Einführung in die

Simulation von AHDL-Designs mit dem Synopsys VHDL Simulator

für das FPGA-Praktikum

A. Klindworth
Arbeitsbereich „Technische Grundlagen der Informatik"
Fachbereich Informatik der Universität Hamburg
1994

Inhaltsverzeichnis

1 Einleitung

Da das im FPGA-Praktikum verwendete Entwurfssystem MAX+plusII nicht über einen eigenen Simulator verfügt, muß für die Simulation eines Designs ein anderer Simulator benutzt werden. Die Entscheidung fiel hier zugunsten des sehr leistungsfähigen Synopsys VHDL Simulators *vhdlsim* aus. Dadurch ist eine Konvertierung eines ursprünglich in AHDL beschriebenen Design-Projekts nach VHDL erforderlich, die durch entsprechende Programme automatisch erfolgt. Die beteiligten Programme übernehmen auch die unmittelbare Anbindung an *vhdlsim* und tragen die Namen *mftg*, *mttg*, *mfsim* und *mtsim*. Sie werden zusammen mit dem realisierten Simulationskonzept in Kapitel 2 erläutert.

Neben der Aufgabe, eine für den verwendeten Simulator verständliche Beschreibung eines Design-Projekts bereitzustellen, müssen für die Durchführung eines Simulationslaufs auch die Signalverläufe an den primären Eingängen, die sogenannten *Simulationsstimuli*, beschrieben werden. Diese Stimuli-Beschreibung erfolgt ebenfalls in VHDL, da VHDL ausdrucksstark genug ist, um auch komplexe Signalverläufe kompakt und in von höheren Programmiersprachen gewohnter Semantik und Syntax „programmieren" zu können. Die für diesen Zweck benötigten Kenntnisse der Konzepte und Sprachkonstrukte von VHDL werden im Kapitel 3 vorgestellt.

Einige Hinweise zur Benutzung des Synopsys VHDL Simulators *vhdlsim* folgen im abschließenden Kapitel 4, in dem die wichtigsten Kommandos zur Steuerung des Simulators aufgelistet und die Möglichkeiten zur Beobachtung von externen und internen Signalen des simulierten Designs vorgestellt werden.

2 Simulationskonzept

Nach Entwurf und Beschreibung einer digitalen Schaltung ist zu überprüfen, ob das Entwurfsergebnis der Spezifikation entspricht. Eine Simulation der entworfenen Hardwarestruktur ist in der Praxis immer noch das übliche Vorgehen, um diese Überprüfung durchzuführen. Dabei können zwei Aspekte des Designs unterschieden werden[1]: Zum einen die Funktionalität der Schaltung, beschrieben durch die logischen Funktionen der Schaltnetze und die Aufeinanderfolge von Zustandsübergängen der speichernden Elemente (Flipflops). Zum anderen das detaillierte zeitliche Verhalten der Schaltungen unter Berücksichtigung der auftretenden Verzögerungszeiten. Beide Aspekte müssen und können durch Simulation(en) mit geeigneten Stimuli dahingehend validiert werden, ob sie der Spezifikation genügen.

Will man die Funktionalität einer Schaltung[2] simulieren, so ist es hilfreich oder zumindest nicht nachteilig, wenn bei der Simulation Verzögerungszeiten zunächst nicht berücksichtigt werden. Wir werden eine solche Simulation im folgenden als *funktionale Simulation* bezeichnen und sie von der Simulation mit allen Verzögerungszeiten unterscheiden, die wir *timing-Simulation* nennen wollen.

Für die Durchführung einer timing-Simulation benötigen wir eine Beschreibung der zu überprüfenden Schaltung, die sämtliche Verzögerungszeiten exakt oder zumindest in ausreichender Näherung enthält. Da hierzu die physikalische Realisierung der Schaltung bekannt sein muß, werden diese Verzögerungszeiten vom MAX+plusII Compiler aus dem Design extrahiert, *nachdem* es auf die Logikzellen des Zielbausteins abgebildet worden ist. Hierzu erzeugt der Compiler optional eine Design-Beschreibung in EDIF, Verilog oder VHDL, in der diese Verzögerungszeiten enthalten sind. Diese Beschreibung ist eine flache Netzliste und die in ihr beschriebene Struktur spiegelt die *physikalische* Gliederung des Designs in Logikzellen und I/O-Blöcken des Zielbausteins wieder. Sowohl die hierachische Strukturierung der ursprünglichen AHDL-Beschreibung des Designs, wie auch dort verwendete Namen für Signale, Instanzen, Automaten und Automatenzustände sind in ihr nicht mehr verfügbar. Lediglich die I/O-Signale des Designs stimmen in der Netzliste mit denen in der AHDL-Beschreibung überein, wobei allerdings ursprünglich als Bus beschriebene Signalgruppen in der Netzliste als Einzelsignale auftreten. Für das Aufspüren von Fehlern in einer AHDL-Beschreibung ist es aber wünschenswert wenn nicht unerläßlich, bei der Simulation auch interne Signale beobachten oder gar zuweisen und diese auch eindeutig den Signalen

[1] Die folgende Unterscheidung trifft so nur für rein synchrone Schaltungen zu. Die Struktur der Funktionsblöcke programmierbarer Logikbausteine ist aber auf die Realisierung synchroner Schaltungen abgestimmt, so daß die allermeisten und insbesondere die im Rahmen des FPGA-Praktikums entworfenen Schaltungen ausschließlich synchrone Schaltwerke enthalten werden.

[2] Gemeint ist wieder eine synchrone Schaltung

in der ursprünglichen Beschreibung zuordnen zu können. Erkennung und Lokalisierung eines Fehlers wird immens erleichtert, wenn die simulierte Schaltungsbeschreibung genau so strukturiert ist wie in der ursprünglichen Beschreibung, d.h.

- Design-Hierarchie,
- Busse,
- Namen *aller* Design-Objekte, also Automaten, Zustandsnamen, Instanznamen und Signalnamen

sollten in dem Simulationsmodell erhalten sein. In der PC-Version verfügt MAX+plusII über einen Simulator, der die funktionale Simulation eines solchen Schaltungsmodells unterstützt. Die zugehörige Stimulibeschreibungssprache ist jedoch für die Simulation komplexer Designs recht unkomfortabel, da z.B. ein Schleifenkonstrukt fehlt. MAX+plusII für Workstations verfügt über keinen integrierten Simulator und erzeugt stattdessen optional die oben erwähnte Netzlistenbeschreibung in einem der Standardformate. In Vorbereitung des Praktikums wurde daher u.a. ein eigener Compiler entwickelt, der aus einer hierarchisch gegliederten AHDL-Beschreibung eine in gleicher Weise gegliederte VHDL-Beschreibung erzeugt. Das zugehörige Programm trägt den Namen *mftg* (für „MAX+plusII Functional Test model Generator"). Mit ihm kann also eine Schaltungsbeschreibung erzeugt werden, die den drei o.g. Anforderungen an ein strukturtreues Simulationsmodell genügt. Dieses Modell enthält aber keinerlei Informationen über Verzögerungszeiten, kann also nur für eine rein funktionale Simulation benutzt werden. Für die Durchführung einer timing-Simulation muß die vom MAX+plusII Compiler optional erzeugte VHDL-Netzliste verwendet werden. Ein Programm namens *mttg* (für „MAX+plusII Timing Test model Generator") sorgt dafür, daß dieses Timing-Model eines Designs kompatibel zu dem funktionalen Modell ist. Hierzu wird die VHDL-Netzliste geringfügig modifiziert, wobei u.a. ursprünglich als Busse beschriebene Signalgruppen restauriert werden. Auf diese Weise kann ein und dasselbe Stimulus-File für beide Typen von Simulationsmodellen benutzt werden.

