Strukturbeschreibung in VHDL

Diese Beschreibung ist wegen des V2 im Praktikum gekürzt.

1 Einleitung

Digitalschaltungen können typischerweise auf drei verschiedene Arten beschrieben werden: als Verhalten (*wenn* x=0, *dann* y<=1 bzw. sequentiell *zuerst* x<=1, *danach* x<=0), als Funktion (x_1=0, x_2=1, x_3=1 *ergibt* y<=1) oder als Struktur (*Modul 1 ist* wie folgt *mit Modul 2 verbunden*).

Die Darstellung als Struktur ist bei einem hierarchischen Entwurf für die oberste(n) Ebene(n) sinnvoll, weil man hier auf größere Funktionsblöcke (Module) zurückgreifen kann und die Struktur einen schnellen Überblick über die Zusammenarbeit der Module im Gesamtsystem erlaubt.

Speziell für die Übersicht als Mensch eignet sich die Darstellung der Struktur in einem Schaltplan sehr gut.

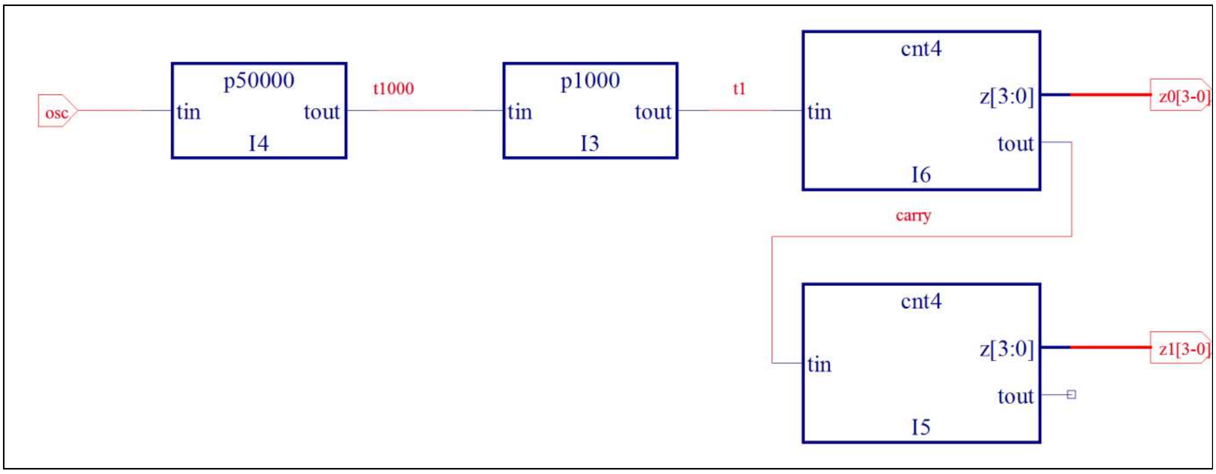

Abbildung 1: Strukturdarstellung mit Schaltplan

Der Schaltplan in Abbildung 1 ist beinahe selbsterklärend: Der am Eingang osc anliegende Takt wird mit Hilfe des Moduls I4 zuerst durch 50000 geteilt und anschließend mit Hilfe des Moduls I3 noch einmal durch 1000. Der neue, interne Takt t1 betreibt einen Zähler (Modul I6), der vier Ausgänge hat uns einerseits einen Zähltakt (*tout*) liefert. Dieser Takt betreibt einen weiteren Zähler (Modul I5) mit denselben Eigenschaften.

Unklar ist hier nur noch die genaue Funktion des Zählers, es könnte sich um ebenso gut um einen aufwärts zählenden Binärzähler (Zählfolge 0, 1, ..., 14, 15, 0) handeln wie um einen abwärts zählenden Dezimalzähler (Zählfolge 1, 0, 9, 8, ...).

Trotzdem wird aus der Struktur sehr schnell die Gesamtfunktion klar, ebenso die Zerlegung des Gesamtzählers in zwei Teilzähler gleicher Funktion.

