nicht durch seine Komponenten, sondern durch gleichzeitig nebeneinander laufende Prozesse, welche die Belegung der Signale des Bausteines funktional bestimmen. Prozesse beschreiben damit wie sich Signale in Abhängigkeit anderer Signale verhalten.

Bevor Sie im nächsten Kapitel beginnen sich mit GHDL zu beschäftigen, sollen im Folgenden einige grundlegenden Konzepte von VHDL erklärt werden. Durch die weite Verbreitung von VHDL finden Sie auch ausreichend Literatur in der Bibliothek oder online.

Zu jeder (simulierbaren und synthetisierbaren) VHDL Beschreibung existiert eine sogenannte Top- Level-Entity. Das Hauptmerkmal dieser Komponente besteht darin, dass diese den Projektnamen trägt und dass alle nach externen geführten Signale aufgelistet sind.

Symbol	Beschreibung
U	nicht initialisiert
X	undefiniert
0	starke logische Null
1	starke logische Eins
Z	Hochohmig
W	schwach Unbekannt
L	schwach logische Null
H	schwach logische Eins
-	nicht beachten (don't care)

Tabelle 1: Signale der 9-wertigen standard logic nach IEEE Standard 1164-1993

1 Bibliotheken

Jeder Baustein beginnt mit dem Einbinden der Bibliotheken (Libraries). Folgende zwei Zeilen werden am Anfang von allen Ihren Bausteinen stehen:

```
1 LIBRARY ieee;
2 USE ieee.std_logic_1164.all;
```

Der Datentyp *std_logic* ist eine nach dem *IEEE Standard 1164-1993* definiert 9-wertige Logik. Dabei können Signale die in Tabelle 2 angegebenen Werte annehmen: Im Gegensatz zu dem Datentyp *bit*, der nur die Werte '0' und '1' annehmen kann, hat *std_logic* den Vorteil alle Zustände die auch auf der Zielplattform auftreten können zu beschreiben. Innerhalb der Übungen werden Sie jedoch fast ausschließlich mit starken logischen Nullen und Einsen ('0' bzw. '1') arbeiten. Jedoch sollten sie trotzdem alle Signale immer als *std_logic* Datentypen deklarieren, da innerhalb des Syntheseprozesses von Quartus II auch andere Zustände der Signale betrachtet werden können, beispielsweise wenn Ihr Signal noch nicht initialisiert wurde ('U') oder wenn der Zustand undefiniert ('X') ist. Sobald sie elementare Rechenoperationen benötigen (wie beispielsweise Plus '+' oder Minus '-') müssen Sie zusätzlich aus der *ieee* Bibliothek Folgendes laden:

```
1 USE ieee.std_logic_unsigned.all;
```

2 Schnittstellenspezifikation

Die Schnittstellen eines Bausteines werden über die Schnittstellenspezifikation (entity Konstrukt) beschrieben. Diese stellt die externe Sicht der Komponente dar. Den Namen des Bausteines können Sie selbst bestimmen (natürlich sollten keine reservierten Worte, Zahlen, Umlaute, Leerzeichen ...verwendet werden). Jedoch muss Ihr oberster Baustein (Top-Level-Entity) den Namen des Projektes tragen. Ein Template sieht folgendermaßen aus:

```
ENTITY baustein IS
   PORT (
      portname : IN     datentyp;
      portname : OUT    datentyp ;
      portname : INOUT  datentyp
   );
END baustein ;
```

Ein typisches Beispiel dafür wäre dann:

```
ENTITY most_used IS
   PORT(
      CLOCK_50: IN STD_LOGIC;
      KEY   : IN STD_LOGIC_VECOTR( 3 downto 0 );
      SW : IN STD_LOGIC_VECOTR( 17 downto 0 ) ;
      LEDG  : OUT STD_LOGIC_VECTOR(7 downto 0);
      LEDR  : OUT STD_LOGIC_VECTOR(17 downto 0)
   );
END most_used;
```

