DIPLOMARBEIT

Entwicklung eines „Hardware Description Language“ Testbench-Generators

ausgeführt am Institut für Computertechnik
der Technischen Universität Wien

unter der Anleitung von Univ. Prof. Dipl.-Ing. Dr. techn. Richard Eier,
Univ. Ass. Dipl.-Ing. Martin Horauer und
Univ. Ass. Dipl.-Ing. Wolfgang Radinger
als verantwortlich mitwirkendem Universitätsassistenten

durch

Csongor Huba Attila Sass
Kornhäuselgasse 5/4/10, 1200 Wien
Matr. Nr. 9227693

Wien, 24.01.2005

_____
(Unterschrift)

Kurzfassung

Die computerunterstützte Simulation hat auch in der Digitaltechnik die Spuren hinterlassen. Um eine Simulation durchführen zu können, wird ein Modell des Entwurfsobjekts benötigt. Die Modellierung der digitalen Systeme wurde durch die HDL (Hardware Description Language) Sprachen vereinfacht. Eine sehr verbreitete Variante der HDL Sprachen ist VHDL (Very High Speed Integrated Circuit Hardware Description Language). Mit VHDL werden die Funktionalität und das zeitliche Verhalten des Entwurfsobjekts modelliert. Durch Simulationswerkzeuge können bereits komplexe VHDL-Modelle simuliert werden. Damit kann die Funktionalität des Entwurfsobjekts vor einer Hardware-Realisierung überprüft werden.

Ein nicht zu unterschätzender Teil einer Hardware-Entwicklung ist die Überprüfung des Entwurfsobjekts. Die Fehler rechtzeitig zu Erkennen und zu Beseitigen, reduzieren die Entwicklungszeiten und die Entwicklungskosten. Um die korrekte Funktion des Modells zu überprüfen, wird eine Testbench realisiert. Die Realisierung einer Testbench ist eine mühsame und kostspielige Arbeit. Die Testbench-Strukturen weisen meistens einen ähnlichen Aufbau auf. Diese bestehen aus einem Datengenerator, aus Stimulidaten und weiteren Modulen für die Auswertung der Ergebnisse.

Im Rahmen dieser Diplomarbeit werden zwei Testbench-Strukturen präsentiert. Ausgehend von diesen Strukturen und von weiteren Überlegungen, um die Entwicklung einer Testbench und die Erfassung der Stimulidaten zu erleichtern, wurde ein Testbench-Generator realisiert. Die notwendigen Daten für eine Testbench werden mit dem Testbench-Generator über eine grafische Benutzeroberfläche erfasst. Die Daten sind auf einer sehr hohen Abstraktionsebene dargestellt. Dadurch können jeweils eine Vielzahl von Testbench-Strukturen, Stimulibefehl Modellen und Dateiformaten unterstützt werden. Die Dateien werden mit dem Testbench-Generator erzeugt. Welche Dateien generiert werden, können vor der Generierung ausgewählt werden. Die derzeit unterstützten Testbench-Strukturen und Stimulibefehl Modelle werden in dieser Arbeit auch präsentiert. Die Erweiterung der unterstützten Testbench-Strukturen, Stimulibefehl Modelle oder weiterer Dateiformate kann mit der Entwicklung neuer Programmmodulen realisiert werden, die mit einem Plugin-Mechanismus in den Testbench-Generator eingebunden werden können.

Abstract

Computer aided simulation left its traces also in the realm of digital technology. In order to carry out a simulation a model of the draft-object is needed. Modeling digital systems is simplified by HDL (Hardware Description Language). One widely used variant of HDL languages is VHDL (Very High Speed Integrated Circuit Hardware Description Language). Using VHDL the functionality and the temporal behaviour of draft object is modelled. Simulation tools are already able to simulate complex VHDL models. In such a way the functionality of a draft object can be verified before the realisation of the hardware.

One Step in developing hardware which should not be underestimated is testing the draft object. Identification and elimination of errors in due time reduces development time and development costs. In order to carry out a test as regards the proper functioning of the model a testbench is realised which is a cumbersome and expensive task. Most of the testbenches have similar structures. These consists a data generator, stimulus data and modules for the evaluation of the results.

Within the scope of my thesis some testbench structures are presented. Outgoing from these structures and from further considerations in order to facilitate the development of a testbench and the collection of the stimulus data a testbench generator is realized. Necessary data for the testbench are gathered with the help of the testbench generator. The data are displayed on a very high level of abstraction. Thus many testbench structures, stimulus command models and other file formats can be supported. The files are produced with the testbench generator. What kind of file is necessary can be selected before the generation. In the context of this thesis already existing structures and command models are also presented. An extension of the supported testbench structures, stimulus command models or other file formats can be realized with the development of new program modules, which can be merged with a plug-in mechanism into the testbench generator.

Danksagung

Ich möchte mich bei allen Personen bedanken, die mich während meines Studiums und meiner Aufwendungen für die Diplomarbeit unterstützt haben.

Für die gute Betreuung möchte ich mich im Besonderen bei Herrn Dipl.-Ing. Martin Horauer und bei Herrn Dipl.-Ing. Wolfgang Radinger bedanken.

Meinen Eltern und meiner Familie gilt ein ganz besonderer Dank.

Meine Frau Judit hat immer Verständnis gezeigt, wenn ich meine Freizeit nicht ihr und meinen Kindern gewidmet, sondern für mein Studium aufgebracht habe.

Für die moralische Unterstützung möchte ich Mag. Dieter Maier, Mag. Christian Lintner und allen Kollegen danken.

Abkürzungen

AMS	Analog und Mixed Signal
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit, Anwendung spezifischen integrierten Schaltungen
EIS	Enterprise Information System
ER	Entity Relationship
HDL	Hardware Description Language
ISS	Instruktion Set Simulator
SQL	Structured Query Language
TBG	Testbench-Generator
UML	Unified Modeling Language
UUT	Unit Under Test
VHDL	Very high speed integrated circuit Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extended Markup Language

Inhaltsverzeichnis

1 Entwicklung von integrierten Schaltkreisen

Für die heutige Generation ist die Verwendung der mikroelektronischen Anlagen in allen Bereichen eine Selbstverständlichkeit. Fast jeder kennt die Bedienung eines Fernsehers, eines Telefongerätes oder eines Bordcomputers. Die Funktionalität dieser basiert auf bestimmten elektronischen Schaltungen, die grundlegend aus zwei Arten von Schaltkreisen bestehen. Einige Entwicklungen verwenden nur Standardschaltkreise, andere benötigen Spezialbausteine, um die gewünschte Funktionalität zu erreichen. Eine integrierte elektronische Schaltung, die für bestimmte Anwendungen erstellt ist, wird ASIC (Application Specific Integrated Circuit) genannt. Der Entwurf solcher Schaltungen wird durch die Verwendung einer Hardware-Beschreibungssprache und von Software-Werkzeugen stark erleichtert.

VHDL (Very high speed integrated circuits Hardware Description Language) ist eine formale, genormte Hardware-Beschreibungssprache. Sie wird immer mehr bei der Entwicklung von Mikrochips, insbesondere von ASIC, eingesetzt. Durch die Simulation können die Kodierungsfehler und die konzeptionellen Fehler des Modells aufgedeckt werden. Das VHDL-Modell wird mit verschiedenen Eingangswerten stimuliert und die Reaktionen des Modells können beobachtet und überprüft werden. Um ein System letztendlich fertigen zu können, muss die Beschreibung immer genauer werden. Die Umformung einer weniger detaillierten Beschreibung in eine mehr detaillierte Beschreibung wird als Synthese bezeichnet. Es gibt Synthesewerkzeuge, die einzelne Konstrukte von Hardwarebeschreibungssprachen direkt auf Hardwareelemente abbilden können. Die Top-Down-Designmethode mit VHDL in Verbindung mit der Simulation und mit der automatischen Logiksynthese ermöglicht die Bewältigung der Komplexität von integrierten Schaltungen. Auf Grund des selbstdokumentierenden Charakters von VHDL kann ein mit VHDL beschriebenes Modell auch eingeschränkt als Dokumentation dienen.

1.1 VHDL Entwicklung

Die Entwicklung von VHDL ging vom amerikanischen Verteidigungsministerium aus. Das Ziel war eine maschinen- und menschenlesbare Sprache zur Beschreibung einer Hardware zu realisieren. VHDL wird ständig erweitert und überarbeitet. Der Standard selbst brauchte 16 Jahre von der ursprünglichen Ideensammlung bis zum offiziellen IEEE Standard. Erstmals wurde VHDL in 1987 standardisiert. Bei der Verabschiedung des Standards einigte man sich, in einem Zeitraum von fünf Jahren, neue überarbeitete Versionen des Standards herauszubringen. Dies führte bisher zu einer Erneuerung des Standards im Jahre 1993, 1998 und 2002. Eine bedeutende Erweiterung ist eine Übermenge von VHDL, nämlich VHDL-AMS (Analog und Mixed Signal). Es handelt sich um eine Erweiterung von analogen Beschreibungskonstrukten. Eine aktuelle Sammlung der Standards der Version 2002 ist in [EDA04] zu finden.

1.2 Modellierung mit VHDL

Die Modellierung hat in der Hardware- und Software- Entwicklung immer mehr an Bedeutung gewonnen. Die Modelle machen die Entwicklungen übersichtlicher. In VHDL wurden drei wichtige Modellierungsmethodiken berücksichtigt:

1. **Abstraktion:** verschiedene Teile eines Modells unterschiedlich detailliert zu beschreiben.
2. **Modularität:** große Funktionsblöcke zu unterteilen und in abgeschlossene Unterblöcke, den so genannten Modulen, zusammenzufassen.
3. **Hierarchie:** ein System aus mehreren Modulen, die wiederum aus mehreren Modulen bestehen können, aufzubauen.

1.2.1 Abstraktion

Die Abstraktion erlaubt es, verschiedene Teile eines Modells unterschiedlich detailliert zu beschreiben. Die Module, die nur für die Simulation gebraucht werden, müssen nicht so genau beschrieben werden, wie Module die für die Synthese gedacht sind.

Die Abstraktion wird definiert als das Weglassen von Detailinformationen. Es wird also zwischen aktuell wichtigen und unwichtigen Informationen unterschieden. Eine neue Abstraktionsebene wird definiert, in dem bestimmt wird, welche Informationen auf dieser Ebene wichtig sind und welche unwichtig. Die unwichtigen Informationen werden bei der Modellierung auf dieser Abstraktionsebene nicht berücksichtigt. Soll ein Modell einen bestimmten Abstraktionsgrad besitzen, so muss jedes Modul des Modells den gleichen Abstraktionsgrad besitzen. Ansonsten entsteht ein gemischtes Modell. In der ASIC Modellierung werden fünf Abstraktionsebenen unterschieden, die graphisch in der Abbildung 1 dargestellt sind:

1. **Verhaltensebene:** gibt die funktionale Beschreibung des Systems wieder. Es gibt keinen Systemtakt und die Signalwechsel sind asynchron, mit Schaltzeiten beschrieben. In der Regel sind solche Beschreibungen nur simulierbar, nicht aber synthetisierbar.
2. **Registerebene:** wird mittels zusammen geschalteter Funktionsblöcke, wie z.B. Register, ALU, Zähler oder Speicherelemente, aufgebaut. Die Funktionsblöcke können auch parallel arbeiten. Die Register Veränderungen und Zustandsübergänge werden mit Wahrheits-, bzw. Zustandsübergangstabellen beschrieben.
3. **Logikebene:** das gesamte Design wird durch Logikgatter und speichernde Elemente beschrieben. Den Schritt von der Registerebene zur Logikebene übernehmen die Synthesewerkzeuge.
4. **Schaltungsebene:** Wenn die topologische Objekte miteinander verknüpft sind,

entsteht eine Schaltung. Die Struktur der Schaltung ist aus passiven und aktiven Bauelementen aufgebaut. Für die Beschreibung des Verhaltens der elektrischen Größen im Zeitbereich werden im allgemeinem Differentialgleichungen verwendet.

5. **Layout Ebene:** Die unterste Ebene bildet das Layout. Hier werden die verschiedenen elektronischen Bauelemente in der entsprechenden Technologie auf den Ebenen des Chips verteilt und verbunden. Nach dem Layout kann anschließend die Schaltung gefertigt werden.

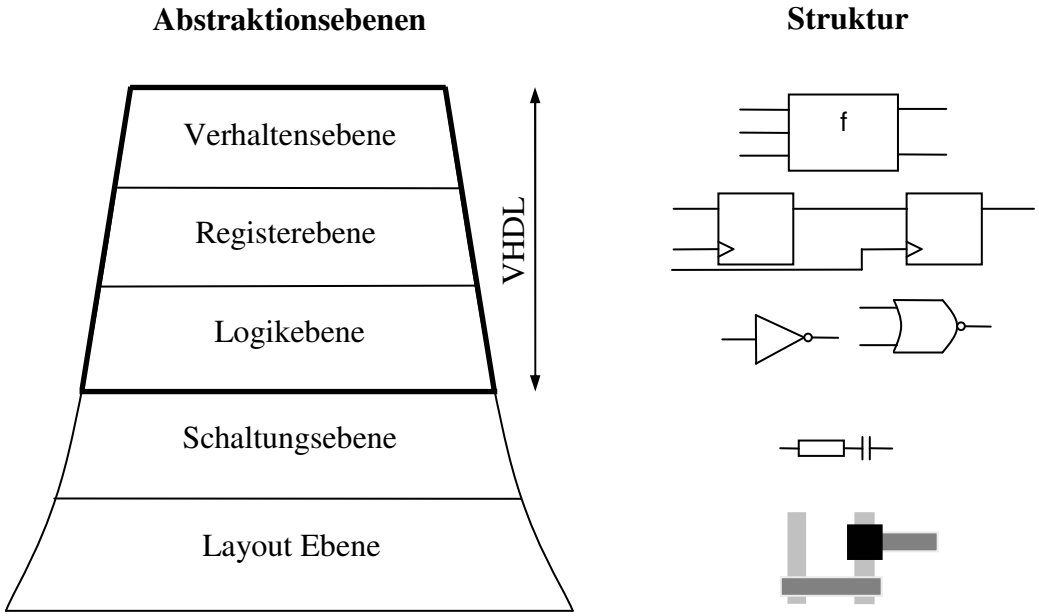

Abbildung 1 – ASIC Modellierung Abstraktionsebenen

In der Literatur gibt es weitere Definitionen der Abstraktionsebenen. In [Arm00] oder in [Hof97] wurden z.B. sechs Ebenen Definiert.

VHDL findet Anwendung in den oberen drei Abstraktionsebenen. Es ist nicht geeignet, ein Layout zu beschreiben. Der Wechsel von einer höheren zu einer tieferen Abstraktionsebene wird unterschiedlich gut von Computerwerkzeugen unterstützt.

1.2.2 Modularität

Die Modularität erlaubt, große Funktionsblöcke zu unterteilen und in abgeschlossene Unterblöcke, so genannte Module, zusammenzufassen. Dadurch kann ein komplexes System in handhabbare Subsysteme unterteilt werden. Die Kriterien, nach denen die Unterteilung vorgenommen wird, können von Entwurf zu Entwurf unterschiedlich sein. Meist erfolgt sie aber nach funktionalen Gesichtspunkten. Dies erlaubt, z.B. die Modellierung des Gesamtsystems auf mehrere Entwickler aufzuteilen. Jeder Entwickler entwirft ein abgeschlossenes Subsystem.

1.2.3 Hierarchie

Die Hierarchie erlaubt, ein System aus mehreren Modulen, die wiederum aus mehreren Modulen bestehen können, aufzubauen. Eine Ebene in der Beschreibungshierarchie kann also ein oder mehrere Module mit unterschiedlichem Abstraktionsgrad, beinhalten. Die Untermodule der darin enthaltenen Modulen bilden die nächste darunter liegende Hierarchieebene.

Die Modularität und die Hierarchie dienen also der Vereinfachung und der Ordnung. Es gibt noch weitere Vorteile: Es ist möglich an beliebigen Stellen unterschiedliche Realisierungen eines Moduls zu untersuchen. Es wird jeweils nur die entsprechende Einbindung im Gesamtmodell geändert. Zu einem Simulationsmodell kann z.B. neben dem zu entwerfenden digitalen Modul auch eine analoge Schnittstelle gehören. Auch andere, bereits als echte Hardware vorliegende Bausteine (ASICs, FPGAs), können als Simulationsmodelle eingebunden werden.

1.3 Design Flow

Die Entwicklung wird auf mehreren Phasen aufgeteilt. Ein Zusammenhang der Phasen wird in der Abbildung 2 dargestellt. Nach der Spezifikation (Funktion und Zeitverhalten) wird ein synthesegerechtes VHDL-Modell erstellt (eventuell über ein Modell auf Verhaltensebene). Dieses kann nun simuliert und damit auf Richtigkeit der Funktion geprüft werden. Zeigt das Modell das gewünschte Verhalten, so kann die VHDL Beschreibung synthetisiert werden. Dabei sucht ein Synthese-Werkzeug aus einer anzugebenden ASIC Bibliothek die entsprechenden Gatter und Flip-Flops heraus, um die Beschreibung nachzubilden. Wesentlich bei diesem so genannten Synthesevorgang ist, dass die Summe der Gatterlaufzeiten der gewählten Gatter auf den längsten Pfaden (von jedem Flip-Flop Ausgang bis zum nächsten Flip-Flop Eingang) kleiner als die Taktperiode ist. Sobald das Modell durch bestimmte Elemente einer ASIC Bibliothek dargestellt ist, kann eine Simulation auf Gatterebene durchgeführt werden. Erst jetzt treten Gatter- und Leitungsverzögerungen bei den einzelnen Signalen auf. Die einzelnen Gatter können als Verzögerung behaftete VHDL-Modelle beschrieben werden. Für die Leitungsverzögerungen werden Schätzwerte berechnet (die tatsächlichen Werte stehen erst nach dem Layout zur Verfügung). Jetzt muss nochmals geprüft werden, ob die Schaltung das vorgegebene Zeitverhalten der Spezifikation einhält.

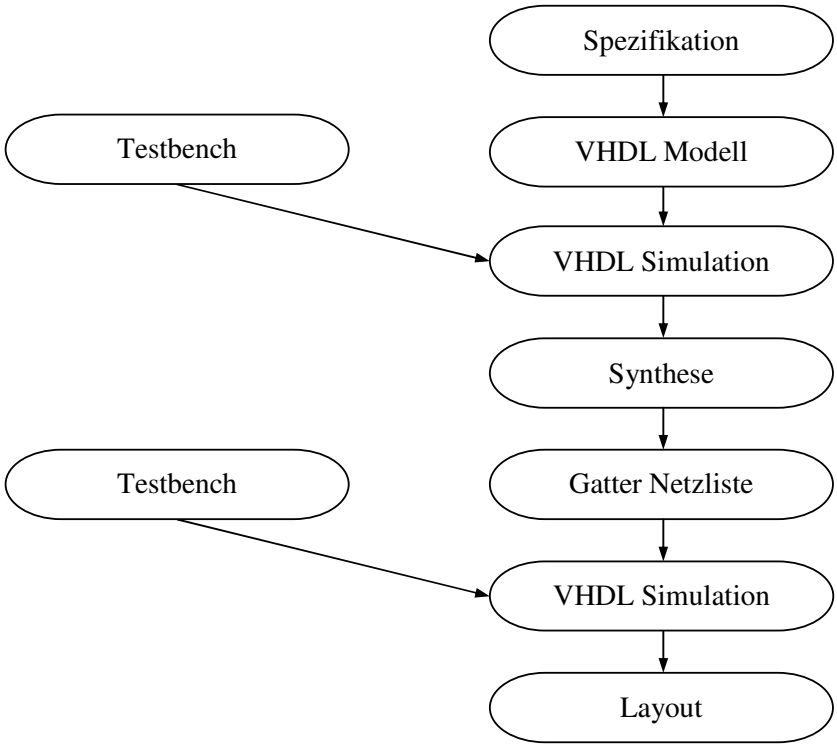

Abbildung 2 – ASIC Design Flow

In der Abbildung 3 sind die Ergebnisse der Modellierung, der Simulation und der Synthese dargestellt. Die Modellierung wird meistens mit einem Texteditor realisiert. Durch die Simulation werden die Signalabläufe in einer grafischen Form dargestellt. Nach der Synthese entsteht eine Gatter und Flip-Flop Schaltung.

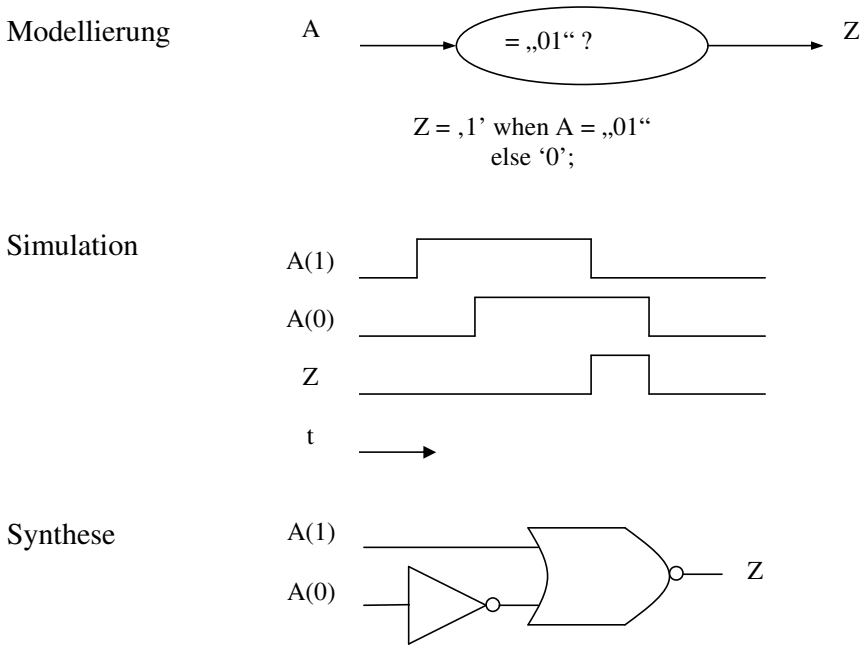

Abbildung 3 – Ergebnisse der Modellierung, Simulation und Synthese

1.4 Testbench-Strukturen

In dem derzeit angewandten Entwicklungsablauf, wird das ASIC in eine Testumgebung eingebaut. Die Testumgebung wird auch Testbench genannt. Die Komplexität des entwickelnden ASICs wirkt sich direkt auf die Komplexität der Testbench aus. Üblicherweise kann der ASIC selbst in Funktionsblöcke unterteilt werden, wobei in der ersten Testphase die Funktion der einzelnen Teile überprüft wird. In weiterer Folge werden die Blöcke miteinander verbunden und getestet. Zum Schluss wird die Überprüfung des gesamten ASIC vorgenommen. Normalerweise benötigt jede Testphase eine eigene Testbench.

Das zu testende ASIC wird auch UUT (Unit Under Test) genannt. Das Zusammenspiel der Testbench mit UUT wird in der Abbildung 4 dargestellt.

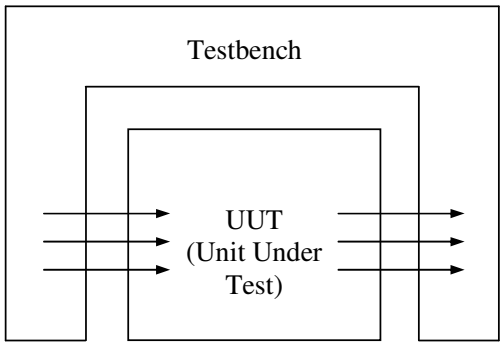

Abbildung 4 – Zusammenspiel der Testbench mit der UUT

Die Testbench ist mit den Eingängen und Ausgängen der UUT verbunden, generiert verschiedene Testsignale an den Eingängen der UUT und bewertet die Reaktion des Designs. Um Verständnisprobleme aufdecken zu können, sollte die Testbench von anderen Entwicklern erstellt werden, als das ASIC selbst. Der Entwickler einer solchen Testbench muss sich also sehr genau mit der zu erwarteten Funktion des ASICs auseinandersetzen, um möglichst realistische Szenarien zu erstellen und alle wichtigen Funktionen auch auszutesten. Der Aufwand für eine solche Testumgebung ist daher bei komplexeren ASICs sehr hoch und erreicht in vielen Fällen den zeitlichen Aufwand, der zur Erstellung des gesamten ASIC Designs notwendig war. Eine Garantie, dass ein so getestetes ASIC später seinen Dienst fehlerfrei ausführt, gibt es nicht. Immer wieder kommt es vor, dass bestimmte Kombinationen nicht in der Testbench berücksichtigt waren. Dies kann zu langwieriger Fehlersuche, kostspieligem Redesign und Verzögerungen bei der Produktfertigstellung führen.

Die Testbench-Strukturen können in zwei Gruppen gegliedert werden: in die herstellerabhängigen Simulationswerkzeuge mit Testbench Funktionalität und in die VHDL Testbench-Strukturen.

Die Simulationswerkzeuge bieten oft die Funktionalität an, womit die ASIC Entwicklungen getestet werden können. Durch die Simulator Befehle können gezielt die Signalwerte gesetzt werden. Die Möglichkeiten sind meistens begrenzt. Die Erweiterung der Funktionalität ist vom Hersteller abhängig. Durch den Austausch der Simulationswerkzeuge können die erfassten Stimulidaten nicht mehr verwendet werden.

Die VHDL Testbench-Strukturen können unterschiedliche Komplexitätsstufen erreichen. Es fängt an mit einem einfachen VHDL-Code um endet mit komplexen VHDL Bibliotheken. Die Testbench Funktionalität deckt dementsprechend sehr begrenzt oder sehr umfangreich die Anforderungen einer komplexen Simulation. Ein Nachteil der komplexen Testbench-Strukturen ist, dass sie viel Rechnerleistung benötigen. Aber mit geeigneter Hardware und Software ist es möglich die gewünschte Leistung zu erzielen. Ein weiterer wichtiger Aspekt der Testbench-Strukturen ist der Aufbau der Stimulibefehle. Die leicht lesbaren Stimulibefehle müssen meistens von Testbench stark verändert werden, bevor an die Eingänge der UUT geführt werden. Die Stimulibefehle, die ohne Veränderungen an die Eingänge der UUT geführt werden können, sind kaum lesbar. Die Entscheidung, welche Stimulibefehle verwendet werden, liegt bei dem Testbench Entwickler. Wenn weniger Rechnerleistung zur Verfügung steht und die Stimulibefehle nachträglich mit einem Texteditor nicht verändert werden sollen, dann sind die Stimulidaten nahe Stimulibefehle zu empfehlen.

Für kleine ASIC Entwicklungen, wie z.B. Prototypen, wird als Testbench ein einfacher VHDL-Code verwendet. Die Funktionalität der Testbench ist nur auf die Weitergabe der Eingabedaten, auch Stimulidaten genannt, an die UUT begrenzt. Die Stimulidaten werden direkt in der Testbench angegeben. Ein Nachteil solcher Testbench ist, dass nach jeder Änderung der Stimulidaten, der VHDL-Code neu übersetzt werden soll. Der VHDL-Code kann sehr groß werden, der Überblick über die Stimulidaten geht verloren. Die Wiederverwendbarkeit der Testbench ist kaum gegeben.

Durch die Auslagerung der Stimulidaten in externe Dateien, kann der VHDL-Code von Stimulidaten befreit werden. Die Stimulidaten werden in eine Textdatei gespeichert und beinhalten alle Werte, die für die Versorgung der UUT notwendig sind. VHDL bietet die Funktionalität um Dateien zu lesen und zu schreiben. Damit ist es möglich, ein Datengenerator Modul zu realisieren, womit die Stimulidaten aus den Dateien sequenziell gelesen und die UUT weitergegeben werden. Der Datengenerator wird über eine Schnittstelle mit der UUT verknüpft und liefert die Bitmustern an die Eingänge der UUT. Da die Stimulidaten in einem binären Format dargestellt sind, sind die Stimulidateien schwer zu lesen. Eine solche Testbench-Struktur ist in der Abbildung 5 dargestellt.