Leider ist die Darstellung als Schaltplan aber nicht portabel, d.h. in verschiedenen Entwicklungswerkzeugen muss der Schaltplan immer neu erstellt werden. Es ist üblicherweise auch nicht möglich, die Module zu parametrisieren.

VHDL bietet die Möglichkeit, eine dem Schaltplan vergleichbare strukturelle Beschreibung zu verfassen. Diese Beschreibung ist dann portabel. Zudem ist es möglich, sehr bequem die Zuordnung von Strukturelementen (Blöcke im Schaltplan) zu den dann tatsächlich verwendeten Modulen (entity und architecure) zu ändern. So kann auch zu einem späten Zeitpunkt für jeden Block das geeignetste Modul gewählt werden, ohne dass die Strukturbeschreibung noch einmal geändert werden müsste.

Darüber hinaus ist es auch möglich, Module zu parametrisieren. Es wäre als möglich, nur ein Teilermodul zu entwerfen, das aber durch einen Teilerfaktor N parametrisiert wird. Diesen Universalteiler kann man dann mit wechselnden Teilerfaktoren N beliebig als Block einbinden.

Sinn dieser Zusammenfassung ist es, das in VHDL verwendete Konzept darzustellen. Dabei werden bei weitem nicht alle Möglichkeiten beschrieben. An manchen Stellen werden auch bewusst Freiheitsgrade weggelassen, so dass eine Aussage (z.B. "der Typ muss identisch sein") strenggenommen falsch ist. Allerdings kann man mit identischen Typen auch keinen Fehler machen, so dass die Einhaltung dieser Regel zunächst das Einfachste ist.

2 Konzept in VHDL

In VHDL wird die Strukturbeschreibung in folgende Teile zerlegt:

1. Definition der benötigten Module
2. Deklaration der Blöcke für die Verwendung in einer Strukturbeschreibung
3. Definition der Strukturansicht
4. Instanziierung und Verbindung der Blöcke untereinander
5. Zuordnung der Module an die Blöcke

2.1 Definition der benötigten Module

Für jedes später verwendete Modul muss eine Beschreibung vorliegen. Diese Beschreibung ist eine *entity* mit mindestens einer *architecture*. Jedes Modul kann in einer eigenen Datei vorliegen, das ergibt von vornherein eine gute Kapselung (Abbildung 2). Jedes Modul könnte also z.B. problemlos von einem anderen Entwickler erstellt und separat getestet werden. Zu einer *entity* können in VHDL mehrere *architectures* gehören. Welche *architecture* dann für einen bestimmten Block verwendet wird, kann im letzen Schritt festgelegt werden.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity p50000 is
port
(
  tin:  in  std_logic;
  tout: out std_logic
);
end;


architecture mch of p50000 is
begin
gekürzt wegen V2
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity p1000 is
port
(
  tin:  in  std_logic;
  tout: out std_logic
);
end;


architecture mch of p1000 is
begin
gekürzt wegen V2
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity cnt4 is
port
(
  tin:  in  std_logic;
  z:    out unsigned(3 downto 0);
  tout: out std_logic
);
end;

