Simulation/Testbench

1 Einführung

Die hier behandelte Aufgabe ist es, eine Schaltung (Hardware) zu einer gegebenen Anforderung zu entwickeln. Die folgenden Angaben beziehen sich auf einen bestimmten Stand der Technik. Dieser Stand ändert sich natürlich über die Jahre. Was vor 20 Jahren noch gar nicht möglich war, kann heute üblich sein. Entsprechend wird man in 20 Jahren vermutlich einen Schaltungsentwurf deutlich abstrakter bzw. mit weniger Einschränkungen formulieren können als heute. Wenn es also heißt „ist nicht möglich“, dann ist das in der Regel als „heute, 2018, noch nicht möglich“ zu verstehen. Wesentliche Begriffe sind im Glossar erklärt.

1.1 Ablauf eines Entwurfs

Abbildung 1 zeigt den üblichen Ablauf einer Schaltungsentwicklung. Zunächst wird, ausgehend von der Anforderung, eine Schaltung auf der sogenannten Register-Transfer-Ebene (RTL-Entwurf) entwickelt. Das ist eine Darstellung, die einerseits noch Angaben über die spätere Hardware enthält, aber andererseits mit aktuellen Werkzeugen in eine Schaltung umgewandelt werden kann (Schaltungssynthese). Man schränkt sich dabei bewusst in der jeweiligen Hardwarebeschreibungssprache auf solche Konstrukte (Sprachelemente und deren Verwendung) so ein, dass eine Synthese möglich ist.

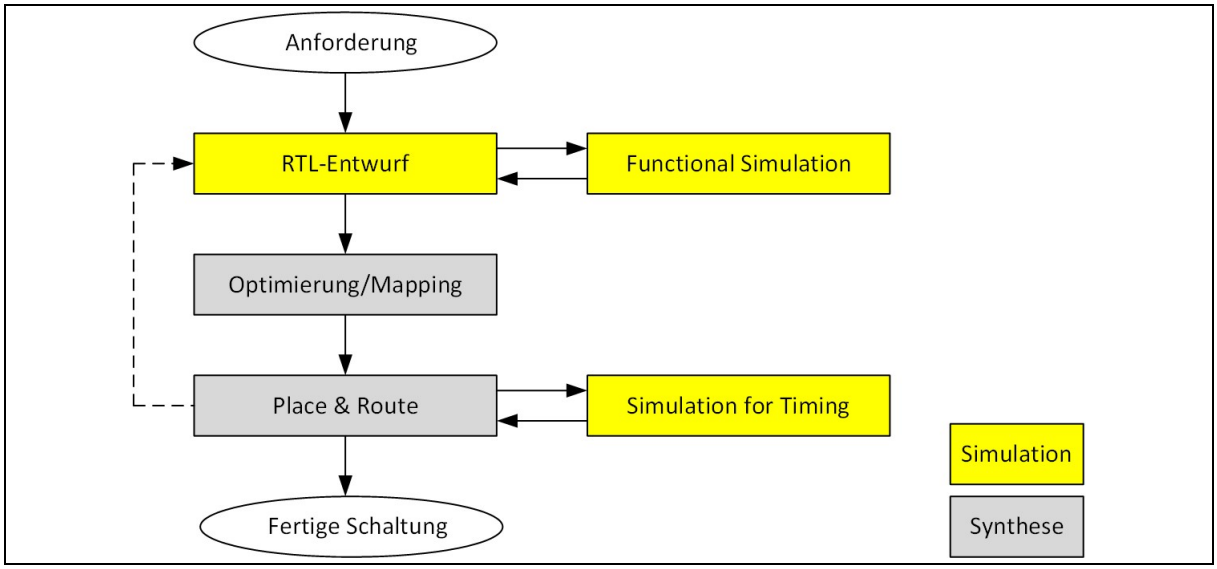


Abbildung 1: Entwurfsvorgang

Hat man den ersten Entwurf auf diese Ebene durchgeführt, folgt eine erste Simulation.

1.2 Simulationsumgebung

Für die Simulation benötigt man ein Modell des zu simulierenden Objekts, Anregungen (Stimuli) und eine Möglichkeit, die Reaktionen des Objekts zu beobachten. Da VHDL von Beginn an für die Schaltungssimulation gedacht war, sind in der Sprache selbst alle für die Simulation benötigten Elemente vorhanden.
In VHDL wird der beschriebene Aufbau für die Simulation als *Testbench* bezeichnet (Abbildung 2).