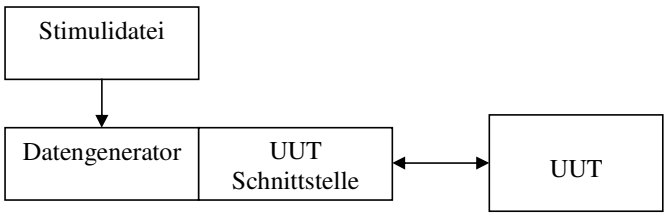

Abbildung 5 – Einfache Testbench-Struktur

Durch die Einführung von Pseudobefehlen können die binären Stimulidaten in einer lesbaren Form dargestellt werden. Um die Pseudobefehle interpretieren zu können wird ein weiteres Modul benötigt. Das Modul interpretiert die Pseudobefehle und generiert die notwendigen binären Stimulidaten. Ein solches Modul zu realisieren ist sehr schwierig und für andere

Applikationen ist nicht wieder verwendbar. Nach der Realisierung oder nach Änderungen soll neu übersetzt werden.

Eine Weiterentwicklung der Testbench ist die Einführung eines Comparators. Die UUT Ausgangsignale und die Referenzdaten werden den Comparator weitergegeben. Die Referenzdaten können entweder aus einer Datei gelesen werden oder von einem Referenzmodul erstellt werden. Wenn die Referenzdaten in eine Datei gespeichert sind, sollen diese vor dem Testlauf berechnet und erfasst werden. Das ist oft nicht einfach. Wenn ein Referenzmodul verwendet wird, dann soll die UUT nachgebaut werden. Das ist doppelter Entwicklungsaufwand. Der Comparator liefert eine Reportdatei, die mit verschiedenen Werkzeugen ausgewertet werden kann. Der Datengenerator, die UUT Schnittstelle, das Referenzmodul und der Comparator sind in VHDL realisiert und müssen nach der Realisierung oder nach Änderungen neu übersetzt werden.

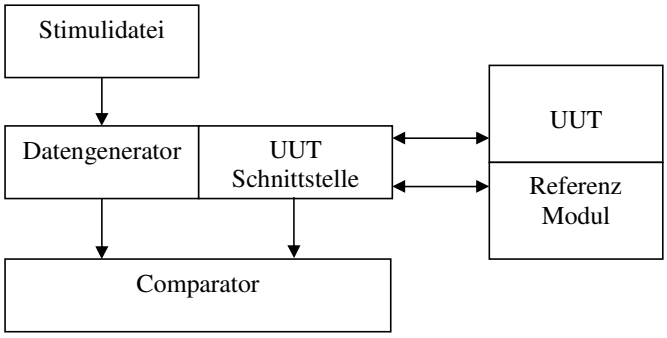

Abbildung 6 – Testbench-Struktur mit Comparator

1.5 Related Works

1.5.1 Kommerzielle Werkzeuge

Es gibt am Markt bereits Firmen, die kommerziellen Werkzeuge für ASIC-Tests anbieten. Diese werden in den folgenden Abschnitten beschrieben. Sie sind meistens sehr teuer oder sie beschränken sich auf einem Simulationswerkzeug oder auf ein Betriebssystem.

Full Circuit Ltd – VHDL TestBench Tool

Die Firma „Full Circuit Ltd“ entwickelt Software und Hardware-Lösungen für die Bahn. Unter anderem hat das Programm „VHDL TestBench Tool” entwickelt. „VHDL TestBench Tool” ist eine Microsoft Excel Anwendung. Diese besteht aus einer Microsoft Excel Dokumentvorlage, einem Microsoft Excel Makro und einem VHDL-Template. Das Werkzeug ermöglicht eine benutzerfreundliche Eingabe und Darstellung der Signalwerte. Die graphische Darstellung der Signale wird von Microsoft Excel unterstützt. Bussignale können nicht dargestellt werden. Ein wesentlicher Vorteil ist, dass die Regeln der Testdaten Definitionen, durch Excel Formeln unterstützt werden können. Die Angabe von komplexen Zyklendefinitionen ist ebenfalls sehr leicht durchführbar. Das Werkzeug generiert auf Grund der definierten Ein- und Ausgangssignalen einen VHDL-Code. Solange keine

Signaldefinition Änderungen gemacht werden, wird der Code nur einmal übersetzt. Die erfassten Testdaten werden in eine Datei gespeichert, die bei der Durchführung des Tests, aus der Datei gelesen werden. Aufgrund seiner Microsoft Excel Abhängigkeit ist es nur unter Microsoft Windows verfügbar. Eine detailliertere Beschreibung ist in [Ful04] angeführt.

SOC central – TestWizard

Die Firma SOC Central mit den Produkten TestWizard und SimCluster bieten dem Benutzer eine Testbench Umgebung, womit VHDL, Verilog oder C Testbench erstellt werden können. Unter anderem werden Eigenschaften wie Zufallsdatengenerierung, benutzerdefinierte Strukturen und Listen, funktionale Domain Analyse, usw. angeboten. Es ist nur für die Betriebssysteme Solaris und Linux verfügbar. Eine detailliertere Beschreibung ist in [SOC04] angeführt.

Mentor Graphics – Seamless

Die Firma Mentor Graphics entwickelt Lösungen für automatisiertes Elektronik Design. Das Produkt Seamless bietet eine Verifikation der Hardware und Software gleichzeitig. Bevor ein Hardware-Prototyp überhaupt entstanden ist, kann das Werkzeug die Hardware simulieren und damit den Zusammenspiel von Hardware und Software testen. Damit ist es möglich die Hardware und Software Schnittstellen Fehler frühzeitig festzustellen und zu isolieren. Die Fehler in dieser Entwicklungsphase zu erkennen reduziert die Kosten, die Entwicklungszeit, die Anzahl der Prototypen und die Risiken der Entwicklung.

Software und Hardware werden oft von unterschiedlichen Entwicklern realisiert. Deswegen werden oft Performanceprobleme bei der Interkommunikation zwischen der Software und Hardware auftreten. Seamless unterstützt die Erkennung der Performanceprobleme.

Zum Zeitpunkt des ASIC-Tests müssen die Software und ein voll funktionsfähiges Modell des Systems zur Verfügung stehen. Um eine Aussage über die Richtigkeit des entwickelnden Systems zu bekommen, ist es wichtig einen großen Teil der Software bereits fertig zu haben. Aber je größer der Software, desto langsamer die Simulation. Um die Simulation zu beschleunigen, wurde für die Abarbeitung der Software ein ISS (Instruktion-Set-Simulator) entwickelt. Der ISS interpretiert sehr schnell die Befehle und steuert damit die Hardware. Die Hardware wird über das Bus-Funktional-Modell angekoppelt. Das simuliert das elektrische Verhalten des Prozessors. Jedes mal, wenn der ISS einen neuen Befehl bearbeitet, sendet der ISS einen Befehl an das Bus-Funktional-Modell. Dieses setzt den Befehl in einen elektrischen Signal um und liefert die aus der Hardware gelesenen Daten an den ISS zurück. Jede Signaländerung in der Schaltung wird vom Simulator verfolgt. Die Verzögerungszeiten der Bausteine und der Leitungen werden auch berücksichtigt.

Seamless ist ein sehr mächtiges Werkzeug, aber die benötigte Hardware und Software Ressourcen sind komplex und teuer. Nur große Firmen können sich das leisten. Eine detailliertere Beschreibung ist in [Men04] angeführt.

1.5.2 Freie Werkzeuge

Die freien Werkzeuge für ASIC-Tests sind meistens Werkzeuge von Universitäten, die sehr an Universitätsbedürfnisse angepasst sind. Diese werden in den folgenden Abschnitten beschrieben.

TU Wien – Hofstätter Testbench

Im Rahmen einer Diplomarbeit [Hof97] an der TU Wien Institut der Computertechnik hat Hr. DI. Michael Hofstätter im Jahre 1997 bereits ein Testbench Konzept entwickelt und sämtliche VHDL Bibliotheken realisiert, womit die VHDL Module testen können. Die Testbench ermöglicht mit einem geringen Entwicklungsaufwand, Testdaten aus einer Stimulidatei der UUT zuzufügen und die Ergebnisse auszuwerten.

Die Testbench besteht aus Testbench-Kontrollblock, ASIC Specific Block, Datengenerator, Takt Generator, Datenbewertung Modul und Referenzmodul. Der Testbench-Kontrollblock steuert sämtliche Module der Testbench. Die Steuerungsbefehle sind in einer Kommandodatei gespeichert und sie werden sequenziell abgearbeitet. Der ASIC Specific Block ist das einzige Modul, das für jede Testbench neu erstellt werden soll. In diesem Modul werden die ASIC spezifischen Befehle definiert. Der Datengenerator generiert die Stimulidaten und das Timing. Er liest die Stimulibefehle aus einer Datei und versorgt die Eingangssignale der UUT. Das Datenbewertung Modul vergleicht die Istwerte mit den Sollwerten und erstellt eine Reportdatei. Die Auswertung der Reportdatei wird mit zusätzlichen Werkzeugen realisiert. Das Referenzmodul simuliert die gleiche Funktionalität wie die UUT. Dadurch ist es möglich im Datenbewertungsmodul die Istwerte mit den Sollwerten zu vergleichen.

Die Testbench ist sehr mächtig und benutzerfreundlich. VHDL-Kenntnisse sind jedoch notwendig. Ein sinnvoller Einsatz des Konzeptes ist ab einem mittelgroßen Projekt aufwärts zu empfehlen.

Universität Erlangen Nürnberg – VHDL Testbench-Generator

Die Universität Erlangen Nürnberg Lehrstuhl für Rechnergestützten Schaltungsentwurf hat einen Online VHDL Testbench-Generator in der Programmiersprache C entwickelt. Um eine Testbench zu generieren ist es ein korrekter VHDL Entity Code des überprüfenden Systems notwendig. Das Entity Code wird in eine Web Maske eingegeben und nach einer kurzen Zeit wird die Testbench angezeigt. Der wesentliche Nachteil dieser Lösung ist, das nur das VHDL-Code generiert wird und keine Testdaten. Eine detailliertere Beschreibung ist in [Erl04] angeführt.

Doulos – VHDL Testbench

Die Firma Doulos hat einen Online Testbench-Generator in der Programmiersprache Perl entwickelt. Der VHDL-Code wird in eine Web Maske eingegeben und nach einer kurze Zeit wird die Testbench angezeigt. Der wesentliche Nachteil dieser Lösung ist, das nur der VHDL-Code generiert wird und keine Testdaten. Eine detailliertere Beschreibung ist in [Dou04]

angeführt.

1.5.3 Vergleich der Werkzeuge

Die erwähnten Werkzeuge können in zwei Gruppen klassifiziert werden:

1. **Komplexe Werkzeuge:** die sehr viel Funktionalität abdecken können und damit das Testen sehr stark unterstützen. Die Nachteile solcher Werkzeuge sind die hohen Kosten und die Komplexität des Einsatzes. Die Einarbeitungszeit ist hoch, die benötigten Schnittstellen sind komplex. Die Werkzeuge sind meistens auf die Bedürfnisse von Chipherstellen abgeschnitten und deswegen sind für einen allgemeinen Einsatz nicht geeignet.
2. **Primitive Werkzeuge:** die schon bei mittelgroßen Anwendungen die Anforderungen nicht mehr erfüllen können.

Folgende Tabelle liefert einen Überblick über einigen wichtigen Merkmalen der Werkzeuge:

Werkzeug	**Vorteile**	**Nachteile**
Full Circuit Ltd: VHDL TestBench Tool	- Eingabe und Visualisierung in Microsoft Excel - Formel- und Zyklen- Definitionen werden unterstützt	- Betriebssystem Microsoft Windows
SOC central: TestWizard	- Zufallsdatengenerierung - Benutzerdefinierte Strukturen und Listen - funktionale Domain Analyse	- Betriebssystem Sun Solaris und Linux - Hardware und Software Ressourcen sind komplex und teuer
Mentor Graphics: Seamless	- Software und Hardware Zusammenspiel testbar - Performanceanalyse	- Testbar nur mit einer fertigen Software und ein voll funktionsfähiges Modell des Systems - Hardware und Software Ressourcen sind komplex und teuer
TU Wien: Hofstätter Testbench	- Sehr mächtig, jahrelange Testbench-Erfahrung wurde eingebracht - Leicht lesbare Testroutinen - Skalierbar	- Hoher Initialaufwand - VHDL Kenntnisse notwendig
Universität Erlangen Nürnberg: VHDL Testbench-Generator	- Das Werkzeug wird im Web Angeboten, deshalb keine lokale Installation ist notwendig	- Generiert nur die Hülle einer rudimentären Testbench
Doulos: VHDL Testbench	- Das Werkzeug wird im Web Angeboten, deshalb keine lokale Installation ist notwendig	- Generiert nur die Hülle einer rudimentären Testbench

1.6 Aufgabenstellung

Die Überprüfung von einem VHDL-Modell stellt einen nicht zu unterschätzenden Anteil der gesamten Entwicklungszeit dar. Die Unterstützung der Simulationswerkzeuge ist in den meisten Fällen nicht ausreichend. Deshalb wird eine geeignete Testumgebung benötigt. Die Testumgebung hängt sehr stark vom entwickelnden ASIC ab. Aber es gibt immer eine ähnliche Entwicklungsrichtlinie der Testumgebung. Es wird immer mit Stimulidaten gearbeitet, diese Stimulidaten werden an den testenden Komponenten weitergeleitet, die Ergebnisse der testenden Komponenten werden mit den Referenzdaten verglichen und ausgewertet. Die Stimulidaten werden oft mit einem Texteditor bearbeitet und in einer Textdatei abgespeichert. Wenn sehr viele Stimulidaten erfasst werden, wird die Textdatei unübersichtlich. Wenn noch zusätzlich die Schnittstellen zu den überprüfenden Komponenten geändert werden, müssen diese Änderungen auch im Testbench berücksichtigt werden. Aber damit werden die Stimulidaten inkonsistent und werden sehr wahrscheinlich auch die Verbindungen mit Komponenten inkonsistent.

Der Schwerpunkt dieser Diplomarbeit ist ein Werkzeug zu realisieren, womit das Erstellen, Modifizieren und Dokumentieren der Testbench und der Stimulidaten unterstützt werden können. Das Werkzeug wird Testbench-Generator genannt. Der modulare Aufbau des Testbench-Generators, sowie eine Sammlung von APIs (Application Programming Interface) sollen die Weiterentwicklung des Werkzeuges ermöglichen. Die genaue Spezifikation und das Konzept sind im Kapitel 2 beschrieben.

2 Testbench-Generator Design

Ausgehend von eine Sammlung Überlegungen, die die Entwicklung einer Testbench und die Erfassung der Stimulidaten erleichtern, und zwei Testbench-Strukturen, präsentiert in den nächsten Abschnitten, wurde das Design des Testbench-Generators realisiert. Das Design beschreibt das Datenmodell und die Architektur des Testbench-Generators.

2.1 Testbench-Generator Spezifikation

Die Funktionalität des Testbench-Generators kann grundsätzlich in Eingabe- und Ausgabefunktionalität gegliedert werden.

Die Eingabefunktionalität ist zuständig für die Eingabe aller Daten, die von einer Testbench benötigt werden. Die Eingabe erfolgt über ein grafisches Interface, wobei folgende Merkmale als Schwerpunkte behandelt werden:

1. Erstellung von Signalverläufen
2. Überprüfung der zeitlichen Relationen zwischen den einzelnen Signalverläufen
3. Zusammenfassung aller Signale zu Befehlen
4. Aneinanderkettung von mehreren Befehlen zu einer Testroutine
5. Nachträgliches Modifizieren, Einfügen, Löschen sämtlicher konfigurierbarer Parameter

Die Ausgabefunktionalität ist für die Generierung der Dateien zuständig. Die generierten Dateien sind die HDL Programme, die Stimulidateien, die Konfigurationsdateien und die Dokumentationen. Folgende Merkmale werden als Schwerpunkte behandelt:

1. VHDL Testbench + Patternfile
2. Verilog Testbench + Patternfile
3. ASCII Ausgabe der Befehlssequenzen
4. Programmsequenzen automatisch dokumentieren
5. LATEX Macro der Signalverläufe

2.1.1 Eingabefunktionalität

Signaldefinition

Die Eingabeprozeduren unterstützen die Erfassung der Signalen oder der Signalbussen. Zur Definition eines Signals oder eines Signalbusses werden im Allgemeinen mehrere Parameter benötigt: Signalname, Signalanzahl, Zykluszeiten, Buszyklusdauer, Default-Werte für inaktive Zustände, usw. Der Signalname bezeichnet das zu charakterisierende Signal (z.B. CS#, DATA, usw.). Die Signalanzahl wird unterschieden, ob es sich um eine einzige Leitung

oder um einen Bus handelt und aus wie vielen letzterer besteht. Mit Hilfe der Zykluszeiten werden alle Zeitpunkte innerhalb eines Buszyklusses, zu denen ein Pegelwechsel der betrachteten Leitung erfolgt, genau festgelegt. Anhand dieser Zeiten wird definiert, wann ein Signal aktiv, bzw. inaktiv ist. Die Zykluszeiten aller Signale müssen sich jeweils in Summe genau zur Dauer des Buszyklusses ergänzen. Jedes Signal kann individuell einen oder mehrere Zeitpunkte aufweisen, zu denen der Wert bzw. Pegel des Signals in ein Ausgangsfile protokolliert wird. Dadurch soll der automatische Vergleich von Patternfiles zu verschiedenen Zeitpunkten in einem Designprozess ermöglicht werden. Weiters sind während der Dauer eines Buszyklusses die einzelnen Signale nur zu definierten Zeiten gültig. In den restlichen so genannten inaktiven Zeitpunkten, nimmt das jeweilige Signal einen definierten Default-Pegel an der für die restliche Betrachtung und Handhabung fixiert wird. Der aktive Teil des jeweiligen Signals wird später entweder bei der Befehlsdefinition (z.B. Chip Select Werte, Byte Enable Pegel, usw.) oder aber gar erst bei der Aneinanderkettung der einzelnen Befehle als Parameter (z.B. DATA, R/W, usw.) festgelegt. Innerhalb einer Buszyklusdauer kann ein Befehl auch mehrmals hintereinander einen aktiven Zustand annehmen. In diesem Fall sind für dieses Signal mehrere Variablen zur Beschreibung nötig.

Befehlsdefinition

Für jeden einzelnen zu definierenden Befehl müssen für die Dauer eines Befehlszyklus jeweils alle Signale getrieben werden. Wird z.B. der Akkumulator-Register einer zu testenden Unit beschrieben oder gelesen, so kann hierfür ein Befehl ACC VAR1, VAR2 definiert werden. Für diesen Befehl werden die Adresssignale während des aktiven Bereichs auf den fixen Wert des Akkumulator Registers eingestellt. Weiters wird das Chip Select Signal zum aktiven Zeitpunkt fix auf logisch Null gelegt. Das Label ACC dient zur Kennzeichnung des Befehls und muss sich von allen anderen Befehlen in der Bezeichnung unterscheiden. Der Parameter VAR1 könnte z.B. den aktiven Zustand der R/W# Signalleitung aufnehmen und würde erst später bei der Befehlsaneinanderkettung festgelegt werden. Ebenso könnte VAR2 die entsprechende Variable für den aktiven Teil der Daten Signalleitungen darstellen. Für den Befehl ACC VAR1, VAR2 würde somit folgende Festlegung entstehen:

Signal	Wert
CS#	0
ADDR	0x0F0
R/W#	VAR1
DATA	VAR2

Tabelle 1 – Befehl ACC

Zu diesem Zeitpunkt ist bereits keinerlei Information über das zeitliche Signalverhalten mehr vorhanden. Dies wird gänzlich von der unterliegenden Signaldefinitionsschicht übernommen.

Befehlsaneinanderkettung zu einer Testroutine

Die zuvor definierten Befehle können in dieser Schicht zu einer Testroutine aneinandergefügt werden. Erst jetzt werden die gesamten Stimulidaten für die Stimulidatei, die bereits

basierend auf der Signaldefinition erstellt wurden, generiert. Die Stimulidatei besteht aus aneinander gefügten Stimulidaten.

Modifikationsmöglichkeiten

Um die Entwurfsgestaltung und Handhabung des Testbench-Generators möglichst komfortabel zu gestallten, sind zahlreiche Möglichkeiten zum nachträglichen Editieren, Einfügen und Löschen von Befehlen, Befehlssequenzen und einzelnen Signalparametern vorzusehen. Hierbei wird vor allem die Modifikation in einer Abstraktionsschicht die entsprechenden Auswirkungen in den anderen Schichten erzeugt. Wird z.B. die Anzahl der Signalbits einer Leitung bei der Signaldefinition geändert, so soll dies auch Auswirkungen in der Befehlsdefinition- und Testroutinen- Schicht bewirken.

2.1.2 Ausgabefunktionalität

VHDL u. VERILOG Testprogramm und Patternfile

Aus den eingegebenen Daten erstellt der Testbench-Generator wahlweise entweder ein VHDL oder ein VERILOG Testprogramm, das sämtliche Datenoperationen über Dateizugriffe der Stimulidatei (Stimulidaten und Referenzdaten) durchführt. Das Portmapping wird vom Testbench-Generator unterstützt. Sowohl das VHDL Testprogramm als auch das VERILOG Testprogramm werden mit denselben Stimulidateien versorgt.

Latex Macro der Signalverläufe

Um das Timing des Buszyklusses gleich automatisch dokumentieren zu können, wird der Buszyklus als Latex Macro abgespeichert. Dadurch werden die Signalverläufe direkt als Waveforms dargestellt und können somit leichter verifiziert werden.

ASCII Ausgabe der Befehlssequenzen

Während eines Testlaufs werden in der Regel die simulierten Waveforms manuell auf ihre Funktionalität kontrolliert. Um hierfür einen Überblick gewährleisten zu können, werden die angelegten Testsequenzen mitdokumentiert. Die Befehlsfolge wird ASCII mäßig exportiert.

2.2 Testbench-Struktur mit Controller

In diesem Abschnitt präsentierte Testbench ist in der Lage sehr komplexe Simulationen durchzuführen. Das Blockschaltbild der Testbench ist in der Abbildung 7 präsentiert und besteht aus folgenden Modulen:

1. **Parser:** parst die Konfigurationsdatei und Stimulidatei.
2. **Controller:** steuert die Testbench durch die Steuerungsbefehlen.
3. **Stimuligenerator:** stellt die Stimulidaten der UUT und des Referenzmoduls, bzw. die Referenzdaten dem Comparator zur Verfügung.
4. **Taktgenerator:** liefert den Systemtakt.

5. **Comparator:** wertet die Ausgabedaten der UUT aufgrund der Referenzdaten aus.
6. **Reportgenerator:** speichert die Simulationsergebnisse in eine Textdatei ab.
7. **UUT:** der zu testenden Einheit.
8. **Referenzmodul:** liefert die Solldaten.

Die Testbench Module sind parametrisierbar. Ein Parameter ist z.B. die Periode des Taktgenerators. Diese werden bei der Initialisierung der Testbench gesetzt, wobei bestimmte Parameter können jeder Zeit mitverändert werden.

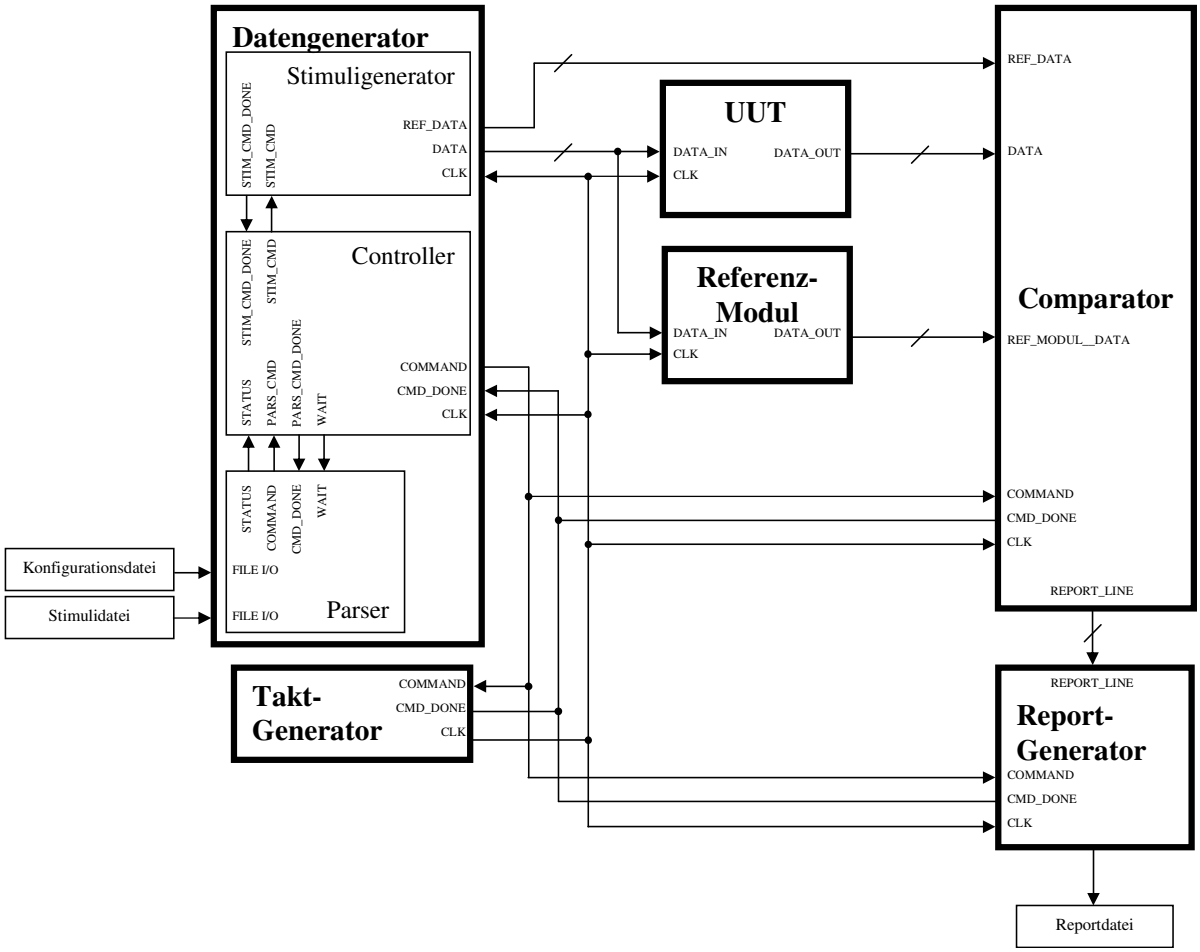

Abbildung 7 – Testbench-Struktur mit einem Controller

Der Datengenerator hat zwei wesentliche Aufgaben. Erstens die Initialisierung und die Steuerung der Testbench Modulen und zweitens die Versorgung der UUT und des Referenzmoduls mit Stimulidaten, bzw. die Versorgung des Comparators mit den zu erwartenden Werten. Die Initialisierung der Module erfolgt zu Beginn der Simulation anhand der Konfigurationsdatei. Der Datengenerator liest die Steuerungsbefehle aus der Konfigurationsdatei und leitet die Befehle über die COMMAND Leitung an den beteiligten Modulen weiter. Die erfolgreiche Übernahme der Werte, wird über die CMD_DONE Leitung bestätigt. Nach der Initialisierung der Testbench, wird die Simulation durchgeführt. Die Module können auch während der Simulation neu konfiguriert werden. In diesem Fall sind die Steuerungsbefehle zwischen den Stimulibefehlen eingetragen. Sobald ein

Steuerungsbefehl in den Stimulidaten vorkommt, wird ein Steuerungsvorgang eingeleitet. Die Werte werden über die COMMAND Leitung an den beteiligten Modulen weitergeleitet und über die CMD_DONE Leitung wird eine Bestätigung abgeholt. Die Versorgung der UUT und des Referenzmoduls mit den Stimulidaten, bzw. der Comparator mit den Referenzdaten, werden anhand der Stimulibefehle aus der Stimulidatei durchgeführt. Die Stimulibefehle werden aus der Stimulidatei gelesen, in Stimulidaten, bzw. in den Referenzdaten konvertiert und anschließend an die DATA, bzw. REF_DATA Leitungen geführt.

