

Diplomvorprüfung

Technische Grundlagen der Informatik

Prof. Dr. Arndt Bode

Wintersemester 2004/2005

15. Februar 2005

Anlage III

Rechnergestützter Schaltungsenwurf/VHDL

Georg Acher, Markus Leberecht

Inhaltsverzeichnis

1 Bereich III: Rechnergestützter Schaltungsentwurf	5
1.1 VHDL-Kurzanleitung	5
1.1.1 Vorbemerkung	5
1.1.1.1 Nomenklatur	5
1.1.2 Bausteine, Module und Bibliotheken	6
1.1.3 Datenobjekte, -typen und Modi	7
1.1.3.1 Datenobjekte	7
1.1.3.2 Datentypen	7
1.1.3.3 Modi bei Signalen	8
1.1.4 Nebenläufige Beschreibung des Aufbaus oder des Verhaltens von Architekturbeschreibungen (Architectures)	10
1.1.4.1 Strukturelle Beschreibung	10
1.1.4.2 Verhaltensbeschreibung	10
1.1.5 Operatoren	14
1.1.6 Attribute	15
1.2 Beispiele zur Benutzung der VHDL-Sprachmittel	16
1.2.1 Eine exemplarische Entity-Deklaration	16
1.2.2 Eine dazu passende Architekturdeklaration mit Prozeß	16
1.2.3 Nebenläufige Zuweisungen	17
1.2.4 Komponenten und ihre Instantiierung	17
1.2.5 Sequentielle Sprachkonstrukte und ihre Anwendung	19
1.2.5.1 Die einfache Abfrage	19
1.2.5.2 Die komplexe Abfrage	19
1.2.5.3 Die For-Schleife	19
1.2.5.4 Die While-Schleife	19
1.2.6 Verbindung von synchroner Logik und asynchronem Verhalten durch Prozesse	21
1.2.6.1 Grundlage: Ein 8-Bit-Register	21
1.2.6.2 8-Bit-Register mit synchronem Reset	21
1.2.6.3 8-Bit-Register mit asynchronem Reset	21
1.2.6.4 Asynchron rücksetzbares Register mit synchroner Ladeberechtigung (Enable)	22
1.2.7 Ein vollständiges Beispiel: Ladbarer Zähler mit Nulldurchlauferkennung	23

1 Bereich III: Rechnergestützter Schaltungsentwurf

1.1 VHDL-Kurzanleitung

1.1.1 Vorbemerkung

VHDL ist eine Abkürzung von Abkürzungen und steht für *VHSIC Hardware Description Language*, wobei VHSIC die Kurzform von *Very High Speed Integrated Circuit* ist. Man erkennt hieran schon, daß diese Sprache in der Hauptsache zur Beschreibung von Schaltungen dient, die im Bereich des Entwurfes integrierter Schaltkreise verwendet werden. Ihre vornehmliche Nutzung erfolgt in der

- Dokumentation bzw. Beschreibung von Schaltungen, in der
- Simulation von Schaltungen und als Grundlage zur
- automatischen Synthese von Schaltungen.

Dokumentation bedeutet in diesem Zusammenhang nicht eine willkürliche Textansammlung ähnlich einem Handbuch, sondern eine funktionale Beschreibung in einer syntaktisch und teilweise semantisch nachprüfbar Form.

Unter automatischer Synthese versteht man dabei in der Regel den Vorgang der maschinellen Realisation einer digitalen elektrischen Schaltung aus einer Beschreibung, die z. B. in VHDL vorliegt.

VHDL selbst ist eine "vollständige" Programmiersprache und bietet unter anderem auch I/O-Möglichkeiten, Zeiger, Funktionen, Prozeduren und einige Objekt-orientierte Konzepte (z.B. Überladung). Diese werden dann praktisch, wenn Beschreibungen auf höherem Abstraktionsebenen oder z.B. auch automatisierte Tests von beschriebenen Schaltungen durchgeführt werden müssen.

In dieser Kurzbeschreibung wird allerdings nur eine kleine Untermenge von VHDL erläutert, die auf die Bedürfnisse der Veranstaltung *Technische Grundlagen der Informatik* in Übung, Klausur und Praktikum zugeschnitten ist.

In Abschnitt 1.2 sind jeweils Beispiele aufgeführt, die die Benutzung der vorgestellten Sprachmittel verdeutlichen sollen.

1.1.1.1 Nomenklatur

GROSSGESCHRIEBENE Wörter Kennzeichnen Schlüsselwörter der Sprache VHDL. VHDL selbst unterscheidet allerdings weder bei Schlüsselwörtern noch bei Variablen- und Typennamen zwischen Groß- und Kleinschreibung.

`<bezeichner>` in spitzen Klammern deuten auf benutzerdefinierte Namen im VHDL-Quelltext hin.

`bezeichner` ohne spitze Klammern stehen für einzusetzende VHDL-Ausdrücke.

VHDL-Ausdrücke wie `[irgend_was]` in eckigen Klammern sind optional.

VHDL-Ausdrücke wie `{irgend_was}` stellen beliebige Wiederholungen des gleichen Ausdrucks dar.

VHDL-Ausdrücke wie `irgend_was | irgend_was_anderes` stellen eine Auswahlmöglichkeit zwischen den beiden Alternativen dar.

Ein Kommentar in VHDL ist stets einzeilig und beginnt mit zwei Bindestrichen `--`.

1.1.2 Bausteine, Module und Bibliotheken

Ein „Baustein“ in einer Schaltung kann als Black Box mit einer Schnittstelle (Interface, entspricht den Anschlussbeinen bzw. Pins bei realen Bauteilen) zu seiner Umwelt verstanden werden. VHDL bietet zur Beschreibung dieses Sachverhalts die beiden Sprachmittel `entity` und `architecture` an.