Die Durchführung einer Simulation nach dem hier vorgestellten Konzept erfolgt in folgenden Schritten:

1. **Schaltungsbeschreibung in AHDL und Übersetzen mit MAX+plusII. Für die Durchführung einer timing-Simulation ist im MAX+plusII Compiler der VHDL Netlist Writer zu aktivieren.**
2. **Erzeugen des Simulationsmodells mit *mftg* für funktionale bzw. *mttg* für timing-Simulation.**
3. **Erstellen bzw. Modifizieren der Simulationsstimuli und -parameter.**
4. **Starten des Synopsys VHDL-Simulators mit *mfsim* für funktionale bzw. *mtsim* für timing-Simulation.**
5. **Auswerten der Simulationsergebnisse und ggf. zurück zu 3. oder 1. .**

In den beiden folgenden Abschnitten werden noch einige nähere Informationen zur funktionalen und der timing-Simulation gegeben, wobei auch die Aufgaben beleuchtet werden, die die beteiligten Programme zu bewältigen haben.

2.1 Funktionale Simulation (*mftg* und *mfsim*)

Die funktionale VHDL-Beschreibung eines Designs erzeugt *mftg* in mehreren Schritten, die in Abbildung 1 zusammengestellt sind.

Zunächst wird anhand des vom MAX+plusII Compiler erzeugten *mtf*-Files (Message Text File) festgestellt, ob die Übersetzung in MAX+plusII fehlerfrei erfolgt ist. Diese Überprüfung ist erforderlich, da die anschließende Analyse der Design-Hierarchie diese aus den vom MAX+plusII Compiler erzeugten *.hif*-Files (Hierarchy Information Files) extrahiert. Diese Analyse und nachfolgende Schritte in *mftg* benötigen eine konsistente Hierarchiebeschreibung in den *.hif*-Files. Als Ergebnis liefert die Hierarchie-Analyse eine sortierte Liste, in der alle an dem Gesamtdesign beteiligten Subdesigns in der Reihenfolge verkettet sind, in der sie *bottom-up* nach VHDL konvertiert werden können. Für jedes Subdesign werden bei der eigentlichen Konvertierung je ein *.io*-File

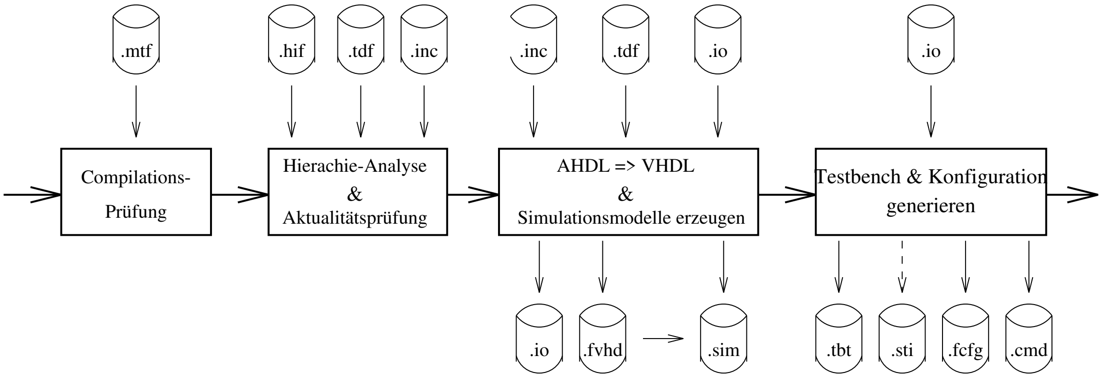

Abbildung 1: Erzeugen des funktionalen Simulationsmodells mit mftg

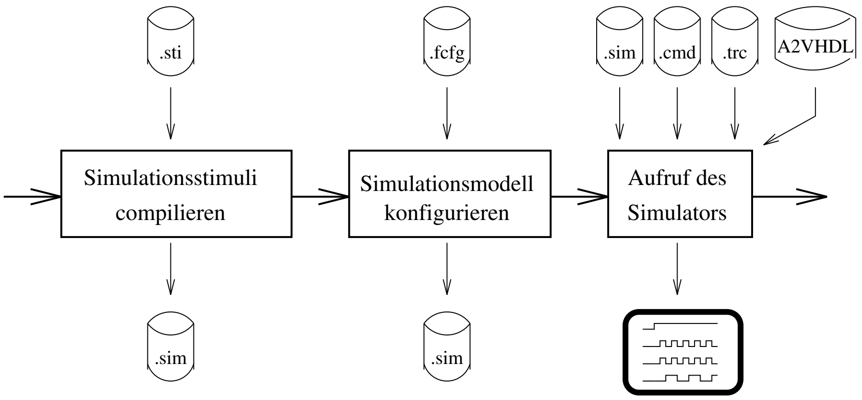

Abbildung 2: Aufruf des Simulators für funktionale Simulation (mfsim)

(Input/Output-File) und eine zur AHDL-Beschreibung funktional äquivalente VHDL-Beschreibung in der Datei mit der Extension *.fvhd* erzeugt. Eine vorgeschaltete Aktualitätsprüfung überprüft anhand der Zeitmarke des jeweils letzten Schreibzugriffes auf die beteiligten Dateien, ob VHDL-Beschreibung und/oder *.io*-File neu erzeugt werden müssen. Dieses ist der Fall, wenn das *.tdf*-File jünger als *.io*- und/oder *.fvhd*-File ist oder wenn sich die Ports einer instanzierten Makrofunktion (sprich: dessen *.io*-File) verändert haben. Unnötige Konvertierungen werden nicht durchgeführt und dadurch Zeit gespart.

Nachdem alle VHDL- und I/O-Beschreibungen der an dem Design-Projekt beteiligten Subdesigns nach VHDL konvertiert worden sind, werden die zugehörigen Simulationsmodelle durch Aufruf des Synopsys VHDL Compilers *vhdlan* aktualisiert. Abschließend werden noch das Template für die Beschreibung der Simulationsstimuli (*.tbt*-File (Testbench Template File)), ein Kommando-File (*.cmd*-File) für den Simulator *vhdlsim* sowie der VHDL Code zur Konfiguration des Designs für funktionale Simulation (*.fcfg*-File) erzeugt.