Der Comparator übernimmt die Ausgabedaten der UUT und des Referenzmoduls, bzw. die Referenzdaten vom Datengenerator. Die Werte werden verglichen und an den Reportgenerator weitergeleitet. Der Reportgenerator überprüft seine Eingänge und erfasst die Daten in eine Reportdatei. Welche Daten mitgeschrieben werden, wird durch den Reportgenerator spezifischen Parameter gesteuert. Der Taktgenerator liefert den Takt an allen Modulen. Dieser wird ebenfalls über die COMMAND Leitung gesteuert. Die UUT, bzw. das Referenzmodul werden nicht beschrieben. Sie sind die Resultate der ASIC Entwicklungen und haben eine klar definierte Schnittstelle nach Außen.

[Hof97] beschreibt eine ähnliche Testbench-Struktur und kann für die Erweitung der Testbench-Struktur paar Ideen liefern.

2.2.1 Parser

Der Parser hat die Aufgabe die Befehle aus der Konfigurationsdatei und Stimulidatei zu lesen und den Controller zur Verfügung stellen. Die Konfigurationsdatei beinhaltet die Initialisierungsbefehle, womit die Module, inklusiv der Parser, am Anfang der Simulation initialisiert werden. Die Stimulidatei beinhaltet die Stimulidaten der UUT, des Referenzmoduls, die Referenzdaten des Comparators und weitere Steuerungsbefehle. Die zwei Dateien sind Textdateien und können mit herkömmlichen Texteditoren bearbeitet werden.

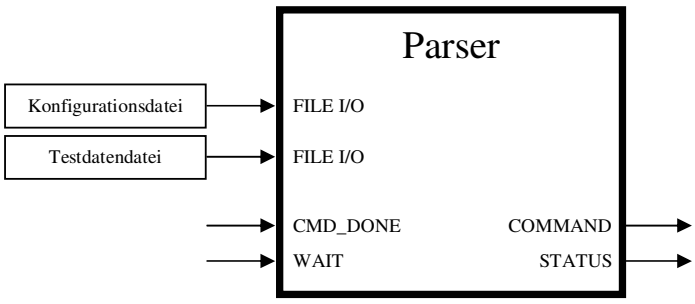

Abbildung 8 – Parser Modul

Die Testbench wird mit den Steuerungsbefehlen aus der Konfigurationsdatei initialisiert. Nach der Initialisierung der Testbench werden die Befehle aus der Stimulidatei sequenziell eingelesen. Die eingelesenen Befehle werden über die COMMAND Leitung als Zeichenkette an den Controller weitergeleitet. Auf der Leitung CMD_DONE wird die erfolgreiche Ausführung der Befehle vom Controller signalisiert. Das zeitliche Verhalten wird somit, bis auf eine Ausnahme, durch die CMD_DONE Leitung bestimmt. Als Ausnahme gilt die WAIT

Leitung, welche dazu verwendet werden kann, den Parser vom Controller vorübergehend anzuhalten. Die Zustände des Parsers werden über die STATUS Leitung dem Controller zur Verfügung gestellt.

Schnittstelle

Die Schnittstelle des Parsers ist in der Abbildung 8 dargestellt und besteht aus folgenden Leitungen:

1. **COMMAND:** ist von Typ String und dient zur Übermittlung der Befehle als Zeichenkette an den Controller. Die Zeichenketten bestehen aus dem Befehlswort und weiteren Parametern. Die Befehle sind modulspezifisch und werden bei der Beschreibung jedes Moduls spezifiziert.
2. **CMD_DONE:** ist von Typ Std_Logic. Wenn der Befehl vom Controller übernommen wurde, wird vom Controller über diese Leitung ein Impuls erzeugt. Damit signalisiert dieser, dass die COMMAND Leitung mit dem nächsten Befehl angelegt werden kann.
3. **WAIT:** ist von Typ Boolean. Wenn auf dieser Leitung der Wert TRUE anliegt, wird mit den Parsen so lange gewartet, bis auf diese Leitung vom Controller wieder den Wert FALSE gesetzt wird. Mit dieser Leitung wird signalisiert, dass der Controller intern beschäftigt ist und keine neue Befehle übernehmen kann. Dies kann z.B. dann auftreten, wenn einem Stimulibefehl ein Signalwertebereich zugewiesen wurde, und diesen Wertebereich der Stimuligenerator zuerst auflösen muss, dann die einzelnen Werte dem Stimulibefehl zuweisen und die Stimulibefehle schließlich ausführen muss.
4. **STATUS:** ist von Typ String und signalisiert den Zustand des Parsers. Je nach dem welchen Zustand der Parser annimmt, wird der Controller seine Tätigkeiten durchführen. Wenn z.B. der Parser in der Initialisierungsphase ist, wird der Controller warten, bis der Parser fertig ist. Folgende Zustände kann der Parser annehmen:
    a. **I:** Die Initialisierung des Parsers wird durchgeführt.
    b. **C:** Ein Befehl zum Verarbeiten liegt an der COMMAND Leitung.
    c. **E *Fehlermeldung*:** Ein Parserfehler ist aufgetreten. Die Fehlermeldung wird auch mitgeliefert.
    d. **R:** Die Bearbeitung der Stimulidatei ist abgeschlossen.

Konfigurationsdatei

In der Konfigurationsdatei werden die Initialisierungsbefehle der Module abgespeichert. Sind modulspezifische Befehle, also Befehle die nicht an die UUT, bzw. an das Referenzmodul bestimmt sind, sondern an den anderen Testbench Modulen. Welche Befehle in der Konfigurationsdatei vorkommen dürfen und wie die Befehle ausschauen, werden in der

Beschreibung der einzelnen Module definiert.

Stimulidatei

Die Stimulidatei beinhaltet die Befehle, die bei der Simulation ausgeführt werden sollen. Die Steuerungsbefehle und die Stimulibefehle werden gemischt angewendet. Dadurch ist es möglich in der Simulationsphase bestimmte Testbench Parameter zu verändern und modifizierte Simulationsbedingungen zu erzeugen. Welche Steuerungsbefehle in der Stimulidatei vorkommen dürfen und wie die Befehle ausschauen, werden in der Beschreibung der einzelnen Module definiert. Die Stimulibefehle beinhalten die Informationen, die für die Versorgung der Eingangssignale der UUT, bzw. des Referenzmoduls notwendig sind.

Parameter

Der Parser kann durch den folgenden Parameter konfiguriert werden:

1. ***KONFIGURATIONSDATEI*** **`string:`** Name der Konfigurationsdatei.
2. ***STIMULIDATEI*** **`string:`** Name der Stimulidatei.

Befehle

Der Steuerungsbefehl des Parsers setzt den STIMULIDATEI Parameter des Parsers. Dieser kann auch während der Simulation aufgerufen werden.

Syntax:

```
command ::=   PARSER STIMULIDATEI string
```

Fehlerbehandlung

Wenn im Parser ein Fehler auftritt, bricht mit dem Parsen ab und setzt den STATUS Leitung auf „E“. Folgende bekannte Fehler können auftreten:

1. **Konfigurationsdatei nicht vorhanden:** wenn die angegebene Konfigurationsdatei nicht im Dateisystem vorhanden ist.
2. **Stimulidatei nicht vorhanden:** wenn die angegebene Stimulidatei nicht im Dateisystem vorhanden ist.
3. **Konfigurationsdatei mit falschen Format:** wenn die Konfigurationsdatei einen falschen Format hat, bzw. unlesbar ist.
4. **Stimulidatei mit falschen Format:** wenn die Stimulidatei einen falschen Format hat, bzw. unlesbar ist.

2.2.2 Controller

Der Controller ist für die Steuerung der Testbench zuständig. Dieser unterscheidet zwischen den Steuerungsbefehlen und Stimulibefehlen und leitet sie weiter.

Die Steuerungsbefehle werden über die COMMAND Ausgangsleitung an den adressierten

Testbench Modulen, wie der Taktgenerator, der Comparator oder der Reportgenerator, weitergeleitet. Die Bestätigung von den Modulen, dass die Befehle angenommen wurden, wird über die CMD_DONE Eingangsleitung gesendet. Da diese Leitung von mehreren Modulen betrieben werden können, ist es sehr wichtig, nach dem Senden der Bestätigung, das aktive Modul von der Leitung abzukoppeln. Der Controller wartet bis auf eine gewisse Zeit auf die Bestätigung. Es kann vorkommen, dass kein Modul den Befehl annimmt und damit wird die Fortsetzung der Befehlsverarbeitung verhindert. Die maximale Wartezeit ist ein Parameter der Controller und er ist jederzeit änderbar.

Die Stimulibefehle werden über die STIM_CMD Leitung an den Stimuligenerator weitergeleitet. Wenn im Stimulibefehl ein Wertbereich angegeben ist, löst der Controller dieses auf und generiert Stimulibefehle mit Werten aus dem Wertebereich. In diesem Vorgang setzt die WAIT Leitung auf TRUE. Damit benachrichtigt den Parser, dass er keine neuen Befehle übernehmen kann, weil er beschäftigt ist.

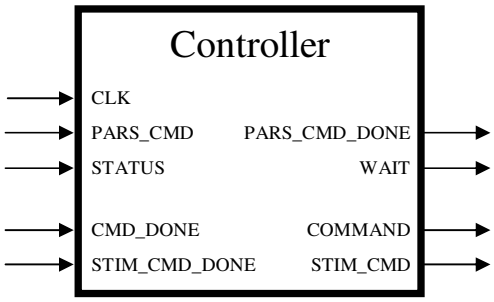

Abbildung 9 – Controller Modul

Schnittstelle

Die Schnittstelle des Controllers ist in der Abbildung 9 dargestellt und besteht aus folgenden Leitungen:

1. **STATUS:** Über diese Leitung wird der Controller über den Zustand des Parsers informiert. Es gibt vier Parserzustände. „I“, wenn der Parser initialisiert wird. In diesem Fall ist der Controller in dem Wartezustand. „C“, wenn ein gültiger Befehl am P_COMMAND Leitung liegt. Der Controller liest den Befehl und verarbeitet ihn. „E“, wenn im Parser ein Fehler aufgetreten ist. Der Controller schickt einen neuen Befehl dem Reportgenerator mit der Fehlermeldung. „R“, wenn die Abarbeitung der Stimulidatei abgeschlossen ist. In diesem Fall wird der Stimuligenerator gestoppt.
2. **PARS_CMD:** Diese Leitung ist der Gegenpool der COMMAND Leitung des Parsers. Sie ist vom Typ String und dient zur Übermittlung der Befehle vom Parser. Die Befehle werden als Zeichenkette übermittelt und sie bestehen aus einem Befehlswort und weiteren Parametern. Ein Teil der Befehle sind dem Controller adressiert und sie werden sofort verarbeitet. Die restlichen Befehle werden den adressierten Modulen weitergeleitet.
3. **PARS_CMD_DONE:** Diese Leitung vom Typ Std_Logic ist der Gegenpool der

CMD_DONE Leitung des Parsers. Wenn der Controller den Befehl des Parsers übernommen hat, sendet dem Parser eine Bestätigung in Form eines Impulses zurück.

4. **WAIT:** Diese Leitung ist vom Typ Boolean. Wenn auf dieser Leitung der Wert TRUE anliegt, wird mit den Parsen so lange gewartet, bis auf diese Leitung vom Controller wieder den Wert FALSE gesetzt wird. Mit dieser Leitung wird signalisiert, dass der Controller intern beschäftigt ist und keine neue Befehle übernehmen kann.
5. **STIM_CMD:** Diese Leitung ist vom Typ String und dient zur Übermittlung der Stimulibefehle an den Stimuligenerator. Die Befehle werden als Zeichenkette übermittelt und sie bestehen aus einem Befehlswort und weiteren Parametern.
6. **STIM_CMD_DONE:** Diese Leitung ist vom Typ Std_Logic. Die Übernahme des Befehls vom Stimuligenerator wird über diese Leitung mit einem Impuls bestätigt.
7. **COMMAND:** Diese Leitung ist vom Typ String und dient zur Übermittlung der Befehle an den anderen Modulen. Die Befehle werden als Zeichenkette übermittelt und sie bestehen aus einem Befehlswort und weiteren Parametern.
8. **CMD_DONE:** Diese Leitung ist vom Typ Std_Logic. Die Übernahme des Befehls von einem Modul wird über diese Leitung mit einem Impuls bestätigt.
9. **CLK:** Bei einer steigenden Flanke (0 → 1) an diesem Eingang, vom Typ Std_Logic, werden die COMMAND und STIM_CMD Leitungen neu gesetzt. Die PARS_CMD_DONE und WAIT Leitungen werden vom CLK nicht beeinflusst.

Parameter

Der Controller kann durch folgende Parameter konfiguriert werden:

***COMMAND_TIMEOUT* `time:`** Die maximal erlaubte Zeit zur Ausführung eines Befehls. Wenn die Zeit überschritten wird, wird der Controller eine Timeout Fehlerbehandlung einleiten. Wenn der Parameter den Wert 0 hat, wird nie ein Timeout Fehler ausgelöst.

Befehle

Der Controller bezieht alle Befehle über die PARS_CMD Leitung. Die Befehle sind Steuerungsbefehle und Stimulibefehle. Es gibt derzeit nur ein Steuerungsbefehl der dem Controller adressiert ist. Dieser setzt den COMMAND_TIMEOUT Parameter des Controllers und kann auch während der Simulation aufgerufen werden.

Syntax:

```
command ::=   CONTROLLER COMMAND_TIMEOUT time
```

Alle anderen Steuerungsbefehle werden an den anderen Modulen weitergeleitet. Die Stimulibefehle werden vom Controller des Stimuligenerators vorbereitet und weitergeleitet. In den meisten Fällen ist keine Vorbereitung notwendig. Eine Vorbereitung ist notwendig, wenn

die Stimulibefehle mit einem Wertbereich gesetzt sind. In diesem Fall wird der Stimulibefehl vom Controller in einzelne Stimulibefehle aufgesplittert und nur dann dem Stimuligenerator weitergeleitet.

Fehlerbehandlung

Ein möglicher Fehler ist das Setzen von fehlerhaften Parametern. Diese treten unmittelbar nach einem Aufruf des CONTROLLER Befehls auf. Vor dem Einsatz der neuen Parameter, werden diese überprüft. Wenn das Timeout keine gültige Zahl ist, werden die Anweisungen ignoriert und der Controller arbeitet wie zuvor weiter. Ein weiterer Fehler ist, wenn ein Timeout der Befehlsverarbeitung ausgelöst wird. In diesem Fall wird der nächste Befehl geholt und die Simulation wird fortgesetzt. Jeder auftretende Fehler bewirkt eine Fehlermeldung in der Reportdatei.

2.2.3 Stimuligenerator

Der Stimuligenerator generiert, anhand der Stimulibefehle, die Stimulidaten für die Eingänge der UUT und des Referenzmoduls, bzw. die Referenzdaten für den Comparator. Der Controller liefert dem Stimuligenerator die Befehle über die STIM_CMD Leitung. Der Stimuligenerator verarbeitet die Stimulibefehle, erstellt die Stimulidaten und sendet jeweils nach jedem Befehl eine Bestätigung dem Controller über die STIM_CMD_DONE Leitung zurück. Die DATA und REF_DATA Leitungen werden mit den Stimulidaten versorgt. Diese repräsentieren symbolisch die Schnittstelle zu UUT und zu Referenzmodul, bzw. zu Comparator. Die DATA Leitungen werden an die UUT und wenn vorhanden an das Referenzmodul angeschlossen. Die REF_DATA Leitungen werden an dem Comparator angeschlossen.

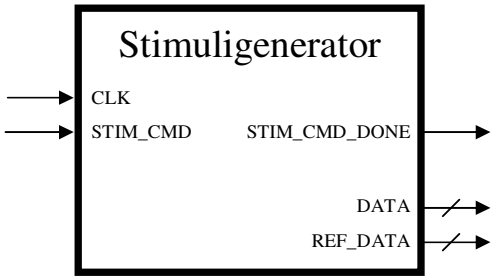

Abbildung 10 – Stimuligenerator Modul

Schnittstelle

Die Schnittstelle des Stimuligenerators ist in der Abbildung 10 dargestellt und besteht aus folgenden Leitungen:

1. **STIM_CMD:** Diese Leitung ist vom Typ String und dient zur Übermittlung der Stimulibefehle vom Controller an den Stimuligenerator. Die Befehle werden als Zeichenkette übermittelt und sie bestehen aus einem Befehlswort und weiteren Parametern.

2. **STIM_CMD_DONE:** Diese Leitung ist vom Typ Std_Logic. Die Ausführung des Befehls wird dem Controller über diese Leitung mit einem Impuls signalisiert.
3. **DATA:** Die DATA Leitungen repräsentieren die Eingangsschnittstelle der UUT und des Referenzmoduls. Sie werden mit Signalen des gleichen Datentyps und der gleichen Breite, wie die Eingänge der UUT und des Referenzmoduls, ersetzt.
4. **REF_DATA:** Die REF_DATA Leitungen stellen die Ausgangsschnittstelle der UUT und des Referenzmoduls dar. Sie werden mit Signalen des gleichen Datentyps und der gleichen Breite, wie die Ausgänge der UUT und des Referenzmoduls, ersetzt. Der Comparator wird über diese Leitungen mit den Referenzdaten versorgt.
5. **CLK:** Bei einer steigenden Flanke (0 → 1) an diesem Eingang, vom Typ Std_Logic, werden die DATA, REF_DATA oder COMMAND Leitungen neu gesetzt. Die STIM_CMD_DONE und WAIT Leitungen werden vom CLK nicht beeinflusst.

Befehle

Der Stimuligenerator erhält keine Steuerungsbefehle, nur Stimulibefehle. Diese werden in Stimulidaten umgewandelt und an die UUT, an das Referenzmodul und an den Comparator weitergegeben.

2.2.4 Taktgenerator

Der Taktgenerator erzeugt eine Impulsfolge am CLK Ausgang. Die Periode und die Dauer der Impulse innerhalb der Taktperiode sind einstellbar. Es können mehrere Impulse pro Periode angegeben werden, um Bausteine, die Mehrpulstakt benötigen, versorgen zu können. Alle Parameter sind während der Laufzeit veränderbar. Die Parameter des Taktgenerators werden durch die Steuerungsbefehle gesetzt. Die Steuerungsbefehle werden über die COMMAND Leitung übermittelt, die Bestätigung, dass der Befehl angenommen wurde, wird über die CMD_DONE Leitung gesendet.

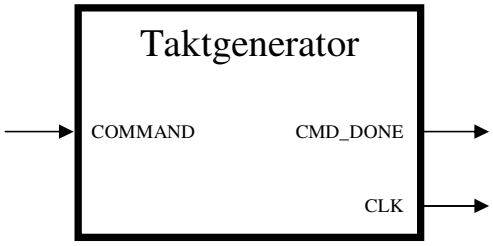

Abbildung 11 – Taktgenerator Modul

Schnittstelle

Die Schnittstelle des Taktgenerators ist in der Abbildung 11 dargestellt und besteht aus folgenden Leitungen:

1. **COMMAND:** Diese Leitung ist vom Typ String und dient zur Übermittlung der Befehle als Zeichenkette vom Controller an den Taktgenerator.

2. **CMD_DONE:** Diese Leitung ist vom Typ Std_Logic. Die Ausführung des Befehls wird dem Controller über diese Leitung mit einem Impuls signalisiert.
3. **CLK:** Diese Leitung ist vom Typ Std_Logic und liefert ein periodisches Taktsignal. Das zeitliche Verhalten ist in der Abbildung 12 dargestellt. Die Zeitangaben der steigenden und fallenden Flanken sind parametrisierbar. Die Parameter werden durch Steuerungsbefehle verändert.

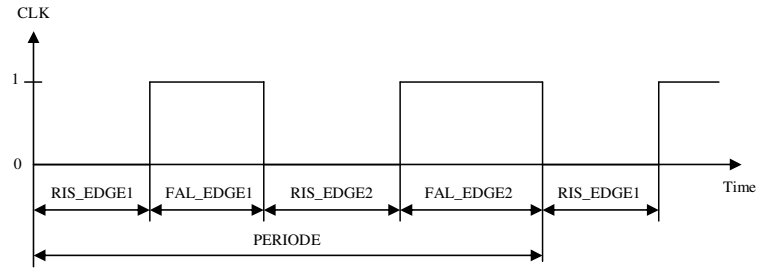

Abbildung 12 – Taktsignal zeitliche Verhalten

Parameter

Der Taktgenerator kann durch den folgenden Parameter konfiguriert werden:

1. ***PERIODE* `time:`** Periodendauer. Sie ist auch während der Simulation änderbar.
2. ***RIS_EDGE1* `time:`** Zeit bis zur steigenden Flanke des ersten Impulses. Sie ist auch während der Simulation änderbar.
3. ***FAL_EDGE1* `time:`** Zeit bis zur fallenden Flanke des ersten Impulses. Sie ist auch während der Simulation änderbar.
4. ***RIS_EDGE2* `time:`** Zeit bis zur steigenden Flanke des zweiten Impulses. Sie ist auch während der Simulation änderbar.
5. ***FAL_EDGE2* `time:`** Zeit bis zur fallenden Flanke des zweiten Impulses. Sie ist auch während der Simulation änderbar.

Befehle

Der Taktgenerator bezieht die Steuerungsbefehle über die COMMAND Leitung und wird durch das Befehlswort CLOCK adressiert. Die Steuerungsbefehle setzen die Parameter des Taktgenerators und können auch während der Simulation aufgerufen werden. Um unerwünschte Nebeneffekte bei dem Wechsel der Zeitangaben zu vermeiden, erfolgt die Umschaltung erst nach beendetem Taktzyklus.

Syntax:

```
command ::=   CLOCK command-element{,command-element}
command-element ::=   PERIODE time
                    | RIS_EDGE1 time
                    | FAL_EDGE1 time
                    | RIS_EDGE2 time
                    | FAL_EDGE2 time
```

Fehlerbehandlung

Die wahrscheinlichsten Fehler sind das Setzen von fehlerhaften Parametern. Diese treten unmittelbar nach einem Aufruf des CLOCK Befehls auf. Vor dem Einsatz der neuen Parameter, werden diese überprüft. Wenn die Summe der RIS_EDGE und FAL_EDGE Zeiten mehr als die angegebene Periode ist, werden die Anweisungen ignoriert und der Taktgenerator arbeitet wie zuvor weiter. Jeder auftretende Fehler bewirkt eine Fehlermeldung in der Reportdatei.

2.2.5 Comparator

Der Comparator vergleicht die Ausgänge der UUT mit den Referenzdaten. Das Ergebnis wird an den Reportgenerator weitergeleitet. Die Referenzdaten können vom Stimuligenerator oder vom Referenzmodul stammen, wobei die Werte vom Stimuligenerator favorisiert werden. Die Parameter des Comparators werden durch die Steuerungsbefehle gesetzt. Die Steuerungsbefehle werden über die COMMAND Leitung übermittelt, die Bestätigung, dass der Befehl angenommen wurde, wird über die CMD_DONE Leitung gesendet.

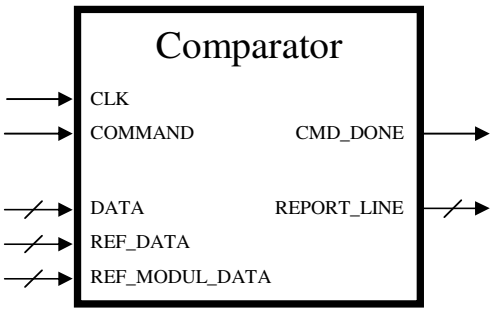

Abbildung 13 – Comparator Modul

Schnittstelle

Die Schnittstelle des Comparators ist in der Abbildung 13 dargestellt und besteht aus folgenden Leitungen:

1. **COMMAND:** Diese Leitung ist vom Typ String und dient zur Übermittlung der Befehle als Zeichenkette vom Controller an den Comparator.
2. **CMD_DONE:** Diese Leitung ist vom Typ Std_Logic. Die Ausführung des Befehls wird dem Controller über diese Leitung mit einem Impuls signalisiert.
3. **CLK:** Taktsignal für Datensynchronisation.
4. **DATA:** Die Ausgangssignale der UUT werden an diesen Leitungen gelegt.
5. **REF_MODUL_DATA:** Die Ausgangssignale des Referenzmoduls werden an diesen Leitungen gelegt. Wenn kein Referenzmodul angeschlossen ist, werden diese Leitungen nicht verwendet.
6. **REF_DATA:** Die Referenzdaten des Stimuligenerators werden an diesen Leitungen gelegt.

7. **REPORT_LINE:** Die Ergebnisse des Vergleichsvorgangs werden an diesem Ausgang im String Format ausgegeben.

Parameter

Der Parser kann durch den folgenden Parameter konfiguriert werden:

TOLERANCE **`time`***:* Toleranzzeit, in der die Signale unterschiedlich sein dürfen.

Befehle

Der Reportgenerator bezieht die Steuerungsbefehle über die COMMAND Leitung und wird durch das Befehlswort REPORT adressiert. Die Steuerungsbefehle setzen die Parameter des Reportgenerators und können auch während der Simulation aufgerufen werden.

Syntax:

```
command ::=   COMPARATOR TOLERANCE time
```

Fehlerbehandlung

Die wahrscheinlichsten Fehler sind das Setzen von fehlerhaften Parametern. Diese treten unmittelbar nach einem Aufruf des COMPARATOR Befehls auf. Vor dem Einsatz der neuen Parameter, werden diese überprüft. Wenn die Toleranz keine gültige Zeit ist, werden die Anweisungen ignoriert und der Comparator arbeitet wie zuvor weiter. Jeder auftretende Fehler bewirkt eine Fehlermeldung in der Reportdatei.

2.2.6 Reportgenerator

Der Reportgenerator sammelt Informationen aus der Testbench und erstellt eine Reportdatei. Die Informationen stammen hauptsächlich vom Comparator, die über die REPORT_LINE Leitung ankommen. Die Parameter des Comparators werden durch die Steuerungsbefehle gesetzt. Die Steuerungsbefehle werden über die COMMAND Leitung übermittelt, die Bestätigung, dass der Befehl angenommen wurde, wird über die CMD_DONE Leitung gesendet.

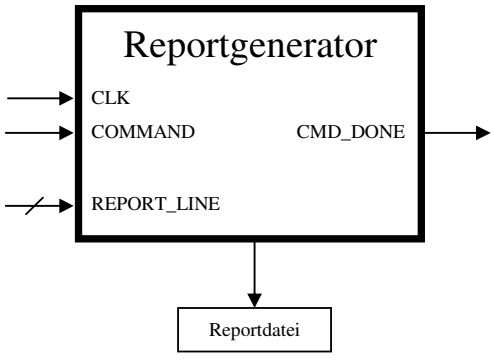

Abbildung 14 – Reportgenerator Modul

Schnittstelle

Die Schnittstelle des Reportgenerators ist in der Abbildung 14 dargestellt und besteht aus folgenden Leitungen:

1. **COMMAND:** Diese Leitung ist vom Typ String und dient zur Übermittlung der Befehle als Zeichenkette vom Controller an den Reportgenerator.
2. **CMD_DONE:** Diese Leitung ist vom Typ Std_Logic. Die Ausführung des Befehls wird dem Controller über diese Leitung mit einem Impuls signalisiert.
3. **CLK:** Taktsignal für Datensynchronisation.
4. **REPORT_LINE:** Über diese Leitung werden die Nachrichten im String Format den Reportgenerator übermittelt.

Reportdatei

Der Reportgenerator schreibt die eintreffenden Nachrichten in die Reportdatei. Diese ist eine Textdatei und kann mit Texteditoren gelesen werden. Die Formatierung des Inhalts ist von der Implementierung des Reportgenerators abhängig. Das Resultat der Simulation wird in der Reportdatei festgehalten.

Parameter

Der Parser kann durch den folgenden Parameter konfiguriert werden:

1. ***REPORTDATEI*** **`string:`** Name der Reportdatei.
2. ***LOGLEVEL*** **`integer:`** Loglevel des Reportgenerators.

Befehle

Der Reportgenerator bezieht die Steuerungsbefehle über die COMMAND Leitung und wird durch das Befehlswort REPORT adressiert. Die Steuerungsbefehle setzen die Parameter des Reportgenerators und können auch während der Simulation aufgerufen werden.