architecture mch of cnt4 is
begin
gekürzt wegen V2
end;
```

Abbildung 2: Definition der Module in separaten Dateien

2.2 Deklaration der Blöcke für die Verwendung in einer Strukturbeschreibung

Damit ein einer Strukturbeschreibung Blöcke verwendet werden können, müssen sie vorab deklariert werden. In der Programmiersprache C entspräche das der Deklaration von Funktionen vor der erstmaligen Verwendung.

> Seit VHDL 93 ist es möglich, Blöcke direkt unter Angabe der entity in einer anderen entity zu verwenden. Die Kapitel 2.2.1 und 2.2.2 können Sie überspringen. In den weiteren Kapiteln ist diese Methode bei der ersten Verwendung jeweils mit roter Schrift hervorgehoben.

2.2.1 Sprachelement *component*

Ein Block wird inhaltlich genauso wie eine *entity* deklariert. Auch hier müssen die Signalnamen am Block, die Signalrichtung und der Signaltyp angegeben werden. Jede *component* wird durch einen eindeutigen Namen bezeichnet. Der Name kann identisch mit dem Namen einer *entity* sein. In diesem Fall kann später eine automatische Zuordnung von *components* zu *entities* erfolgen. Die Namen können sich aber auch unterscheiden. Dann muss später (letzter Schritt, Zuordnung der Module an die Blöcke) noch angegeben werden, welche *entity* zu welcher *component* gehören soll.
Dasselbe gilt für die Signalnamen. Nur Richtung und Typ müssen identisch sein.
Die erste Methode spart zunächst Schreibarbeit und ist bei kleinen Entwürfen völlig ausreichend. Die zweite Methode bietet die größere Flexibilität. Dann können recht einfach fremde Modulbibliotheken für eine eigene Strukturbeschreibung verwendet werden.
Für das Beispiel wurden dieselben Namen an allen Stellen gewählt.

```
entity cnt4 is
  port
  (
    tin:         in  std_logic;
    z:           out unsigned (3 downto 0);
    tout:        out std_logic
  );
end;
```

```
component cnt4 is
  port
  (
    tin:         in  std_logic;
    z:           out unsigned (3 downto 0);
    tout:        out std_logic
  );
 end component;
```

Abbildung 3: Beispiel für eine *component*

Abbildung 3 zeigt, dass der Aufbau einer *component* identisch zu einer *entity* ist. Lediglich das Sprachelement *component* muss am Ende zwingend wiederholt worden.

2.2.2 Sprachelement *package*

Es wäre möglich, alle *components*, die man in einer Strukturbeschreibung verwenden möchte, lokal, d.h. in derselben Datei zu deklarieren. Das wird aber erstens sehr schnell sehr unübersichtlich und zweitens müsste man dann jede Änderung in allen Strukturbeschreibungen nachziehen. Das ist natürlich sehr fehleranfällig. Daher ist es besser, alle *components* in einer separaten Datei als *package* abzulegen. In der Programmiersprache C entspräche dies einem Header, in dem alle extern aufrufbaren Funktionen deklariert sind. Dieses *package* kann dann in der Strukturbeschreibung mit den bereits bekannten *library* und *use*-Anweisungen eingebunden werden.

Abbildung 4 links zeigt den formalen Aufbau eines *package*. Jedes *package* wird über einen eindeutigen Namen identifiziert (hier *pack_mch*). Das muss nicht der Name der Datei, in der das *package* abgelegt ist, sein.

Innerhalb des Deklarationsteils des package werden dann die einzelnen *components* beschrieben.

Separate Datei mit Deklarationen

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package pack_mch is

  component p50000 is
  port
  (
    tin:         in  std_logic;
    tout:        out std_logic
  );
  end component;

  component p1000 is
  port
  (
    tin:         in  std_logic;
    tout:        out std_logic
```

Einbindung in einer anderen Datei

```
library work;
use work.pack_mch.all;
```

```
  );
  end component;

  component cnt4 is
  port
  (
    tin:         in  std_logic;
    z:           out unsigned (3 downto 0);
    tout:        out std_logic
  );
  end component;

end;
```

Abbildung 4: Aufbau eines *packages*

In einer anderen Datei kann auf die Deklarationen des *package* zugegriffen werden, nachdem es eingebunden wurde (Abbildung 4 rechts). Die eigene Arbeitsumgebung heißt *work*. Mit der *library*-Anweisung wird diese Umgebung vor dem Einbinden (*use*) aktiviert.