Der Aufruf des VHDL Simulators für funktionale Simulation erfolgt mit dem Shell-Script *mfsim*. Einzelne Teilaufgaben, die *mfsim* zu erledigen hat, bevor der eigentliche Simulator *vhdlsim* aufgerufen werden kann, sind in Abbildung 2 aufgeführt. Zunächst aktualisiert *mfsim* das Simulationsmodell zu dem VHDL File mit der Extension *.sti*, in dem der Benutzer die Simulationsstimuli beschrieben hat. Hierzu wird gegebenenfalls wiederum der VHDL Compiler *vhdlan* aufgerufen. Anschließend wird die Testbench für funktionale Simulation konfiguriert, indem das *.fcfg*-File mit *vhdlan* übersetzt wird. *mfsim* endet mit dem Aufruf von *vhdlsim*, wobei automatisch die Kommandos im *.cmd*-File ausgeführt werden. Dieses enthält u.a. das Kommando `trace *'signal`, wodurch sämtliche externen Anschlüsse des Designs bei der Simulation beobachtet werden. Zusätzlich hat der Benutzer

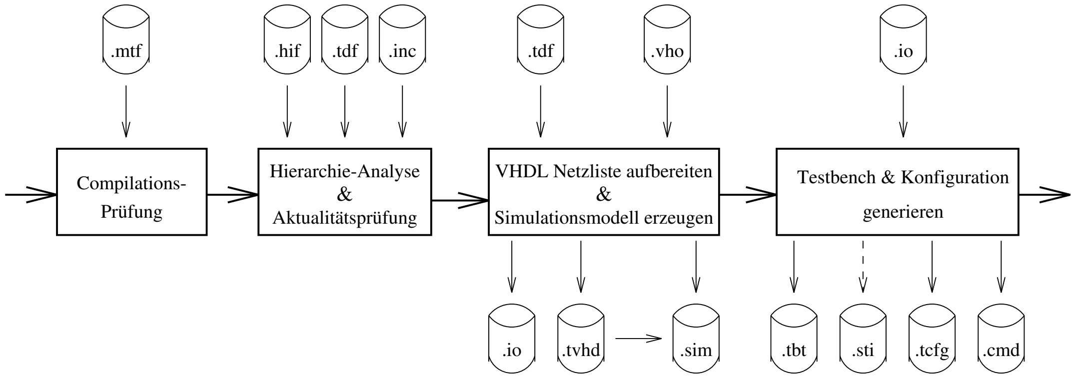

Abbildung 3: Erzeugen des Simulationsmodells mit Verzögerungszeiten (mttg)

die Möglichkeit, ein weiteres Kommando-File bereitzustellen, daß bei Aufruf von *vhdlsim* ebenfalls automatisch und im Anschluß an das *cmd*-File ausgeführt wird. Dieses optionale File muß hierzu die Extension *.trc* (Trace File) haben. Im *.trc*-File werden vorzugsweise `trace`-Kommandos zur Beobachtung interner Signale des Designs enthalten sein.[3]

2.2 Timing Simulation (*mttg* und *mtsim*)

Zunächst wird wie bei *mftg* anhand des vom MAX+plusII Compiler erzeugten *mtf*-Files (Message Text File) festgestellt, ob die Übersetzung in MAX+plusII fehlerfrei erfolgt ist. Anschließend wird die VHDL Beschreibung mit vollständiger Timing-Information generiert. Bei aktiviertem *VHDL Netlist Writer* im MAX+plusII Compiler liegt eine solche Beschreibung bereits in dem File mit der Extension *.vho* vor. Diese wird von *mttg* also so modifiziert, daß sie zu dem von *mftg* erzeugten VHDL Modell kompatibel ist. Diese modifizierte VHDL Beschreibung wird in dem File mit der Extension *.tvhd* abgelegt und das zugehörige Simulationsmodell durch Aufruf des VHDL Compilers erzeugt bzw. aktualisiert. Deswiteren erzeugt/aktualisiert *mttg* in diesem Schritt auch das *.io*-File, daß anschließend zur Generierung des Testbench Template Files sowie des Kommando-Files für den Simulator benötigt wird. *mttg* endet mit der Bereitstellung des VHDL Konfigurations Files für die Timing-Simulation.

Abbildung 3 zeigt die von *mttg* realisierten Teilschritte.

Der Aufruf des VHDL Simulators für Timing-Simulation erfolgt mit dem Shell-Script *mtsim*, das sich von *mfsim* dahingehend unterscheidet, daß durch Übersetzung des *.tcfg*-File die Testbench für Timing-Simulation konfiguriert wird. Da dieses VHDL-Modell nicht hierarchisch strukturiert ist und die Namen der internen Signale keine Zuordnung zu denen in der ursprünglichen AHDL-Beschreibung erlauben, macht es keinen Sinn, interne Signale zu beobachten. Beim abschließenden Aufruf von *vhdlsim* wird daher außer dem von *mftg* oder *mttg* erzeugten Standard-Kommando-File (*.cmd*-File) ein vorhandenes *.trc*-File nicht berücksichtigt.

3 Einführung in die Stimulibeschreibung mit VHDL

VHDL (für „**V**ery High Speed Integrated Circuit **H**ardware **D**escription **L**anguage") kann als die derzeitige Standard-Hardwarebeschreibungssprache angesehen werden. Sie wurde unter Regie des DoD entwickelt und als IEEE Standard seit 1987 etabliert mit dem Ziel, für die Beschreibung von Struktur und Verhalten nicht nur digitaler Schaltungen *eine* einheitlich über alle Entwurfsebenen hinweg verwendbare Sprache zur Verfügung zu haben. VHDL ist daher eine sehr leistungsfähige und vielseitig verwendbare Sprache, deren Beherrschung zu erlernen nur empfohlen werden kann. Das FPGA-Praktikum ist hierzu jedoch aus zeitlichen Gründen sicher nicht

[3] Für eine kurze Übersicht über die wichtigstens Kommandos von *vhdlsim*, nicht nur zum Setzen von *traces*, siehe Kapitel 4.

AHDL	VHDL
Subdesign	Entity
Subdesign Ports	Entity Ports
Logic Section	Architecture
Primitive	vordefinierte Entities
Function Prototype	Component Deklaration
Node	Signal vom Typ `node`
Instanz-Deklaration	Component Instanziierung
inline Funktionsaufruf	Component Instanziierung

Tabelle 1: AHDL und zugehörige VHDL Nomenklatur

der geeignete Rahmen. Hier wird VHDL von den Teilnehmern ausschließlich zur Beschreibung von Simulationsstimuli benutzt, wozu Kenntnisse über wenige Grundkonzepte und Sprachkonstrukte ausreichen. Sie werden in dieser Kurzeinführung vermittelt. Wer bereits aus anderen Lehrveranstaltungen oder durch Selbststudium über Erfahrungen im Umgang mit VHDL verfügt, mag diese nutzbringend einsetzen, wenn er/sie die verwendeten sprachlichen Konstrukte sicher beherrscht. Aber bitte keine übermäßige Experimentierfreude, um besonders schöne VHDL-Programme zu schreiben! Im Rahmen des Praktikums ist VHDL nur ein unvermeidliches Mittel zum Zweck.