Syntax:

```
command ::=   REPORT command-element{,command-element}
command-element ::=   REPORTDATEI string
                      | LOGLEVEL integer
```

Fehlerbehandlung

Die wahrscheinlichsten Fehler sind das Setzen von fehlerhaften Parametern. Diese treten unmittelbar nach einem Aufruf des REPORT Befehls auf. Vor dem Einsatz der neuen Parameter, werden diese überprüft. Wenn die Reportdatei nicht angelegt werden kann oder wenn der Loglevel unbekannt ist, werden die Anweisungen ignoriert und der Reportgenerator arbeitet wie zuvor weiter.

2.3 Testbench-Struktur ohne Controller

In diesem Abschnitt präsentierte Testbench ist für kleine bis mittelgroße ASIC Entwicklungen geeignet. Die Testbench arbeitet nur mit Stimulibefehlen. Die Konfiguration der Module wird am Anfang der Simulation mit den Parametern aus der Konfigurationsdatei durchgeführt. Das Blockschaltbild der Testbench ist in der Abbildung 15 präsentiert und besteht aus folgenden Modulen:

1. **Stimuligenerator:** parst die Stimulidatei und stellt die Stimulidaten der UUT und dem Referenzmodul, bzw. die Referenzdaten dem Comparator zur Verfügung.
2. **Taktgenerator:** liefert den Systemtakt.
3. **Comparator mit Reportgenerator:** wertet die Ausgabedaten der UUT aufgrund der Referenzdaten aus und speichert die Ergebnisse in eine Textdatei ab.
4. **UUT:** der zu testenden Einheit.
5. **Referenzmodul:** liefert die Solldaten.

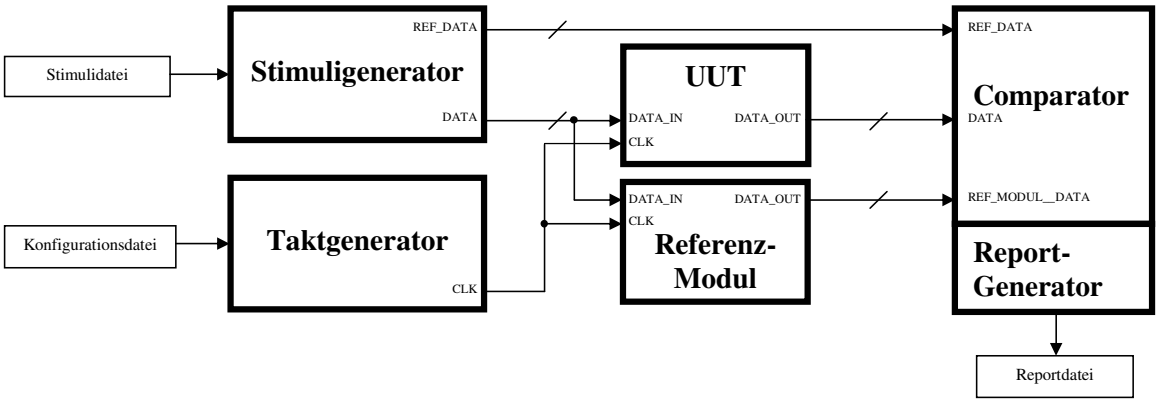

Abbildung 15 – Testbench-Struktur ohne Controller

Der Stimuligenerator ist für die Versorgung der UUT und des Referenzmoduls mit den Stimulidaten, bzw. für die Versorgung des Comparators mit den Referenzdaten zuständig. Die Stimulidaten und die Referenzdaten werden aus den Stimulibefehlen generiert, welche in der Stimulidatei gespeichert sind. Die Stimulibefehle werden sequentiell aus der Stimulidatei gelesen, in Stimulidaten und in Referenzdaten konvertiert und anschließend an die DATA, bzw. REF_DATA Leitungen geführt. Der Comparator übernimmt die Ausgabedaten der UUT und des Referenzmoduls, bzw. die Referenzdaten vom Stimuligenerator. Die Werte werden verglichen und das Resultat in eine Reportdatei geschrieben. Der Taktgenerator liefert den Takt der UUT und dem Referenzmodul. Die UUT, bzw. das Referenzmodul werden nicht beschrieben. Sie sind die Resultate der ASIC Entwicklungen und haben eine klar definierte Schnittstelle nach Außen.

2.4 Stimulibefehl Modelle

Die Stimulibefehle können verschiedene Modelle haben. In diesem Abschnitt werden drei

Varianten präsentiert.

2.4.1 Stimulibefehl Modell 1

Der Stimulibefehl ist als eine Bitfolge dargestellt. Die Bitfolge besteht aus den Stimulidaten der Eingangsleitungen und aus den zu erwartenden Werten (Referenzwerte) der Ausgangsleitungen der UUT. Die zu erwartenden Werte sind mit einem „R“ gekennzeichnet. Das Kennzeichen wurde wegen die IN/OUT Signalen eingeführt, da diese Signale abwechselnd entweder mit Stimulidaten versorgt oder mit den zu erwartenden Werten verglichen werden. An der letzten Stelle des Befehls wird die Dauer des Befehls mitgespeichert. Für jeden Pegelwechsel eines Signals ist ein neuer Befehl notwendig. Der Vorteil solcher Befehle ist, dass sie sehr schnell verarbeitet werden können. Für zeitkritische Entwicklungen ist eine ideale Lösung. Der Nachteil ist, dass die langen Bitfolgen schwer lesbar sind.

Beispiel: Der zeitliche Ablauf der Signale aus der Abbildung 16 wird in der Tabelle 2 als eine Sequenz von Stimulibefehlen dargestellt.

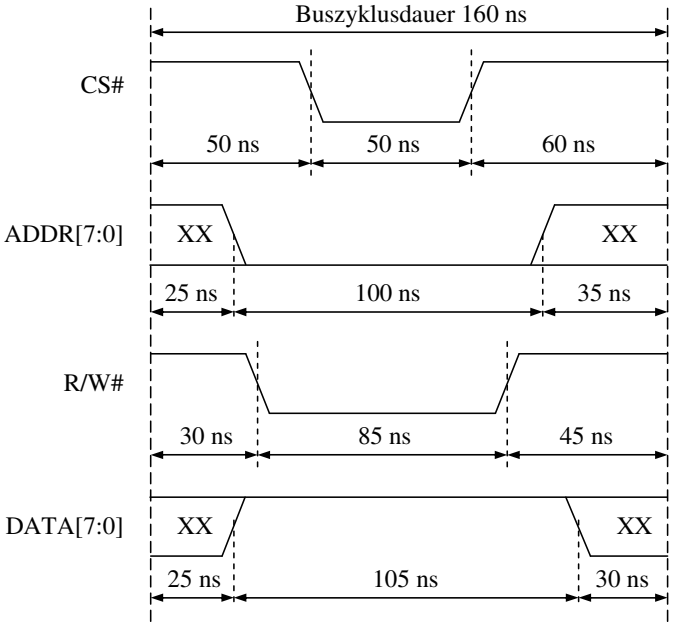

Abbildung 16 – Signalabläufe einer Buszyklus

CS	ADDR	RW	DATA	Time
1	XXXXXXXX	1	XXXXXXXX	25ns
1	00000000	1	11111111	5ns
1	00000000	0	11111111	20ns
0	00000000	0	11111111	50ns
1	00000000	0	11111111	15ns
1	00000000	1	11111111	10ns
1	XXXXXXXX	1	11111111	5ns
1	XXXXXXXX	1	XXXXXXXX	30ns

Tabelle 2 – Stimulibefehle der Variante 1

Nach jedem Pegelwechsel des Timingdiagramms wird ein neuer Befehl angelegt. Die

Bitpositionen der Signale bleiben immer an der selber Stelle. Im Beispiel wird die CS Leitung immer mit dem Wert aus der Position 1 der Bitfolge versorgt. Wenn die Zeitangabe am Ende des Befehls nicht definiert wurde, dann wird sofort der nächste Befehl ausgeführt.

2.4.2 Stimulibefehl Modell 2

Der Stimulibefehl besteht aus einem Signalnamen und aus einer Bitfolge. Der Signalname ist die Bezeichnung einer Leitung der UUT. Wenn die Leitung eine Eingangsleitung ist, besteht die Bitfolge aus den Stimulidaten der Eingangsleitung. Wenn die Leitung eine Ausgangsleitung ist, besteht die Bitfolge aus den zu erwartenden Werten der Ausgangsleitung. Die zu erwartenden Werte sind mit einem „R“ Zeichen gekennzeichnet. Das Kennzeichen wurde wegen der IN/OUT Signalen eingeführt, da diese Signale abwechselnd entweder mit Stimulidaten versorgt oder mit den zu erwartenden Werten verglichen werden. Die Befehle werden sequenziell ausgeführt. Für jeden Pegelwechsel eines Signals ist ein neuer Befehl notwendig. Zwischen zwei Pegelwechsel wird die Befehlverarbeitung mit einem WAIT Statement angehalten. Der Vorteil solcher Befehle ist, dass sie sich nur auf einzelne Signale beziehen, nur bei Pegelwechsel der einzelnen Signale aufgerufen werden sollen. Der Nachteil ist, dass die langen Befehlslisten schwer lesbar sind.

Beispiel: Der zeitliche Ablauf der Signale aus der Abbildung 16 wird in der Tabelle 3 als eine Sequenz von Stimulibefehlen dargestellt.

Signal	**Wert**
CS	1
ADDR	XXXXXXXX
RW	1
DATA	XXXXXXXX
WAIT	25ns
ADDR	00000000
DATA	11111111
WAIT	5ns
RW	0
WAIT	20ns
CS	0
WAIT	50ns
CS	1
WAIT	15ns
RW	1
WAIT	10ns
ADDR	XXXXXXXX
WAIT	5ns
DATA	XXXXXXXX
WAIT	30ns

Tabelle 3 – Stimulibefehle der Variante 2

Nach jeden Pegelwechseln des Timingdiagramms wird ein neuer Befehl angelegt. Der Befehl beinhaltet nur ein einziges Signal. Wenn gleichzeitig mehrere Pegelwechsel stattfinden, werden dementsprechend mehrere Befehle erstellt.

2.4.3 Stimulibefehl Modell 3

Der Stimulibefehl wird als einen Pseudobefehl dargestellt. Das Erzeugen von Stimulidaten aus dem Befehl ist wesentlich komplexer, als im Modell 1 und 2. Jeder Befehldefinition wird mit einem HDL Prozess modelliert. Der Vorteil solcher Befehle ist, dass sie sehr kompakt und leicht zu lesen sind. Der Nachteil ist, dass die Verarbeitung der Befehle viel Rechenleistung benötigt.

Beispiel: `ACC 0,0x92`

2.5 Testbench-Generator Datenmodell

Im folgenden Abschnitt wird das Datenmodell des Testbench-Generators definiert. Ein wichtiger Aspekt des Modells ist die redundanten Informationen zu vermeiden. Eine nachträgliche Änderung soll auf alle abhängigen Objekte Auswirkung haben.

Die wichtigsten Business Objekte des Modells sind die Signaldefinition, die Signalwertdefinition, die Befehlsdefinition, die Programmdefinition und die Moduldefinition. Die Signaldefinition enthält alle Eigenschaften, wie Signalname, Signalbreite und die Timinginformationen eines Signals. Die Timinginformationen sind durch die Signalwertdefinitionen modelliert. Für eine Zeitspanne zwischen zwei möglichen Flanken eines Signals wird ein Signalwert definiert. Dieser ist durch die Periode der Zeitspanne und der Signalwert charakterisiert. Die Befehlsdefinition fasst jene Signaldefinitionen zusammen, die betroffen sind um eine gewünschte Funktion der UUT anzusteuern. Die Signalwerte können auch auf der Befehldefinitionsebene modifiziert werden. Sie überschreiben den Signalwert der Signaldefinition. Um das Überschreiben nur auf der jeweiligen Befehlsdefinition zu beschränken, wird nur eine Signaldefinitionsinstanz dem Befehl zugewiesen. Die Dokumentation der Signale auf der Befehlsdefinitionsebene ist damit auch gewährleistet. Es wird nur die Signaldefinitionsinstanz dokumentiert. Die Programmdefinition fasst Befehlsdefinitionen zusammen. Die Signalwerte können auch auf der Programmdefinitionsebene modifiziert werden. Sie überschreiben den Signalwert der Befehlsdefinition. Um das Überschreiben nur auf der jeweiligen Programmdefinition zu beschränken, wird nur eine Befehlsdefinitionsinstanz dem Programm zugewiesen. Die Dokumentation der Befehle auf der Programmdefinitionsebene ist damit auch gewährleistet. Es wird nur die Befehlsdefinitionsinstanz dokumentiert. Die Moduldefinition fasst alle Signaldefinitionen zusammen womit die Verbindung zwischen die Testbench und die UUT realisiert werden soll. Die Mapping Informationen zwischen die Testbench und die UUT Leitungen werden auch abgespeichert.

Eine grafische Darstellung der Abhängigkeiten ist in das ER (Entity Relationship) Diagramm des Datenmodells in der Abbildung 17 dargestellt.

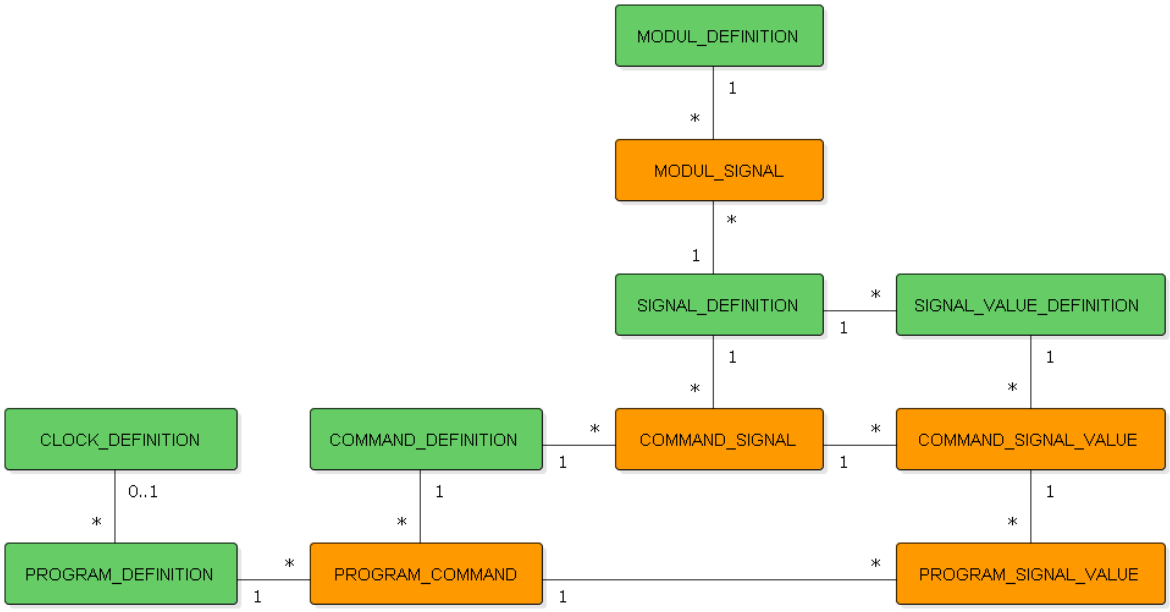

Abbildung 17 – ER Diagramm des Datenmodells

Die Beziehungen sind mit Linien dargestellt. Zwischen den Signaldefinitionen (SIGNAL_DEFINITION) und den Signalwertdefinitionen (SIGNAL_VALUE_DEFINITION) ist eine 1 zu n Beziehung definiert, also ein Signal kann mehrere Signalwerte besitzen. Zwischen den Befehlsdefinitionen (COMMAND_DEFINITION) und den Signaldefinitionen ist eine m zu n Beziehung definiert, die durch die Befehlssignale (COMMAND_SIGNAL) aufgelöst wurde. Ein Befehl kann aus mehreren Signalen bestehen, genauso ein Signal kann zu mehreren Befehlen zugewiesen werden. Ein solches Befehlsignal kann den Signalwert überschreiben und der überschriebene Wert wird in der Befehlsignalwert (COMMAND_SIGNAL_VALUE) abgespeichert. Zwischen den Programmdefinitionen (PROGRAM_DEFINITION) und den Befehlsdefinitionen ist eine m zu n Beziehung definiert, die durch die Programmbefehle (PROGRAM_COMMAND) aufgelöst wurde. Ein Programm kann aus mehreren Befehlen bestehen, genauso ein Befehl kann zu mehreren Programmen gehören. Ein Programmbefehl kann den Signalwert überschreiben und der überschriebene Wert wird in den Programmsignalwert (PROGRAM_SIGNAL_VALUE) abgespeichert. Eine Programmdefinition kann eine Taktsignal Definition (CLOCK_DEFINITION) besitzen. Die Moduldefinitionen (MODUL_DEFINITION) sind mit den Signaldefinitionen auch in eine m zu n Beziehung, die durch Modulsignale (MODUL_SIGNAL) aufgelöst werden.

Das UML Klassendiagramm des Datenmodells ist in der Abbildung 18 dargestellt.

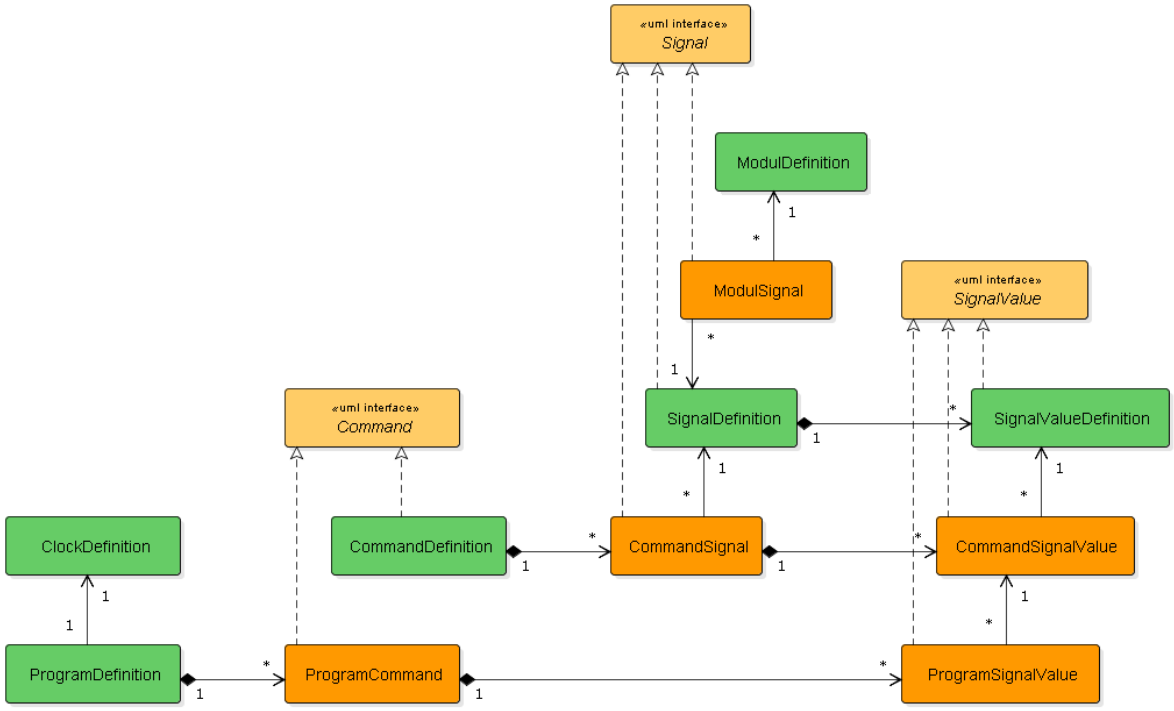

Abbildung 18 – UML Class Diagramm des Datenmodells

Die Beziehungen zwischen den Klassen können auf die Beziehungen des ER Modells zurückgeführt werden. Durch die Einführung der Interfaceelemente kann das Modell verfeinert werden. Das Signal-Interface deklariert alle Methoden der Signaldefinition und der Befehlssignal (CommandSignal). Damit haben die Signaldefinition Klasse und das Befehlssignal Klasse die gleiche Signatur nach Außen. Das SignalValue Interface deklariert die Methoden der Signalwertdefinition (SignalValueDefinition), des Befehlssignalwertes (CommandSignalValue) und des Programmsignalwertes (ProgramSignalValue) und das Befehl-Interface (Command) deklariert die Methoden der Befehlsdefinition (CommandDefinition) und des Programmbefehls (ProgramCommand).

Die Spezifikationen der Business Objekte werden in den folgenden Unterkapiteln, teilweise auch mit Timing Diagramm Beispielen, näher beschrieben.

2.5.1 Signaldefinition

Ein Signal definiert die Schnittstelle zwischen der Testbench und der UUT. Zur Definition eines Signals oder Signalbusses werden im Allgemeinen mehrere Parameter benötigt. Die einzelnen Parameter werden anhand der Abbildung 19 etwas näher illustriert.

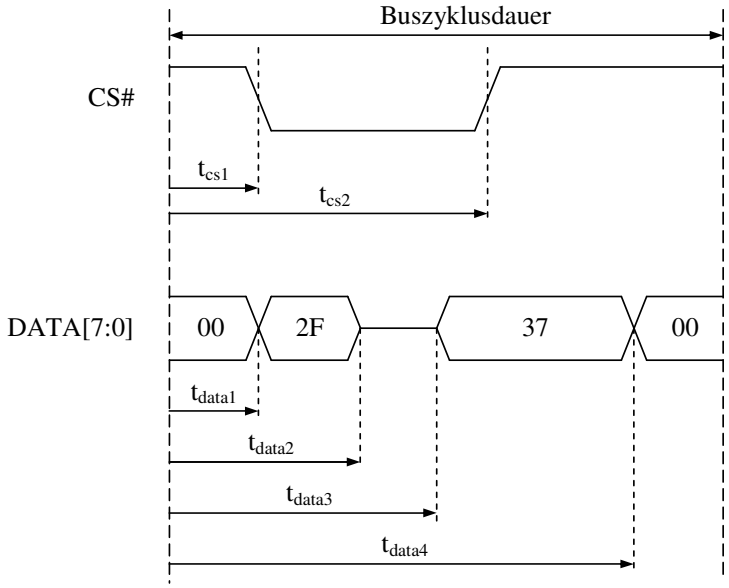

Abbildung 19 – Signaldefinition

1. **Signalname:** Bezeichnet das zu charakterisierende Signal (z.B. CS#, DATA, usw.). Es soll eine eindeutige Bezeichnung des Signals sein.
2. **Signalbreite:** Definiert die Anzahl der Leitungen eines Signals (1 Bit, 8 Bit, usw.). Dadurch wird sich unterscheiden, ob es sich um eine einzige Leitung oder um einen Bus handelt und aus wie vielen Letztere besteht.
3. **Signaltyp:** Gibt an ob das Signal ein Eingang (IN Signal), ein Ausgang (OUT Signal) oder ein Eingang/Ausgang (IN/OUT Signal) der UUT ist. Die IN/OUT Signale sind zu vermeiden. Sie benötigen eine sehr aufmerksame Behandlung. Man muss sehr genau wissen, wann als Eingang, bzw. als Ausgang tätig sind. Das hängt oft von anderen Signalen ab. Wenn aber von keinem externen Signal abhängig ist, sondern z.B. von einem internen Zustand der UUT, ist die Umschaltzeitpunkt Außen nicht sichtbar. Also in manchen Fällen ist es ratsam ein IN/OUT Signal der UUT für Testzwecke in einem IN Signal und OUT Signal aufzuteilen.
4. **Initial Wert:** Die Signale werden vor einer Simulation mit diesem Wert initialisiert.
5. **Default Wert:** Wird eingesetzt, wenn der Signalpegel für eine Zeitspanne undefiniert bleibt. Es wird meistens für die inaktiven Zustände verwendet. Eine Konsistenzprüfung ist notwendig, um Default Werte, die die Signalbreite nicht abdeckt, ausfindig zu machen.
6. **Zykluszeiten:** Mit Hilfe der Zykluszeiten werden alle Zeitpunkte innerhalb eines Buszyklus, zu denen ein Pegelwechsel der betrachteten Leitung erfolgt, genau festgelegt. Anhand dieser Zeiten wird definiert, wann ein Signal aktiv, bzw. inaktiv ist.

Wenn die Testbench-Strukturen weitere Parameter benötigen, kann die Parameterliste erweitert werden. Die neuen Parameterdefinitionen werden in eine XML Konfigurationsdatei

erfasst. Die Übersetzung des Testbench-Generators ist nicht notwendig.

Die Plausibilitätsprüfung der Signaldefinition wird in einer Business Funktion abgebildet. Diese überprüft den Signalnamen, bzw. den Wert und die Breite des Default-Wertes. Ein Signalname darf nur einmal vorkommen. Der Wert und die Breite des Default-Wertes sollen konsistent sein.

Die Zykluszeiten teilen das Signal in Signalbereiche auf. Ein Signal ist eine Verkettung von Signalbereiche, wobei ein Signalbereich durch eine Zeitspanne und Signalzustand charakterisiert ist. Solche Signalbereiche werden in dieser Arbeit durch einen Signalwert modelliert.

2.5.2 Signalwertdefinition

Die Signalwerte modellieren den Ablauf eines Signals. Ein Signal besitzt eine Aneinanderkettung von Signalwerten. Die Eigenschaften der Signalwerte definieren das Timing des Signals. Zur Definition eines Signalwertes werden folgende Parameter benötigt:

1. **Signalwertname:** Bezeichnet das zu charakterisierende Signalwert (z.B. tlx1, tlx2, usw. aus Abbildung 19). Es soll eine eindeutige Bezeichnung des Signalwertes innerhalb eines Signals sein.
2. **Dauer:** Gibt eine Mindestdauer des Signalwertes an. Wenn die Dauer nicht angegeben ist, kann von 0 bis zu einem Buszyklus variieren. Welchen Wert dann annimmt, ist von den anderen Signalwerten abhängig.
3. **Wert:** Gibt den Signalpegel für die angegebene Dauer an. Eine Konsistenzprüfung ist notwendig, um die Werte, die die Signalbreite nicht abdecken, ausfindig zu machen.
4. **IN/OUT:** Wenn ein Signal, als Ausgangssignal einer UUT definiert wird, dient der angegebene Signalpegel als Vergleichswert. Ansonsten ist als Stimulidaten zu sehen.
5. **Dehnbar:** Durch diesen Parameter wird angegeben, ob die angegebene Mindestdauer erhöht werden darf oder nicht. Wenn ein Befehl länger dauert, als die Summe der Mindestdauer aller Signalwerte eines Signals, dann werden die ausdehnbare Signalwerte so ausgedehnt, dass die Dauer des Befehls (Buszyklus) erreicht wird. Wenn die Mindestdauer nicht angegeben wurde, wird der Signalwert immer ausgedehnt, also wird dieser Parameter nicht berücksichtigt.

Wenn die Testbench-Strukturen weitere Parameter benötigen, kann die Parameterliste erweitert werden. Die neuen Parameterdefinitionen werden in eine XML Konfigurationsdatei erfasst. Die Übersetzung der Testbench-Generator ist nicht notwendig.

Die Plausibilitätsprüfung der Signalwertdefinition wird in einer Business Funktion abgebildet. Diese überprüft den Signalwertnamen und die Breite des Signalwertes. Der Signalwertname darf in eine Signaldefinition nur einmal vorkommen. Der Wert des Signalwertes darf nicht

breiter sein, als die Signalbreite.

Beispiel: In der Abbildung 20 ist ein Signal mit drei Signalwerten definiert, „Setup“, „Valid“ und „Finish“. „Setup“ und „Finish“ sind nicht dehnbar, „Valid“ schon. Wenn der Buszyklus von 47 ns auf 62 ns geändert wird, bleiben die Dauer von „Setup“ und „Finish“ unverändert, also 10 ns und 17 ns, aber die Dauer von „Valid“ wird von 20 ns auf 35 ns erhöht.