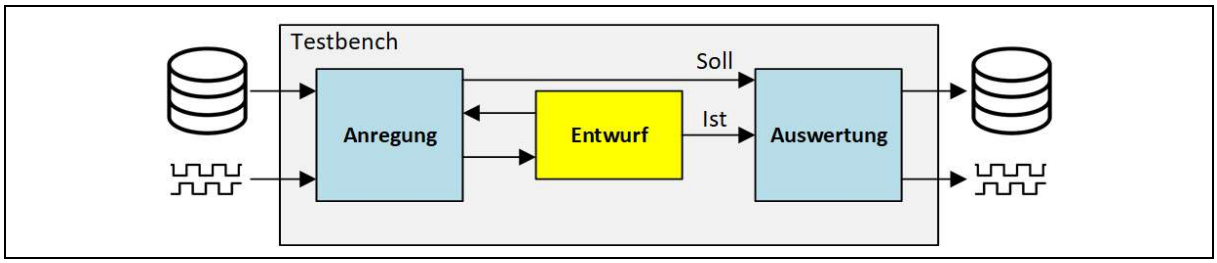


Abbildung 2: Testbench

Anregung, Entwurf und Auswertung können allen mit Sprachmitteln von VHDL durchgeführt werden. Das bezieht das Einlesen von Anregungen (Testvektoren) aus Dateien und die Auswertung (Soll-/Ist-Vergleich, Ausgabe von Ergebnissen in Dateien) mit ein. Dazu können natürlich noch anderer Methoden der Anregung bzw. der Beobachtung von Signalen je nach Werkzeughersteller kommen.

2 Testbench

Die Testbench (TB) besteht dabei wie jede Entwurfseinheit in VHDL aus einer *entity* und einer *architecture*. Die port-Liste der *entity* ist leer. Es gibt keine Signale, die in die TB führen oder aus ihr kommen.
In der *architecture* wird der Entwurf, der getestet werden soll, instanziiert. Das geschieht genau so wie einer beliebigen anderen Instanziierung auch (strukturelle Beschreibung). Der zu testende Entwurf wird als Device Under Test (DUT) bezeichnet. Seine Signale werden mit Signalen verbunden, die, ebenfalls wie üblich, in der *architecture* lokal deklariert worden sind.
Die *architecture* enthält dann aber zusätzlich Anweisungen, in der Regel Prozesse, die den lokalen Signalen Werte zuweisen. Dazu werden Anweisungen benutzt (z.B. *wait for* <time>), die nicht synthesefähig sind, es aber auch nicht sein müssen. So können (endlose) Takte oder beliebige zeitliche Signalverläufe erzeugt werden.
Ebenso können mit Anweisungen wie *assert* Sollwerte mit Istwerten verglichen werden, Auffälligkeiten mittels *report* ausgegeben werden und mit *severity* auch Aktionen wie den Abbruch der Simulation ausgelöst werden.
Die Simulation mittels der Testbench ist zwar textuell aufwendig(er), dafür aber zwischen allen Werkzeugen, die den VHDL-Standard umsetzen, portabel.

2.1 Sprachmittel

Für die Simulation sind in VHDL einige Sprachmittel vorhanden, die für den Schaltungsentwurf entweder nicht benötigt werden (z.B. Ein-/Ausgabe über Dateien) oder die nicht in eine Schaltung umsetzbar sind (z.B. Abwarten einer vorgegebenen Zeit). Einige dieser Sprachmittel werden im Folgenden vorgestellt, da sie in einer Testbench typischerweise verwendet werden. Falls Zeiten vorkommen, dann beziehen sie sicher immer auf die Simulationszeit, d.h. die im Simulator verwendete Zeit. Die tatsächlich in einer Schaltung auftretenden Zeiten können damit nicht vorgegeben werden.

2.2 wait

Diese Anweisung kann nur in einer sequentiellen Umgebung (*process*, *procedure*) verwendet werden. Sie dient dort dazu, den Ablauf anzuhalten, bis eine Bedingung erfüllt ist. Die möglichen Formen sind in Abbildung 3 gezeigt. Wird die wait-Anweisung in einem *process* verwendet, dann darf der *process* keine Empfindlichkeitsliste haben.