3.1 VHDL-Nomenklatur für Design-Objekte

Die VHDL Nomenklatur für Design-Objekte weicht von der in AHDL ab. Für unsere Zwecke können die in Tabelle 1 aufgeführten annähernden Entsprechungen als gültig angesehen werden.

3.2 Das Konzept der Testbench

Bei der Stimuli-Beschreibung mit VHDL stellt sich im Rahmen des FPGA-Praktikums das Simulations-„Programm" dar als ein dem zu testenden Design hierarchisch übergeordnetes Entity, das als *Testbench* bezeichnet wird. In der Architecture dieser Testbench ist für jeden Port des zu simulierenden Designs ein mit ihm verbundenes Signal deklariert. Im Anweisungsteil der Testbench-Architecture können in einem oder mehreren VHDL-Prozessen diesen Port-Signalen Werte zugewiesen und die resultierende Antwort an den Output-Port – Signalen beobachtet werden.

Das Stimulus-File zu einem Design-Projekt enthält demnach die VHDL-Beschreibung dieser Testbench und muß den Namen `<design_name>.sti` haben, um von den Konvertierungs- und Simulations-Tools erkannt zu werden. Es wird bei der erstmaligen Konvertierung eines AHDL-Designs mit *mftg* oder *mttg* als Kopie des Files `<design_name>.tbt` erstellt. Es enthält dann einen Rahmen für die Beschreibung der Testbench und nimmt dem Benutzer einige Schreibarbeit ab, indem

- benötigte VHDL-Libraries und -Packages,[4]
- Entity und Architecture der Testbench,
- Signale für alle externen Anschlüsse des Design-Projekts, sowie
- eine Test-Instanz des Design-Projekts

bereits deklariert bzw. erzeugt sind. Der Benutzer fügt diesem Rahmen dann an den durch Kommentare markierten Stellen Stimuli-Prozesse und hierfür benötigte Deklarationen hinzu.

Die Abbildungen 4 bis 6 skizzieren hierzu das Beispiels eines 4-Bit Addierers mit Ausgangsregister.

[4] Das *Package*-Konstrukt dient in VHDL der Zusammenfassung zusammengehörige Typ- und Funktionsdeklarationen.

```
SUBDESIGN add4_akku (
  clear, clk, a[3..0], b[3..0]: INPUT;
  s[3..0]: OUTPUT;
)
...
```

Abbildung 4: Subdesign-Section des Top-Level Design-Files

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
LIBRARY A2VHDL;
USE A2VHDL.basics.all;
USE A2VHDL.functions.all;

ENTITY VHDL_add4_akku_TB IS
END VHDL_add4_akku_TB;

ARCHITECTURE VHDL_test OF VHDL_add4_akku_TB IS
  COMPONENT add4_akku
    PORT ( clear: IN node;
           clk: IN node;
           a: IN node_vector(3 downto 0);
           b: IN node_vector(3 downto 0);
           s: OUT node_vector(3 downto 0)
    );
  END COMPONENT;

  SIGNAL clear: node := 'X';
  SIGNAL clk: node := 'X';
  SIGNAL a: node_vector(3 downto 0) := "XXXX";
  SIGNAL b: node_vector(3 downto 0) := "XXXX";
  SIGNAL s: node_vector(3 downto 0);

-- Put declarations of global procedures, functions and constants here

BEGIN

-- Put stimuli processes here

  PROCESS
  BEGIN

    WAIT;
  END PROCESS;

add4_akku_INST: add4_akku PORT MAP (clear,clk,a,b,s);

END VHDL_test;
```

Abbildung 5: Erzeugtes Template für das Stimulus-File

```
LIBRARY IEEE;

...              -- erster Teil unveraendert

  SIGNAL s: node_vector(3 downto 0);

  CONSTANT nr_of_clock_cycles: INTEGER := 256;

BEGIN

  PROCESS  -- R"ucksetzen zu Beginn der Simulation
   BEGIN
    clear <= '1';
    WAIT FOR 25 NS; -- clear f"ur 25 NS auf '1'
    clear <= '0'
    WAIT;           -- danach '0'; Prozess beenden
  END PROCESS;

  PROCESS  -- Erzeugen eines Taktsignals von 10 MHz
   BEGIN
    FOR i IN 0 to nr_of_clock_cycles LOOP
      clk <= '0';
      WAIT FOR 50 NS;
      clk <= '1';
      WAIT FOR 50 NS;
    END LOOP;
    WAIT;
  END PROCESS;

  PROCESS  -- Stimuli f"ur die Dateneing"ange a und b
   BEGIN
    WAIT FOR 25 NS;
    FOR i IN 0 TO 15 LOOP
      a <= To_node(i,4);  -- INTEGER => node_vector der L"ange 4
      FOR j IN 0 TO 15 LOOP
        b <= To_node(i, 4);
        WAIT FOR 100 NS;  -- Signalzuweisungen aktivieren, 100 NS warten
      END LOOP;
    END LOOP;
    WAIT;      -- Proze"s beenden
  END PROCESS;

add4_akku_INST: add4_akku PORT MAP (clear,clk,a,b,s);