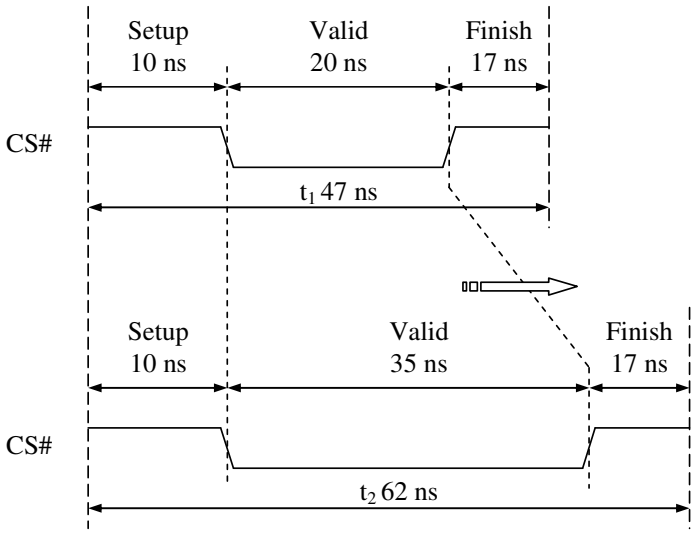

Abbildung 20 – Signalwert Definition

Die Signalwerte werden in der Signaldefinition festgelegt. Die Parameter der Signalwerte, die nicht unbedingt bei der Signaldefinition gesetzt werden müssen, sind der Wert und die Dauer. Wenn der Wert bei der Signaldefinition nicht gesetzt wird, kann noch beim Einbinden der Signale in die Befehle festgelegt werden. Wenn diese noch immer nicht festgelegt wurde, kann noch bei der Befehlsaneinanderkettung, also Programmerstellung festgelegt werden. Wenn nie festgelegt wird, dann wird der Default Wert des Signals als Wert genommen. Die Dauer des Signalwertes kann noch beim Einbinden der Signale in die Befehle modifiziert werden. Die IN/OUT Signale besitzen auch IN/OUT Signalwerte. Die Auswahl, ob IN oder OUT ist, kann noch beim Einbinden der Signale in die Befehle modifiziert werden.

Die Aneinanderkettung von Signalwerten ist in der Tabelle 4 dargestellt. Das Signal SIG1 hat eine Busbreite von ANZ1 und besitzt n Signalwerte. Der erste Signalwert hat die Dauer t_{11} und den Wert w_{11}. Der zweite Signalwert hat die Dauer t_{12} und den Wert w_{12}. Der letzte Signalwert hat die Dauer t_{1n} und den Wert w_{1n}. Die Dauer t, die Werte w und die Parameter IN/OUT müssen nicht gesetzt werden. Innerhalb von einem Signal ist nur ein dehnbarer Signalwert definierbar.

Signalname	**Signalbreite**	**Signalwerte**		
		Dauer	**Wert**	**IN/OUT**
SIG_1	ANZ_1	t_{11}	w_{11}	IN/OUT
		t_{12}	w_{12}	IN/OUT
		...	...	...
		t_{1n}	w_{1n}	IN/OUT

...	...	...	...	...
SIG_m	ANZ_m	t_{m1}	w_{m1}	IN/OUT
		...	...	...
		t_{mk}	w_{mk}	IN/OUT

Tabelle 4 – Aneinanderkettung von Signalwerten

Beispiel: In der Abbildung 20 wurde das IN Signal CS# dargestellt. Die Signalwertaneinanderkettung ist in der Tabelle 5 dargestellt und besteht aus drei Signalwerten. Der erste Signalwert hat eine Dauer von 10 ns und den Wert 0, der zweite Signalwert hat eine Dauer von 20 ns und den Wert 1 und der dritte Signalwert hat eine Dauer von 17 ns und den Wert 0.

Signalname	**Signalbreite**	**Signalwerte**		
		Dauer	**Wert**	**IN/OUT**
CS#	1	10 ns	1	IN
		20 ns	0	IN
		17 ns	1	IN

Tabelle 5 – Beispiel mit Signalwerteaneinanderkettung

Beispiel: In der Abbildung 21 wurde das IN/OUT Signal DATA dargestellt. Die Signalwertaneinanderkettung ist in der Tabelle 6 dargestellt und besteht aus fünf Signalwerten. Der erste Signalwert hat eine Dauer von 10 ns und den Wert 0x00. Der zweite Signalwert hat eine Dauer von 10 ns und den Wert 0x2F. Der dritte Signalwert hat eine Dauer von 10 ns und den Wert ZZh. Der vierte Signalwert hat eine Dauer von 18 ns und den Wert VAR, also bleibt vorerst noch nicht definiert und der fünfte Signalwert hat eine Dauer von 10 ns und den Wert 0x00. Der dritte Signalwert kann auch als OUT definiert werden. Wenn als OUT definiert wird, ist es möglich einen Referenzwert anzugeben, wie in diesem Beispiel 0x0FF. Dieser Referenzwert wird am Ausgang der UUT erwartet.

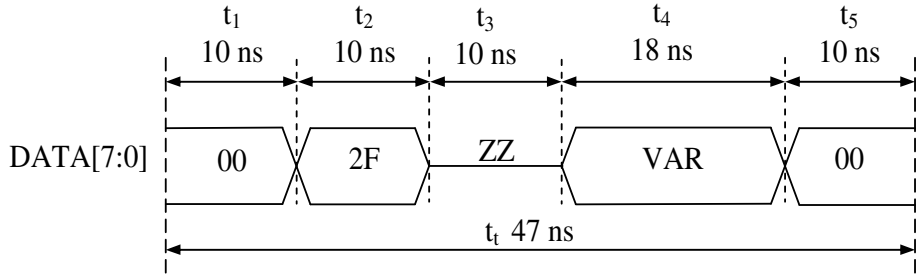

Abbildung 21 – Ablauf der DATA Signal

Signalname	Signalbreite	Signalwerte		
		Dauer	Wert	IN/OUT
DATA	8	10 ns	0x000	IN
		10 ns	0x02F	IN
		10 ns	0x0ZZ	OUT
		18 ns	VAR	IN
		10 ns	0x000	IN

Tabelle 6 – Beispiel mit Signalwerteaneinanderkettung

2.5.3 Befehlsdefinition

Ein Befehl ist die Zusammenlegung der Signale. Für jeden einzelnen zu definierenden Befehl müssen für die Dauer eines Befehlszyklusses jeweils alle Signale getrieben werden. Bereits in der Signaldefinitionsschicht wird das zeitliche Signalverhalten durch Signalwerte definiert. Das zeitliche Signalverhalten kann noch in der Befehlschicht modifiziert werden. Wenn der Pegel der Signalwerte in der Signaldefinitionsschicht nicht gesetzt wird, kann als Befehlsvariable oder als Programmvariable verwendet werden. Wenn die Dauer der Signalwerte in der Signaldefinitionsschicht nicht gesetzt wird, wird als Befehlsvariable verwendet. In der Tabelle 7 ist ein Befehl BEF_1 mit den Signalen SIG_1 bis SIG_m definiert. Das zeitliche Signalverhalten der Signale, wie z.B. SIG_1 wird durch eine Reihe von Signalwerten definiert. Die Signalzykluszeiten t_{11}, ..., t_{1n} werden teilweise in der Signalsdefinitionsschicht definiert und der Rest in der Befehlsdefinitionsschicht. Die Werte w_{11}, ..., w_{1n} werden teilweise in der Signalsdefinitionsschicht definiert und der Rest in der Befehlsdefinitionsschicht oder in der Programmschicht.

Befehlname	Signalname	Signalwerte		
		Dauer	Wert	IN/OUT
BEF_1	SIG_1	t_{11}	w_{11}	IN/OUT
		...	...	...
		t_{1n}	w_{1n}	IN/OUT
	...	...	...	...
	SIG_m	t_{m1}	w_{m1}	IN/OUT
		...	...	...
		t_{mk}	w_{mk}	IN/OUT

Tabelle 7 – Befehlsdefinition

Beispiel: Um den Akkumulator Register einer UUT zu beschreiben, wird der Befehl ACW VAR definiert. Das Timing des Befehls ist in der Abbildung 22 dargestellt. Für diesen Befehl werden alle Adresssignale während des aktiven Bereichs auf den fixen Wert des Akkumulator

Registers eingestellt. Weiters wird das Chip Select Signal (CS#) zum aktiven Zeitpunkt fix auf logisch 0, bzw. das R/W# Signal auf logisch 0 gelegt. Das Label ACW dient zur Kennzeichnung des Befehls und muss sich von allen anderen Befehlen in der Bezeichnung unterscheiden.

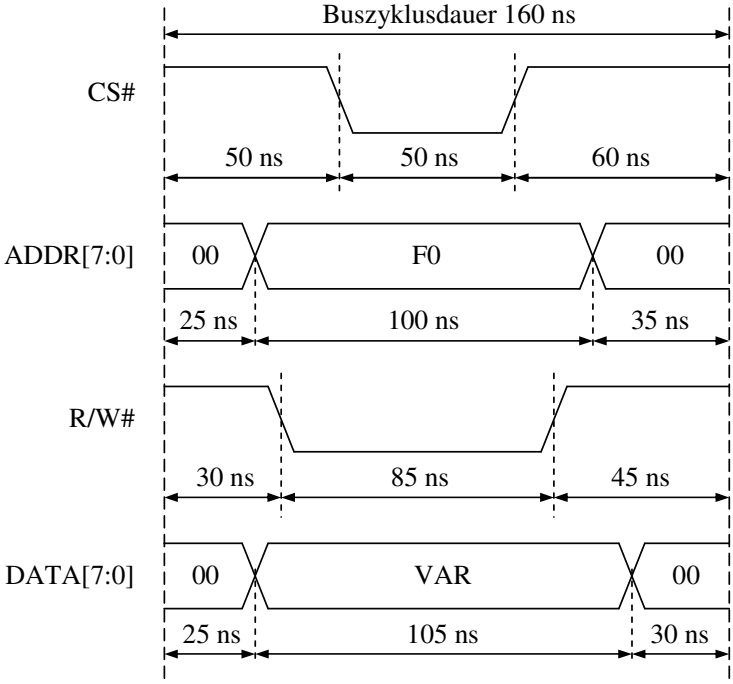

Abbildung 22 – Signalabläufe des ACW Befehls

Der Parameter VAR stellt den aktiven Teil der DATA Signalleitungen dar. Dieser wird bei der Befehlsaneinanderkettung festgelegt. Für den Befehl ACW VAR würde somit die Festlegung aus der Tabelle 8 entstehen.

Signalname	Wert	IN/OUT
CS#	0	IN
ADDR	0x0F0	IN
R/W#	0	IN
DATA	VAR	IN

Tabelle 8 – ACW Befehlsdefinition

Der CS# Signal ist aus drei Signalwerten zusammengesetzt, die in der Signaldefinitionsschicht definiert wurden. Der erste Signalwert hat die Dauer von 50 ns und wurde in der Signaldefinitionsschicht auf den Wert 1 gesetzt. Der zweite Signalwert hat eine Dauer von ebenfalls 50 ns und wurde in der Befehlsdefinitionsschicht auf den Wert 0 gesetzt. Der dritte Signalwert mit der Dauer von 60 ns wurde in der Signaldefinitionsschicht auf den Wert 1 gesetzt.

Der Adressbus ADDR ist ebenfalls aus drei Signalwerten zusammengesetzt. Der erste Signalwert hat die Dauer von 25 ns und wurde in der Signaldefinitionsschicht auf den Wert 0x00 gesetzt. Der zweite Signalwert hat eine Dauer von 50 ns und wurde in der

Befehlsdefinitionsschicht auf den Wert 0x0F0 gesetzt. Der dritte Signalwert mit der Dauer von 35 ns wurde in der Signaldefinitionsschicht auf den Wert 0x00 gesetzt.

Der R/W# Signal besteht auch aus drei Signalwerten. Der erste Signalwert hat die Dauer von 30 ns und wurde in der Signaldefinitionsschicht auf den Wert 1 gesetzt. Der zweite Signalwert hat die Dauer von 85 ns und wurde in der Befehlsdefinitionsschicht auf den Wert 0 gesetzt. Der dritte Signalwert mit der Dauer von 45 ns wurde in der Signaldefinitionsschicht auf den Wert 1 gesetzt.

Der Datenbus DATA hat auch drei Signalwerte. Der erste Signalwert hat die Dauer von 25 ns und wurde in der Signaldefinitionsschicht auf den Wert 0x00 gesetzt. Der zweite Signalwert hat die Dauer von 95 ns und wurde nicht gesetzt. Das bleibt variabel und wird in der Befehlsaneinanderkettung mit dem gewünschten Wert gesetzt. Der dritte Signalwert mit der Dauer von 30 ns wurde in der Signaldefinitionsschicht auf den Wert 0x00 gesetzt.

Befehlname	**Signalname**	**Signalwerte**		
		Dauer	**Wert**	**IN/OUT**
ACW	CS#	50 ns	1	IN
		50 ns	**0**	IN
		60 ns	1	IN
	ADDR	25 ns	0x000	IN
		100 ns	**0x0F0**	IN
		35 ns	0x000	IN
	R/W#	30 ns	1	IN
		85 ns	**0**	IN
		45 ns	1	IN
	DATA	25 ns	0x00	IN
		95 ns	**VAR**	IN
		30 ns	0x00	IN

Tabelle 9 – ACW Befehlsdefinition mit Signalwerten

Beispiel: Um den Akkumulator Register einer UUT zu lesen, wird der Befehl ACR VAR definiert. Das Timing des Befehls ist in der Abbildung 23 dargestellt. Für diesen Befehl werden alle Adresssignale während des aktiven Bereichs auf den fixen Wert des Akkumulator Registers eingestellt. Weiters wird das Chip Select Signal (CS#) zum aktiven Zeitpunkt fix auf logisch 0, bzw. das R/W# Signal auf logisch 1 gelegt. Das Label ACR dient zur Kennzeichnung des Befehls und muss sich von allen anderen Befehlen in der Bezeichnung unterscheiden.

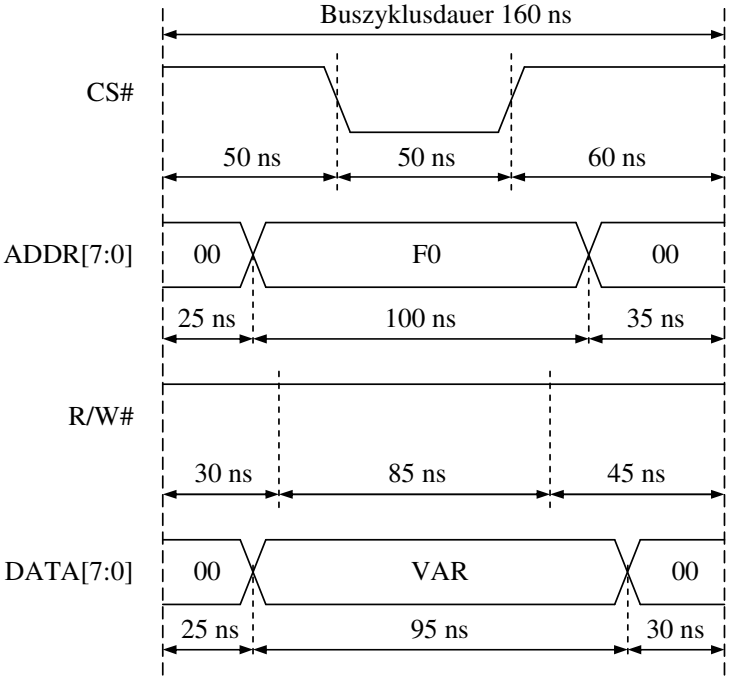

Abbildung 23 – Signalabläufe des ACR Befehls

Der Parameter VAR stellt den Referenzwert der DATA Signalleitungen dar. Dieser kann bei der Befehlsaneinanderkettung festgelegt werden. Die DATA Signalleitungen sind OUT Signale. Alle Ports vom angekoppelten ASICs sollen dabei auf Hochimpedanz geschaltet werden. Wenn der Parameter nicht gesetzt wird, kann kein Vergleich mit der UUT DATA durchgeführt werden. Für den Befehl ACR VAR würde somit die Festlegung aus der Tabelle 10 entstehen.

Signalname	Wert	IN/OUT
CS#	0	IN
ADDR	0x0F0	IN
R/W#	1	IN
DATA	VAR	OUT

Tabelle 10 – ACR Befehlsdefinition

Zur Definition eines Befehls werden folgende Parameter benötigt:

1. **Befehlname:** Bezeichnet den zu charakterisierenden Befehl (z.B. ACW, usw.). Es soll eine eindeutige Bezeichnung des Befehls sein.
2. **Dauer:** Die Dauer eines Befehlszyklus. In diesem Zeitraum müssen die Signale gesetzt werden. Die Mindestdauer jedes Signals darf die Befehlsdauer nicht überschreiten, also die Periode des Befehls darf nicht kürzer sein, als sein längstes Signal. Wenn die Mindestdauer eines Signals kürzer ist als die Befehlsdauer, soll das Signal ausdehnbar sein.

Wenn die Testbench-Strukturen weitere Parameter benötigen, kann die Parameterliste erweitert werden. Die neuen Parameterdefinitionen werden in eine XML Konfigurationsdatei

erfasst. Die Übersetzung des Testbench-Generators ist nicht notwendig.

Die Plausibilitätsprüfung der Befehlsdefinition wird in einer Business Funktion abgebildet. Diese überprüft den Befehlsnamen und die Periode des Befehls. Die Periode des Befehls soll mindestens so lang sein, wie die längste Signalperiode der Befehlssignale.

2.5.4 Programmdefinition

Nach dem die Befehle definiert wurden, können sie in Programmen (Testroutinen) verwendet werden. Ein Programm besteht aus einer Reihe Befehlsinstanzen und Konfigurationsparametern. Eine Befehlsinstanz ist eine Referenz auf einer Befehlsdefinition. Die Änderungen in der Befehlsdefinition haben auch auf die Befehlsinstanzen Auswirkung. Die noch nicht gesetzten Signalwerte werden in der Befehlsinstanz gesetzt. Bereits in der Signaldefinitionsschicht, bzw. in der Befehlsdefinitionsschicht werden die zeitlichen Signalverhalten durch Signalwerte definiert. Die Informationen über das zeitliche Signalverhalten sind also in der Programmdefinition nicht mehr relevant. Zur Definition eines Programms wird der Default Zeiteinheit Parameter benötigt. Dieser gibt an mit welchen default Zeiteinheiten die Zeiten im Modell zu verstehen sind.

Wenn die Testbench-Strukturen weitere Parameter benötigen, kann die Parameterliste erweitert werden. Die neuen Parameterdefinitionen werden in eine XML Konfigurationsdatei erfasst. Die Übersetzung der Testbench-Generator ist nicht notwendig.

Beispiel: In der Tabelle 11 wird ein Programm präsentiert, womit der Akkumulator mit dem Wert 0x92 beschrieben und anschließend der Akkumulator gelesen wird. Der Referenzwert der ACR Befehl ist 0x92. Die Angabe des Referenzwertes ist nicht Pflicht.

```
ACW 0x92
ACR 0x92
```

Tabelle 11 – Programmsequenz

Das Timing Diagramm des Programms ist in der Abbildung 24 dargestellt. Der hohe Abstraktionsgrad des Programms kann festgestellt werden. Durch zwei Zeilen Programmcode wird ein komplexes Timing definiert.

Die textuelle Eingabe der Programmsequenzen wird durch eine Business Funktion der Programmdefinition unterstützt. Diese parst die Programmsequenzen und erstellt die Befehlsinstanzen. Die Auflösung der Signalwertebereiche in einzelne Signalwerte wird auch von einer Business Funktion der Programmdefinition durchgeführt.

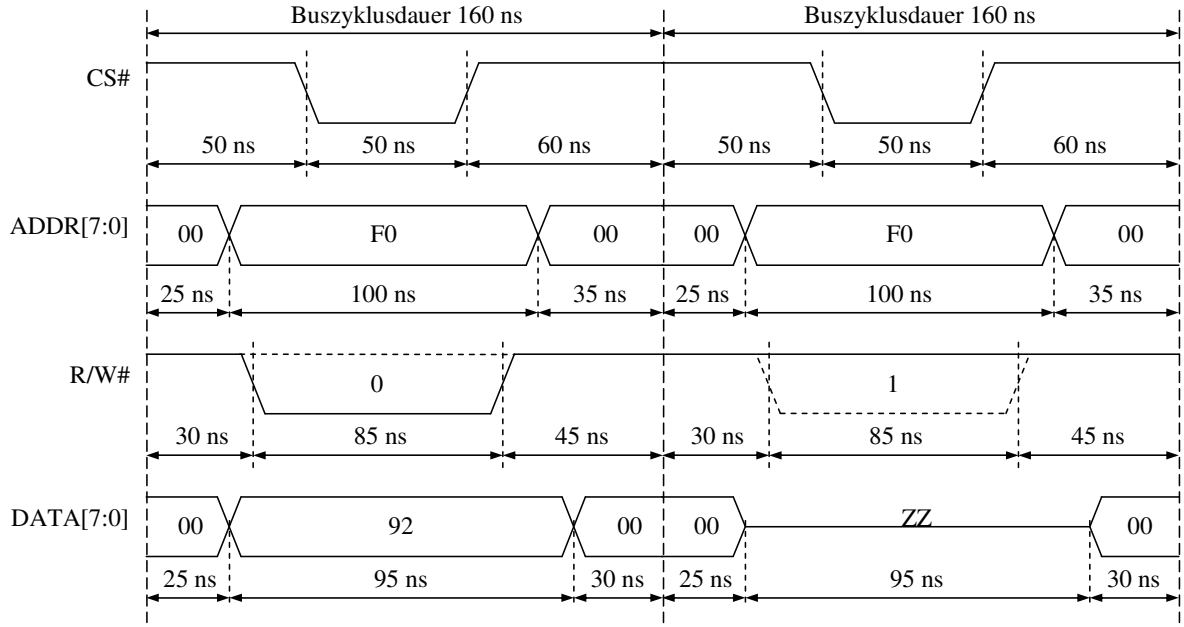

Abbildung 24 – Signalabläufe einer Programmsequenz

2.5.5 Moduldefinition

Die Moduldefinition modelliert die Schnittstelle der UUT. Für jede einzelne UUT wird eine Moduldefinition angelegt, die mit einem Modulnamen gekennzeichnet wird. Der Generator generiert aus der Moduldefinition die Testbench Schnittstellen der UUT.

Wenn die Testbench-Strukturen Parameter benötigen, können Parameter erfasst werden. Die neuen Parameterdefinitionen werden in eine XML Konfigurationsdatei erfasst. Die Übersetzung der Testbench-Generator ist nicht notwendig.

Die Plausibilitätsprüfung der Moduldefinition wird in einer Business Funktion abgebildet. Diese überprüft den Modulnamen. Der Modulname darf nur einmal vorkommen.

Beispiel: In der Abbildung 25 ist eine UUT mit fünf Eingängen dargestellt, zwei Ausgängen und einem Eingang/Ausgang (IN/OUT). Das entsprechende Modul hat die gleichen Signaldefinitionen, wie die UUT. Der Generator kann auf Grund dieser Informationen die Schnittstelle zwischen der Testbench und die UUT erstellen.

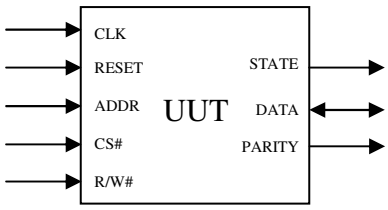

Abbildung 25 – Moduldefinition

2.5.6 Taktdefinition

Die Taktdefinition modelliert ein Taktsignal. Das Taktsignal ist ein spezielles Signal. Dieses

kann von den anderen Signalen, von Befehlen oder von Modulen genauso wie ein herkömmliches Signal angesprochen werden. Zur Definition eines Taktsignals werden folgende Parameter benötigt:

1. **Periode:** Definiert die Länge eines einzelnen Taktes.
2. **Steigende Flanke 1:** Zeit bis zur steigenden Flanke des ersten Impulses. Es kann entweder einen absoluten Wert annehmen oder einen Prozentsatz der Periode.
3. **Fallende Flanke 1:** Zeit bis zur fallenden Flanke des ersten Impulses. Es kann entweder einen absoluten Wert annehmen oder einen Prozentsatz der Periode.
4. **Steigende Flanke 2:** Zeit bis zur steigenden Flanke des zweiten Impulses. Es kann entweder einen absoluten Wert annehmen oder einen Prozentsatz der Periode.
5. **Offset Zeit:** Zeit vom Einschalten des Generators bis zum Beginn der ersten Periode (Initialisierungszeit für andere Komponenten).

Die einzelnen Parameter der Taktdefinition werden anhand der Abbildung 26 näher illustriert.

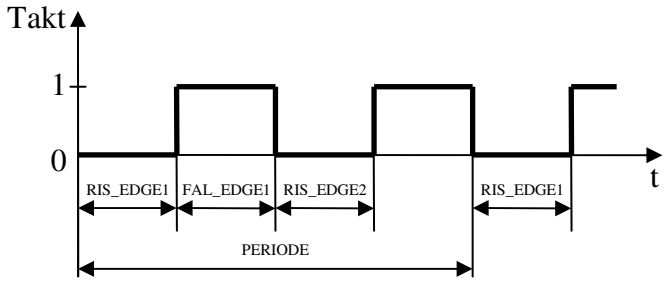

Abbildung 26 – Taktdefinition

Wenn die Testbench-Strukturen weitere Parameter benötigen, kann die Parameterliste erweitert werden. Die neuen Parameterdefinitionen werden in eine XML Konfigurationsdatei erfasst. Die Übersetzung der Testbench-Generator ist nicht notwendig.

Die Plausibilitätsprüfung der Taktdefinition wird in einer Business Funktion abgebildet. Es wird geprüft, ob die Periode einen sinnvollen Wert hat oder nicht.

2.5.7 Comparator Definition

Der Comparator modelliert einen Signalcomparator. Dieser hat die Aufgabe die Ausgangssignale der UUT mit den Ausgangssignalen eines Referenzmoduls der UUT zu vergleichen.

Da die unterschiedlichen Testbench-Strukturen unterschiedliche Parameter benötigen, werden keine fixen Parameter definiert. Die Parameterdefinitionen pro Testbench-Struktur werden bei Bedarf in XML Konfigurationsdateien erfasst. Die Übersetzung der Testbench-Generator ist nicht notwendig.

2.5.8 Reportgenerator Definition

Der Reportgenerator hat die Aufgabe die Simulationsergebnisse und die Statusinformationen aus der Testbench zu sammeln und in eine Textdatei zu schreiben.

Da die unterschiedlichen Testbench-Strukturen unterschiedliche Parameter benötigen, werden keine fixen Parameter definiert. Die Parameterdefinitionen pro Testbench-Struktur werden bei Bedarf in XML Konfigurationsdateien erfasst. Die Übersetzung der Testbench-Generator ist nicht notwendig.

2.5.9 Konstantendefinition

Die Konstanten definieren globale Zeitangaben und Signalwerte der Testbench. Diese können von den Signaldefinitionen, Signalwertdefinitionen, Befehldefinitionen und Programmdefinitionen als Zeitangaben, bzw. als Signalwerte verwendet werden.

Beispiel: `ZERO32 ::= 0x00000000`

2.6 Testbench-Generator Architektur

Die Architektur der Testbench-Generator ist in der Abbildung 27 dargestellt. Es sind drei Schichten zu unterscheiden. Die Client-Schicht, wo die Eingabe und die Visualisierung der Daten stattfindet, die Business-Schicht, wo die Datenverarbeitung stattfindet und die EIS (Enterprise Information System) Schicht, wo die Daten dauerhaft gespeichert werden.

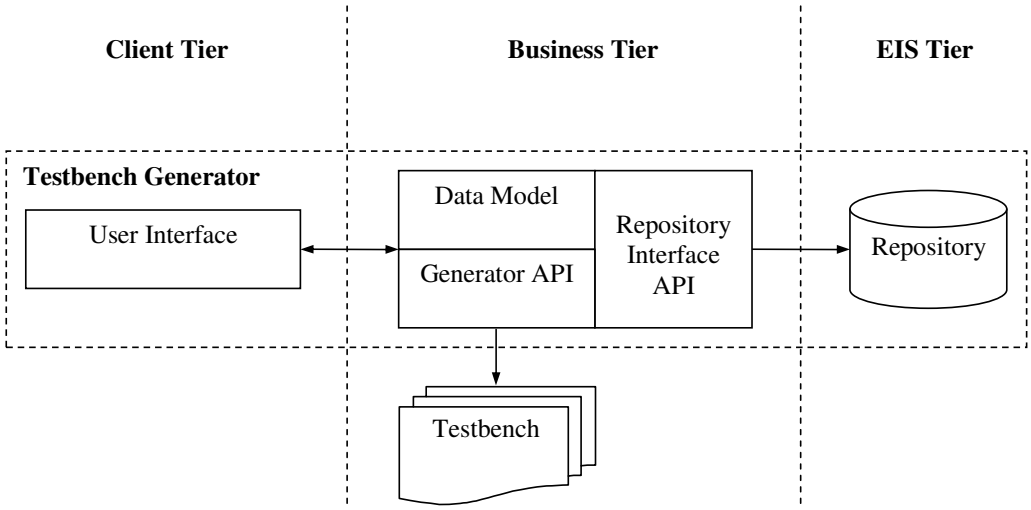

Abbildung 27 – Architektur der Testbench-Generator

Folgende Module sind dabei zu unterscheiden:

1. **User Interface:** Ermöglicht den Anwendern die Eingabe der Daten und die Steuerung des Testbench-Generators. Diese kann als eine Client Anwendung oder als eine Web Anwendung realisiert werden.
2. **Data Model:** In diesem Bereich werden die Daten für die Schnittstellen

bereitgehalten. Die Businesslogik, die Konsistenzprüfungen und die Transaktionssicherheit werden hier realisiert.