2.3 Definition der Strukturansicht

Die Struktur bildet insgesamt wieder ein Modul und wird in VHDL entsprechend mit einer *entity* und einer *architecture* beschrieben. Die *entity* enthält wie üblich die Signale an den Modulgrenzen, im Beispiel die Signale *osc*, *z0[3..0]* und *z1[3..0]* mit den zugehörigen Richtungen und Typen. Die Signale *t1000*, *t1* und *carry* sind lokal, sie werden also wie üblich in der *architecture* deklariert.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.pack_mch.all;

entity top is
port
(
  osc:     in  std_logic;
  z0, z1:  out unsigned (3 downto 0)
);
end;


architecture mch of top is
  signal t1000, t1, carry: std_logic;

begin

end;
```

Abbildung 5: Strukturansicht (noch ohne Blöcke)

Wie Abbildung 5 zeigt, unterscheidet sich die Definition einer Strukturansicht in nichts von der bereits bekannten Modulansicht. Hier wird nur noch die eigenen Komponentenbibliothek *pack_mch* mit eingebunden, weil deren Elemente später benötigt werden. Bei der direkten Einbindung entfällt das (rot markierte Zeilen in Abbildung 5).

2.4 Instanziierung und Verbindung der Blöcke untereinander

In der *architecture* müssen jetzt die einzelnen Blöcke instanziiert, d.h. angelegt werden. Jede Instanz muss einen eindeutigen Namen bekommen. Für jede Instanz wird dann angegeben, um welche *component* es sich handelt. Anschließend werden die Anschlüsse dieser Instanz mit lokalen Signalen der *architecture* oder mit Signalen der *entity* (*ports*) verbunden. Es ist nicht möglich, Instanzen direkt, d.h. ohne ein Zwischensignal, zu verbinden. Der Aufbau einer Instanziierung sieht wie folgt aus:

```
Instanzname: Komponente port map (Signalzuordnungsliste); -- erste architecture (default)
Instanzname: Komponente(architecture) port map (Signalzuordnungsliste); -- bestimmte architecture
```

Wenn die direkte Einbindung verwendet wird, dann muss hier die entity des Blocks angegeben werden. Ist mehr als eine architecture zu dieser entity vorhanden, dann kann auch hier optional die architecture mit angegeben werden, die für diese Instanz des Block verwendet werden soll.

```
Instanz: entity name port map (Signalzuordnungsliste); -- erste architecture (default)
Instanz: entity name(architecture) port map (Signalzuordnungsliste); -- bestimmte architecture
```

Für die Signalzuordnungsliste gibt es zwei Möglichkeiten: namensgebunden und positionsgebunden.

2.4.1 Signalzuordnung nach Namen

Bei der Zuordnung nach Namen werden die Signale der *component* (so, wie sie dort deklariert sind) mit lokalen Signalen verbunden. Die Reihenfolge der Zuordnung ist beliebig. Die Signale müssen in Richtung und Typ übereinstimmen. Da lokale Signale keine Richtung haben, können sie an alle Signale der *component* angeschlossen werden. Signale aus der *entity* (*ports*) haben eine Richtung, dort muss auf die Übereinstimmung geachtet werden. Die einzelnen Zuordnungen werden durch Komma getrennt. Eine Zuordnung hat folgenden Aufbau: Signal der Komponente => lokales Signal.

```
I0: RSFF port map (R => oeffner, S => schliesser, Q => takt);
```

Für den Block I0 wird eine Komponente RSFF (vermutlich ein RS-Flipflop) eingesetzt. Die Komponente hat die Signale R, S und Q. Diese Signale werden den lokalen Signalen oeffner, schliesser und takt verbunden.

2.4.2 Signalzuordnung nach Position

Hier werden nur die lokalen Signale in der Liste durch Komma getrennt angegeben. Die Zuordnung der Signale erfolgt über die Position in der Liste. Das erste lokale Signal in der Liste wird mit dem ersten Signal in der *port*-Liste der *component* verbunden. Beispiel:

```
I0: RSFF port map (oeffner, schliesser, takt);
```

Bei dieser Methode muss man zwar weniger schreiben, aber sie ist fehleranfälliger. Erstens kann man kann sich selber bei der Reihenfolge irren. Zweitens kann es sein, dass sich die Reihenfolge ändert, weil die *component*-Deklaration extern geändert wird (anderes *package*). Ein Vertauschen der Reihenfolge von *R* und *S* kann der VHDL-Compiler dann nicht bemerken, die Beschreibung ist ja immer noch legal.

2.4.3 Offene Signale

Nicht immer braucht man alle Ein- oder Ausgänge. Eingänge muss man immer mit einem Signal verbinden, auch dann, wenn der Wert an diesem Signal für die Schaltung ohne Belang ist. Am einfachsten deklariert man dann in der *architecture* ein lokales Signal und weist ihm einen konstanten Wert zu.

Unbenutzte Ausgänge werden bei der Signalzuordnung mit ***open*** angegeben.

2.4.4 Gesamtbeschreibung

Abbildung 6 bzw. Abbildung 7 zeigt die gesamte Beschreibung für die Struktur, so wie sie im Schaltplan (Abbildung 1) gezeigt ist, in VHDL.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library work;
use work.pack_mch.all;

entity top is
port
(
  osc:    in  std_logic;
  z0, z1: out unsigned (3 downto 0)
);
end;
```

```
architecture mch of top is
  signal t1000, t1, carry: std_logic;
  signal z0i, z1i: unsigned(3 downto 0);

begin
  I4: p50000 port map(tin => osc, tout => t1000);
  I3: p1000  port map(t1000, t1);
  I6: cnt4   port map(tin => t1, z => z0i, tout => carry);
  I5: cnt4   port map(carry, z1i, open);

  z0 <= not z0i; -- wegen der Low-aktiven LED auf dem Board
  z1 <= not z1i;
end;
```

Abbildung 6: Gesamtbeschreibung Struktur (mit component und package)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top is
port
(
  osc:    in  std_logic;
  z0, z1: out unsigned (3 downto 0)
);
end;
```

```
architecture mch of top is
  signal t1000, t1, carry: std_logic;
  signal z0i, z1i: unsigned(3 downto 0);

begin
  I4: entity p50000 port map(tin => osc, tout => t1000);
  I3: entity p1000 port map(t1000, t1);
  I6: entity cnt4 port map(tin => t1, z => z0i, tout => carry);
  I5: entity cnt4 port map(carry, z1i, open);

  z0 <= not z0i; -- wegen der Low-aktiven LED auf dem Board
  z1 <= not z1i;
end;
```

Abbildung 7: Gesamtbeschreibung Struktur (direkte Einbindung)

Zur Illustration wurde bei I3 und I5 die nicht empfehlenswerte Zuordnung nach Position verwendet.

3 Verwendung universeller Module

Sieht man sich den Aufbau der Schaltung in Abbildung 1 an, dann stellt man fest, dass sich die beiden Taktteiler I3 und I4 nur in dem jeweiligen Teiler unterscheiden. Der innere Aufbau beider Module ist gleich, sie unterscheiden sich nur durch den jeweiligen Endwert des Zählers. In VHDL können Module durch die Verwendung von Parametern bei jeder Verwendung an die jeweilige Anforderung angepasst werden. Zu beachten ist dabei, dass die Parameter zum Zeitpunkt der Schaltungssynthese bekannt (berechenbar) sein müssen. Das Modul wird dann mit den berechneten Parametern in die Schaltung eingebaut, d.h. diese Parameter sind in der fertigen Schaltung nicht mehr sichtbar oder änderbar. Man kann also damit im Fall des Taktteilers den Teiler im Betrieb nicht mehr ändern.

3.1 Parameterverwendung in einem Modul

Parameter werden an ein Modul in der *entity* als Liste übergeben, diese Liste heißt *generic*. Sie enthält analog zu der Liste *port* für die Ein-/Ausgänge alle Parameternamen und deren Typ. Die Richtungsabgabe entfällt, weil ein Modul nur Parameter empfängt. Dafür können Parameter mit einem Defaultwert vorbelegt werden, der benutzt wird, wenn bei der Instanziierung des Moduls kein anderer Wert angegeben wird.