END VHDL_test;
```

Abbildung 6: Stimulus-File nach Bearbeitung durch den Benutzer

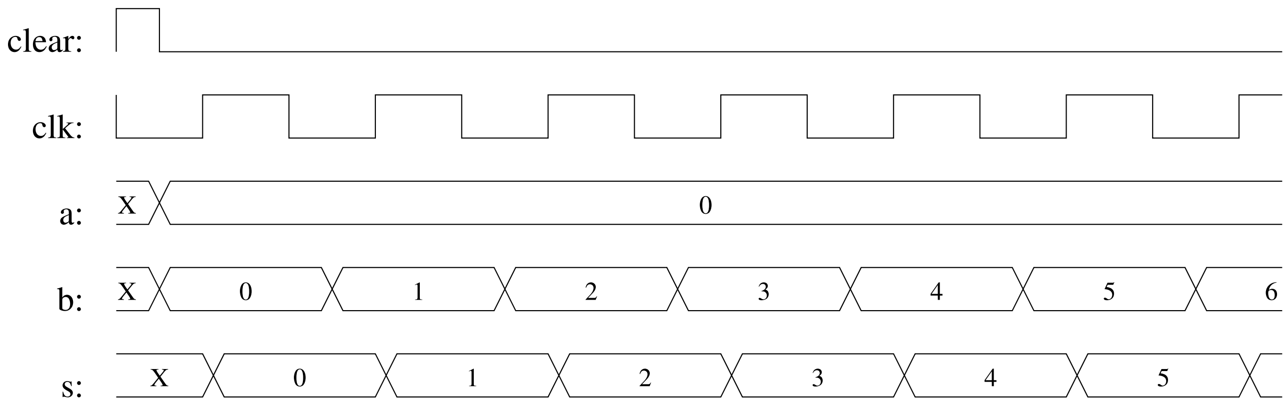

Abbildung 7: Signalverlauf bei der Simulation

In Abbildung 4 ist die Subdesign-Section des Top-Level-Files des Design-Projekts aufgeführt. Abbildung 5 zeigt das hieraus von *mftg* bzw. *mttg* erzeugte Testbench Template File. In Abbildung 6 schließlich hat der Benutzer zwei Prozesse zur Beschreibung von Stimuli in das Template eingefügt. Mit dem Programm *mfsim* bzw. *mtsim* kann dieses Stimulus-File nun übersetzt und simuliert werden. In Abbildung 7 ist der erste Teil der resultierenden Signalverläufe während der Simulation skizziert. Beachten Sie insbesondere, daß drei Stimulus-Prozesse beschrieben wurden, die alle parallel ausgeführt werden.

Wird das Design-Projekt später modifiziert, so daß es erneut mit *mftg* bzw. *mttg* konvertiert werden muß, so wird nur das Testbench Template File `<design_name>.tbt`, nicht jedoch `<design_name>.sti` neu erzeugt. Auf diese Weise bleiben die vom Benutzer in das Stimulus-File eingefügten Anweisungen erhalten. Ihm obliegt es dann aber auch, seine Testbench an die möglicherweise veränderte Anzahl, Wortbreite oder Benennung der externen Anschlüsse des Designs anzupassen. Hierauf machen *mftg* und *mttg* den Benutzer ggf. mit einer entsprechenden Warnmeldung aufmerksam.

Die Programme *mftg*, *mttg*, *mfsim* und *mtsim* sind so aufeinander abgestimmt, daß für funktionale und Timing-Simulation das gleiche Stimulus-File benutzt wird. Der Benutzer kann also beliebig zwischen funktionaler und Timing-Simulation wechseln, ohne jeweils ein neues Stimulus-File zu erzeugen bzw. Änderungen an dem File vornehmen zu müssen.

3.3 Konkurrente und Sequentielle Umgebungen

Ein wesentliches Konzept von VHDL ist die Unterscheidung zwischen konkurrenten und sequentiellen Umgebungen (*concurrent* und *sequential environments*). Die Architecture eines Entity ist eine konkurrente Umgebung, d.h. alle Anweisungen sind gleichzeitig aktiv. Zwischen den konkurrenten Anweisungen können Daten nur über einen besonderen Typ von Variablen, sog. *Signalen* ausgetauscht werden, die hierzu auch als solche deklariert sein müssen.[5]

Konkurrente Anweisungen sind

- Instanziierungen von anderen Design-Entity's, sog. *Components*,
- Zuweisungen an Signale und
- Prozesse.

Die VHDL-Beschreibung in Abbildung 6 des vorhergehenden Abschnitts zeigte hierzu bereits beispielhaft einen Anweisungsteil zu einer Architecture, in der eine Komponente vom Typ `add4_akku` instanziiert wird. Diese Komponente hat den von *mftg* generierten Namen `add4_akku_INST`. Konkurrent zu dieser Anweisung sind drei Prozesse spezifiziert, über die den Signalen Werte zugewiesen werden.

[5] Für die Simulation von AHDL-Designs sollten Sie darauf verzichten, selbst Signale zu deklarieren. Nur so bleibt die gewünschte Entsprechung von VHDL-Signalen mit den AHDL-Nodes bestehen.

3.4 Der Prozeß als sequentielle Umgebung

Mit einem Prozeß wird eine sequentielle Umgebung eröffnet, in der Anweisungen strikt sequentiell ausgeführt werden. In einem Prozeß-Environment können „gewöhnliche" Variablen, Konstanten, Prozeduren und Funktionen, genau wie in einer höheren Programmiersprache deklariert und verwendet werden. Auch im Anweisungsteil eines Prozesses stehen Sprachkonstrukte für Zuweisungen, bedingte Verzweigungen und Schleifen zur Verfügung, deren Semantik weitgehend der von Pascal- oder C-Anweisungen gleicht.

Ein Prozeß tritt mit zu ihm konkurrenten Anweisungen/Prozessen nur über Signale in Wechselwirkung, deren Wert in einem Prozeß gelesen und geschrieben werden kann. Zu beachten ist dabei die besondere Semantik von Zuweisungen an Signale innerhalb eines Prozesses:

> **Sämtliche Signalzuweisungen, die innerhalb einer Prozeß-Umgebung ausgeführt werden, werden erst nach Beenigung oder Unterbrechung des Prozesses mittels einer `WAIT`-Anweisung nach außen sichtbar.**

In einem Prozeß werden die im Programmfluß zwischen zwei `WAIT`-Anweisungen liegenden Anweisungen atomar, d.h. an einem ununterbrochenen Stück ausgeführt.[6] Erst nach Ausführung eines `WAIT`-Statements kommt ein anderer, zu diesem konkurrenter Prozeß zur Ausführung. Insbesondere werden die Anweisungen eines Prozesses zyklisch ausgeführt, d.h. nach Beendigung der letzten Anweisung wird mit der ersten fortgesetzt. Ein beliebter Fehler ist, daß einen Prozeß beendende `WAIT`-Statement zu vergessen.

Ein Prozeß gliedert sich wie ein hochsprachliches Programm in einen Deklarationsteil, in dem Datentypen, Variablen, Konstanten, Prozeduren und Funktionen deklariert werden, und einen Anweisungsteil.

3.4.1 Deklarationsteil

Im Deklarationsteil eines Prozesses können Typen, Konstanten und Variablen deklariert werden.

3.4.1.1 Typdeklarationen VHDL ist voll typisiert und der Benutzer hat die Möglichkeit, eigene Datentypen zu deklarieren. Die Syntax hierzu ist:

```
<type_declaration> ::=  TYPE <type_name> IS <type_definition>;
```

Als `<type_definition>` können skalare Typen (z.B. vordefinierter Typ INTEGER oder Aufzählungstypen), andere zuvor deklarierte Datentypen oder Arrays und Records hiervon angegeben werden. Beispiele für gültige Typdeklarationen sind

```
TYPE logic_value IS ('0', '1', 'Z', 'X');
TYPE byte IS ARRAY (7 downto 0) OF logic_value;
TYPE table_of_bytes IS ARRAY (1 to 100) OF byte;
TYPE logic_vector IS ARRAY (NATURAL RANGE <>) OF logic_value;
```

Insbesondere die letzte Deklaration enthält eine nützliche Erweiterung gegenüber vielen höheren Programmiersprachen, indem die Indexgrenzen nicht bereits bei der Deklaration eines Array-Typs festgelegt werden müssen. `logic_vector` ist dadurch einfach definiert als ein Array beliebiger Länge mit Komponenten vom Typ `logic_value`. Erst bei der Deklaration eines Objekts von dem Typ werden die Indexgrenzen festgelegt, z.B.

[6] vgl. hierzu das Beispiel in Abschnitt 3.4.2.1