3. **Generator API:** Ermöglicht die Abfrage aller erfassten Modelldaten.
4. **Repository Interface API:** Ist eine klar definierte Schnittstelle, um viele Arten von Repositories anbinden zu können, wie z.B. eine SQL (Structured Query Language) Datenbank oder Dateien.
5. **Repository:** Ist für die dauerhafte Datenhaltung zuständig. Im Repository werden die Befehlsbibliotheken, die Programme, die Modulen und die Konfigurationsparameter gespeichert. Als mögliche Repositories sind die SQL Datenbanken oder Dateien.

2.6.1 User Interface

Die Benutzeroberfläche ermöglicht den Anwendern die Erfassung aller Testbench Daten und die Steuerung des Testbench-Generators. Die Daten sind in zwei Gruppen zu ordnen. Die wieder verwendbaren Daten und die Programmdaten. Die wieder verwendbaren Daten bestehen aus den Signaldefinitionen und aus den Befehlsdefinitionen und sie bilden die Befehlsbibliotheken. Die Realisierung der Befehlsbibliotheken wird durch die Benutzeroberfläche unterstützt. Die Programme werden auf bestehenden Befehlsbibliotheken aufgebaut. Ohne Angabe einer Befehlsbibliothek können keine Programme erstellt werden. Die Aneinanderkettung der Befehle resultiert den Programmablauf. Die zusätzlichen Programmparameter wie z.B. die Taktdefinition bilden die Programm Metadaten. Zu den Metadaten können mehrere Programm Abläufe realisiert werden.

2.6.2 Data Model

Das Datenmodell stellt die Daten der Benutzeroberfläche und dem Generator API zur Verfügung. In der Business Logik des Datenmodells werden die gewünschten Funktionalitäten realisiert, wie z.B. die Syntax- und Plausibilitätsprüfung der erfassten Daten oder das Parsen der Textprogramme und die Umwandlung in Befehlinstanzen. Wenn die Transaktionssicherheit gewünscht ist, wird diese im Datenmodell realisiert.

2.6.3 Generator API

Das Generator API ermöglicht die Abfrage aller erfassten Modelldaten. Auf Grund dieser Daten können Generatoren für neue Testbench-Strukturen oder Generatoren für verschiedene Dateiformate implementiert werden.

2.6.4 Repository

Durch das Repository Interface wurde Ermöglicht die Modelldaten in unterschiedlichen Medien und Formate abzuspeichern. Die am häufigsten eingesetzten Medien sind die SQL

Datenbanken oder die Dateisysteme.

File Repository

Als Repository werden Dateien eingesetzt. Die Modelldaten werden im XML Format oder in Dateien in einer serialisierten Form abgespeichert. Die Serialisierung ist eine neue Technik Objekte aus dem Hauptspeicher über ein Stream zu übertragen. Um die Wiederverwendbarkeit der Befehle zu gewährleisten, werden die Befehlsdefinitionen und die Signaldefinitionen in eigene Befehlsbibliothek Dateien abgespeichert. Die Programme werden auch in Dateien abgelegt. Damit ist die Verteilung der Befehlsbibliotheken sehr leicht durchzuführen.

Datenbank Repository

Eine Alternative zu einem File Repository ist eine SQL Datenbank. Die Modelldaten werden in Datenbanktabellen abgespeichert. Der Einsatz der SQL Datenbanken ist dann sinnvoll, wenn gleichzeitig mehrere Benutzer am selben Projekt arbeiten und die Daten Zentral gehalten werden müssen.

3 Testbench-Generator Implementierung

Bei der Implementierung des Testbench-Generators wurde versucht möglichst alle Anforderungen zu berücksichtigen. Die Modelldefinitionen aus dem Kapitel 2 wurden gänzlich implementiert. Zusätzlich wurde auf die Flexibilität und Erweiterbarkeit sehr viel Wert gelegt. Als Programmiersprache wurde Java gewählt. Diese hat den Vorteil, dass die resultierende Anwendungen Plattformunabhängig (Betriebssystemunabhängig, soweit eine Java Runtime Umgebung für das Betriebsystem vorhanden ist) sind. Damit wird gewährleistet, dass der Testbench-Generator auf unterschiedlichen Betriebsystemen betrieben werden kann. Als Java Referenzhandbuch siehe [Eck02]. Durch einige Schnittstellen (APIs) können weitere Module dazu entwickelt werden. Z.B. durch ein neues Generator Modul können weitere Testbench-Strukturen unterstützt werden.

Das Blockschaltbild des Testbench-Generators ist in der Abbildung 28 dargestellt. Der Testbench-Generator ist eine eigenständige Client-Applikation. Alle definierten Schichten wurden zusammengefasst. Die Daten werden im Hauptspeicher des Client Rechners verarbeitet und in Dateien zwischengespeichert.

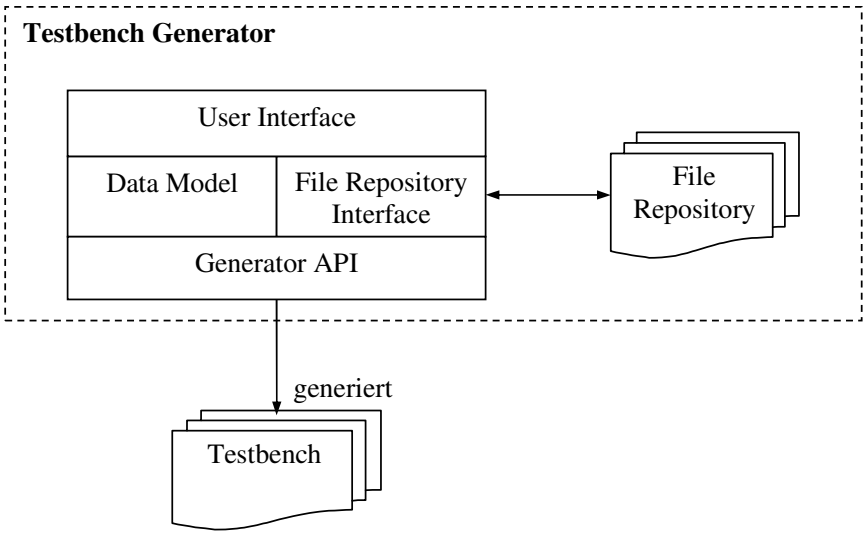

Abbildung 28 – Blockschaltbild der Testbench-Generator

3.1 User Interface

Die Benutzeroberfläche ist eine Java Client Oberfläche und benötigt mindestens den Java Runtime Version 1.4.0. Die Layout Einstellungen können über eine Properties-Datei verändert werden. Die Benutzeroberfläche besteht aus mehreren Eingabemasken, die unterstützen die Erfassung aller Daten, die für einen Testbench notwendig sind. Die Steuerung des Testbench-Generators wurde Menügesteuert realisiert. Die Erweiterung der Parameterlisten jegliches Datenmodells wird durch XML Konfigurationsdateien realisiert. Eine Neuübersetzung des Testbench-Generators ist damit nicht notwendig. In den nächsten Abschnitten werden die Masken präsentiert.

3.1.1 Masken für die Befehlsbibliothekerstellung

Für die Erfassung der Befehlsbibliotheken stehen zwei Masken zur Verfügung: die Signaldefinitionsmaske und die Befehlsdefinitionsmaske. In der Signaldefinitionsmaske werden die Signaldefinitionen mit Signalwertdefinitionen erfasst. Der Screenshot der Maske ist in der Abbildung 29 dargestellt. Die Signalwerte können in unterschiedlichen Zahlensystemen eingegeben werden. Die Zahlen ohne zusätzliches Kennzeichen werden als Binärzahlen interpretiert. Die Hexadezimalzahlen müssen mit dem „0x“ Zeichen gekennzeichnet werden.

Die Signaldefinition wird einer Plausibilitätsprüfung unterzogen, die im Datenmodell definiert wurde. Die Fehlermeldungen werden in der Maske präsentiert.

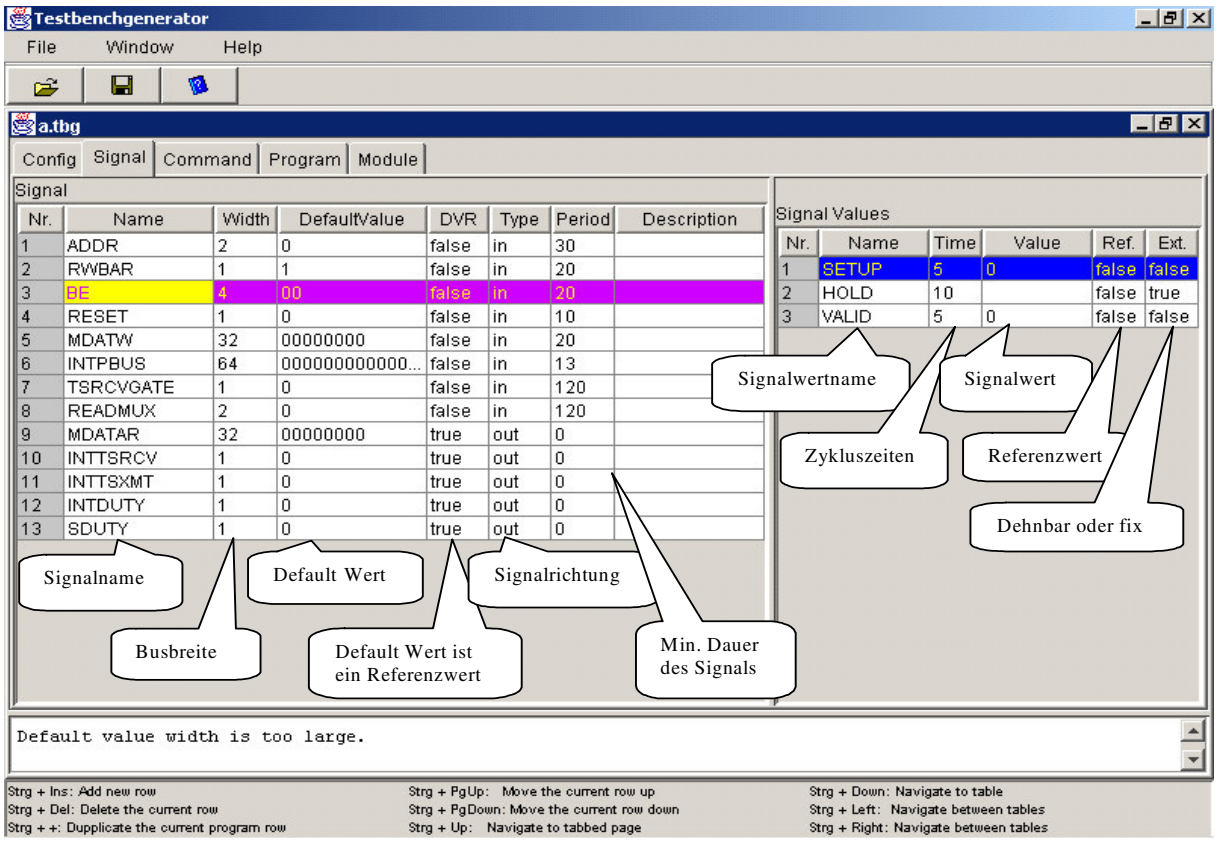

Abbildung 29 – Signaldefinitionsmaske

In der Befehlsdefinitionsmaske werden die Signale in Befehle zusammengefasst und die Befehlsparameter gesetzt. Die Zeitangaben und die Wertangaben der Signalwerte können überschrieben werden. Das Feld „Name“ beinhaltet den Befehlnamen und darf nur einmal vergeben werden. Die Befehlsdefinition wird, wie bei der Signaldefinition, auch einer Plausibilitätsprüfung unterzogen. Der Screenshot der Maske ist in der Abbildung 30 präsentiert.

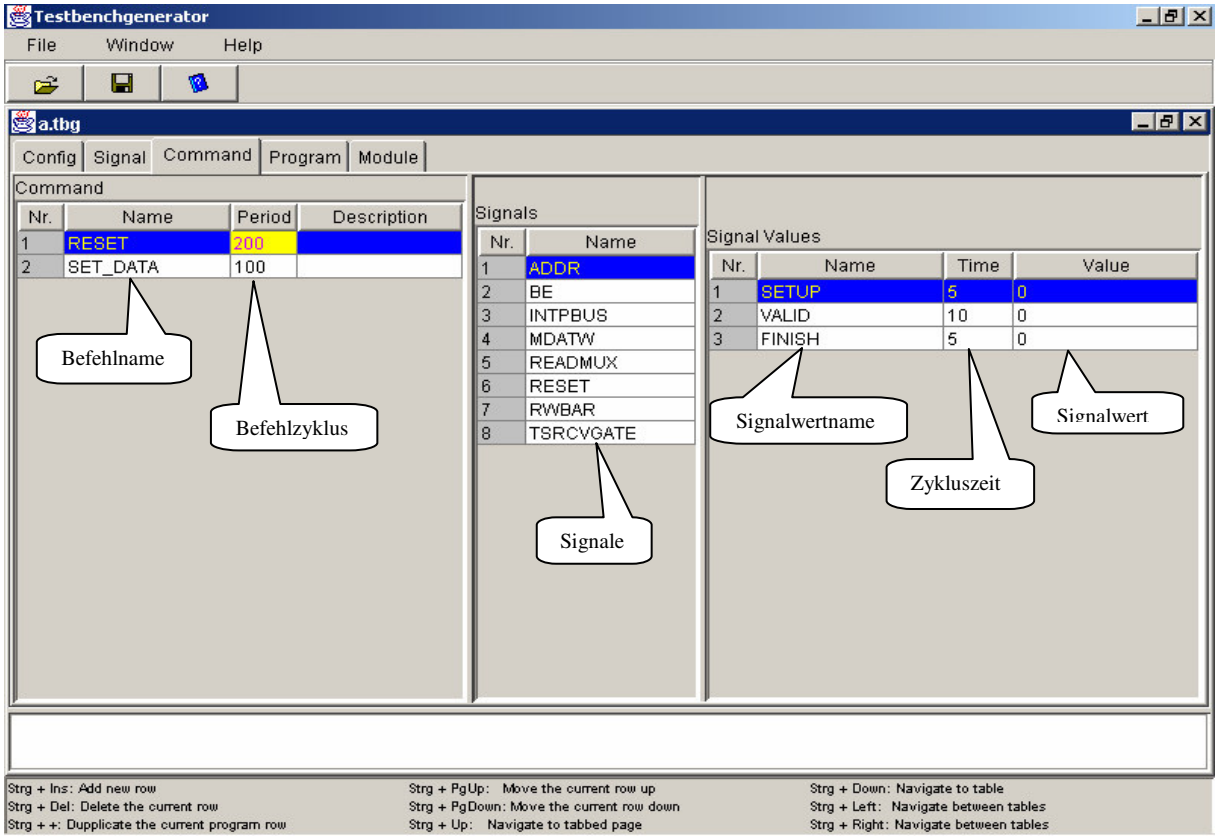

Abbildung 30 – Befehldefinitionsmaske

3.1.2 Masken für die Programmerstellung

Bevor ein Programm erstellt werden kann, soll eine Befehlsbibliothek vorhanden sein. Wenn die Bibliothek vorhanden ist, kann mit der Programmerstellung angefangen werden. Die Programmerstellung besteht aus zwei Schritten. Im ersten Schritt werden die Programm Metadaten erfasst, in dem Zweiten die Befehle werden aneinandergekettet. Die Metadaten sind das Taktsignal, die UUT Schnittstellen, die Referenzmodule und alle zusätzlichen Parameter die für die Testbench Generierung notwendig sind. Die Abbildung 31 zeigt die Eingabemaske, womit das Taktsignal und die Zeiteinheit erfasst werden können.

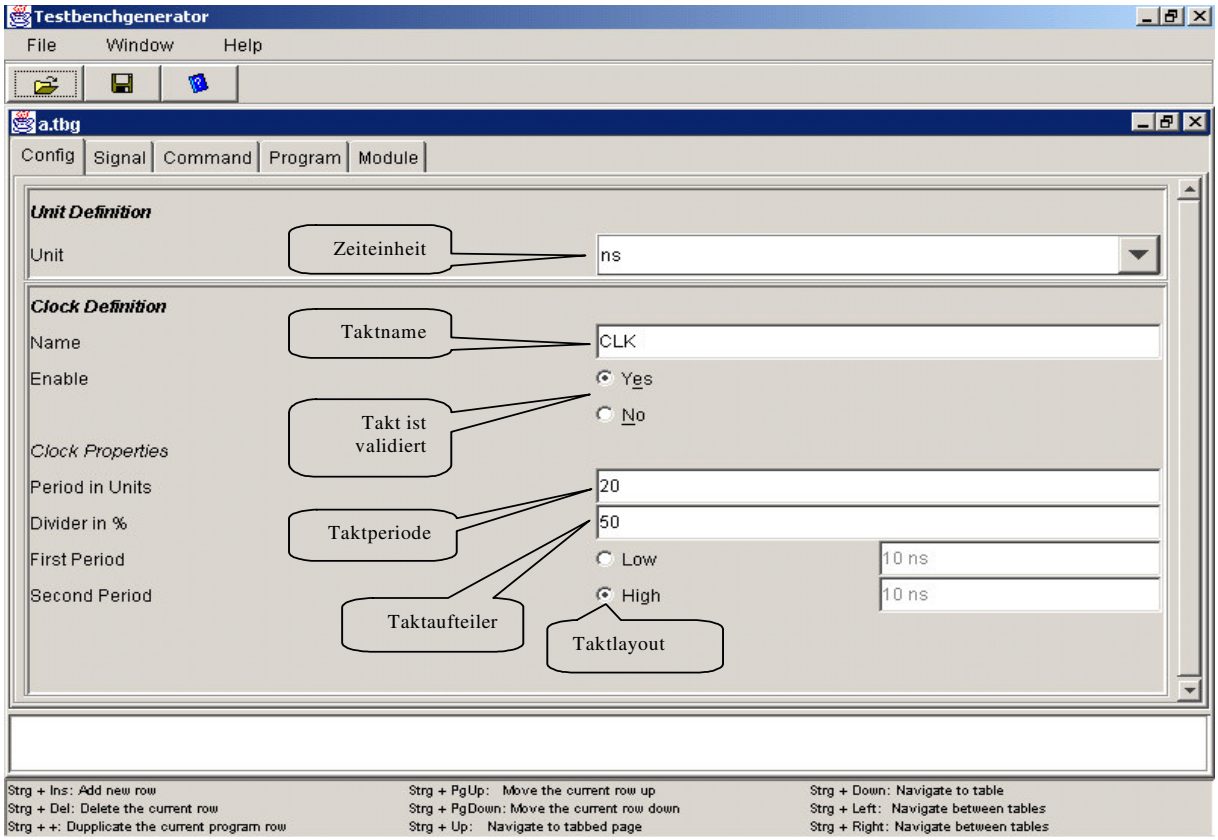

Abbildung 31 – Taktgenerator Definitionsmaske

Die Schnittstellendefinitionen der UUT Modulen bilden die restlichen Metadaten der Programmdefinition. Diese werden auch mit einer Maske unterstützt. In der Abbildung 32 ist die Definition einer solchen Schnittstelle dargestellt.

Bei der Aneinanderkettung der Befehle in Programme werden die Befehlsdefinitionen aus einer Befehlsbibliothek entnommen. Der Vorgang wird durch zwei Masken unterstützt. In einer Maske werden die Befehle in eine tabellarische Struktur eingetragen, so wie in der Abbildung 33 gezeigt ist. Die zweite Maske ist ein reiner Texteditor, der nach jeder eingegebenen Zeile den Text parst und die Befehle generiert. Die so entstandenen Befehle werden in der Eingabemaske mit der tabellarischen Struktur automatisch synchronisiert. Die Syntax der Texteingabe des Programmcodes ist:

```
command ::= command-name variable{,variable}
command-name ::= Name des Befehls
variable ::= Wert der Befehlsvariable
```

Pro eingegebene Textzeile darf nur ein Befehl vorkommen. Durch Nichteinhaltung der Syntaxspezifikation resultiert eine Fehlermeldung. Die Funktionalität des Parsens wurde im Datenmodell realisiert.

Die überschreibbare, bzw. die noch nicht gesetzte Signalwerte werden bei der Aneinanderkettung der Befehle gesetzt. Die erstellten Programme werden auch einer Plausibilitätsprüfung unterzogen, die im Datenmodell definiert wurde. Es wird vor Allem die Richtigkeit der angegebenen Signalwerte überprüft.

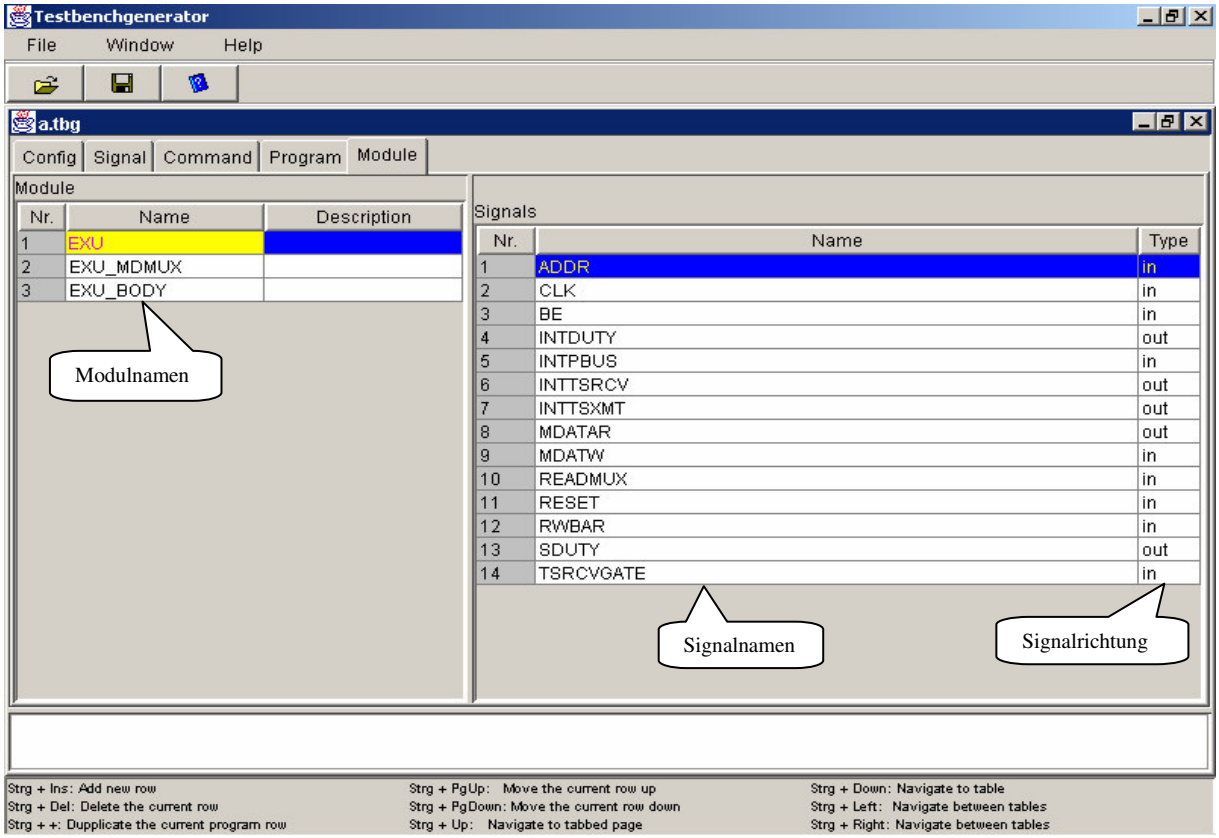

Abbildung 32 – Moduldefinitionsmaske

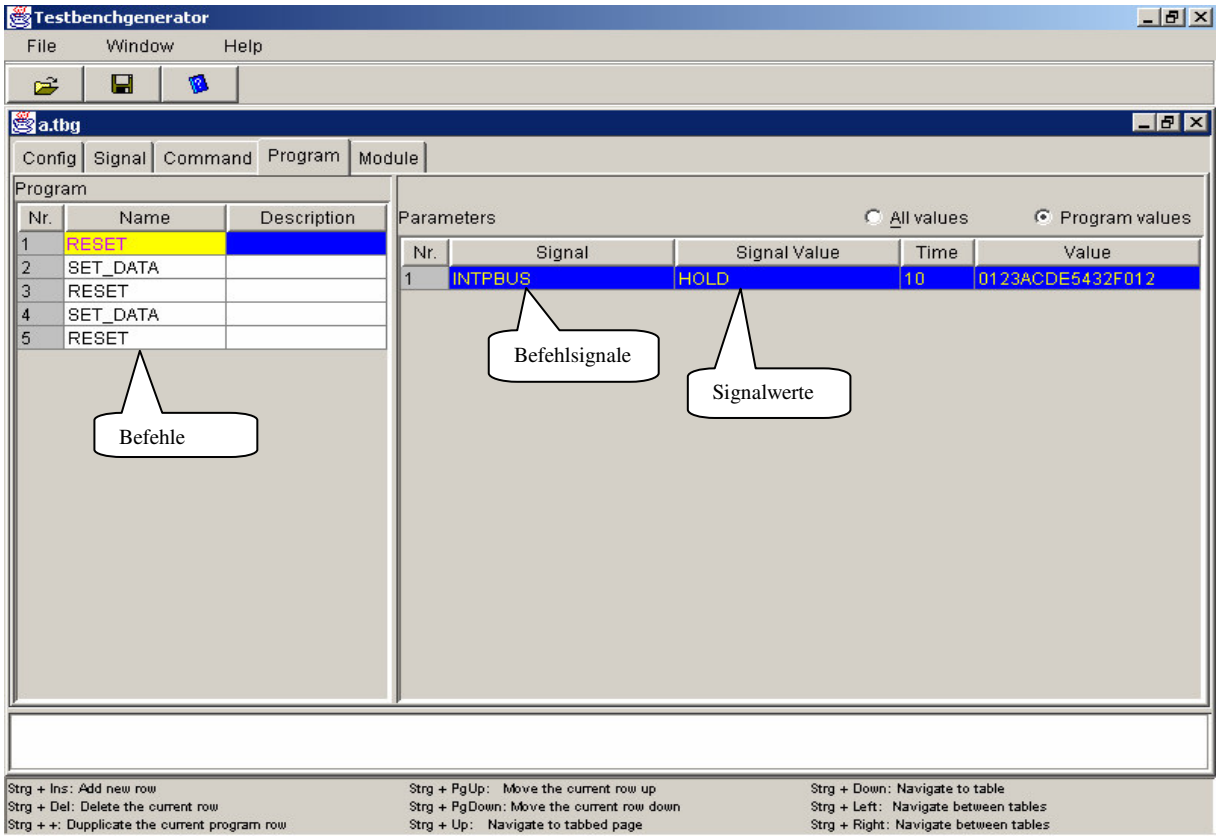

Abbildung 33 – Programmdefinitionsmaske

3.2 Generatoren

Das Generator API ermöglicht die Realisierung von Generatoren für Testbench-Strukturen, Stimulidaten oder andere Dateiformate. Im Rahmen dieser Diplomarbeit wurden zwei Generatoren implementiert. Die erste Implementierung ist ein Generator für die Generierung der Testbench-Struktur präsentiert im Kapitel 2.3. Die Testbench wird im VHDL-Code generiert. Als VHDL-Unterstützung wurde [Arm00] und [Hag95] verwendet. Die zweite Implementierung ist ein Latex Makro Generator. Damit ist es möglich aus den Testbench-Programmen Latex Makros für Zeitdiagramme zu erstellen. Die Latex Makros sind in [May00] beschrieben.