Für die Bezeichnungen der externen Schnittstellen sollten Sie auf die im Handbuch angegebenen Bezeichnungen zurückgreifen. Dieses macht Ihnen die Arbeit leichter, da die jeweiligen Hardware- komponenten auf dem Board auch mit den gleichen Bezeichnungen beschriftet sind. Die in diesen Übungen am meisten benutzten Ein- und Ausgaben sind im Beispiel *most_used* aufgelistet. Technisch gesehen sind die PINs des FPGA direkt über Leiterbahnen mit den jeweiligen elektronischen Bauteilen verbunden. Jedoch erscheint es sehr umständlich die PINs als Schnittstellen anzugeben. Daher ist es wesentlich einfacher, den PINs Namen und Vektoren zuzuordnen, so dass beispielsweise alle grünen LEDs einen gemeinsamen Vektor LEDG haben. Dieser ist nun 8-bit Breit und läuft damit von 7 bis 0. Um Ihnen die Arbeit abzunehmen, gibt es von Altera schon die Datei, *DE2-pinassignments.csv*

Vorweg greifend für den praktischen Teil, soll noch darauf hingewiesen werden, dass alle nicht verwendeten PINs auf tri-state gesetzt werden sollen. Dieses verhindert, dass ungewollt Spannungspegel an Bauteile und PINs gelegt werden, die nicht verwendet werden. Vor der Programmierung des FPGAs sollten alle offenen Pins noch auf tri-state gesetzt werden.

3 Beschreibung von Architekturen

Nachdem Sie Ihre Bibliotheken eingebunden haben und die Schnittstellen definiert haben, können Sie die Komponente durch sein Verhalten, Datenfluss und/oder Struktur beschreiben. Der grundsätzliche Aufbau sieht folgendermaßen:

```
1 ARCHITECTURE behavior OF baustein IS
2 -- Hier koennen Signale definiert werden
3 BEGIN
4    -- Code
5 END behavior ;
```

Grundsätzlich beschreibt eine Architektur das Verhalten, bzw. die interne Realisierung einer vorher entity. Dabei können mehrere Architekturen für eine entity deklariert werden, welche später über eine configuration ausgewählt werden kann. Dabei wird zwischenVerhalten, Datenfluss und Struktur unterschieden:

3.1 Verhalten

Das Grundelement einer Verhaltensbeschreibung ist der ***process***. Der interne Aufbau eines VHDL-Prozesses ist vergleichbar mit dem sequenziellen Task einer Programmiersprache

i. sequenzielle Abarbeitung der Anweisungen

ii. Kontrollanweisungen zur Steuerung des Ablaufs

iii. Verwendung lokaler Variablen, -Datentypen

iv. Unterprogrammtechniken (Prozeduren und Funktionen)

Da das Ziel der VHDL Beschreibung ein durch den Simulator/Emulator ausführbares Verhalten ist, gibt es spezielle Konstrukte, die festlegen wann der Prozess zu aktivieren ist — im Gegensatz zu Programmen in herkömmlichen Sinne sind Hardwareelemente immer, gleichzeitig (parallel) aktiv. Ein Prozess hat deshalb entweder eine ***sensitivity list*** oder enthält ***wait***-Anweisungen. Beide Methoden bewirken, dass bei der Änderung von Eingangswerten der Architektur der Prozess von Simulator aktiviert wird, die Anweisungen sequenziell abgearbeitet werden und dadurch neue Ausgangswerte erzeugt werden. Eingeschränkt werden Prozesse durch in ihrer Kommunikation nach außen, welche nur über Signale stattfinden darf. Diese Signale können zum einen Ein- und Ausgänge der Schaltung sein, zum anderen können auch mehrere Prozesse über (architektur-)interne Signale kommunizieren. In Gegensatz zu den Signalen können die Variablen des Prozesses nur prozessintern verwendet werden und stellen so etwas wie lokalen Speicher dar.

```
1 architecture behavior of COMPARE is begin
2    P1: process (A, B) is --Prozess, sensitiv zu Eingängen A und B begin
```

```
3       if (A = B) then
4          C <= '1' after 1 ns;
5       else
6          C <= '0' after 2 ns;
7       end if;
8    end process P1;
9 end architecture behavior;
```