Deklaration einer Entity

```
ENTITY <entity_name> IS
    PORT (
        [signal] <identifier> {,<identifier>}:[mode] <type_mark>
        {:[signal] <identifier> {,<identifier>}:[mode] <type_mark>}
    );
END [<entity_name>];
```

Die Entity stellt das Interface eines Bausteins oder einer Schaltung nach außen dar. In der PORT-Deklaration werden alle Ein- und Ausgänge mit Namen und ihren Datentypen aufgeführt. Es ist übrigens zulässig, daß eine Entity keine PORT-Deklaration besitzt, dies ist für bestimmte Spezialfälle sinnvoll.

Deklaration einer Architecture

```
ARCHITECTURE <architecture_name> OF <entity_name> IS
    type_declaration
    | signal_declaration
    | constant_declaration
    | component_declaration
BEGIN
    {process_statement
    | concurrent_signal_assignment_statement
    | component_instantiation_statement}
END [<architecture_name>];
```

Eine Architecture stellt das zu einer Entity gehörige „Innenleben“ bzw. die Realisierung einer Schaltung dar. Aus diesem Grunde existiert zu einer Entity immer mindestens eine Architecture. Da es zu einer Entity durchaus mehrere Realisierungsvarianten (also mehrere Architecture-Definitionen) geben kann, wird der konkrete Realisierungsname mit `<architecture_name>` definiert. Dieser Name sollte aussagekräftig sein, hat aber ausser für die Unterscheidung und Variantenauswahl keine weitere Funktion.

Innerhalb des Architecture-Kopfes können Datentypen, Signale, Konstanten und Komponenten, also Kopien anderer Bausteine, deklariert werden.

Eine Architecture selber besteht aus Prozessen, nebenläufigen Signalzuweisungen und instantiierten („eingebauten“) Komponenten. Diese Komponenten sind nichts anderes als die Einbindung anderer Architekturen.

Deklaration einer Komponente

```
COMPONENT <component_name> IS
    PORT (
        [signal] <identifier> {,<identifier>}:[mode] <type_mark>
        {:[signal] <identifier> {,<identifier>}:[mode] <type_mark>}
    );
END COMPONENT;
```

Soll ein Baustein als Komponente innerhalb einer übergeordneten Architecture verwendet werden, so muß er zusätzlich als Component in der angegebenen Form deklariert werden. Möglich, aber hier nicht näher beschrieben, ist auch die Deklaration in Bibliotheken (sog. „Packages“).

1.1.3 Datenobjekte, -typen und Modi

1.1.3.1 Datenobjekte

VHDL beinhaltet drei unterschiedliche Arten von Datenobjekten, die wiederum unterschiedlichen Typs sein können und nur an bestimmten Stellen deklariert werden können.

Signals (Signale) sind die in der Praxis am häufigsten vorkommenden Datenobjekte in VHDL und verwirklichen den Datentransport und die Datenspeicherung innerhalb von Architectures und über deren Grenzen (Schnittstellen) hinaus. Eine Zuweisung an ein Signal sowie das Benutzen des Wertes eines Signales sind in das Zeitmodell von VHDL eingebunden: Alle nebenläufigen Zuweisungen (siehe dort) finden gleichzeitig statt, alle Signalzuweisungen innerhalb der sequentiellen Anweisungen eines Prozesses erfolgen erst durch das `END PROCESS`-Statement bzw. `WAIT`-Statement. Weiterhin besitzen Signale neben ihrem eigentlichen Wert auch noch zusätzliche Attribute, die verschiedene weitere statische und dynamische Eigenschaften beinhalten.

Constants (Konstanten) stellen statische Werte dar, ganz entsprechend herkömmlichen Programmiersprachen, und werden zur Verbesserung der Lesbarkeit des Quellcodes verwendet.

Variables VHDL-Beschreibungen können Anteile enthalten, die ihr Verhalten auf eine sequentielle Weise beschreiben (Prozesse und - nicht in dieser Kurzanleitung - Unterprogramme). Sollen innerhalb solcher Bereiche lokale Informationen zwischenzeitlich gespeichert werden, so kann man sich dazu der **Variables** (Variablen) bedienen. Variablen können aber im Gegensatz zu Signalen keine Information außerhalb ihrer Blöcke transportieren. Variablen werden auch nicht wie Signale erst zu gewissen Zeitpunkten zugewiesen, sondern erhalten ihren Wert innerhalb der sequentiellen Reihenfolge der Abarbeitung von Prozessen.

Wichtig: Variablen sollten nur benutzt werden, wenn man **genau** weiß, was man erreichen will. Für fast alle TGI-Zwecke sind Signale völlig ausreichend!

1.1.3.2 Datentypen

VHDL bietet zahlreiche Möglichkeiten zur flexiblen Benutzung und zur Definition neuer Datentypen. Der Inhalt dieser Kurzbeschreibung soll aber lediglich eine Reihe von gebräuchlichen Typen enthalten.

Typ Integer wird in der Regel im Rahmen von Konstanten verwendet und stellt den üblichen Ganzzahltyp dar. Normalerweise werden Zahlen im Zehnersystem erwartet, allerdings kann auch eine Basis angegeben werden. Beispielsweise ist `16#ab#` hexadezimal und entspricht dem Wert 171).

Typ Boolean beinhaltet die beiden Werte `TRUE` und `FALSE` und wird zur Darstellung von Wahrheitswerten genutzt.