3.3 Repository

Als Repository wird eine File Repository eingesetzt. Die Modelldaten werden im XML Format oder in eine serialisierte Form in Dateien abgespeichert. Um die Wiederverwendbarkeit der Befehle zu gewährleisten, werden die Befehlsdefinitionen und die Signaldefinitionen in eigene Befehlsbibliothek Dateien abgespeichert. Die Programme werden auch in Dateien abgelegt. Damit ist die Verteilung der Befehlsbibliotheken sehr leicht durchzuführen.

4 Anwendungszenario (Case Study)

Die Realisierung einer Testbench wird in vier Aufgabenbereiche aufgeteilt:

1. Testfälle identifizieren
2. Design der Testbench
3. Realisierung der Testbench
4. Simulation und Auswertung der Ergebnisse

Als Grundlage der Aufteilung wurde das Vorgehensmodell eines Softwareentwicklungsprojektes aus [Hit99] genommen.

4.1 Aufgabenbereiche

4.1.1 Testfälle identifizieren

Die Planung der Testbench beginnt mit der Auswahl der Module, die überprüft werden sollen. In den meisten Fällen werden alle Submodule und das Hauptmodul überprüft. Für die ausgewählten Module werden die Testfälle definiert, womit die Funktionalität des ASICs überprüft wird. Die Änderung der Testfälle kann in einer späteren Entwicklungsphase sehr viel Arbeit mit sich ziehen. Eine gründliche Testplanung ist somit von großer Bedeutung.

4.1.2 Design der Testbench

Das Design der Testbench wird nach einer gründlichen Analysephase realisiert. Der Testbench-Generator kommt erstmals zum Einsatz. Die Signale der Module werden identifiziert und mit dem Testbench-Generator erfasst. Die IN/OUT Signale sollten möglichst in einzelne IN, bzw. OUT Signale aufgelöst werden. Auf die Signale wird der Befehlsatz des ASICs aufgesetzt. Dieser sollte noch Testbench-Struktur unabhängig bleiben. Eine Umsetzung auf einer VHDL, Verilog oder irgendeiner anderen Testbench sollte noch frei bleiben.

4.1.3 Realisierung der Testbench

Die Testbench-Struktur wird ausgewählt und die Testbench spezifischen Parameter, bzw. die Parameter der speziellen Testbench Komponenten werden erfasst. Wenn keine Testbench-Struktur die gewünschte Funktionalität darstellt, kann auf Grund des Generators API eine neue Testbench-Struktur implementiert werden. Das Referenzmodul wird realisiert. Der Testbench-Generator kann auf Grund der erfassten Daten die Testbench generieren. Der generierte Code wird mit einem entsprechenden Werkzeug übersetzt. Für eine Testbench können beliebig viele Testprogramme geschrieben werden, welche die vorgegebenen Testfälle abbilden. Wenn kein Referenzmodul realisiert wurde, werden die Referenwerte in

den Testprogrammen miterfasst. Die Stimulidaten werden aus den Testprogrammen generiert. Diese müssen nicht übersetzt werden und sie können auch mit einem Texteditor nachbearbeitet werden. Wenn der Befehlsatz geändert wird, sollen alle Stimulidaten neu generiert werden. Die Testbench soll nur dann neu generiert werden, wenn die Signaldefinitionen verändert werden. Letztendlich werden die zu testende Module definiert und die Signaldefinitionen werden den Modulen zugeordnet.

4.1.4 Simulation und Auswertung der Ergebnisse

Nach dem Erstellen der Testbench und der Stimulidaten, wird die Simulation durchgeführt. Das Resultat der Simulation ist in den meisten Fällen eine Ergebnis-Datei. Diese beinhaltet Informationen bezüglich der Richtigkeit der Funktionalität der getesteten Module. Im Falle einer Fehlfunktion des getesteten Moduls, wird der Fehler ausgebessert und das Modul neu übersetzt. Die Testbench und die Stimulidaten müssen nicht neu erstellt werden. Die Simulation kann wiederholt werden.

4.2 Beispiel

4.2.1 Beschreibung des Testobjekts

Das Anwendungszenario des Testbench-Generators wird anhand eines Beispieles aus [Hor96] illustriert. Das Blockschaltbild ist in der Abbildung 34 dargestellt. Das Hauptmodul EXU besteht aus zwei Teilmodulen. Das EXU_BODY implementiert die Kernfunktionalität des Hauptmoduls und das EXU_MDMUX implementiert einen wieder verwendbaren Multiplexer. Das VHDL Programm der EXU ist in der Abbildung 35 dargestellt.

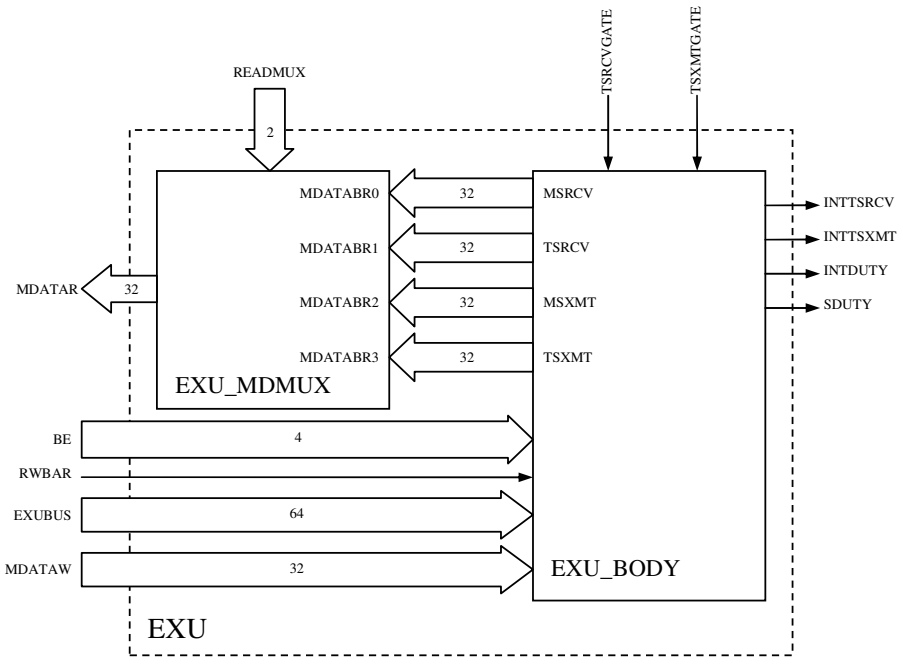

Abbildung 34 – Beispiel EXU (Example Unit) [Hor96]

```
-------------------------------------------------------------------------------
-- I C T Computer Technology Dept
-- University of Technology, Vienna
-- All Rights Reserved
-- Project: UTCSU
-- Designer: Martin HORAUER
-- Version: 1.0
-- File name: exu_ent.vhd
-- Title: Example UNIT
-- Module: entity
-- Tools: Synopsys
-- Ref. lib.: IEEE
-- Ref. pkg.: 1164
-- Roadmap:
-- Date: 10/96
-------------------------------------------------------------------------------
library ieee;
  use ieee.std_logic_1164.all;
entity EXU is
  port(
    ADDR: in std_logic_vector(1 downto 0);
    RWBAR: in std_logic;
    BE: in std_logic_vector(3 downto 0);
    CLK: in std_logic;
    RESET: in std_logic;
    MDATAR: out std_logic_vector(31 downto 0);
    MDATAW: in std_logic_vector(31 downto 0);
    READMUX: in std_logic_vector(1 downto 0);
    EXUBUS: in std_logic_vector(63 downto 0);
    TSRCVGATE: in std_logic;
    TSXMTGATE: in std_logic;
    INTTSRCV: out std_logic;
    INTTSXMT: out std_logic;
    INTDUTY: out std_logic;
    SDUTY: out std_logic
  );
end EXU;
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- I C T Computer Technology Dept
-- University of Technology, Vienna
-- All Rights Reserved
-- Project: UTCSU
-- Designer: Martin HORAUER
-- Version: 1.0
-- File name: exu_beh_arch.vhd
-- Title: Example UNIT
-- Module: architecture
-- Tools: Synopsys
-- Ref. lib.: IEEE
-- Ref. pkg.: 1164
-- Roadmap:
-- Date: 10/96
-------------------------------------------------------------------------------
--     >>>>> EXU
--           |
--       +---+---+
--       |       |
--   exu_body exu_mdmux
-------------------------------------------------------------------------------
architecture behavioral of EXU is
  signal MDATABR0: std_logic_vector(31 downto 0);
  signal MDATABR1: std_logic_vector(31 downto 0);
  signal MDATABR2: std_logic_vector(31 downto 0);
  signal MDATABR3: std_logic_vector(31 downto 0);
  component EXU_BODY
    port(
      EXUBUS: in std_logic_vector(63 downto 0);
      MDATAW: in std_logic_vector(31 downto 0);
      RESET: in std_logic;
      CLK: in std_logic;
      ADDR: in std_logic_vector(1 downto 0);
      BE: in std_logic_vector(3 downto 0);
      RWBAR: in std_logic;
      INTTSRCV: out std_logic;
      TSRCVGATE: in std_logic;
```

```
      TSXMTGATE: in std_logic;
      INTTSXMT: out std_logic;
      INTDUTY: out std_logic;
      SDUTY: out std_logic;
      MSRCV: out std_logic_vector(31 downto 0);
      TSRCV: out std_logic_vector(31 downto 0);
      MSXMT: out std_logic_vector(31 downto 0);
      TSXMT: out std_logic_vector(31 downto 0)
    );
  end component;
  component EXU_MDMUX
    port(
      MDATABR0: in std_logic_vector(31 downto 0);
      MDATABR1: in std_logic_vector(31 downto 0);
      MDATABR2: in std_logic_vector(31 downto 0);
      MDATABR3: in std_logic_vector(31 downto 0);
      MDATAR: out std_logic_vector(31 downto 0);
      READMUX: in std_logic_vector(1 downto 0)
    );
  end component;
begin
  UEXUMDMUX: exu_mdmux port map(MDATABR0, MDATABR1, MDATABR2, MDATABR3, MDATAR, READMUX);
  UEXUBODY: exu_body port map(EXUBUS, MDATAW, RESET, CLK, ADDR, BE, RWBAR, INTTSRCV,
                              TSRCVGATE, TSXMTGATE, INTTSXMT, INTDUTY, SDUTY,
                              MDATABR0, MDATABR1, MDATABR2, MDATABR3
                             );
end behavioral;
--------------------------------------------------------------------------------
```

Abbildung 35 - EXU VHDL Programm [Hor96]

Nach der Spezifikation des Hauptmoduls, werden die zwei Submodule präsentiert. Das Submodul EXU_MDMUX ist in der Abbildung 36 illustriert und stellt einen Multiplexer dar. Eine der vier Bus-Eingänge, abhängig vom READMUX Signal, wird zum Ausgang MDATAR geleitet. Das VHDL Programm ist in der Abbildung 37 präsentiert.

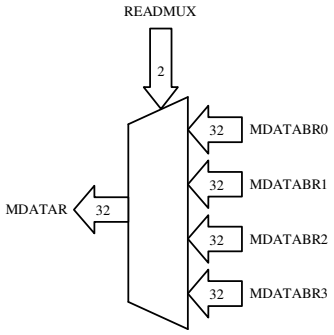

Abbildung 36 – Submodul EXU_MDMUX [Hor96]

```
--------------------------------------------------------------------------------
-- I C T Computer Technology Dept
-- University of Technology, Vienna
-- All Rights Reserved
-- Project: UTCSU
-- Designer: Martin HORAUER
-- Version: 1.0
-- File name: exupkg_mdmux_ent.vhd
-- Title: Modul Data Bus Multiplexer
-- Module: entity
-- Tools: Synopsys
-- Ref. lib.: IEEE
-- Ref. pkg.: 1164
-- Roadmap:
-- Date :10/96
--------------------------------------------------------------------------------
library IEEE;
  use IEEE.std_logic_1164.all;
  use work.exupkg.all;
entity EXU_MDMUX is
```

```
  port(
    MDATABR0: in std_logic_vector(31 downto 0);
    MDATABR1: in std_logic_vector(31 downto 0);
    MDATABR2: in std_logic_vector(31 downto 0);
    MDATABR3: in std_logic_vector(31 downto 0);
    MDATAR: out std_logic_vector(31 downto 0);
    READMUX: in std_logic_vector(1 downto 0)
  );
end EXU_MDMUX;
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- I C T Computer Technology Dept
-- University of Technology, Vienna
-- All Rights Reserved
-- Project: UTCSU
-- Designer: Martin HORAUER
-- Version: 1.0
-- File name: exu_mdmux_beh_arch.vhd
-- Title: Modul Data Bus Multiplexer adopted for EXU
-- Module: architecture
-- Tools: Synopsys
-- Ref. lib.: IEEE
-- Ref. pkg.: 1164
-- Roadmap:
-- Date: 10/96
-------------------------------------------------------------------------------
architecture behavioral of EXU_MDMUX is
begin
  -----------------------------------------------------------------------------
  -- Process: P_READMUX
  -- Purpose: Data Multiplexer
  -- Inputs: MDATABRx, READMUX
  -- Outputs: MDATAR
  -----------------------------------------------------------------------------
  P_READMUX: process(READMUX, MDATABR3, MDATABR2, MDATABR1, MDATABR0)
  begin
    if   (READMUX = "11") then MDATAR <= MDATABR3;   -- TSXMT
    elsif(READMUX = "10") then MDATAR <= MDATABR2;   -- MSXMT
    elsif(READMUX = "01") then MDATAR <= MDATABR1;   -- TSRCV
    elsif(READMUX = "00") then MDATAR <= MDATABR0;   -- MSRCV
    else MDATAR <= ZERO32;
    end if;
  end process;
end behavioral;
-------------------------------------------------------------------------------
```

Abbildung 37 – EXU_MDMUX VHDL Programm [Hor96]

Das Submodul EXU_ BODY ist in der Abbildung 38 illustriert. Das EXUBUS versorgt das EXU Modul mit Daten. Die Register MSRCV, TSRCV, MSXMT und TSXMT sammeln die Daten von EXUBUS, gesteuert von den Signalen TSRCVGATE, bzw. TSXMTGATE. Die gespeicherten Daten können durch die MDATAR Leitungen abgefragt werden. Die zwei Register, DUTYH und DUTYL, führen ein Zwischenspeichern der MDATAW Daten durch. Der DUTY Comparator wird durch das siebzehnte Bit der DUTYL Register gesteuert. Die DUTY Register werden mit dem EXUBUS[55,8] verglichen. Wenn der EXUBUS größer oder gleich ist als DUTY, wird am INTDUTY Ausgang ein Impuls für eine Taktperiode erzeugt. Das Ausgang SDUTY wird so lange aktiv bleiben, bis EXUBUS größer oder gleich ist als DUTY. Das VHDL Programm ist in der Abbildung 39 präsentiert.

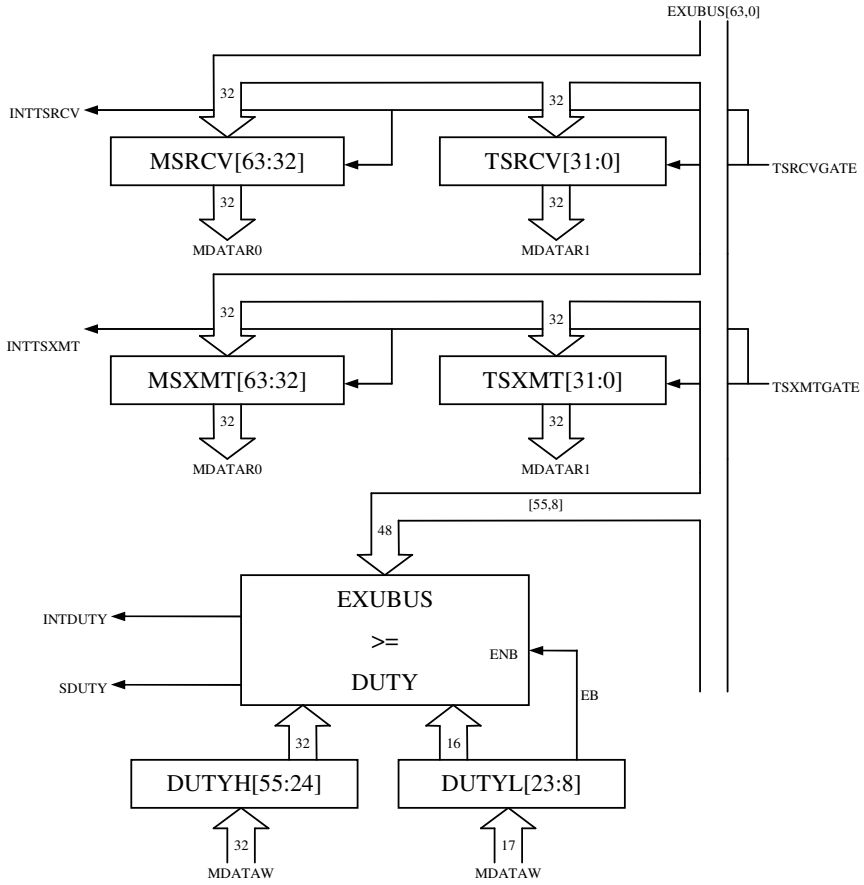

Abbildung 38 – Submodul EXU_BODY [Hor96]

```
-------------------------------------------------------------------------------
-- I C T Computer Technology Dept
-- University of Technology, Vienna
-- All Rights Reserved
-- Project: UTCSU
-- Designer: Martin HORAUER
-- Version: 1.0
-- File name: exu_body_ent.vhd
-- Title: Example Unit
-- Module: entity
-- Tools: Synopsys
-- Ref. lib.: IEEE
-- Ref. pkg.: 116
-- Roadmap:
-- Date: 10/96
-------------------------------------------------------------------------------
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.">=";
  use work.exupkg.all;
entity EXU_BODY is
  port(
    EXUBUS: in std_logic_vector(63 downto 0);
    MDATAW: in std_logic_vector(31 downto 0);
    RESET: in std_logic;
    CLK: in std_logic;
    ADDR: in std_logic_vector(1 downto 0);
    BE: in std_logic_vector(3 downto 0);
    RWBAR: in std_logic;
    INTTSRCV: out std_logic;
    TSRCVGATE: in std_logic;
    TSXMTGATE: in std_logic;
    INTTSXMT: out std_logic;
    INTDUTY: out std_logic;
    SDUTY: out std_logic;
    MSRCV: out std_logic_vector(31 downto 0);
```

```
    TSRCV: out std_logic_vector(31 downto 0);
    MSXMT: out std_logic_vector(31 downto 0);
    TSXMT: out std_logic_vector(31 downto 0)
  );
end EXU_BODY;
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
-- I C T Computer Technology Dept
-- University of Technology, Vienna
-- All Rights Reserved
-- Project: UTCSU
-- Designer: Martin HORAUER
-- Version: 1.0
-- File name: exu_body_beh_arch.vhd
-- Title: Example Unit
-- Module: architecture
-- Tools: Synopsys
-- Ref. lib.: IEEE
-- Ref. pkg.: 1164
-- Roadmap:
-- Date: 10/96
-------------------------------------------------------------------------------
-- The example unit samples timestamps on external events, as
-- can be transmit or receive pacckets. Receive packets trigger via TSRCV a time-
-- stamp, which is sampled into the MSRCV and TSRCV registers. Transmit packets
-- trigger TSXMT wich samples a timestamp into MSXMT and TSXMT.
-- The duty-timer DUTY is responsible to periodically trigger S/R/D/A-duties.
-- The duty-registers are loaded via two 32-bit wide registers and are enabled via the
-- 17th bit of the lower register. When the EXUBUS>=DUTY an INTDUTY + SDUTY are
-- driven active. EXUBUS<DUTY restores SDUTY w. the consecutive rising edge of CLK.
-- Otherwise SDUTY is restored one clock-period after the falling edge of the enable
-- of the current timer. The INTDUTY is just one period active.
-------------------------------------------------------------------------------
architecture behavioral of EXU_BODY is
  signal DUTYH: std_logic_vector(31 downto 0);
  signal DUTYL: std_logic_vector(15 downto 0);
  signal EB: std_logic;
  signal LSDUTY: std_logic;
  signal TMPINTDUTY: std_logic;
  signal TMPSDUTY: std_logic;
  signal QBARSDUTY: std_logic;
begin
  -----------------------------------------------------------------------------
  -- instantiation
  -----------------------------------------------------------------------------
  INTTSRCV <= TSRCVGATE;
  INTTSXMT <= TSXMTGATE;
  SDUTY <= LSDUTY;
  -----------------------------------------------------------------------------
  -- Process: P_MSRCV
  -- Purpose:
  -- Inputs: EXUBUS, RESET, MTCLK, TSRCVGATE
  -- Outputs: MSRCV
  -----------------------------------------------------------------------------
  P_MSCRV: process(EXUBUS, TSRCVGATE, CLK, RESET)
  begin
    if(RESET = '0')
    then
      MSRCV <= ZERO32;
    else
      if(CLK = '1' and CLK'event)
      then
        if(TSRCVGATE = '1')
        then
          MSRCV <= EXUBUS(63 downto 32);
        end if;
      end if;
    end if;
  end process P_MSCRV;
  -----------------------------------------------------------------------------
  -- Process: P_TSRCV
  -- Purpose:
  -- Inputs: EXUBUS, RESET, CLK, TSRCVGATE
  -- Outputs: TSRCV
  -----------------------------------------------------------------------------
  P_TSRCV: process(RESET, CLK, TSRCVGATE, EXUBUS)
```

```
  begin
    if(RESET = '0')
    then
      TSRCV <= ZERO32;
    else
      if(CLK = '1' and CLK'event)
      then
        if (TSRCVGATE='1')
        then
          TSRCV <= EXUBUS(31 downto 0);
        end if;
      end if;
    end if;
  end process P_TSRCV;
  -------------------------------------------------------------------------------
  -- Process: P_MSXMT
  -- Purpose:
  -- Inputs: CLK, RESET, EXUBUS, TSXMTGATE
  -- Outputs: MSXMT
  -------------------------------------------------------------------------------
  P_MSXMT: process(CLK, RESET, EXUBUS, TSXMTGATE)
  begin
    if(RESET='0')
    then
      MSXMT <= ZERO32;
    else
      if (CLK = '1' and CLK'event)
      then
        if (TSXMTGATE = '1')
        then
          MSXMT <= EXUBUS(63 downto 32);
        end if;
      end if;
    end if;
  end process P_MSXMT;
  -------------------------------------------------------------------------------
  -- Process: P_TSXMT
  -- Purpose:
  -- Inputs: RESET, CLK, EXUBUS, TSXMTGATE
  -- Outputs: TSXMT
  -------------------------------------------------------------------------------
  P_TSXMT: process(RESET, CLK, EXUBUS, TSXMTGATE)
  begin
    if(RESET = '0')
    then
      TSXMT <= ZERO32;
    else
      if(CLK = '1' and CLK'event)
      then
        if(TSXMTGATE = '1')
        then
          TSXMT <= EXUBUS(31 downto 0);
        end if;
      end if;
    end if;
  end process P_TSXMT;
  -------------------------------------------------------------------------------
  -- Process: P_DUTYH
  -- Purpose:
  -- Inputs: RWBAR, BE, RESET, CLK, MDATAW
  -- Outputs: DUTYH
  -------------------------------------------------------------------------------
  P_DUTYH: process(RWBAR, BE, RESET, CLK, MDATAW)
  begin
    if(RESET = '0')
    then
      DUTYH <= ZERO32;
    else
      if(CLK = '1' AND CLK'event)
      then
        if   (ADDR = "10" and RWBAR = '0' and BE = "0001")
        then
          DUTYH( 7 downto  0) <= MDATAW( 7 downto  0);
        elsif(ADDR = "10" and RWBAR = '0' and BE = "0010")
        then
          DUTYH(15 downto  8) <= MDATAW(15 downto  8);
        elsif(ADDR = "10" and RWBAR = '0' and BE = "0100")
```

```
        then
          DUTYH(23 downto 16) <= MDATAW(23 downto 16);
        elsif(ADDR = "10" and RWBAR = '0' and BE = "1000")
        then
          DUTYH(31 downto 24) <= MDATAW(31 downto 24);
        elsif(ADDR = "10" and RWBAR = '0' and BE = "0011")
        then
          DUTYH(15 downto  0) <= MDATAW(15 downto  0);
        elsif(ADDR = "10" and RWBAR = '0' and BE = "1100")
        then
          DUTYH(31 downto 16) <= MDATAW(31 downto 16);
        elsif(ADDR = "10" and RWBAR = '0' and BE = "1111")
        then
          DUTYH               <= MDATAW;
        end if;
      end if;
    end if;
  end process P_DUTYH;
  ------------------------------------------------------------------------------
  -- Process: P_DUTYL
  -- Purpose:
  -- Inputs: ADDR, RWBAR, BE, RESET, CLK, MDATAW
  -- Outputs: DUTYL, EB
  ------------------------------------------------------------------------------
  P_DUTYL: process(RWBAR, BE, RESET, CLK, MDATAW)
  begin
    if(RESET='0')
    then
      DUTYL <= ZERO16;
      EB <= '0';
    else
      if(CLK='1' and CLK'event)
      then
        if   (ADDR = "01" and RWBAR = '0' and BE = "0001")
        then
          DUTYL( 7 downto 0) <= MDATAW( 7 downto 0);
        elsif(ADDR = "01" and RWBAR = '0' and BE = "0010")
        then
          DUTYL(15 downto 8) <= MDATAW(15 downto 8);
        elsif(ADDR = "01" and RWBAR = '0' and BE = "0100")
        then
          EB <= MDATAW(16);
        elsif(ADDR = "01" and RWBAR = '0' and BE = "0011")
        then
          DUTYL(15 downto 0) <= MDATAW(15 downto 0);
        elsif(ADDR = "01" and RWBAR = '0' and BE = "1100")
        then
          EB <= MDATAW(16);
        elsif(ADDR = "01" and RWBAR = '0' and BE = "1111")
        then
          DUTYL <= MDATAW(15 downto 0);
          EB <= MDATAW(16);
        end if;
      end if;
    end if;
  end process P_DUTYL;
  --
  INTDUTY <= (TMPSDUTY and QBARSDUTY);
  LSDUTY <= TMPSDUTY;
  ------------------------------------------------------------------------------
  -- Process: P_DUTY
  -- Purpose:
  -- Inputs: DUTYH, DUTYL, EB, EXUBUS, CLK, RESET
  -- Outputs: INTDUTY, SDUTY
  ------------------------------------------------------------------------------
  P_DUTY: process(DUTYH, DUTYL, EB, EXUBUS, CLK, RESET)
  begin
    if(RESET = '0')
    then
      TMPSDUTY <= '0';
    else
      if (CLK = '1' AND CLK'event)
      then
        if   ((EXUBUS(55 downto 8) >= (DUTYH & DUTYL)) and EB = '1') then TMPSDUTY <= '1';
        elsif((EXUBUS(55 downto 8) >= (DUTYH & DUTYL)) and EB = '0') then TMPSDUTY <= '0';
        else                                                            TMPSDUTY <= '0';
        end if;
```

```
      end if;
    end if;
  end process P_DUTY;
  -------------------------------------------------------------------------
  -- Process: P_DUTYGATE
  -- Purpose:
  -- Inputs: CLK, TMPSDUTY, RESET
  -- Outputs: QBARSDUTY
  -------------------------------------------------------------------------
  P_DUTYGATE: process(TMPSDUTY, CLK, RESET)
  begin
    if(RESET = '0')
    then
      QBARSDUTY <= '0';
    else
      if(CLK = '1' and CLK'event)
      then
        QBARSDUTY <= not (TMPSDUTY);
      end if;
    end if;
  end process P_DUTYGATE;
end behavioral;
-------------------------------------------------------------------------
```

Abbildung 39 – EXU_BODY VHDL Programm [Hor96]

4.2.2 Testfälle identifizieren

Es werden beide Submodule (EXU_MDMUX und EXU_BODY) und das Hauptmodul EXU getestet. Das EXU_MDMUX Modul ist ein Multiplexer und wird getestet, ob die richtige Eingangsignale an den Ausgang umgeleitet werden. Die Eingangsignale und die Adressleitungen werden mit unterschiedlichen Werten gesetzt und das Ausgangsignal wird mit einem Referenzwert verglichen. Das EXU_BODY Modul speichert die Busdaten in vier Registern ab. Zu überprüfen ist, ob die abgespeicherten Daten richtig sind oder nicht. Zusätzlich wird auch die Funktionalität des Comparators getestet. Das Zusammenspiel der EXU_MDMUX und EXU_BODY wird mit der Überprüfung des Hauptmoduls EXU getestet. Der Einsatz eines Referenzmoduls ist in keinem Fall notwendig.