Listing 1: Beispiel Verhalten

3.2 Datenfluss

Bei dieser Beschreibung wird der Datenfluss über kombinatorische logische Funktionen (Addierer, Komparatoren, Decoder, Gatter. . .) modelliert.

```
1 architecture dataflow of COMPARE is begin
2    C <= not (A xor B) after 1 ns; --Datenfluss von Eingängen, über
3 end architecture dataflow;         --xor und not, zum Ausgang
```

Listing 2: Beispiel Datenfluss

3.3 Struktur

Strukturbeschreibungen sind Netzlisten aus Bibliothekselementen: diese Elemente werden instanziiert und über Signale miteinander verbunden.

```
1  architecture structure of COMPARE is
2     signal I: bit;                  --lokales Signal
3
4     component XR2 is                -- 1.Komponetnbedeklaration
5        port (X,Y: in bit; Z: out bit);
6     end component XR2;
7
8     component INV is                -- 2.Komponetnbedeklaration
9        port (X: in bit; Z: out bit);
10    end component INV;
11 begin                              --Beschreibung der Netzliste
12    I0: XR2 port map (A,B,I);       --Benutzung der Komponenten
13    I1: INV port map (I,C);
14 end architecture structure;
```

Listing 3: Beispiel Struktur

Sie sollten sich selbständig, neben der Einführung in der Vorlesung und dem Crashkurs, über die Unterschiede zwischen einer Verhaltens- und Strukturbeschreibung informieren. Im folgenden finden Sie einige oft verwendete Routinen. Es soll noch auf folgendes hingewiesen werden:

Sie können Sich innerhalb der Komponente Hilfssignale definieren. Diese sollten auch für die Schnittstellensignale verwendet werden. Sollten sich die Signalnamen der Komponente ändern (beispielsweise weil sie andere LEDs benutzen oder reservierte Namen verwendet haben) müssen sie nur die Verdrahtung am Anfang Ihrer Komponente ändern und nicht alle Namen innerhalb der Komponente ersetzten.

4 Simulation und Synthese

Ein wichtiger Unterschied besteht zwischen simulierbarem (oder funktionellem) und synthetisierbaren Code. Wie der Name schon sagt, lässt sich simulierbarer Code nicht unbedingt auf der Hardware ausführen. In den Übungen während des Onlin-Studiums benötigen wir ausschließlich simulierbaren Code benötigt. Bei einem Wechsel zur Praktischen Übung im Labor benötigen wir auschliesslich synthetisierbarenVHDL-Code. Dieser zeichnet sich dadurch aus, dass er vom Übersetzungsprogramm/Synthesetool (in unserem Fall Quartus II) in eine Netzliste übersetzt werden kann und somit auf den FPGA programmiert werden kann. Der simulierbarer Code dagegen kann auch Elemente enthalten welche nur für die Simulation übersetzt werden können, woraus aber keine Synthese und damit keine Netzliste erstellt werden kann. Dieses soll an folgendem, abstrakten Beispiel anschaulicher gemacht werden:

```
1  ARCHITECTURE Oszillator OF Simulationsbeispiel IS
2     CONSTANT clk : time := 1000 ns ;
3     BEGIN
4     PROCESS( clk , input ) BEGIN
5        out<=REJECT 10 ns INERTIAL NOT input AFTER 10 ns;
6        WAIT FOR clk /2;
7        out <= REJECT 10 ns INERTIAL input AFTER 10 ns;
8        WAIT FOR clk /2;
9     END PROCESS;
10 END Oszillator;
```