Der Aufzählungstyp ist benutzerdefiniert und kann beispielsweise zur einfachen Definition von Zuständen eines Automaten verwendet werden.

```
TYPE <enumeration_type_name> IS
  ( <value_name> {, <value_name>} );
```

Typ Standard-Logik

$$\text{STD_LOGIC} \in \{ 'U', 'X', '0', '1', 'Z' \}$$

Im einzelnen stehen die Werte¹ für folgende Bedeutungen:

- 'U' steht für einen noch uninitialisierten bzw. unbekanntem Datenwert.
- 'X' steht für einen Datenwert, der durch eine noch unvollständige oder fehlerhafte Berechnung bzw. Signalzusammenschaltung entstanden ist.
- '0' stellt die logische '0' dar.
- '1' stellt die logische '1' dar.
- 'Z' heißt *hochohmig* (oder auch *Tristate* bzw. *Three-state*) und bezeichnet einen Wert, der von den allen anderen stets überschrieben werden kann.

'U' und 'X' werden selten direkt verwendet. Tauchen sie als Signalwert auf, weisen sie meist auf Probleme in der Beschreibung hin. 'U' und 'X' existieren nur in der Simulation, nicht im physikalischen Bauteil. Genau deshalb ist ein Auftreten dieser Werte ein sicheres Zeichen dafür, daß die Schaltung in der Realität eher nicht wie gewünscht funktionieren wird.

'Z' wird u.a. benutzt, um mehrere Ausgänge abwechselnd auf ein Signal zu schalten, ohne 'X' zu erhalten. Typisches Beispiel hierzu sind bidirektionale Datenbusse.

STD_LOGIC und die nachfolgend beschriebenen Ableitungen sind die für TGI wichtigste Datentypen.

Typ Standard-Logik-Vektor und Subtypen signed/unsigned

```
STD_LOGIC_VECTOR (<n1> TO | DOWNTO <n2>)
```

Der `std_logic_vector` mit den Indizes von $n1$ nach $n2$ (Schlüsselwort `TO` nur bei $n1 < n2$, sonst `DOWNTO`!) ist eine Zusammenfassung von einzelnen `std_logic`-Elementen und kann so beispielsweise für Busse verwendet werden.

Konstante Vektoren werden mit doppelten Anführungszeichen angegeben (z.B. "100101"). Hexadezimale Werte mit vielfachen von 4 Binärstellen können mit einem vorangestellten `x` beschrieben werden (z.B. `x"ab"`).

Wichtig: `STD_LOGIC_VECTOR` ist an sich eine einfache Sequenz von Bits ohne jegliche Interpretation, damit sind zunächst außer booleschen Operationen (AND, OR, NOT etc.) **keine** weiteren arithmetischen Operationen und Vergleiche (+, -, > usw.) möglich.

Werden diese Operationen benötigt, muß ein Datentyp benutzt werden, der eine Interpretation der Bits besitzt. Dazu sind in der Bibliothek `IEEE.numeric_std.all` die beiden Subtypen `UNSIGNED` und `SIGNED` definiert. Auf diesen sind dann auch die üblichen arithmetischen Funktionen definiert. Zusätzlich wird durch eine Funktionsüberladung eine einfache Anwendung mit dem Integertyp ermöglicht (z.B. bei `a+1` wobei `a` vom Typ `UNSIGNED` und die `1` vom Typ `Integer` ist).

1.1.3.3 Modi bei Signalen

Signale, die zum Datenaustausch zwischen einzelnen Komponenten einer VHDL-Beschreibung dienen, müssen mit einer Datenrichtung (dem Modus) ausgestattet werden, welche in der Signaldefinition in der Entity vor den Typen geschrieben wird. Ein Modus muß einen der folgenden Werte besitzen:

`IN` wird ausschließlich als Eingangssignal verwendet. An diese Signale ist innerhalb der Entity, für den sie definiert wurden, keine Zuweisung möglich.

¹Der eigentliche Standard-Logik-Typ in VHDL hat sogar neun logische Werte und ist ein kein eingebauter, sondern nur einer im Rahmen einer Standardbibliothek eingebauter Datentyp. Für die Zwecke von TGI jedoch soll die obige Form ausreichen.

OUT wird ausschließlich als Ausgangssignal verwendet. Von diesen Signalen kann innerhalb der Entity, für den sie definiert wurden, **nicht** gelesen werden.

INOUT wird verwendet, wenn das Signal bidirektional benutzt wird: Zuweisungen sowohl als auch Auslesen seiner Werte können sowohl innerhalb als auch außerhalb der definierenden Entity erfolgen.

Wichtig: Modi können nur für die Signale in der Entity benutzt werden! Signale, die im Architekturkopf definiert werden, sind nur innerhalb dieser benutzbar, können dafür aber immer gelesen und beschrieben werden.

1.1.4 Nebenläufige Beschreibung des Aufbaus oder des Verhaltens von Architekturbeschreibungen (Architectures)

In Hardware finden in der Regel viele Vorgänge in Abhängigkeit voneinander, dennoch aber zur gleichen Zeit statt. Dieses Verhalten, welches man als *nebenläufig* bezeichnet, kann auch in VHDL beschrieben werden.

1.1.4.1 Strukturelle Beschreibung

Eine gängige Variante der Beschreibung einer Schaltung ist die Darstellung ihrer Struktur. Eine typische Strukturbeschreibung ist z.B. ein Schaltplan, der Anschlüsse von Funktionsblöcken über Linien (Netze, Signale) mit anderen Blöcken verbindet und nach aussen wiederum Anschlüsse bietet. Genau diese Struktur (sog. Netzliste) kann mit VHDL auch textuell nachgebildet werden.

Dabei wird ein übergeordneter Block (z. B. eine Architecture) aus untergeordneten kleineren Modulen zusammengesetzt.