Syntax	**Bedeutung**	**Beispiel**
wait;	Bedingungsloses Warten (Halt)	`wait;`
wait for <Zeitspanne>;	Abwarten einer Zeitspanne	`wait for 10 ns;`

wait until <Bedingung>;	Warten, bis eine Bedingung wahr wird	`wait until clk = '1';`
wait on <Signalliste>;	Warten auf eine beliebige Änderung an einem oder mehreren Signalen	`wait on a, b;`

Abbildung 3: Wait-Statement

Diese Anweisung ist sehr nützlich, um in der Testbench Anregungen zu erzeugen.

```
-- Taktgenerator 50 MHz
process
        variable x: std_logic := '0';
begin
        clock <= x;
        wait for 10 ns;
        x := not x;
end process;
```

```
-- Einmaliger Reset für 25 ns
process
begin
        reset <= '1';
        wait for 25 ns;
        reset <= '0';
        wait;
end process;
```

Abbildung 4: Beispiele für die wait-Anweisung

In Abbildung 4 links wird ein endloser Takt mit 50 MHz und einem Tastverhältnis von 1:1 erzeugt. Weder die Initialisierung der Variable x noch die Zeitangabe sind im Allgemeinen synthesefähig. Für den Simulator ist beides kein Problem. Der Takt wird mit dem Wert 0 beginnen und anschließend bis zum Ende der Simulation alle 10ns seinen Wert wechseln.
In Abbildung 4 rechts wird ein einmaliger Reset erzeugt. Das Signal reset wird zu Beginn der Simulation für 25ns den Wert 1 haben und danach bis zum Ende der Simulation den Wert 0.

2.3 waveform

Eine Anweisung mit dem Schlüsselwort waveform gibt es in VHDL zwar nicht, aber der Begriff waveform kommt in der Sprachreferenz öfter vor. Er bezeichnet dabei eine Zuweisung an, der Informationen über den zeitlichen Verlauf mitgegeben werden.

Syntax	**Bemerkungen**
`ziel <= [modell] waveform;`	Die Angabe des Modells ist optional.

`modell`
`  **transport**`
`  **reject** zeit`
`  **inertial**`
	Es gibt zwei Modelle in VHDL (*inertial* und *transport*). Für die funktionale Simulation ist das Modell in der Regel unwichtig und wird weggelassen. Ohne Angabe wird *inertial* verwendet.

`waveform`
`  element [, element]`
`  **unaffected**`
	Eine Zuweisung kann aus mehreren zeitlich gestaffelten Elementen bestehen.

`element`
`  wert [**after** zeit]`
`  **null** [**after** zeit]`
	Jedes Element besteht aus einem Wert, der dem Ziel nach der angegebenen Zeit zugewiesen wird.

Abbildung 5: Syntaktischer Aufbau einer Zuweisung mit Zeitverlauf (waveform)

Der Aufbau einer Zuweisung mit Zeitangaben ist in Abbildung 5 gezeigt. Die Angaben in eckigen Klammern sind optional. Mit *wert* ist ein beliebiger Ausdruck gemeint, der einen Wert für das Ziel erzeugt. Das kann eine Konstante, aber auch eine Formel sein.
Die Zeitangaben nach *after* beziehen sich alle auf denselben Startzeitpunkt, sie bauen also nicht aufeinander auf. Die besonderen Werte *unaffected* und *null* können mit „keine Veränderung“ übersetzt werden, sind aber nicht immer zulässig und werden in der Regel auch nicht benötigt. Die unterschiedlichen Modelle sind für die „Simulation for Timing“ interessant und werden dann von den Werkzeugen selbst nach Bedarf eingesetzt und mit den jeweiligen Zeitangaben versehen. Für die Funktionssimulation braucht man darüber noch nichts zu wissen.

Zulässig	Unzulässig
`reset <= '1' after 0 ns, '0' after 25 ns;`	`reset <= '1' after 0 ns;` `reset <= '0' after 25 ns;`

Abbildung 6: Zulässige und unzulässige Verwendung einer waveform

Abbildung 6 zeigt links eine zulässige Zuweisung mit Zeitangaben. Der Zeitverlauf entspricht genau dem Beispiel aus Abbildung 4, rechts. Diese Zuweisung braucht nicht in einem *process* zu stehen. Die Aufteilung in zwei Anweisungen rechts ist unzulässig. Die beiden Zuweisungen werden gleichzeitig ausgeführt, das führt aber zu einer Kollision.

2.4 for … loop, while … loop

VHDL kennt sowohl die *for*- als auch die *while*-Schleife. Man kann diese Anweisungen gut benutzen, um wiederholte Anregungen zu erzeugen. Bei der *for*-Schleife wird immer eine lokale Integervariable deklariert. Der Wertebereich der Variable muss konstant sein (d.h. zum Zeitpunkt der Kompilation bekannt). Schleifen können mit einem Label (Bezeichnung) versehen werden. Dann kann man bei geschachtelten Schleifen bei *exit* oder *next* gezielt die Ebene angeben, auf die sich die Anweisung bezieht. Abbildung 7 zeigt die Syntax.