```
VARIABLE a: logic_vector(1 to 10);
```

Variablen, Signale und Konstanten von einem Array-Type verfügen über sog. Attribute. Diese Attribute geben u.a. Auskunft über den Indexbereich. Für die vorgehend deklarierte Variable `a` existieren beispielsweise Attribute, die den linken, rechten, größten oder kleinsten Indexwert angeben (Attribut `LEFT`, `RIGHT`, `HIGH`, `LOW`). Für `a` haben diese Attribute die Werte 1, 10, 10 und 1; Für den gesamten Indexbereich existieren die Attribute `RANGE` und `REVERSE_RANGE`, die für `a` die Werte `1 TO 10` und `10 DOWNTO 1` haben. Auf die Attribute kann lesend zugegriffen werden, indem hinter dem Variablenname, durch ein Hochkomma getrennt, der Name des gewünschten Attributs angegeben wird. Um beispielsweise alle Komponenten von `a` in einer `FOR`-Schleife zu durchlaufen, kann folgende Formulierung benutzt werden:

```
FOR i IN a'RANGE LOOP
  ... a(i) ...
  ...
END LOOP;
```

Die Laufvariable `i` nimmt dadurch nacheinander die Werte 1,2,3,...,9,10 an.

3.4.1.2 Konstantendeklaration Konstanten sind in VHDL voll typisiert und werden mit folgender Syntax deklariert:

```
<constant_declaration> ::=

  CONSTANT <constant_name> : <type_name> := <constant_value>;
```

Beispiele für gültige Konstanten-Deklarationen sind

```
CONSTANT offset: INTEGER := 12;
CONSTANT default_address: logic_vector(7 downto 0) := "01101101";
CONSTANT value_tab: table_of_bytes := ( "00000001", "00000010",
                                        "00000100", "00001000",
                                        "00010000", "00100000",
                                        "01000000", "10000000",
                                        "XXXXXXXX", "ZZZZZZZZ"  );
```

3.4.1.3 Variablendeklaration Variablen werden in der gleichen Weise wie Konstanten deklariert und können auch in der Deklaration initialisiert werden. Zur Kennzeichnung, daß der Wert einer Variable im Gegensatz zu dem einer Konstante auch verändert werden kann, ist statt `CONSTANT` das Schlüsselwort `VARIABLE` zu verwenden.

> **Variablen dürfen nur innerhalb einer sequentiellen Umgebung, also in Prozessen, Prozeduren oder Funktionen, deklariert und benutzt werden.**

Eine Deklaration von globalen Variablen ist nicht möglich und auch nicht nötig, da für den Zweck des Datenaustausches zwischen verschiedenen Prozessen Signale zur Verfügung stehen.

3.4.1.4 Prozeduren und Funktionen Prozeduren und Funktionen können sowohl global, wie auch lokal deklariert werden und bilden eine sequentielle Umgebung. Wie gewohnt liefern Funktionen im Gegensatz zu Prozeduren einen Funktionswert zurück und können damit in Ausdrücken aufgerufen werden. Die Syntax zur Deklaration einer Funktion/Prozedur lautet:

```
<procedure_definition> ::=

  PROCEDURE <procedure_name> [ (<formal_parameter_list>) ] IS
    <subprogram_declarations>
   BEGIN
    { <sequential_statement>; }

  END <procedure_name>;


<function_definition> ::=

  FUNCTION <function_name> [ (<formal_parameter_list>) ] RETURNS <return_type> IS
    <subprogram_declarations>
   BEGIN
    { <sequential_statement>; }

  END <procedure_name>;
```

Der Funktionswert wird innerhalb des Anweisungsteils wie in C mit einem Statement der Form `RETURN <value>` spezifiziert.

In der Liste der formalen Parameter können Signale, Variablen und Konstanten enthalten sein. Das folgende Beispiel zeigt hierzu eine Prozedur `apply_bits`, die einem beliebigen Signal vom Typ `logic_vector` verschiedene Werte mit einem konstanten Zeitabstand nacheinander zuweist.

```
PROCEDURE apply_bits (SIGNAL target: OUT logic_value;
                      VARIABLE values: IN logic_vector;
                      CONSTANT time_step: IN TIME) IS
  VARIABLE akku_time: TIME;
 BEGIN
  akku_time := Now;
  FOR i IN values'RANGE LOOP
    akku_time := akku_time + time_step;
    target <= TRANSPORT values(i) AFTER akku_time;
  END LOOP;
END apply_bits;
```

Dieses Beispiel enthält einiges, daß hier bisher noch nicht erläutert wurde. Für den Parameter `target` wird vereinbart, daß es sich dabei um ein Signal vom Typ `logic_value` handelt. Die zusätzliche Modus-Angabe `OUT` zeigt an, daß dieser Parameter innerhalb der Prozedur nur geschrieben, jedoch nicht gelesen werden darf. Gleichzeitig ist damit für `target` *call-by-reference* vereinbart.

Die zuzuweisenden Werte werden der Prozedur in dem Parameter `values` übergeben, der hierzu als Variable vom Typ `logic_vector`, also als Array beliebiger Größe mit Komponenten vom Typ `logic_value` deklariert wird. Als Modus ist `IN` angegeben, sodaß `values` in der Prozedur nur gelesen und nicht beschrieben werden kann.

Der dritte Parameter, `time_step`, ist eine Konstante vom vordefinierten Typ `TIME` und gibt den (konstanten) zeitlichen Abstand zwischen den einzelnen Zuweisungen an `target` an.

Lokal wird eine Variable `akku_time` vom Typ `TIME` zur Akkumulation der Verzögerungen zwischen den einzelnen Signalzuweisungen deklariert und in dem Anweisungteil mit der aktuellen Simulationszeit `Now` (vordefiniert) initialisiert. Anschließend werden die einzelnen Komponenten von `values` mit der entsprechenden Verzögerung an `target` zugewiesen. Beachten Sie die Verwendung des Atrributs `RANGE` zur Bereichsangabe für die Laufvariable `i` in der `FOR`-Schleife.

Als Ergebnis der Prozedur ist `target` eine Tabelle mit Wertezuweisungen der Form

```
(<zuzuweisender Wert>,<Zuweisungs-Zeitpunkt>)
```

zugeordnet. Die jeweils nächste Zuweisung in dieser Tabelle wird durchgeführt, wenn die Simulationszeit den zugehörigen Zeitpunkt erreicht hat.

Einen zulässigen Aufruf der Prozedur `apply_bits` zeigt das folgende Beispiel:

```
SIGNAL a: logic_value;

PROCESS
  VARIABLE bits: logic_vector(1 to 10);
 BEGIN
  bits := "1001110100";
  apply(a, bits, 100 NS);
END PROCESS;
```

3.4.2 Sequentielle Anweisungen

3.4.2.1 Zuweisungen Wie bereits zuvor mehrfach erwähnt, ist zwischen Zuweisungen an Signale („`<=`") und Zuweisungen an Variable („`:=`") zu unterscheiden. Erstere werden erst wirksam, wenn der Prozeß ein `WAIT`-Statement ausführt, während letztere sofort Wirksamkeit erlangen. Beispielsweise wird in dem folgenden Anweisungsteil bei der zweiten Zuweisung noch der alte Wert des Signals `sig_a` gelesen, obwohl `sig_a` vorher bereits zugewiesen wurde

```
SIGNAL sig_a: logic_value;
...