Eine Form dieser Submodule sind die Komponenten, die durch ihre Instantiierung in einer Architecture vervielfältigt werden (**Instantiierung von Komponenten**):

```
<instantiation_label>:
  <component_name> PORT MAP (
    <signal_name> | OPEN {, <signal_name> | OPEN}
  );
```

Hierbei dient `<instantiation_label>` zur Unterscheidung mehrfach instantiierter Komponenten, während die Port-Map zur Zuordnung von übergeordneten Signalen zu den Ein- und Ausgängen der Komponente benutzt wird.

Wichtig: Die „Portparameter“ einer Komponenteninstantiierung dürfen im allgemeinen nur Signale sein! Konstanten oder direkt in der Instantiierung berechnete Signalwerte sind nicht erlaubt. Grund dieser vermeintlich unsinnigen Einschränkung gegenüber anderen Sprachen ist einerseits die mögliche Bidirektionalität der Parameter als auch eine potentielle Überladung der VHDL-Operatoren. Die Instantiierung sieht zwar einem Funktionsaufruf in anderen Sprachen ähnlich, ist aber wegen der kontinuierlichen Verarbeitung des Datenflusses an den Ports nicht vergleichbar.

1.1.4.2 Verhaltensbeschreibung

Neben der Beschreibung der Struktur ihres Aufbaus kann eine Schaltungsfunktion auch durch ihr Verhalten beschrieben werden. VHDL eröffnet hierfür mehrere Möglichkeiten.

Einfache Zuweisungen ermöglichen es, Signalen Werte zuzuordnen:

```
<signal_name> <= value { operator value };
```

wobei `value` aus Signal, Variable oder Konstante besteht und die üblichen Operatoren verwendet werden können. Da VHDL eine strenge Typprüfung hat, muss der Datentyp beider Seiten übereinstimmen.

Natürlich kann die rechte Seite auch aus komplizierten Ausdrücken bestehen, solange die Syntax des Ausdrucks und der resultierende Datentyp gültig sind.

Bedingte Zuweisungen:

```
<signal_name> <= value_true WHEN condition ELSE value_false;
```

An das Signal wird nur dann `value_true` zugewiesen, wenn die Bedingung `condition` (siehe Vergleichsoperatoren) `TRUE` oder `'1'` ergibt. Ansonsten erhält `<signal_value>` den Wert `value_false`.

Wichtig: Bedingte Zuweisungen sind nur als „Concurrent Statements“ benutzbar, nicht aber in Prozessen (s.u.). Sie sind allerdings kaskadierbar (z.B. `a when b else c when d else e`).

Prozesse: Der **Prozeß** in VHDL ist ein Block, dessen Verhalten über einen sequentiellen Algorithmus beschrieben wird. Bei dessen Ausführung muß allerdings stets bedacht werden, daß die simulierte Zeit für die ganze VHDL-Beschreibung während der sequentiellen Ausführung nicht fortschreitet.

```
[<process_label>:]
PROCESS (sensitivity_list)
    {type_declaration
    | constant_declaration
    | variable_declaration}
BEGIN
    {wait_statement
    | signal_assignment_statement
    | variable_assignment_statement
    | if_statement
    | case_statement
    | loop_statement}
END PROCESS [<process_label>;
```

Jeder Prozeß besitzt eine sogenannte Sensitivity List, eine Aufzählung von Signalen, bei deren Änderung der zwischen `BEGIN` und `END` eingeklammerte Block ausgeführt wird. Im Prozeßkopf können Typen, Konstanten und Variablen vereinbart werden, aber keine Signale.

Innerhalb des Prozesses stehen eine Reihe von Sprachmitteln zur Verfügung, die z. T. auch in anderen herkömmlichen Programmiersprachen existieren.

Der Ablauf der Kommandos erfolgt wie in anderen Sprachen sequentiell, allerdings werden alle Signal-Zuweisungen bis zum Ende des Prozesses zurückgehalten. Damit arbeiten alle Kommandos und die rechten Seiten von Zuweisungen mit den Signalwerten, wie sie beim Eintritt in den Prozeß vorgefunden wurden. Erst das `END PROCESS`-Statement führt die Zuweisungen aus und simuliert somit die scheinbar sofortige Ausführung.

Signalwerte, die in einem Prozeß zugewiesen werden, werden anderen Teilen einer VHDL-Beschreibung also erst nach Beendigung dieses Prozesses bekanntgegeben.

Wichtig: Werden in der sequentiellen Prozeßbeschreibung einem Signal unterschiedliche Werte zugewiesen, erhält das Signal am Prozeßende den letzten zugewiesenen Wert!

Neben der Signalzuweisung (die wieder alle Operatoren nutzen kann, nicht aber `when...else`), sind unter anderem folgende Kommandos erlaubt:

Variablenzuweisung

```
variablen_name := <expression>
```

Variablen, die nur Prozeß-intern deklariert sein können, werden mit `:=` zugewiesen. Im Gegensatz zu Signalen erfolgt die Zuweisung sofort, d.h. der nächste Befehl kann auf den geänderten Wert zurückgreifen.

Wartekommando

```
WAIT UNTIL condition;
WAIT FOR duration unit;
```

Hiermit wird der Ablauf der Kommandos im Prozeß angehalten, bis die angegebene Bedingung wahr oder die angegebene Zeit verstrichen ist. Dieses Kommando ist nur in Prozessen ohne Sensitivity-List nutzbar.

Wichtig: Da WAIT einige Komplikationen für die Realisierung einer Schaltung nach sich zieht bzw. deren Realisierung sogar unmöglich macht, ist das Kommando für TGI ohne umfangreiches VHDL-Wissen bzw. außer bei explizit erwähnte Ausnahmen nicht sinnvoll nutzbar und sollte vermieden werden.