Syntax	Bemerkungen
[label:] [**while** bedingung] **loop** Sequentielle Anweisungen **end loop;**	Ohne **while** handelt es sich um eine Endlosschleife
[label:] **for** variable **in** range **loop** Sequentielle Anweisungen **end loop;**	Die Variable ist vom Typ integer und wird hier automatisch deklariert. Sie ist nur in der Schleife gültig. Der range kann wie bei einem Feld (Vektor) aufwärts (to) oder abwärts (downto) angegeben werden. Die Schrittweite ist immer +1 bzw. -1.
exit [label]][**when** bedingung];	Mit **exit** kann die Schleife vorzeitig verlassen werden. Dabei kann die Schleifenebene optional angegeben werden. Ohne Angabe ist die aktuelle Schleife gemeint.
next [label]][**when** bedingung];	Mit **next** wird der aktuelle Durchlauf beendet und der nächste Durchlauf begonnen. Dabei kann die Schleifenebene optional angegeben werden. Ohne Angabe ist die aktuelle Schleife gemeint.

Abbildung 7: Syntax der for- und der while-Schleife

2.5 assert / report

Diese Anweisungen dienen der Auswertung. Mit *assert* kann geprüft werden, ob zu einem bestimmten Zeitpunkt eine Bedingung erfüllt ist. Falls ja, geschieht weiter nichts. Falls nein, dann kann man mit *report* den Simulator (optional) eine Meldung ausgeben lassen. Man kann zudem angeben, wie schwerwiegend eine nicht erfüllte Bedingung ist.

```
assert bedingung [report meldung] [severity schwere];
meldung: String (z.B. "Zaehlerstand falsch")
schwere: Name einer Schwere bei nicht erfüllter Bedingung.
         Vordefiniert sind NOTE, WARNING, ERROR und FAILURE.
```

Abbildung 8: Syntax assert/report

Die Bedingung kann alles sein, was einen Wahrheitswert ergibt. Bei der Schwere *FAILURE* wird die Simulation beendet. Ohne Angabe einer Schwere wird *ERROR* verwendet.
Die *report*-Anweisung kann auch eigenständig verwendet werden.
Ohne Angabe der *report*-Anweisung wird *"Assertion violation."* verwendet.

2.6 Ende der Simulation

Normalerweise läuft die Simulation (ohne Abbruch durch den Benutzer), bis alle Anregungen abgearbeitet sind. Das kann der Simulator selber feststellen. Falls man einen ständig laufenden Prozess hat (siehe Abbildung 4, links), dann muss man dem Simulator das Ende explizit mitteilen. Abbildung 9 zeigt zwei Möglichkeiten, dies nach einer voreingestellten Zeit zu tun.

bis VHDL 2008	ab VHDL 2008

`-- End Of Simulation after 250 ns`
`process`
`begin`
`  wait for 250 ns;`
`  assert FALSE report "Ende erreicht" severity FAILURE;`
`end process;`
	`-- End Of Simulation after 250ns`
	`process`
	`begin`
	`  wait for 250 ns;`
	`  report "Ende ereicht" severity NOTE;`
	`  std.env.finish;`
	`end process;`

Abbildung 9: Beschreibungsmöglichkeiten für das Ende der Simulation

Glossar

2.7 Register

Mit „Register“ wird in der Digitalelektronik ein Speicher aus parallelgeschalteten DFF bezeichnet. Ein Register hat also immer einen Takt. Mit der *aktiven* Taktflanke übernehmen alle DFF in dem Register gleichzeitig die Daten an den Eingängen und geben sie für die gesamte folgende Taktperiode an den Ausgängen aus. Die *Breite* eines Registers bezeichnet die Anzahl der parallelgeschalteten DFF. In einem Register werden zusammengehörige Bits (z.B. ein Zählerstand) gespeichert. Für einzelne DFF der Begriff „Register“ unüblich, da es dann ja offenbar keine weiteren zugehörigen Daten gibt. Speicher, dessen einzelne Elemente nicht alle zur gleichen Zeit zugänglich sind, wird nicht als Register bezeichnet.
Abbildung 10 zeigt links eine typische Beschreibung eines Registers in VHDL. Die Breite des Registers kann dabei über den Parameter N bei einer Instanziierung der entity eingestellt werden. Rechts ist der Aufbau (N=6) des Registers aus DFF sowie ein übliches Symbol in einem Schaltplan dargestellt.