4.2.3 Design der Testbench

Signaldefinitionen

Die Signale werden auf Grund der Eingänge und Ausgänge des Hauptmoduls EXU, bzw. die Verbindungsleitungen der EXU_MDMUX mit EXU_BODY Module definiert. In der Tabelle 12 sind die Signaldefinitionen aufgelistet:

Signalname	Breite	Signaltyp	Initial Wert	Default Wert
CLK	1	CLOCK		
RESET	1	IN	1	1
ADDR	2	IN		0b00
BE	4	IN		0b0000
EXUBUS	64	IN		
MDATAW	32	IN		0x00000000
RWBAR	1	IN		1
TSRCVGATE	1	IN		
TSXMTGATE	1	IN		
INTDUTY	1	OUT		
INTTSRCV	1	OUT		
INTTSXMT	1	OUT		
SDUTY	1	OUT		
MSRCV	32	OUT		
TSRCV	32	OUT		
MSXMT	32	OUT		
TSXMT	32	OUT		
READMUX	2	IN	0b00	0b00
MDATABR0	32	IN		
MDATABR1	32	IN		
MDATABR2	32	IN		
MDATABR3	32	IN		
MDATAR	32	OUT		

Tabelle 12 – EXU Signaldefinitionen

Die Einführung der Konstanten erleichtert die Übersichtlichkeit der Signalwertdefinitionen. In der Tabelle 13 sind die Konstanten der EXU Module aufgelistet:

Signalname	Type	Wert
WAITCYCLE	time	CLK.PERIOD * 0
ADDR_INIT	time	5ns
ADDR_HOLD	time	5ns
BE_INIT	time	5ns
BE_HOLD	time	5ns
MDATAW_INIT	time	10ns
MDATAW_HOLD	time	5ns
RWBAR_INIT	time	5ns
RWBAR_HOLD	time	5ns

Tabelle 13 – EXU Konstantendefinitionen

Die Signalwertdefinitionen der EXU Module sind in der Tabelle 14 dargestellt:

Signalname	Signalwerte			
	Name	Dauer	Wert	Dehnbar
RESET	SETUP	3 * CLK.PERIOD + WAITCYCLE		
ADDR	SETUP	CLK.PERIOD – ADDR_INIT	0b00	
	VALID	CLK.PERIOD + ADDR_INIT + ADDR_HOLD + WAITCYCLE		
	FINISH	CLK.PERIOD – ADDR_HOLD	0b00	
BE	SETUP	ADDR.SETUP	0x00	
	VALID	ADDR.VALID + BE_INIT + BE_HOLD		
	FINISH	ADDR.FINISH – BE_INIT – BE.HOLD	0x00	
EXUBUS	SETUP	3ns		
	VALID	3 * CLK.PERIOD – EXUBUS.SETUP + WAITCYCLE		
MDATAW	SETUP	2 * CLK.PERIOD – MDATAW_INIT + WAITCYCLE	0x00000000	
	VALID	MDATAW_INIT + MDATAW_HOLD		
	FINISH	CLK.PERIOD – MDATAW_HOLD	0x00000000	
RWBAR	SETUP	CLK.PERIOD – RWBAR_INIT	1	
	VALID	ADDR.VALID + RWBAR_INIT + RWBAR_HOLD		
	FINISH	ADDR.FINISH – RWBAR_INIT – RWBAR_HOLD	1	
TSRCVGATE	VALID	3 * CLK.PERIOD + WAITCYCLE		
	FINISH			
TSXMTGATE	VALID	3 * CLK.PERIOD + WAITCYCLE		
	FINISH			
INTDUTY	VALID			ja
INTTSRCV	VALID			ja
INTTSXMT	VALID			ja
SDUTY	VALID			ja
MSRCV	VALID			ja
TSRCV	VALID			ja
MSXMT	VALID			ja
TSXMT	VALID			ja
READMUX	VALID	3 * CLK.PERIOD + WAITCYCLE		
MDATABR0	VALID			ja
MDATABR1	VALID			ja
MDATABR2	VALID			ja
MDATABR3	VALID			ja
MDATAR	VALID			

Tabelle 14 – EXU Signalwertdefinitionen

Befehlsatz

Der EXU Befehlsatz beinhaltet alle Befehle, womit die Testprogramme realisiert werden können. Die Befehle werden aus den EXU Signaldefinitionen zusammengesetzt. Die nicht gesetzten Signalwerte werden bei der Programmerstellung gesetzt. Die Reihenfolge der nicht gesetzten Signalwerte bei der Befehlsdefinition ist bei der Programmerstellung zu berücksichtigen. In der Tabelle 12 sind die Befehlsdefinitionen aufgelistet:

Befehl	**Beschreibung**	**Signalwerte**	
		Name	**Wert**
RESET	Setzt das System in den Initialzustand.	RESET.SETUP	0
DUTYH	Setzt den DUTYH Register.	ADDR.VALID	0b10
		BE.VALID	0x0F
		EXUBUS.VALID	
		MDATAW.VALID	
		READMUX.VALID	0b00
		RWBAR.VALID	0
		TSRCVGATE.VALID	0
		TSXMTGATE.VALID	0
DUTYL	Setzt den DUTYL Register.	ADDR.VALID	0b01
		BE.VALID	0x0F
		EXUBUS.VALID	
		MDATAW.VALID	
		READMUX.VALID	0b00
		RWBAR.VALID	0
		TSRCVGATE.VALID	0
		TSXMTGATE.VALID	0
TSRCV	Setzt den TSRCV Register.	ADDR.VALID	0b01
		BE.VALID	0x0F
		EXUBUS.VALID	
		MDATAW.VALID	
		READMUX.VALID	0b01
		RWBAR.VALID	1
		TSRCVGATE.VALID	1
		TSXMTGATE.VALID	0

TSXMT	Setzt den TSXMT Register.	ADDR.VALID	0b11
		BE.VALID	0x0F
		EXUBUS.VALID	
		MDATAW.VALID	
		READMUX.VALID	0b11
		RWBAR.VALID	1
		TSRCVGATE.VALID	0
		TSXMTGATE.VALID	1
MSRCV	Setzt den MSRCV Register.	ADDR.VALID	0b00
		BE.VALID	0x0F
		EXUBUS.VALID	
		MDATAW.VALID	
		READMUX.VALID	0b00
		RWBAR.VALID	1
		TSRCVGATE.VALID	1
		TSXMTGATE.VALID	0
MSXMT	Setzt den MSXMT Register.	ADDR.VALID	0b10
		BE.VALID	0x0F
		EXUBUS.VALID	
		MDATAW.VALID	
		READMUX.VALID	0b10
		RWBAR.VALID	1
		TSRCVGATE.VALID	0
		TSXMTGATE.VALID	1
NULL	Für Erstellen einer Zeitverzögerung oder als Platzhalter verwendbar.	ADDR.VALID	0b00
		BE.VALID	0x0F
		EXUBUS.VALID	0x0FFFFFFFFFFFFFFF
		MDATAW.VALID	0x0FFFFFFF
		READMUX.VALID	0b00
		RWBAR.VALID	1
		TSRCVGATE.VALID	0
		TSXMTGATE.VALID	0

Tabelle 15 – EXU Befehlsatz

In dem nächsten Abschnitt werden die Testprogramme mit den definierten Befehlsatz realisiert.

4.2.4 Realisierung der Testbench

Testbench-Struktur und Parameter

Für die Überprüfung der EXU Module reicht die Testbench-Struktur aus dem Kapitel 2.3 aus. Die Testbench besteht aus einem Stimuligenerator, Taktgenerator, Comparator und Reportgenerator. Der Stimuligenerator liest die Stimulidaten aus der Stimulidatei und setzt die Signalwerte. Der Taktgenerator liest die Taktdefinition aus der Konfigurationsdatei und erstellt das Taktsignal. Der Comparator vergleicht die Ausgangsdaten mit den Referenzwerten und das Resultat schreibt der Reportgenerator in eine Report Datei.

Die default Zeiteinheit wird auf „ns“ gesetzt. Die Parameter der Taktdefinition sind die Periodendauer und die Zeit bis zur steigenden Flanke des ersten Impulses. Der zweite Impuls wird nicht verwendet. Die Periodendauer wird auf 40ns gesetzt und die Zeit bis zur steigenden Flanke des ersten Impulses auf 20ns. Die Parameter der Taktdefinition werden in die Konfigurationsdatei abgespeichert und sind in der Abbildung 40 dargestellt. Die Parameter der Reportgenerator beschränken sich auf den Dateiname der Reportdatei.

```
...
--CLOCK  PERIODE  RIS_EDGE1  FAL_EDGE1  RIS_EDGE2  FAL_EDGE2  OFFSET
  CLOCK  40ns     20ns       20ns       0ns        0ns        0ns
...
```

Abbildung 40 – Parameter der Taktdefinition

Anschließend werden die Moduldefinitionen mit den Signalzuordnungen für EXU, EXU_MDMUX und EXU_BODY erstellt. Die Testbench wird jeweils pro Moduldefinition generiert. In der Abbildung 41 – EXU Testbench wird der Testbench für das EXU Modul dargestellt. Der Name der EXU Entity und Architecture werden auf EXU_TB_ENTITY und EXU_TB_ARCHITECTURE gesetzt.

```
-------------------------------------------------------------------------------
entity EXU_TB_ENTITY is
end EXU_TB_ENTITY;
-------------------------------------------------------------------------------

-------------------------------------------------------------------------------
architecture EXU_TB_ARCHITECTURE of EXU_TB_ENTITY is
  --
  signal VSTART: bit;
  --
  variable VPERIODE: time;
  variable VRISEDGE1: time;
  variable VFALEDGE1: time;
  variable VRISEDGE2: time;
  variable VFALEDGE2: time;
  variable VOFFSET: time;
  --
  signal ADDR: std_logic_vector(1 downto 0);
  signal RWBAR: std_logic;
  signal BE: std_logic_vector(3 downto 0);
  signal CLK: std_logic;
  signal RESET: std_logic;
  signal MDATAR: std_logic_vector(31 downto 0);
  ...
  component EXU
    port(
      ADDR: in std_logic_vector(1 downto 0);
      RWBAR: in std_logic;
      BE: in std_logic_vector(3 downto 0);
      CLK: in std_logic;
```

```
      RESET: in std_logic;
      MDATAR: out std_logic_vector(31 downto 0);
      ...
    );
  end component;
  --
begin
  --
  COMP_EXU: EXU
    port map(
      ADDR,
      RWBAR,
      BE,
      CLK,
      RESET,
      MDATAR,
      ...
    );
  ------------------------------------------------------------------------
  -- Process: PROC_CONFIG
  -- Purpose: Read the configuration file
  ------------------------------------------------------------------------
  PROC_CONFIG: process
    file CONFIGFILE: text is in "muxConfig.cfg";
    variable VLINE: line;
    variable VCONTROL: string;
    variable VCLOCKFOUND: boolean;
  begin
    --
    -- Set the start flag
    --
    VSTART <= '0';
    --
    -- Read the clock parameters from the configfile
    --
    VCLOCKFOUND := false;
    while not (endfile(CONFIGFILE))
    loop
      --
      readline(CONFIGFILE, VLINE);
      read(VLINE, VCONTROL);
      --
      if(VCONTROL = "CLOCK")
      then
        read(VLINE, VPERIODE);
        read(VLINE, VRISEDGE1);
        read(VLINE, VFALEDGE1);
        read(VLINE, VRISEDGE2);
        read(VLINE, VFALEDGE2);
        read(VLINE, VOFFSET);
        VCLOCKFOUND := true;
      end if;
      ...
    end loop;
    --
    -- Checks the clock parameters
    --
    if(VCLOCKFOUND = false)
    then
      assert false report "Clock Parameters not found";
      wait;
    end if;
    ...
    --
    -- Set the start flag
    --
    VSTART <= '1';
    --
  end process PROC_CONFIG;
  ------------------------------------------------------------------------
  -- Process: PROC_CLK
  -- Purpose: Generate the clock signal
  ------------------------------------------------------------------------
  PROC_CLK: process
  begin
    --
    wait on VSTART until VSTART = '1';
```

```
    --
    wait for VOFFSET;
    while true
    loop
      CLK <= '0'; wait for VRISEDGE1;
      CLK <= '1'; wait for VFALEDGE1;
      CLK <= '0'; wait for VRISEDGE2;
      CLK <= '1'; wait for VFALEDGE2;
    end loop;
  end process PROC_CLK;
  ------------------------------------------------------------------------------
  -- Process: PROC_STIMULI
  -- Purpose: Read the stimulifile and generate the stimuli and reference signals
  ------------------------------------------------------------------------------
  PROC_STIMULI: process
    file STIMULIFILE: text is in "muxStimuli.cfg";
    variable VLINE: line;
    variable VSIGNALNAME: string;
    variable VREFERENCE: character;
    variable VWAITFOR: time;
    --
    -- IN Signals
    --
    variable VADDR: std_logic_vector(1 downto 0);
    ...
    --
    -- OUT Signals
    --
    variable VMDATAR: std_logic_vector(31 downto 0);
    variable REF_VMDATAR: std_logic_vector(31 downto 0);
    ...
  begin
    --
    wait on VSTART until VSTART = '1';
    --
    while not (endfile(STIMULIFILE))
    loop
      readline(STIMULIFILE, VLINE);
      read(VLINE, VSIGNALNAME);
      --
      -- IN Signals
      --
      if (VSIGNALNAME = "ADDR") then read(VLINE, VADDR); ADDR <= VADDR; end if;
      ...
      --
      -- OUT Signals
      --
      if (VSIGNALNAME = "MDATAR") then read(VLINE, VMDATAR); REF_MDATAR <= VMDATAR; end if;
      ...
      --
      -- IN/OUT Signals - yet not exists
      --
      if(VSIGNALNAME = "XXX")
      then
        read(VLINE, VXXX);
        read(VLINE, VREFERENCE);
        if(VREFERENCE = 'R') then REF_XXX <= VXXX;
        else                      XXX <= VXXX;
        end if;
      end if;
      ...
      --
      -- WAIT
      --
      if(VSIGNALNAME = "WAIT") then read(VLINE, VWAITFOR); wait for VWAITFOR; end if;
      --
    end loop;
    --
    -- Stimulifile terminated
    --
    assert false report "Stimulifile terminated";
    wait;
    --
  end process PROC_STIMULI;
  ------------------------------------------------------------------------------
  -- Process: PROC_COMPARATOR
  -- Purpose: Compare the out signals with the reference signals
```

```
  ------------------------------------------------------------------------------
  PROC_COMPARATOR: process(MDATAR,   REF_MDATAR,
                           INTTSRCV, REF_INTTSRCV,
                           ...)
    file REPORTFILE: text is out "muxReport.rep";
  begin
    if(MDATAR'event or REF_MDATAR'event)
    then
      --
      write(VLINE, "MDATAR ");     write(VLINE, MDATAR);     write(VLINE, " ");
      write(VLINE, "REF_MDATAR "); write(VLINE, REF_MDATAR);
      --
      if(MDATAR = REF_MDATAR) then write(VLINE, " OK");
      else                         write(VLINE, " ERROR");
      end if;
      --
      writeline(REPORTFILE, VLINE);
      --
    end if;
    ...
  end process PROC_COMPARATOR;
  --
end EXU_TB_ARCHITECTURE;
------------------------------------------------------------------------------
```

Abbildung 41 – EXU Testbench

Testprogramme und Stimulidaten

Das Testprogramm präsentiert in der Abbildung 42 testet die Funktionalität des EXU Hauptmoduls. Nach einem Initialisierungsvorgang werden der Comparator und anschließend die vier Register getestet.

```
...
RESET
DUTYH 0x000FFFFFFFFFFD00,0x0FFFFFFF
DUTYL 0x000FFFFFFFFFFD00,0x00A37FFFE
NULL
NULL
DUTYH 0x000FFFFFFFFF0000,0x0FFF0FFFF
DUTYL 0x000FFFFFFFFFFD00,0x00A37FFFE
NULL
NULL
DUTYL 0x000FFFFFFFFFFF00,0x00A30FFFE
NULL
TSRCV 0x0000000FFFFFFFE00,0x022226789
NULL
TSXMT 0x00123ACDE5432F012,0x022226789
MSRCV 0x00123ACDE5432F012,0x003C10E23
TSRCV 0x00123ACDE5432F012,0x003C10E23
MSXMT 0x00123ACDE5432F012,0x003C10E23
TSXMT 0x00123ACDE5432F012,0x003C10E23
...
```

Abbildung 42 – EXU Testprogramm

Aus dem EXU Testprogramm resultieren die Stimulidaten. Eine Sequenz der Stimulidaten ist in der Abbildung 43 präsentiert.

```
...
------------------------------------------------------------------------------
--RESET
------------------------------------------------------------------------------
RESET        '0'
ADDR         "00"
BE           X"0"
EXUBUS       X"XXXX_XXXX_XXXX_XXXX"
MDATAW       X"0000_0000"
READMUX      "00"
RWBAR        '1'
TSRCVGATE    'X'
```

```
TSXMTGATE    'X'
         WAIT         120ns
-------------------------------------------------------------------------------
--DUTYH 0x000FFFFFFFFFFD00,0x0FFFFFFF
-------------------------------------------------------------------------------
RESET        '1'
ADDR         "00"
BE           X"0"
EXUBUS       X"XXXX_XXXX_XXXX_XXXX"
MDATAW       X"0000_0000"
READMUX      "00"
RWBAR        '1'
TSRCVGATE    '0'
TSXMTGATE    '0'
INTDUTY      '0'                        R
SDUTY        '0'                        R
INTTSRCV     '0'                        R
INTTSXMT     '0'                        R
MDATAR       X"0000_0000"               R
         WAIT         3ns
EXUBUS       X"00FF_FFFF_FFFF_FD00"
         WAIT         32ns
ADDR         "10"
BE           X"F"
RWBAR        '0'
         WAIT         35ns
MDATAW       X"FFFF_FFFF"
         WAIT         15ns
ADDR         "00"
MDATAW       X"0000_0000"
         WAIT         10ns
BE           X"0"
RWBAR        '1'
         WAIT         30ns
-------------------------------------------------------------------------------
--DUTYL 0x000FFFFFFFFFFD00,0x00A37FFFE
-------------------------------------------------------------------------------
RESET        '1'
ADDR         "00"
BE           X"0"
EXUBUS       X"XXXX_XXXX_XXXX_XXXX"
MDATAW       X"0000_0000"
READMUX      "00"
RWBAR        '1'
TSRCVGATE    '0'
TSXMTGATE    '0'
INTDUTY      '0'                        R
SDUTY        '0'                        R
INTTSRCV     '0'                        R
INTTSXMT     '0'                        R
MDATAR       X"0000_0000"               R
         WAIT         3ns
EXUBUS       X"00FF_FFFF_FFFF_FD00"
         WAIT         32ns
ADDR         "01"
BE           X"F"
RWBAR        '0'
         WAIT         35ns
MDATAW       X"0A37_FFFE"
         WAIT         15ns
ADDR         "00"
MDATAW       X"0000_0000"
         WAIT         10ns
BE           X"0"
RWBAR        '1'
         WAIT         25ns
-------------------------------------------------------------------------------
--NULL
-------------------------------------------------------------------------------
RESET        '1'
ADDR         "00"
BE           X"F"
EXUBUS       X"FFFF_FFFF_FFFF_FFFF"
MDATAW       X"FFFF_FFFF"
READMUX      "00"
RWBAR        '1'
TSRCVGATE    '0'
```

```
TSXMTGATE    '0'
         WAIT         120ns
...
```

Abbildung 43 – EXU Stimulidaten

Aus dem EXU Testprogramm können auch die Latex Makro Daten generiert werden. Eine Sequenz der Latex Makro Datei ist in der Abbildung 44 präsentiert.

```
\begin{timing}[null]{null}
\tin{1}{CLK}
\tin{2}{RWBAR}
\tin{3}{BE}
\tin{4}{RESET}
\til{1}{LLHHLLHHLLHHLLHHLLHHLLHHLLHHLLHHLLHHLLHH}
\til{2}{HLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLH}
\til{3}{XXVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVX}
\til{4}{LLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLLL}
...
\end{timing}
```

Abbildung 44 – Latex Macro Daten

Aus diesen Makros können die Zeitdiagramme der Signalabläufe erstellt werden.

4.2.5 Simulation und Auswertung der Ergebnisse

Nach der Simulation wird die Report Datei ausgewertet. Wenn die Ausgangssignalwerte und die Referenzwerte nicht übereinstimmen, wird in der Report Datei die Stelle mit „ERROR“ markiert. Ein Ergebnis über die korrekte Funktionalität des getesteten Moduls kann sehr schnell getroffen werden.

5 Analyse

5.1 Zusammenfassung

Die präsentierten Testbench-Strukturen mit oder ohne Controller können bei den meisten ASIC Entwicklungen, unabhängig von den verwendeten HDL Sprache, eingesetzt werden. Diese ermöglichen die Versorgung der Testobjekte mit Stimulidaten und die Auswertung der Reaktionen. Die Stimulidaten und Referenzdaten sind in einer Stimulidatei abgelegt und werden während der Simulation herausgelesen.

Ausgehend von den Testbench-Strukturen und von Überlegungen, die die Entwicklung einer Testbench und die Erfassung der Stimulidaten erleichtern, wurde das Design des Testbench-Generators realisiert. Dabei wurden das Datenmodell und die Architektur des Testbench-Generators definiert. Das Datenmodell des Testbench-Generators definiert alle Daten, die für eine Testbench notwendig sind. Ein wichtiger Aspekt des Modells ist das Vermeiden der redundanten Informationen. Eine nachträgliche Änderung soll auf alle abhängigen Objekte Auswirkung haben. Die wichtigsten Business Objekte des Modells sind die Signaldefinition, die Signalwertdefinition, die Befehlsdefinition, die Programmdefinition und die Moduldefinition. Die Architektur des Testbench-Generators definiert drei Schichten. Die Client-Schicht, wo die Eingabe und die Visualisierung der Daten stattfinden, die Business-Schicht, wo die Datenverarbeitung stattfindet und die EIS (Enterprise Information System) Schicht, wo die Daten dauerhaft gespeichert werden.

Die Implementierung des Testbench-Generators wurde mit der Programmiersprache Java als eine eigenständige Client-Applikation realisiert. Die definierten Schichten wurden zusammengelegt. Die Benutzeroberfläche besteht aus mehreren Eingabemasken, die unterstützen die Erfassung aller Daten, die für eine Testbench notwendig sind. Die Steuerung des Testbench-Generators wurde Menügesteuert realisiert. Die unterstützte Testbench-Struktur ist die Struktur ohne Controller. Eine weitere Generator-Implementierung unterstützt die Generierung der Latex Dateien. Die erfassten Daten werden im XML Format oder, in einer serialisierten Form, in Dateien abgespeichert. Um die Wiederverwendbarkeit der Befehle zu gewährleisten, werden die Befehlsdefinitionen und die Signaldefinitionen in eigene Befehlsbibliothek Dateien abgespeichert.

Eine mögliche Vorgehensweise der Testbench-Entwicklung wurde beschrieben. Das Design und Realisierung der Testbench wird vom Testbench-Generator unterstützt. Das Beispiel demonstriert die Realisierung einer Testbench mit Hilfe des Testbench-Generators.

Der Testbench-Generator ist derzeit noch als Prototyp zu betrachten. Um ein professionelles Werkzeug zu realisieren ist eine Weiterentwicklung notwendig. Für eine Weiterentwicklung stehen das Datenmodell, der Java Source-Code und die Schnittstellen (APIs) zur Verfügung.

5.2 Ausblick

Der Testbench-Generator kann auch als eine Web-Anwendung implementiert werden. Als Middleware Plattform wäre eine J2EE- oder .Net-Plattform geeignet. Eine J2EE-Implementierung hat den Vorteil, dass die bereits realisierten Java Programmmodulen teilweise wieder verwendbar sind. Als J2EE Referenzhandbuch siehe [J2E04]. Als weiteren Verbesserungen der Benutzeroberfläche können die Mehrsprachigkeit, die Mehrbenutzerfähigkeit oder die Speicherung der Benutzereinstellungen erwähnt werden.

Das Setzen der Befehlsvariablen mit Wertebereichen ermöglicht eine flächendeckende Domainanalyse der UUT. Die Wertebereiche werden entweder in der Testbench oder in der Stimulidatei in allen einzelnen Werten des Wertebereichs aufgelöst.

Syntax:

```
domain ::= min_value..max_value
min_value ::= der kleinste Wert des Domains
max_value ::= der größte Wert des Domains
```

Beispiel: Die Instanz des „`ACC var1, var2`“ Befehls, mit var1 aus dem Wertebereich 1 bis 10 und var2 mit dem Wert 0x00, ist „`ACC 1..10, 0x00`“.

Ein Zufallsdatengenerator könnte die Befehlsvariablen mit Daten versorgen. Das Problem mit den Zufallsdaten ist, dass die Referenzwerte für die Zufallsdaten nicht definiert werden können. Der Einsatz von Zufallswerten ist nur in einer Testbench mit einem Referenzmodul sinnvoll.

Die Rückkopplung der UUT Ausgangssignale ermöglicht die Realisierung eines adaptiven Tests. Die bedingte Befehlsausführung kommt oft bei Protokolltests zum Einsatz. Der Testbench-Generator soll die Realisierung von Testprogrammen mit bedingter Befehlsausführung ermöglichen. Mit einfachen „if else“ Bedingungen kann bereits ein adaptiver Test realisiert werden.

Die Implementierung eines VERILOG Generators ermöglicht den Einsatz des Testbench-Generators auch für die ASICs, die in VERILOG geschrieben wurden. Der VERILOG Generator erstellt die VERILOG Testbench und die VERILOG Testdaten.

Literaturverzeichnis

[Arm00] James R. Armstrong, F. Gail Gray:
VHDL Design. Representation and Synthesis (Second Edition), Prentice Hall PTR, 2000

[Dou04] Doulos:
VHDL Testbench, http http://www.doulos.com/knowhow/perl/, 2004

[Eck02] Bruce Eckel:
Thinking in Java, 3rd Edition, Prentice Hall, 2002

[EDA04] EDA Industry Working Groups:
VHDL-2002 revision, http://www.eda.org/isac, 2004

[Erl04] Universität Erlangen Nürnberg:
VHDL Testbench-Generator, http://www.vhdl-online.de/TB-GEN, 2004

[Ful04] Full Circuit Ltd:
VHDL TestBench Tool, http://www.fullcircuit.com/TBtool.htm, 2004

[Hag95] Klaus ten Hagen:
Abstrakte Modellierung digitaler Schaltungen, Springer Verlag, 1995

[Hit99] Martin Hitz, Gerti Kappel:
UML @ Work. Von der Analyse zur Realisierung, dpunkt.verlag, 1999

[Hof97] Michael Hofstätter:
Diplomarbeit. VHDL-Package zur Unterstützung des VHDL Testbench-Entwurfs, TU Wien, 1997

[Hor96] Martin Horauer:
Synopsys, http://www.ict.tuwien.ac.at/horauer/Publikationen/cad.pdf, 1996, S. 7-21

[J2E04] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, Eric Jendrock:
The J2EE 1.4 Tutorial, Sun Microsystems, 2004

[May00] Ludwig May, Jens Leilich:
TIMING.STY Satz von Zeitdiagrammen für digitale Schaltungen, 2000

[Men04] Mentor Graphics:
Seamless, http://www.mentor.com/seamless/, 2004

[SOC04] SOC central:
TestWizard, http://www.soccentral.com/results.asp?entryID=1734, 2004