If-Kommando

```
IF condition THEN
    sequence_of_statements
    {ELSIF condition THEN sequence_of_statements}
    [ELSE sequence_of_statements]
END IF;
```

Mit dem If-Kommando werden wie üblich Entscheidungen innerhalb eines Prozesses getroffen. Elself ist dabei eine Zusammenfassung von else und if und benötigt kein eigenes end if.

Vorsicht ist geboten bei der Benutzung von If ohne einen else-Zweig, sofern die Beschreibung zur Synthese eingesetzt wird: Eine fortgelassene Alternative impliziert die Benutzung von Speicherelementen bei einer Zuweisung, da nur so der vorherige Wert erhalten werden kann.

Case-When-Kommando

```
CASE expression IS
    {WHEN constant_value | OTHERS => sequence_of_statements}
END CASE;
```

Das Case-When-Kommando dient zur Auswahl aus mehreren Möglichkeiten. Zur Abdeckung aller Alternativen dient das Schlüsselwort OTHERS, welches die Aktion bei sämtlichen nicht explizit spezifizierten Möglichkeiten ausführt. OTHERS muß immer angegeben werden, wenn die anderen Fälle nicht alle Möglichkeiten abdecken.

For-Loop-Kommando

```
[<loop_label>:]
FOR <variable_name> IN <n1> TO | DOWNTO <n2> LOOP
    sequence_of_statements
END LOOP [<loop_label>];
```

Das For-Loop-Kommando implementiert eine Schleife, deren Laufvariable nacheinander die Integer-Werte von $n1$ bis $n2$ annimmt (Benutzung von TO und DOWNTO wieder wie bei den Vektorgrenzen) und für jeden Wert die nachfolgenden Kommandos ausführt. $n1$ und $n2$ können dabei Ausdrücke aus Variablen und Konstanten sein, nicht aber aus Signalen. Da die Schleife im Prozess-Kontext abläuft, werden Signalzuweisungen darin ebenfalls erst am Prozeßende ausgeführt, damit arbeitet die Schleife auch immer mit denselben Signalwerten.

While-Loop-Kommando

```
[<loop_label>:]
WHILE condition LOOP
    sequence_of_statements
END LOOP [<loop_label>];
```

Das While-Loop-Kommando implementiert eine Schleife, die so lange ausgeführt wird, wie die Bedingung condition wahr ist. condition selber kann wiederum ein Ausdruck aus Variablen, Konstanten und Signalen sein. Allerdings haben die verzögerten Signalzuweisungen im Prozeß (wie bei der FOR-Schleife) keine direkten Auswirkungen auf die Bedingung im nächsten Durchlauf.

Wichtig: Die `WHILE`-Schleife kann ähnlich wie `WAIT` sehr leicht zu Problemen in der schaltungstechnischen Realisierung (Synthese) führen. In TGI sollte sie nur benutzt werden, wenn sie explizit gefordert ist!

1.1.5 Operatoren

Operatoren in VHDL werden in Bedingungen, bei Werten und bei Ausdrücken verwendet. Wie jede andere Programmiersprache bietet VHDL hiervon die üblichen arithmetischen und logischen Funktionen an.

Logik:

AND Logisches UND (z.B. $a \text{ AND } b$)

OR Logisches ODER

NAND Logisches NAND

NOR Logisches NOR

XOR Logisches Exklusiv-ODER

XNOR Logisches Exklusiv-NOR

Vergleich:

= Gleichheit (z.B. $a = b$)

/= Ungleichheit

< Kleiner (für Integer, Signed, Unsigned)

<= Kleiner oder gleich (für Integer, Signed, Unsigned)

> Größer (für Integer, Signed, Unsigned)

>= Größer oder gleich (für Integer, Signed, Unsigned)

Schieben:

SLL Linksseitiges logisches Schieben (z.B. $\text{SLL}(a, 2)$)

SRL Rechtsseitiges logisches Schieben

SLA Linksseitiges arithmetisches Schieben

SRA Rechtsseitiges arithmetisches Schieben

ROL Linksseitiges Rotieren

ROR Rechtsseitiges Rotieren

Addition:

+ Addition (z.B. $a + b$)

- Subtraktion

Vorzeichen:

- + Positives Vorzeichen
- Negatives Vorzeichen (z.B. -1)

Multiplikation:

- * Multiplikation (für Integer)
- / Division (für Integer)
- MOD Modulo (für Integer) (z.B. a MOD b)

Verschiedene:

- ** Potenzierung (für Integer) (z.B. 2**a)
- ABS Absolutwert (für Integer)
- NOT logische Negierung (boolean oder Standard-Logik)

Jede Gruppe von Operatoren hat die gleiche Präzedenz. Die Gruppen selber sind nach aufsteigender Präzedenz geordnet.

Wichtig:

- Die logischen Operatoren arbeiten sowohl auf booleschen Werten als auch bitweise auf `std_logic`, `std_logic_vector`, `signed` und `unsigned`.
- Insbesondere AND und OR haben im Gegensatz zu anderen Sprachen gleiche Priorität! Gemischte AND/OR-Ausdrücke müssen daher passend geklammert werden, um Syntaxfehler durch Mehrdeutigkeiten („Ambiguity“) zu vermeiden.
- *, / und MOD können in der Synthese sehr aufwendig werden. Eine Ersetzung durch +, -, Schiebeoperationen bzw. geänderte Algorithmenbeschreibungen ist in vielen Fällen möglich und effizienter.

1.1.6 Attribute

Attribute machen im „großen“ VHDL zusätzliche Eigenschaftsangaben über Entity- oder Architecture-Blöcke, Signale und Typen. Für TGI beschränken wir uns jedoch lediglich auf die Nutzung eines bestimmten Attributes.

```
<signal_name>'EVENT
```

nimmt genau dann den Wert TRUE an, wenn sich der Wert des Signals <signal_name> ändert.