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

entity reg_n is
generic
(
  n:    positive :=1 -- Breite des
Registers
);
port
(
  clk: in  std_logic;
  d:   in  std_logic_vector(n-1 downto 0);
  q:   out std_logic_vector(n-1 downto 0)
);
end;

architecture functional of reg_n is
begin
  process(clk)
  begin
    if rising_edge(clk)
    then
      q <= d;
    end if;
  end process;
end;
```

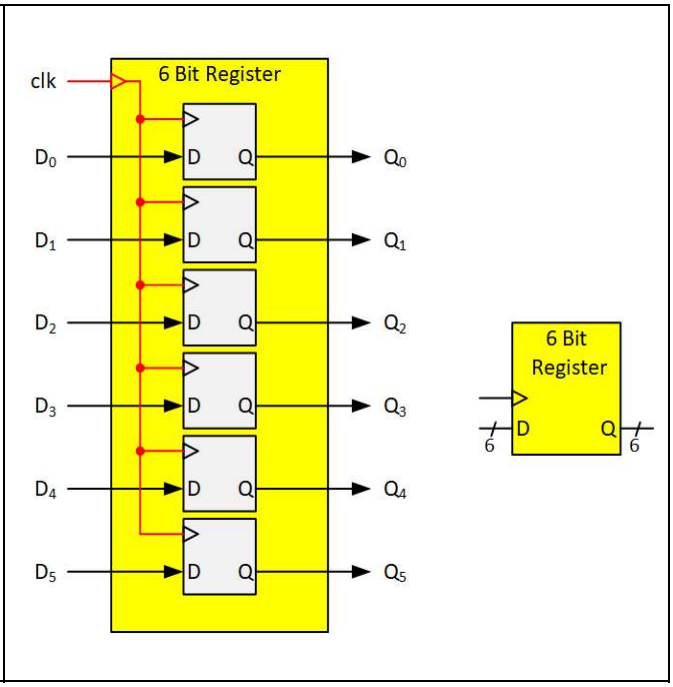


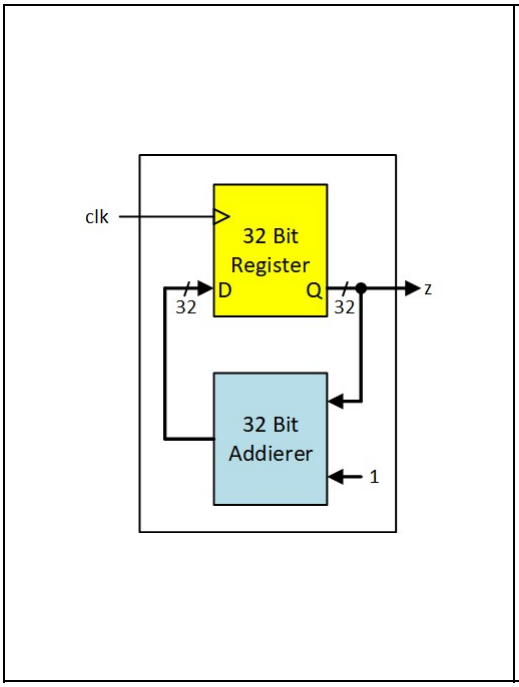
Abbildung 10: Register (VHDL, Aufbau, Symbol)

2.8 Taktsynchroner Entwurf

Bei dieser Entwurfsmethode sind alle DFF (d.h. auch alle Register) in einem Modul an denselben Takt angeschlossen. Der Bereich (d.h. alle Module), in dem dieser Takt verwendet wird, heißt *clock domain.* Viele Schaltungen haben nur eine einzige clock domain. Diese Methode hat zwar auch Nachteile (z.B. elektrische Lastspitzen bei der aktiven Taktflanke), aber die Vorteile überwiegen so deutlich, dass dies die fast ausschließlich benutzte Entwurfsmethode ist.
Hat man mehr als eine clock domain, dann erfordert der Signalübergang von einer clock domain (Quelle) zur nächsten (Ziel) besondere Aufmerksamkeit. Speziell bei unabhängigen Takten muss immer mit einer Verletzung der Setup- und Holdzeiten an den DFF des Ziels gerechnet werden. Dann müssen spezielle Techniken (Synchronisierer, Übergabeprotokolle) eingesetzt werden. Da das aufwendig ist, versucht man gerne, mit nur einer clock domain auszukommen.

2.9 Register-Transfer-Ebene (RTL, Register Transfer Level)

Dieser Ausdruck bezeichnet eine Abstraktionsebene (Level) eines taktsynchronen Entwurfs. Das zugrundeliegende Modell ist der Automat. Die Funktion der Schaltung wird durch eine Folge von Einzelschritten abgebildet. Ein Schritt wird jeweils durch die aktive Taktflanke ausgelöst. Der aktuelle Zustand der Schaltung (des Automaten) wird in Registern gespeichert. Pro Schritt werden die Inhalte von Register zu Register weitergegeben (das ist der Transfer). Im Automatenmodell entspricht das der Zustandsübergangsfunktion. Oft gibt es nur ein Register (in einem überschaubaren Modul), das dann gleichzeitig Quelle und Ziel des Transfers ist. Ein einfaches Beispiel ist ein Zähler. Er benötigt ein einziges Register, in dem der aktuelle Zählerstand gespeichert ist. Der Transfer erfolgt von den Ausgängen dieses Registers (Quelle) über einen Addierer (+1) zu den Eingängen desselben Registers (Ziel). Bei jedem Takt wird damit der Wert, der in dem Register gespeichert ist, um eins erhöht – eben ein Zähler. Abbildung 11 zeigt einen solchen Zähler.



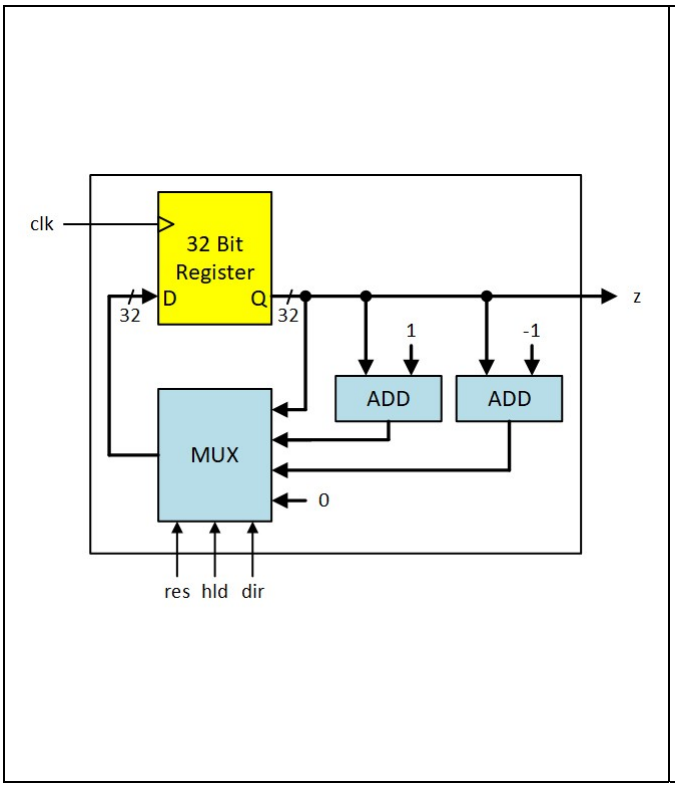