```
entity vorteiler is
  generic
  (
    N: natural
  );
  port
  (
    tin:  in  std_logic;
    tout: out std_logic
  );
end;
```

```
architecture mch of vorteiler is
begin
  signal i: natural range 0 to N-1;
gekürzt wegen V2
end;
```

Listing 1: Verwendung von Parametern (generic) in einem Modul

In Listing 1 wird dem Modul nur ein Parameter namens N vom Typ *natural* übergeben. Ein Defaultwert ist hier nicht sinnvoll, er könnte aber wie folgt angegeben werden: *N: natural :=100*. Wie in der Liste *port* auch entfällt nach dem letzten Parameter der schließende Strichpunkt. In der architecture kann der Parameter dann beliebig verwendet werden. Man sieht hier, dass mit der *range*-Anweisung jeweils nur so viele Binärstellen für die Variable *i* angelegt werden, wie unbedingt nötig.

3.2 Parameterübergabe an ein Modul

Bei der Instanziierung eines Moduls werden die Parameter mit der Anweisung *generic map* zugewiesen (Listing 2). Damit sind für diese Schaltung nur noch zwei Module nötig (vorteiler und cnt4), zudem kann der universelle Vorteiler jetzt ohne Änderung in anderen Schaltungen verwendet werden.

```
I4: vorteiler generic map (N => 50000) port map(tin => osc, tout => t1000);
I3: vorteiler generic map (1000) port map(t1000, t1);
```

Listing 2: Zuweisung von Parametern bei der Instanziierung

Auch hier ist die Zuordnung nach Name (bei I4) oder Position (bei I3) möglich. Parameter, die in der entity mit einen Defaultwert versehen sind, brauchen bei der Instanziierung nicht angegeben werden - in dem Fall wird eben der Defaultwert verwendet.

Die Parameter müssen natürlich in der *component*-Deklaration ebenfalls als Liste *generic* erscheinen.

4 Verwendung von components

Auch wenn es seit VHDL93 möglich ist, eine entity direkt in einer anderen architecture zu verwenden, kann es gute Gründe für den „Umweg" über eine Komponente geben.

In der Praxis dürfte das Haupteinsatzgebiet bei der Verwendung von „black boxes" liegen. Eine "black box" ist ein Modul, für das nur die Schnittstelle (Außenansicht) vorhanden ist. Bei einem eigenen Design wäre das die *entity*. Es kann aber sein, dass man ein bestimmtes Modul von einem Drittanbieter kauft, der die Innenansicht nicht weitergeben will.

In dem Fall liefert er zu dem Modul eine *component* und ggf. noch weitere Dateien, z.B. für die Simulation. Die Innenansicht wird aber nicht als VHDL-Quelldatei sondern in einer „unleserlichen" Bibliothek geliefert. Nun kann man das Modul trotzdem in einem eigenen Entwurf verwenden und bei der Synthese wird dann auf die Bibliothek zurückgegriffen.

Die Beschreibung in der Bibliothek kann dabei auf verschiedenen Abstraktionsebenen vorliegen. Sie kann beispielsweise schon eine fertige Platzierung des gesamten Moduls für ein bestimmtes FPGA enthalten. Das spart beim Gesamtentwurf einiges an Rechenzeit (place & route) und kann zudem zu besseren Ergebnissen führen.

Liegt das Modul dagegen noch ohne solche Information vor, kann es freier platziert werden und so eventuell vorhandene Ressourcen besser nutzen.

Man spricht dann von „Hard Macros" und „Soft Macros". Das Hard Macro ist die bereits auf das FPGA fertig abgebildete Variante, das Soft Macro überlässt das place & route dem Anwender.

Module, die als Black Box vorliegen, können meist nicht mit Hilfe von *generic* / *generic map* parametrisiert werden, denn dazu müsste der Compiler den Quelltext in VHDL kennen.