PROCESS BEGIN
  VARIABLE a,b,c: logic_value;
 BEGIN
 ...                -- sei sig_a = '1'
  sig_a <= 'X';     -- Zuweisung von 'X' wird nicht sofort wirksam,
  a := sig_a;       -- also erh"alt a den alten Wert von sig_a -> a = '1'
  sig_a <= NOT a;   -- Erneute Zuweisung an sig_a macht fr"uhere Zuweisung unwirksam
  WAIT FOR 10 NS;   -- Letzte Zuweisung an sig_a wird wirksam -> sig_a = '0'
  b := sig_a;       -- b erh"alt den Wert '0';
  c := a;           -- c erh"alt den Wert '1'
 ...
END PROCESS;
```

3.4.2.2 Programmschleifen (FOR, WHILE) Zur wiederholten Ausführung eines Anweisungsblocks können Schleifenkonstrukte benutzt werden, wobei `FOR` - und `WHILE` - Schleifen zur Verfügung stehen. Die Syntax einer `FOR` - Schleife lautet

```
<for_loop> ::=  FOR i IN <range_specification> LOOP
                  { <sequential_statement> }
                END LOOP;
```

Insbesondere muß dabei die Laufvariable der Schleife nicht zuvor deklariert werden. Sie kann in dem Schleifenkörper nur gelesen, nicht beschrieben werden.

Eine `WHILE`-Schleife wird in folgender Weise angegeben:

```
<while_loop> ::=  WHILE <condition> LOOP
                    { <sequential_statement> }
                  END LOOP;
```

Um eine Schleife vorzeitig zu verlassen, kann eine `EXIT`-Anweisung (entspricht `break` in C) ausgeführt werden.

3.4.2.3 IF-Statement Zur bedingten Ausführung von Anwesiungsblöcken steht ein `IF-THEN-ELSE` - Konstrukt zur Verfügung. Die Syntax lautet

```
<if_statement> ::=    IF <condition> THEN
                        { <sequential_statement> }
                    { ELSIF <condition> THEN
                        { <sequential_statement> } }
                    [ ELSE
                        { <sequential_statement> } ]
                      END IF;
```

wobei `<condition>` ein beliebiger Ausdruck vom Typ `BOOLEAN` ist. Insbesondere kann also ein Logik-Signal nicht wie in AHDL als Bedingung angegeben werden. Stattdessen muß eine Formulierung der Form

```
IF sig_a = '1' THEN ...
```

benutzt werden.

3.4.2.4 CASE-Statement Will man einen Wert nacheinander mit verschiedenen Konstanten vergleichen, so kann hierfür wie in einer höheren Programmiersprache ein `CASE`-Statement anstelle einer `IF-THEN-ELSE`-Kette benutzt werden. Eine solche Auswahlanweisung hat die folgende syntaktische Form

```
<case_statement> ::=   CASE <selector> IS
                     { WHEN <choices> => { <sequential_statement> } }
                     [ WHEN OTHERS    => { <sequential_statement> } ]
                       END CASE;

<choices> ::=  <choice> { | <choice> }
```

wobei `<choice>` eine Konstante vom gleichen Typ wie `<selector>` sein muß.

3.5 Concurrent Signal Assignments

Neben der Möglichkeit, Signale in einer Prozeßumgebung zuzuweisen, können in einer konkurrenten Umgebung auch sog. *concurrent signal assignments* benutzt werden. Folgende Signaldeklaration mit anschließender Zuweisung erzeugt beispielsweise für `a` ein 5 MHz Taktsignal:

```
SIGNAL a: logic_value := '0';
...
a <= TRANPORT NOT a AFTER 100 NS;
```

Man beachte insbesondere das besondere Symbol „`<=`" für die Zuweisung an Signale sowie die hier benutzte Möglichkeit, Signalzuweisungen zeitlich zu verzögern (Schlüsselworte `TRANSPORT` und `AFTER`).

3.6 Instanziierung von Design-Komponenten

Für die hierachisch gegliederte Strukturbeschreibung eines Design-Projekts verfügt VHDL selbstverständlich über Sprachkonstrukte zur Instanziierung von Design-Komponenten, die über die Möglichkeiten in AHDL weit hinaus gehen. Im Rahmen des FPGA-Praktikums wird VHDL von den Teilnehmern jedoch *ausschließlich für die Beschreibung von Simulations-Stimuli* – also zur Beschreibung von *Verhalten*, nicht *Struktur* – verwendet. Auf die Erläuterung der Möglichkeiten in VHDL zur Strukturbeschreibung kann daher an dieser Stelle ganz verzichtet werden.

3.7 Die Library A2VHDL

Die von den Konvertierungsprogrammen *mftg* und *mttg* erzeugten VHDL-Modelle benutzen eine spezielle Library namens `A2VHDL`. In dieser Library sind die AHDL-Primitive ebenso als VHDL-Entities definiert, wie der für alle Signale durchgängig verwendete Datentyp `node` bzw. `node_vector` nebst zugehöriger Operationen. Darüberhinaus sind im Package `basics` Konvertierungsfunktionen bereitgestellt, um Integer-Zahlen in Bitvektoren vom Typ `node_vector` bzw. einzelne Bits vom Typ `node` umzuwandeln und umgekehrt. Diese Funktionen haben die Namen `To_int`, `To_signed_int` und `To_node` und sind in `A2VHDL.basics` folgendermaßen deklariert:

```
FUNCTION To_int (a: node) RETURN INTEGER;
FUNCTION To_int (a: node_vector) RETURN INTEGER;
FUNCTION To_signed_int (a: node_vector) RETURN INTEGER;
FUNCTION To_node (i: INTEGER; width: INTEGER) RETURN node_vector;
FUNCTION To_node (i: INTEGER) RETURN node;
FUNCTION To_node (a: BOOLEAN) RETURN node;
```

Bei der Umwandlung eines Integers in einen `node_vector` der Länge `width`, werden negative Zahlen im 2er-Komplement dargestellt. Reicht die Breite `width` des `node_vectors` nicht für die Aufnahme aller signifikanten Bits aus, wird während der Simulation eine entsprechende Warnungsmeldung ausgegeben.

Bei der Umwandlung eines `node_vector` in einen Integer-Wert interpretiert die Funktion `To_int` den Vektor als eine vorzeichenlose Binärzahl, wohingegen `To_signed_int` eine vorzeichenbehaftete Binärzahl im 2er-Komplement unterstellt.

4 Durchführung der Simulation

4.1 Vorbereitungen und Starten des Simulators

Nachdem ein AHDL-Design mit *mftg* oder *mttg* erfolgreich nach VHDL konvertiert worden ist, können Simulationsläufe mit dem VHDL Simulator durchgeführt werden. Hierzu ist folgendermaßen vorzugehen:

1. Erstellen oder Aktualisieren des Stimulus-Files `<design_name>`*.sti*.
2. Wird eine funktionale Simulation durchgeführt, so kann ein Kommando-File für *vhdlsim* erstellt werden, das beim Aufruf des Simulators automatisch ausgeführt wird. Es muß hierzu den Namen `<design_name>`.trc haben und enthält üblicherweise *trace*-Anweisungen zur Beobachtung interner Signalknoten des Designs.
3. Übersetzen der an dem Simulationsmodell beteiligten VHDL-Beschreibungen in das von *vhdlsim* benötigte interne Format. Hierzu ist für jedes VHDL-File der Design-Hierarchie *bottom-up* der Synopsys VHDL Analyzer *vhdlan* aufzurufen.
4. Aufruf des Simulators *vhdlsim*.

5. Starten eines Simulationslaufs mit dem im File <design_name>*.sti* beschriebenen Stimuli mit Hilfe des Kommandos `run`.

Die Schritte 3 und 4 werden von den Programmen *mfsim* für funktionale bzw. *mtsim* für Timing-Simulation automatisch durchgeführt. Die Schritte 1 und ggf. 2 liegen natürlich in der Verantwortung des Benuzters, ebenso Schritt 5.

4.2 Bedienung des Simulators

Die folgende Auflistung führt die wichtigsten Kommandos des Synopsys VHDL Simulators *vhdlsim* auf und erläutert sie kurz. Beachten und nutzen Sie insbesondere die Online-Hilfe des Simulators.

`help`	online-Hilfe Als optionaler Parameter kann das Thema (z.B. ein Kommando) angegeben werden, zu dem Hilfe gewünscht wird.
`ls`	zeigt die auf der aktuellen Hierarchieebene verfügbaren Signale, Instanzen (Components), und anderen Objekte. Als Parameter kann ein regulärer Ausdruck und/oder ein Attribut (z.B. 'signal) angegeben werden, um aus der Liste aller Objekte einen Subset auszuwählen
`cd`	Navigation in der Design-Hierarchie Wechselt die aktuelle Ebene in der Design-Hierarchie
`set`	erlaubt die Deklaration von Variablen, z.B. für häufig benötigte Pfade.
`assign`	erlaubt die Zuweisung von Werten an Signale
`run`	Startet die Simulation Als optionaler Parameter kann eine Zeitspanne (z.B. 100us) angegeben werden, nach der die Simulation unterbrochen werden soll. Eine laufende Simulation kann mit `CTRL-C` unterbrochen und später mit `run` fortgesetzt werden.
`restart`	startet den Simulator neu, wobei alle Kommandos bis zum ersten `run` wieder ausgeführt werden. Dieses Kommando ist nützlich, wenn man einen Simulationslauf wiederholen und dabei weitere Signale beobachten möchte.
`trace`	legt zu beobachtende Signale fest, wobei als Parameter ein regulärer Ausdruck für die Auswahl der zu beobachtenden Signale anzugeben ist
`include`	liest Kommandos aus einem Text-File
`quit`	beendet den Simulator

```
SUBDESIGN top_level ( ... )
...
VARIABLE
  adder1, adder2: add_4bit;
...
```

```
SUBDESIGN add_4bit ( ... )
...
BEGIN
  ...
  (sum0, c_in1) = fulladd(a0, b0, c_in);  % 1. anonyme Instanz
  (sum1, c_in2) = fulladd(a1, b1, c_in1); % 2. anonyme Instanz
  ...
END;
```

Abbildung 8: Hierarchische AHDL-Beschreibungen mit zwei anonymen Instanzen

4.3 Zusätzliche Möglichkeiten bei der funktionalen Simulation

Innerhalb des Synopsys VHDL-Simulators kann man in der gleichen Weise in der Design-Hierarchie navigieren wie in dem Verzeichnisbaum eines Betriebsystems. Die Wurzel des Hierarchiebaums bildet hier die Top-Level Architecture des Designs. In unserem Fall ist dieses die Testbench. Die Instanz des zu testenden Designs bildet den Tochterknoten der Testbench und hat den Namen `<design_name>_INST`. Darin instanziierte Subdesigns bilden dann die nächste Ebene des Hierarchiebaums usw. - bis hin zu den Blättern mit den Subdesigns, die keine Instanziierungen enthalten.

Auf jeder Ebene der Designhierarchie können die in dem zugehörigen Subdesign deklarierten internen Knoten und benannten Instanzen, also solchen Instanzen, die über eine Deklaration in der Variable Section erzeugt wurden, mit dem gleichen Namen angesprochen werden, der in der ursprünglichen AHDL-Beschreibung verwendet wurde. Gleiches gilt für die Inputs eines Subdesigns, wohingegen die bidirektionalen und die Output- Ports eines Subdesigns den Namen `n_<ahdl_port_name>` haben.

Im Gegensatz zu Signal-Arrays (Busse), sind Instanzen-Arrays in dem VHDL-Modell nicht erhalten. Vielmehr wird ein Instanzen-Array in einzelne Instanzen aufgelöst, deren Namen sich aus dem ursprünglichen Namen des Arrays gefolgt von dem jeweiligen Indexwert zusammensetzen. In der AHDL-Beschreibung per inline-Funktionsaufruf erzeugte Instanzen haben in der Reihenfolge ihres Auftretens in der AHDL-Beschreibung die Namen `<subdesign_name>_INST1`, `<subdesign_name>_INST2`, etc.

Um zu verdeutlichen, unter welchem Namen einzelne Objekte einer AHDL-Beschreibung in dem zugehörigen funktionalen VHDL-Modell anzusprechen sind, zeigt Abbildung 8 einen Ausschnitt einer exemplarischen AHDL-Beschreibung, die sowohl anonyme wie auch benannte Instanziierungen enthält.

Innerhalb des VHDL Simulationsmodells können die anonymen Instanzen des Subdesigns `fulladd` durch Verwendung der vollständigen Pfade referenziert werden, also

```
/top_level_INST/adder1/fulladd_INST1
/top_level_INST/adder1/fulladd_INST2
/top_level_INST/adder2/fulladd_INST1
/top_level_INST/adder2/fulladd_INST2
```

Sind für das Subdesign `fulladd` der Input-Port `a` und der Output-Port `s` definiert, so haben die zugehörigen Signale für die zweite anonyme Instanz in `adder1` die vollständigen Namen

```
/top_level_INST/adder1/fulladd_INST2/a
```

und

```
/top_level_INST/adder1/fulladd_INST2/N_s
```

Sind diese Ports von `fulladd` mit den Signalen `a1` und `sum1` von `adder1` verbunden, so sind die vollständigen Namen

```
/top_level_INST/adder1/a1
```

und

```
/top_level_INST/adder1/sum1
```

Synonyme für die zugehörigen Port-Signale der `fulladd`-Instanz.