```
entity counter_1 is
port
(
  z:   out unsigned(31 downto 0);
  clk: in  std_logic
);
end;

architecture rtl of counter_1 is
  signal curcount, newcount: unsigned(31 downto 0);
begin

  -- Register (Speicher)
  process(clk)
  begin
    if rising_edge(clk)
    then
      curcount <= newcount;
    end if;
  end process;

  -- Transfer (Datenpfad)
  newcount <= curcount + 1;

  -- Ausgabe
  z <= curcount;
end;
```

Abbildung 11: RTL-Beschreibung Aufwärtszähler

Links ist der Speicher (hier 32 Bit) und der Datenpfad für den Transfer (über den Addierer) zu sehen. Rechts ist eine sehr explizite Umsetzung in VHDL angegeben. In der Praxis würde man die Addition im Prozess mit erledigen.

Der Datenfluss wird in der Regel von Bedingungen abhängen. In Abbildung 12 ist links der prinzipielle Datenfluss bei einem Universalzähler (Reset, Hold, Up, Down) gezeigt. Das ist natürlich nur ein Prinzipschaltbild. Ebenso gut könnte man nur einen Addierer verwenden und dafür an dessen zweitem Eingang zwischen +1 und -1 mit dem Signal *dir* auswählen. Rechts ist wiederum eine sehr explizite Beschreibung in VHDL angegeben[1].

[1] Das Konstrukt `(others=>'0')` füllt alle Stellen des Vektors mit Null.




```
entity counter_2 is
port
(
  z:              out unsigned(31 downto 0);
  dir, res, hld: in  std_logic;
  clk:            in  std_logic
);
end;

architecture rtl of counter_2 is
  signal cc, nc: unsigned(31 downto 0);
begin

  -- Register
  process(clk)
  begin
    if rising_edge(clk)
    then
      cc <= nc;
    end if;
  end process;

  -- Transfer (Datenpfad)
  nc <=   (others=>'0') when res='1'
     else cc            when hld='1'
     else cc+1          when dir='1'
     else cc-1;