1.2 Beispiele zur Benutzung der VHDL-Sprachmittel

1.2.1 Eine exemplarische Entity-Deklaration

```
ENTITY register8 IS
  PORT (
    clk, rst, en:   IN std_logic;
    data:          IN std_logic_vector(7 DOWNTO 0);
    q:            OUT std_logic_vector(7 DOWNTO 0)
  );
END register8;
```

Zu beachten sind hier zwei Dinge:

- Das fehlende Semikolon hinter dem Vektor q: In VHDL ist die PORT-Liste eine mit Semikolon getrennte Aufzählung von Signalen, so daß am Ende das Semikolon auf jeden Fall fehlt.
- Die Benutzung von DOWNTO anstelle von TO führt zu einer natürlicheren Benutzung von Bitvektoren. Ein konstanter Vektor kann somit von links nach rechts mit der herkömmlichen Bedeutung interpretiert werden: MSB kommt als erstes. "10011011" wird daher als hexadezimal 9B (=155₁₀) anstatt als D9 (=217₁₀) aufgefaßt.

1.2.2 Eine dazu passende Architekturdeklaration mit Prozeß

```
ARCHITECTURE archregister8 OF register8 IS
BEGIN
  PROCESS (rst, clk)
  BEGIN
    IF rst='1' THEN
      q <= "00000000"; -- alternativ in Hex x"00"
    ELSEIF clk'EVENT AND clk='1' THEN
      IF en='1' THEN
        q <= data;
      ELSE -- kann wegfallen, da ohne Zuweisung
        q <= q; -- Wert implizit gespeichert wird
      END IF;
    END IF;
  END PROCESS;
END archregister8;
```

Zu beachten sind hierbei folgende Dinge:

- Die Architecture kann eine von mehreren sein, die zu einer Entity gehören. Man kann somit unterschiedliche Implementationen eines Moduls oder unterschiedliche Abstraktionen desselben (z. B. eine Struktur- und eine Verhaltensbeschreibung) im Vergleich leicht austesten.
- Die Sensitivity List des Prozesses gibt an, welche Signaländerungen eine Aktivierung des Prozesses auslösen. Zusammen mit der geschachtelten IF-Anweisung wird hiermit folgendes Verhalten implementiert:
 - Ein asynchroner Reset: Sobald das Signal `rst` logisch 1 wird, egal zu welchem Zeitpunkt, so wird `q` immer auf den Wert "00000000" (also 0) gesetzt.
 - Ein synchroner Wertewechsel mit steigender Taktflanke: Wenn `clk` seinen Wert ändert und kein Reset angelegt wird, so wird die IF-Abfrage `ELSEIF clk'EVENT AND clk='1' THEN` ausgeführt, die nur bei einer erfolgten Wertveränderung *und* aktuellem logischem 1-Wert (also einer steigenden Flanke) wahr wird.
- Einzelwert und Vektoren: Man beachte die Schreibweise für ein einzelnes Bits ('0') und für einen Vektor ("00000000").

- Innerhalb des durch `clk`-Aktivität aktivierten `ELSEIF`-Blocks sind alle weiteren Abfragen (hier ja nur eine, die nach dem Signal `en`) vollständig ausdekodiert. Hier bewirkt die explizite Nennung von `q <= q`, daß in der Synthese ein Speicherelement, ein Flipflop beispielsweise, eingefügt wird. Eine andere recht häufige aber nicht so gut lesbare Methode mit dem gleichen Effekt hätte darin bestanden, den `ELSE`-Zweig einfach vollständig wegzulassen. Hierbei hätte die Semantik von VHDL zur Folge gehabt, daß ein Speicherelement eingefügt werden müßte, um den Wert von `q` über eine einzige Aktivierung des Prozesses hinaus zu erhalten.

1.2.3 Nebenläufige Zuweisungen

Beispiele für einfache nebenläufige Zuweisungen, das heißt also Zuweisungen von Signalen, die gleichzeitig stattfinden, wären:

```
v <= a AND NOT b;
w <= a OR (b AND c);
y <= a NAND b XOR c;
```

Zu beachten ist hierbei, daß alle booleschen Operatoren die *gleiche* Präzedenz besitzen und daher bei mehr als einem Operator eine explizite Klammerung notwendig wird.

Die bedingte Zuweisung ausserhalb eines Prozesses sieht dann entsprechend aus:

```
a <= '1' WHEN b = c ELSE '0';
```

Nur im Falle von Gleichheit von `b` und `c` wird hierbei `a` auf logisch 1 gesetzt. Diese Form einer Zuweisung ist besonders dann bequem und effizient, wenn nur ein ganz bestimmter Fall aus einer großen Menge von Möglichkeiten für eine bestimmte Zuweisung sorgt².

1.2.4 Komponenten und ihre Instanziierung

Als Beispiel für eine Deklaration einer Komponente soll mal wieder das 8-Bit-breite Registermodul dienen:

```
COMPONENT register8 IS
  PORT (
    clk, rst, en:   IN std_logic;
    data:          IN std_logic_vector(7 DOWNTO 0);
    q:             OUT std_logic_vector(7 DOWNTO 0)
  );
END COMPONENT;
```

Steht diese Deklaration zusätzlich zu einem Entity-Architecture-Paar im Quelltext, so heißt dies, das von nun an die Komponente `register8` wie ein Bauteil verwendet werden kann, und zwar beispielsweise so:

```
ARCHITECTURE use_of_register8 OF example_reg8 IS
  COMPONENT register8 IS PORT (      -- Deklaration des Interface
    clk, rst, en:   IN std_logic;
    data:          IN std_logic_vector(7 DOWNTO 0);
    q:             OUT std_logic_vector(7 DOWNTO 0)
  );
  END COMPONENT;
  SIGNAL clock, reset, enable:      STD_LOGIC;
  SIGNAL data_in, data_out:         STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
  -- irgendwelcher anderer Code (Prozesse, Zuweisungen, etc.) hier
```