Mit den Befehlen **REJECT**, **INERTIAL** und **AFTER** kann das (zeitliche) Verhalten von realen Bausteinen beschrieben werden. Soll der Code synthetisiert werden, ist dieses nicht mehr nötig bzw. möglich, da die Beschreibung auf den realen Bauelementen läuft und daher die Zeitangaben nicht mehr zum tragen kommen. Bei der **WAIT** Anweisung handelt es sich ebenfalls um eine nicht synthetisierbare Anweisung. Die reale Hardware kann nicht wie eine Simulation oder ein Softwareprozess einfach "anhalten". Soll ein Warten implementiert werden, muss dieses indirekt über einen Zähler realisiert werden. Auch das angeben von Konstanten (sowie von Variablen) ist nur sehr eingeschränkt möglich. Das Zuweisen von direkten Zeiten, wie im Beispiel, ist nicht möglich. Daher kann auf diese Weise kein Taktgeber realisiert werden. Allgemein können Variablen wie Integer nur dazu verwendet werden, den "Schreibaufwand" zu minimieren, jedoch nicht als Variablen, die innerhalb der Laufzeit benutzt werden können.

5 Hilfreiche Routinen

5.1 Bedingungen

```
ARCHITECTURE behavior OF Bedingung IS
   SIGNAL a: STD_LOGIC_VECTOR (1 DOWNTO 0);
   SIGNAL b: STD_LOGIC_VECTOR (2 DOWNTO 0);
   BEGIN
   PROCESS( a , b )
   BEGIN
      IF (a="01") THEN
         b<="001" ;
      ELSIF (a="11") THEN
         b<="000" ;
      ELSE
         b<="111"';
      END IF;
   END PROCESS;
END behavior;
```

5.2 Fallunterscheidung

```
ARCHITECTURE behavior OF Fallunterscheidung IS
   SIGNAL a,b: STD_LOGIC_VECTOR (2 DOWNTO 0);
   BEGIN
   PROCESS( a , b )
   BEGIN
      CASE a IS
      WHEN "001" => b <= "000";
      WHEN"011" | "110"=>b<="100";
      WHEN OTHERS => b <= "111" ;
      END CASE;
   END PROCESS;
END behavior;
```

5.3 Bedingung mit „WITH . . . SELECT“

```
ARCHITECTURE structure OF Verzweigung IS
   SIGNAL a :STD_LOGIC_VECTOR(1 DOWNTO 0);
   SIGNAL out,c,d,e : STD_LOGIC;
   BEGIN
      WITH a SELECT b<= c WHEN "10",
      d WHEN "01",
      e WHEN OTHERS;
END structure;
```

5.4 Bedingung für Signale außerhalb eines Prozesses

```
ARCHITECTURE structure OF Signalbedingung IS
   SIGNAL a,b,c,d: STD_LOGIC;
   BEGIN
      a<= NOT a WHEN b = '1' ELSE ( c XOR d ) ;
END structure;
```

5.5 Clock-Signale

```
ARCHITECTURE behavior OF Clocksignal IS
   SIGNAL clk : STD_LOGIC ;
   SIGNAL count : STD_LOGIC_VECTOR(4 DOWNTO 0);
   BEGIN
      clk <= clock_50 ; PROCESS(clk)
         IF (clk='1' AND clk' event) THEN --wenn clk steigende Flanke
         count <= count + 1;
      ELSE
         count <= count ;
      END IF;
   END PROCESS;
END behavior
```

5.6 Variablen und Schleifen

```
ARCHITECTURE behavior OF Schleife IS
   SIGNAL clk , reset : STD_LOGIC;
   SIGNAL register : STD_LOGIC_VECTOR(6 DOWNTO 0);
   BEGIN
      clk <= clock_50 ;
      reset <=NOT KEY(0);
   PROCESS( c l k )
      VARIABLE i : INTEGER := 0;
      IF reset ='1' THEN
         FOR i IN 0 TO 5 LOOP
            IF i = 0 OR i = 1 THEN
               register(i) <=NOT register(i+1);
            ELSE
               register ( i ) <= register ( i +1);
            END IF;
         END LOOP;
      END IF;
   END PROCESS;
END behavior;
```