  -- Ausgabe
  z <= cc;
end;
```

Abbildung 12: RTL-Beschreibung Universalzähler

2.10 Synthese vs. Simulation

Mit dem Begriff *Synthese* wird die Konstruktion einer funktionsfähigen Schaltung aus einer Beschreibung bezeichnet. Man kann bei Weitem nicht alle Beschreibungen automatisch synthetisieren. Die Einschränkungen bei der Beschreibung sind zum Teil durch den Stand der Technik bei den Werkzeugen gegeben, zum Teil aber auch prinzipbedingt. Wenn man in der Beschreibung angibt, dass die Verzögerung eines Signals eine bestimmte Zeit dauern soll (z.B. 10 ns), dann müsste es ein Schaltelement als Hardware geben, das genau diese Verzögerung bereitstellt. Das ist aktuell nicht möglich. Wenn man in einem Prozess eine Bedingung formuliert, die auf die steigende oder fallende Flanke reagiert, dann müsste in der Schaltung ein DFF vorhanden sein, das auf beide Flanken reagiert. Auch das ist aktuell nicht Stand der Technik.

2.11 Synthese: Behavioural, Functional, Structural

Mit diesen Begriffen werden beim Entwurf unterschiedliche Beschreibungsformen bezeichnet. Es handelt sich nicht (notwendigerweise) um Abstraktionsebenen. „Behavioural“ steht für eine Verhaltensbeschreibung. Im Entwurf ist dafür die Formulierung „wenn, dann“ typisch. Die Anforderung an eine Schaltung könnte lauten: „*Wenn* nach dem Einschalten der Kochplatte nach drei Minuten das Wasser noch nicht kocht, *dann* melde schalte die Kochplatte ab und melde einen Fehler.“ Auf dieser Ebene ist ein automatisch synthetisierbarer Entwurf in der Regel nicht möglich. Für einfachere Module, z.B. ein RS-Flipflop, geht das aber sehr gut: *Wenn* der Reset-Eingang aktiv ist, *dann* setze den Ausgang auf 0. *Sonst* prüfe den Set-Eingang. Ist er aktiv, *dann* setze den Ausgang auf 1. *Andernfalls* lasse den Wert am Ausgang unverändert. Abbildung 13 zeigt links eine Verhaltensbeschreibung eines RS-FlipFlops, dessen Eingänge lowaktiv sind. Hier kann man auch sehr gut Kommentare zu jeder Bedingung setzen.
Eine funktionale Darstellung des Entwurfs beschreibt die Zusammenhänge zwischen Ein- und Ausgängen analog zu mathematischen Funktionen der Form $f(x, y, z) = \sqrt{x} + y \sin(z)$. In der Digitalelektronik sind Wahrheitstabellen und Boole´sche Gleichungen (inklusive Arithmetik)

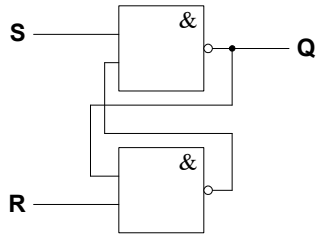
typische funktionale Darstellungen. Abbildung 13 zeigt rechts eine mögliche Funktionsbeschreibung desselben RS-FlipFlops.

```
entity rsff is
port
(
  r, s:  in  std_logic;
  q:     out std_logic
);
end;

architecture behavioural of rsff is
  signal qi: std_logic;
begin

  qi <=   '0' when (r='0') -- Reset
     else '1' when (s='0') -- Set
     else qi;              -- Store

  q <= qi;
end;
```

```
entity rsff is
port
(
  r, s: in  std_logic;
  q:    out std_logic
);
end;

architecture functional of rsff is
  signal qi: std_logic;
begin

  qi <= r and (not s or qi);

  q <= qi;
end;
```

Abbildung 13: Verhaltensbeschreibung (links) und Funktionsbeschreibung (rechts) beim Entwurf

Setzt man die Schaltung aus Teillösungen zusammen, indem man Blöcke miteinander verbindet, dann hat man eine strukturellen Darstellung. Die typische grafische Eingabeform ist der Schaltplan. In VHDL entspricht dem der Entwurf aus mehreren *entities*, die in einer übergeordneten *entity* instanziiert und verbunden werden. Abbildung 14 zeigt links eine Darstellung als Schaltplan und rechts eine Beschreibung in VHDL desselben RS-FlipFlops). Die Darstellung als Schaltplan ist nicht portabel, man wird bei einem Werkzeugwechsel Gefahr laufen, den Schaltplan erneut zeichnen zu müssen.