²Als Alternative kann man ja mal vergleichen, wieviele Zeilen VHDL man braucht, um das gleiche Verhalten mithilfe eines Prozesses zu beschreiben.

```
First_reg8: register8 PORT MAP (  
    clock, reset, enable, data_in, data_out );  
  
-- weiterer Code hier  
  
END use_of_register8;
```

Zu beachten ist hier:

- Der Name `First_reg8` ist der Name dieser einen Instantiierung. Eine oder mehrere weitere Instanzen dieser Komponente können selbstverständlich mit anderem Namen noch hinzugefügt werden.
- Die Architecture-internen Signale (z. B. `clock` oder `reset`) stehen in der Port-Liste der instantiierten Komponente an der gleichen Stelle wie ihre entsprechenden Ein- und Ausgänge in der Komponentendeklaration. Man nennt dies *Positional Association*, welche im Rahmen von TGI ausschließlich genutzt wird³.

³VHDL kann mehr, z. B. die sogenannte *Named Association*, die aber lediglich eine Lesehilfe ist.

1.2.5 Sequentielle Sprachkonstrukte und ihre Anwendung

Die sequentiellen Sprachkonstrukte werden ausschließlich innerhalb von Prozessen angewandt.

1.2.5.1 Die einfache Abfrage

Mithilfe von IF-THEN-ELSEIF-ELSE kann eine komplexe Abfrage von Bedingungen in VHDL realisiert werden:

```
-- geht nur innerhalb von process()!
IF (count = "00") THEN
  a <= b;
ELSIF (count = "10") THEN
  a <= c;
ELSE
  a <= d;
END IF;
```

1.2.5.2 Die komplexe Abfrage

Das CASE-Statement dient zur effizienten Auswahl aus einer ganzen Reihe von Möglichkeiten. Ein Ausdruck wird dazu mit einem Angebot von Konstanten verglichen⁴.

```
CASE count IS
  WHEN "00" =>
    a <= b;
  WHEN "10" =>
    a <= c;
  WHEN OTHERS =>
    a <= d;
END CASE;
```

Hierbei ist zu beachten, daß das Schlüsselwort OTHERS alle Möglichkeiten für nicht abgedeckte Fälle der Auswahl selektiert.

1.2.5.3 Die For-Schleife

Für einen diskreten Bereich von Werten kann ein Schleifenblock ausgeführt werden:

```
meine_for_schleife:
FOR i IN 3 DOWNT0 0 LOOP
  IF reset(i) = '1' THEN
    data_out(i) <= '0';
  END IF;
END LOOP meine_for_schleife;
```

Der Bezeichner *i* repräsentiert hierbei eine Variable und kein Signal. Die Variable hat implizit immer den Typ integer.

1.2.5.4 Die While-Schleife

```
meine_while_schleife:
WHILE (count > 0) LOOP
  count := count - 1;
  result <= result + data_in;
END LOOP meine_while_schleife;
```

⁴Im vollständigen VHDL können auch Ausdrücke anstelle von Konstanten verwendet werden, was die Auswahl noch flexibler macht.

Zu beachten ist hier der Unterschied zwischen `count` (einer Variablen, erkennbar an der Zuweisung `:=`) und `result` (einem Signal, auch erkennbar an der Zuweisung `<=`).

1.2.6 Verbindung von synchroner Logik und asynchronem Verhalten durch Prozesse

Unter synchroner Logik versteht man Zustandsautomaten, deren Zustandsänderung jeweils nur *synchron* mit einem Taktsignal stattfindet. Von Zeit zu Zeit ist es jedoch auch nötig, unabhängig vom zeitlichen Verlauf des Taktsignals, also *asynchron* zu ihm, eine Zustandsänderung des Automaten hervorzurufen. Dazu gibt es bestimmte, oft verwendete Konstrukte als Ausgangspunkt.

1.2.6.1 Grundlage: Ein 8-Bit-Register

Zugrundegelegt sei hier ein 8 Bit breites Register, welches zum Zeitpunkt der steigenden Flanke des Taktsignales `clk` geladen wird.

```
reg8_no_reset:
PROCESS (clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        q <= data;
    END IF;
END PROCESS reg8_no_reset;
```

Dabei seien `clk` vom Typ `STD_LOGIC` sowie `q` und `data` vom Typ `STD_LOGIC_VECTOR (7 DOWNTO 0)`. Der Prozeß wird innerhalb einer Architecture verwendet. Vor der ersten steigenden Taktflanke ist `q` undefiniert.

Alternativ zu `clk'EVENT AND clk = '1'` kann auch die vordefinierte Funktion `rising_edge(clk)` benutzt werden.

1.2.6.2 8-Bit-Register mit synchronem Reset

Benötigt man eine Rücksetzfunktion lediglich synchron mit dem Taktsignal, so kann folgender Ansatz verwendet werden.

```
reg8_sync_reset:
PROCESS (clk)
BEGIN
    IF clk'EVENT AND clk = '1' THEN
        IF sync_reset = '1' THEN
            q <= "00000000";
        ELSE
            q <= data;
        END IF;
    END IF;
END PROCESS;
```

Das zusätzliche Signal `sync_reset` sei erneut vom Typ `STD_LOGIC`.