```
entity NAND2 is
port
(
  a, b : in  std_logic;
  y:     out std_logic
);
end;

architecture functional of NAND2 is
begin
  y <= not (a and b);
end;

entity rsff is
port
(
  r, s: in  std_logic;
  q:    out std_logic
);
end;

architecture structural of rsff is
  signal qi1, qi2: std_logic;
begin
  GATE_1: entity NAND2 port map(a => r, b => qi1, y => qi2);
  GATE_2: entity NAND2 port map(a => s, b => qi2, y => qi1);

  q <= qi1;
end;
```

Abbildung 14: Strukturbeschreibung (Schaltplan und VHDL) beim Entwurf

Alle drei Beschreibungsformen liefern dasselbe Ergebnis. Man kann im Entwurf die Formen beliebig mischen. Für größere Entwürfe ist ein Blockschaltbild auf oberster Ebene sinnvoll. Damit lässt sich eine große Aufgabe in kleinere Aufgaben zerlegen. Wo es möglich ist, kann man dann eine Verhaltensbeschreibung benutzen (z. B. für Automaten). Auch gibt es Teile, bei denen die Funktion (z.B. bei einem Zähler, y <= y+1;) die einfachste Beschreibungsform ist.

2.12 Simulation: Functional, Timing, Verification

Bei der Simulation unterscheidet man zwischen Verifikation, Simulation für die Funktion und Simulation für das Zeitverhalten. Der Begriff Verifikation ist nicht ganz glücklich gewählt, weil in diesem Zusammenhang keine formale Verifikation sondern nur eine Simulation mit so vielen Testvektoren gemeint ist, dass man die Funktion der Schaltung für gesichert hält. Tatsächlich werden aber nicht alle Fälle getestet, so dass eine Restfehlerwahrscheinlichkeit bleibt. Die Verifikation ist also nur der automatisch durchgeführte Soll-/Istvergleich vieler einzelner Simulationen, in der Regel auf der Funktionsebene.

2.12.1 Functional Simulation

Die Simulation für die Funktion ist zeitlos in dem Sinn, dass Signale entweder gar keine Laufzeiten haben oder diese Laufzeiten zu Testzwecken ohne Bezug zu einer tatsächlichen Schaltung angegeben werden. Es gibt aber einen (oder bei mehreren *clock domains* mehr als einen) Takt, der vom Simulator selbst erzeugt wird. Mit dieser Simulation kann geprüft werden, ob der Entwurf auf RTL-Ebene das gewünschte Verhalten Takt für Takt liefert. Man kann die Eingänge der Schaltung entsprechend pro Takt neu belegen und automatisch prüfen lassen, ob sich in den folgenden Takten die erwarteten Werte in den Registern bzw. an den Ausgängen zeigen. Für diese Simulation wird noch keine Synthesewerkzeug benötigt. In VHDL ist dafür die *Testbench* das Standardmittel. Simuliert wird unmittelbar der Entwurf, so wie er nach dem Zusammensetzen der einzelnen Module (in VHDL *elaboration* genannt) entstanden ist.

2.12.2 Simulation for Timing

Hier wird eine fertig umgesetzte Schaltung mit den tatsächlich erwarteten Signallaufzeiten simuliert. Diese Simulation wird in aller Regel erst nach dem „Place and Route“ durchgeführt, da erst dann die zu erwartenden Zeiten bekannt sind. Diese Zeiten hängen sehr stark davon ab, wie weit (elektrisch gesehen) Signalquellen und Signalziele voneinander entfernt sind. Das erste Ziel ist dabei, sicherzustellen, dass die Funktion bei der geforderten Taktfrequenz gewährleistet ist. Das zweite Ziel kann darin bestehen, die mögliche Taktfrequenz zu erhöhen oder alternativ die Schaltung robuster gegen Parameterschwankungen zu machen. Dazu kann man über die Simulation kritische Pfade ermitteln und dann versuchen, diese mit geänderten Vorgaben zur Synthese oder zum Place and Route zu entschärfen.
Falls sich die geforderte Taktfrequenz (Primärziel) nicht einhalten lässt, dann muss man eventuell den RTL-Entwurf ändern (siehe Abbildung 1 gestrichelter Pfeil).
Simuliert wird ein einzelner Synthesestand, den das Werkzeug nach bestimmten Vorgaben und für eine bestimmte Technologie erzeugt hat.

2.13 Backannotation

Damit wird die Übertragung tatsächlich ermittelter Signallaufzeiten in den Entwurf mit dem Ziel einer realistischen Simulation bezeichnet. Das „back“ bezieht sich darauf, dass die Information von einem späteren Entwurfsschritt (dem Place and Route) in einen früheren Schritt (z.B. den RTL-Entwurf) zurückfließt.
Tatsächlich wird aber der RTL-Entwurf nicht verändert, sondern einfach eine weitere Beschreibung des Entwurfs in VHDL erzeugt. Diese Beschreibung bildet dann die Schaltung nach dem Place and Route ab und enthält auch die erwarteten Signallaufzeiten.