1.2.6.3 8-Bit-Register mit asynchronem Reset

Bei einer asynchronen Funktion muß das auslösende Signal mit in die Sensitivity List des Prozesses aufgenommen werden.

```
reg8_async_reset:
PROCESS (clk, async_reset)
BEGIN
    IF async_reset = '1' THEN
        q <= "00000000";
    ELSIF clk'EVENT AND clk = '1' THEN
```

```

        q <= data;
    END IF;
END PROCESS;

```

Der Prozeß wird nun in VHDL auch „ausgeführt“, sobald sich eine Signaländerung am Signal `async_reset` (Typ `STD_LOGIC`) ergibt. Die Präzedenz der durch die Signale `clk` und `async_reset` ausgelösten Aktionen, also ihre Vorrangigkeit, kann bestimmt werden durch die Position des entsprechenden Signales in einer Reihe von `IF ... ELSIF ... END IF` Abfragen bestimmt. Der bemerkenswerte Unterschied zwischen beiden Signalen im Beispiel ist das Fehlen der Abfrage des `EVENT`-Attributes beim Signal `async_reset`. Dies bedeutet, daß die Resetbedingung auch dann als erstes gilt, wenn sich `async_reset` nicht selber geändert hat⁵. Selbst wenn also der Takt ordnungsgemäß weiterläuft, so kann bei gesetztem Resetsignal kein Wert aus `data` in das Register geladen werden.

1.2.6.4 Asynchron rücksetzbares Register mit synchroner Ladeberechtigung (Enable)

Synchrone und asynchrone Funktionen lassen sich auch kombinieren.

```

reg8_sync_assign_async_reset:
PROCESS (reset, clk)
BEGIN
    IF reset = '1' THEN
        q <= "00000000";
    ELSIF clk'EVENT AND clk = '1' THEN
        IF enable = '1' THEN
            q <= data;
        ELSE
            q <= q;
        END IF;
    END IF;
END PROCESS;

```

Abhängig von `clk`, also synchron zum Takt, wird hier die Abfrage von `enable` (Typ `STD_LOGIC`) eingesetzt, um so ein Laden des Zählers lediglich bei aktivierter Ladeberechtigung zuzulassen. Wie schon in 1.2.2 beschrieben, sorgt die explizite Angabe von `q <= q;` dafür, daß ein Synthesewerkzeug hierbei notwendigerweise Speicherelemente (Flipflops) einbaut, um diese Bedingung zu erfüllen. Alternativ dazu ist es auch möglich, den `ELSE`-Zweig dieser Abfrage fortzulassen. Die Semantik von VHDL, die besagt, daß jede Signalinformation eines Prozesses, die nicht innerhalb des Prozesses verändert wird, über mehrere Aufrufe hinweg erhalten bleibt, sorgt implizit dafür, daß auch in einem solchen Fall Speicherelemente benutzt würden. Man spricht im letzteren Fall von *Inferred Logic*.

⁵Aus digitaltechnischer Sicht bedeutet dies, daß der Reset ein pegelgesteuerter Eingang ist, während der Takt ein flankengesteuertes Signal ist.

1.2.7 Ein vollständiges Beispiel: Ladbarer Zähler mit Nulldurchlauferkennung

Der folgende 8-Bit Zähler, der zurückgesetzt und geladen werden kann sowie aufwärts und abwärts zählen kann ist ein gutes Beispiel zum Üben von VHDL. Der Code ist mit den Bibliotheksanweisungen vollständig nutzbar.

```

library IEEE;
use IEEE.std_logic_1164.all; -- für STD_LOGIC
use IEEE.numeric_std.all;    -- für UNSIGNED

ENTITY up_down_count IS
  PORT (
    in_data:    IN UNSIGNED(7 DOWNT0 0);
    value:      OUT UNSIGNED(7 DOWNT0 0);
    zero:       OUT STD_LOGIC;
    clk:        IN STD_LOGIC;
    reset:      IN STD_LOGIC;
    load:       IN STD_LOGIC;    -- '0' = count, '1' = load
    up_down:    IN STD_LOGIC    -- '0' = up, '1' = down
  );
END up_down_count;

ARCHITECTURE arch_ud_count OF up_down_count IS
  SIGNAL value_intern: UNSIGNED(7 DOWNT0 0);
BEGIN

  PROCESS (clk,reset)
  BEGIN
    IF reset = '1' THEN
      value_intern <= "00000000";
    ELSIF clk'EVENT AND clk = '1' THEN
      IF load = '1' THEN
        value_intern <= in_data;
      ELSIF up_down = '0' THEN
        value_intern <= value_intern + 1;
      ELSE
        value_intern <= value_intern - 1;
      END IF;
    END IF;
  END PROCESS;

  zero <= '1' WHEN value_intern = "00000000" ELSE '0';
  value <= value_intern;

END arch_ud_count;

-- Für die Instanziierung in einer anderen Architecture wird folgendes gebraucht:
COMPONENT up_down_count IS
  PORT (
    in_data:    IN UNSIGNED(7 DOWNT0 0);
    value:      OUT UNSIGNED(7 DOWNT0 0);
    zero:       OUT STD_LOGIC;
    clk:        IN STD_LOGIC;
    reset:      IN STD_LOGIC;
    load:       IN STD_LOGIC;    -- '0' = count, '1' = load
    up_down:    IN STD_LOGIC    -- '0' = up, '1' = down
  );
END COMPONENT;

```

Man beachte:

- Die Bibliotheken, die STD_LOGIC und UNSIGNED überhaupt erst benutzbar machen.
- Die Nutzung eines nur in der Architektur deklarierten Signals, da der als OUT definierte Ausgangsport intern nicht gelesen werden kann.
- Die implizite erfolgte Überladung von '+' bzw. '-' (UNSIGNED + INTEGER → UNSIGNED)
- Die Benutzung von Concurrent Statements zur Ausgabe von value_intern und der Berechnung von zero.